
MECADEMIC
INDUSTRIAL ROBOTICS

MC-PM-MECA500

Revision number: 11.3.84

Mecademic Robotics

December 02, 2025

Contents

1 Programming Manual for the Meca500 Industrial Robot	1
2 About this manual	2
3 Basic theory and definitions	5
4 TCP/IP communication	29
5 Communicating over cyclic protocols	50
6 EtherCAT communication	112
7 EtherNet/IP communication	117
8 PROFINET communication	123
9 Troubleshooting	132
10 Motion commands	135
11 Robot control commands	185
12 Data request commands	228
13 Real-time data request commands	271
14 Work zone supervision and collision prevention commands	296
15 Commands for optional accessories	310
16 Commands for managing variables (beta)	352
17 Terminology	364

Programming Manual for the Meca500 Industrial Robot

```
22     with initializer.RobotWithTools() as robot:
23         # CHECK THAT IP ADDRESS IS CORRECT! #
24         try:
25             robot.Connect(address='192.168.0.100')
26         except mdr.CommunicationError as e:
27             logger.info(f'Robot failed to connect. Is the IP address correct? {e}')
28             raise e
29
30         try:
31             # Reset the robot configuration (joint limits, work zone, etc.)
32             #initializer.reset_robot_configuration(robot)
33
34             # Send the commands to get the robot ready for operation.
35             logger.info('Activating and homing robot...')
36             initializer.reset_motion_queue(robot, activate_home=True)
37             initializer.reset_vacuum_module(robot)
38
39             # Pause execution until robot is homed.
40             robot.WaitHomed()
41             logger.info('Robot is homed and ready.')
42
43             # Send motion commands to have the robot draw out a square.
44             if tools.robot_model_is_meca500(robot.GetRobotInfo().robot_model):
45                 robot.MovePose(200, 0, 300, 0, 90, 0)
```

For firmware version: 11.3

Document revision: A

Online release date: December 02, 2025

Document ID: MC-PM-MECA500

The information contained herein is the property of Mecademic Inc. and shall not be reproduced in whole or in part without prior written approval of Mecademic Inc. The information herein is subject to change without notice and should not be construed as a commitment by Mecademic Inc. This manual will be periodically reviewed and revised.

Mecademic Inc. assumes no responsibility for any errors or omissions in this document.

© Copyright 2025, Mecademic Inc.

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-MECA500](#)) 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 Meca500 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
A	November 24, 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 ($^{\circ}$) and time is in seconds (s), except for timestamps.

Joint numbering

The joints of the Meca500 are numbered in ascending order, starting from the base, as shown in Figure 1a. Figure 1 also shows the zero joint positions. In that “zero” robot position, the axis of joint 3 intersects the axis of joint 1, the axes of joints 4 and 6 are aligned and normal to the axis of joint 1, and the axes of joints 2, 3, and 5 are parallel. Finally, note the location of the small screw in the robot’s flange, which is the 20-mm disk with threaded holes at the extremity of the robot arm.

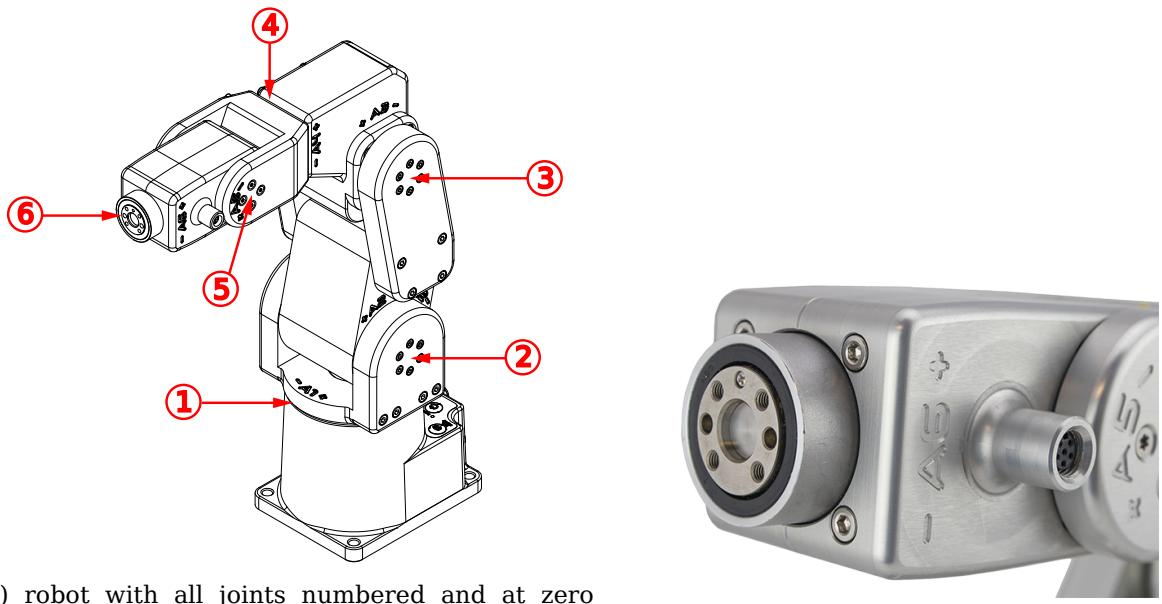


Figure 1: Meca500’s joint numbering and zero-degree joint position

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 364): *Base reference frame* (page 364). 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 366): *World reference frame* (page 366). The main static reference frame coincides with the BRF by default. It can be defined with respect to the BRF using the *SetWrf* (page 184) command.
- **FRF** (page 364): *Flange reference frame* (page 364). Mobile reference frame fixed to the robot's flange. The z axis coincides with the axis of joint 6, and points outwards. Its origin lies on the plane passing through the flange's mating surface. Finally, when all joints are at zero, the y axis of the FRF has the same direction as the y axis of the BRF.
- **FCP** (page 364): *Flange center point* (page 364). Origin of the FRF.
- **TRF** (page 366): *Tool reference frame* (page 366). 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 182) command.
- **TCP** (page 366): *Tool center point* (page 366). Origin of the TRF. (Not to be confused with the Transmission Control Protocol acronym, which is also used in this document.)

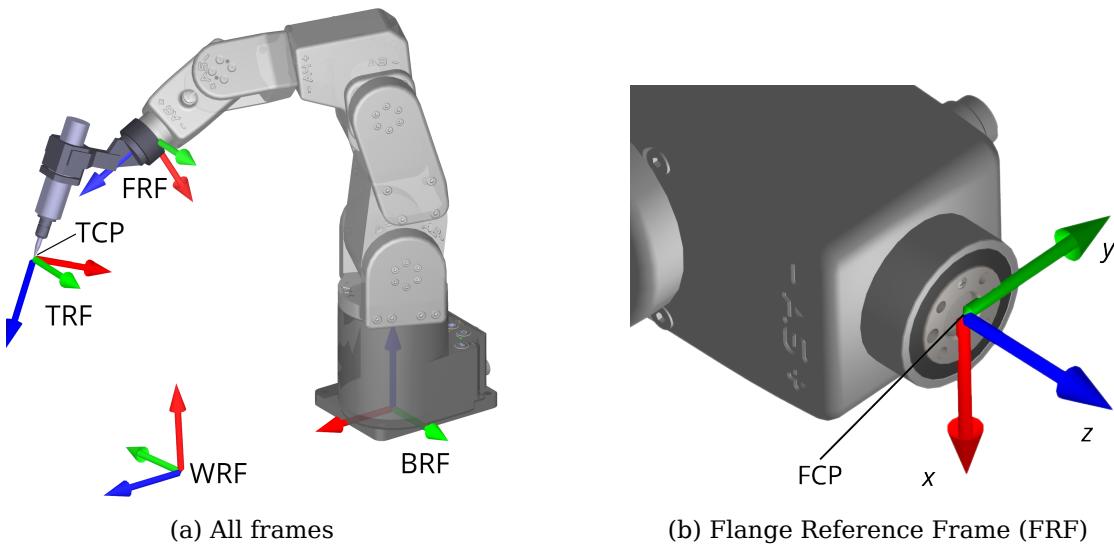


Figure 2: Reference frames for the Meca500

Pose and Euler angles

Some Mecademic commands accept *pose* (page 365) (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 an orientation specified in *Euler angles* (page 364), $\{\alpha, \beta, \gamma\}$, according to the *mobile XYZ convention* (also referred to as XYZ intrinsic rotations, RxRyRz, or XY"Z"). In this convention, if the orientation of a frame F_1 with respect to a frame F_0 is described by the Euler angles $\{\alpha, \beta, \gamma\}$, it means that if you align a frame F_m with frame F_0 , then rotate F_m about its x axis by α (alpha) degrees, then about its y axis by β (beta) degrees, and finally about its z axis by γ (gamma) degrees, the final orientation of frame F_m will be the same as that of frame F_1 .

Figure 3 shows an example of specifying orientation using the mobile XYZ Euler angle

convention. The diagram on the right shows the black reference frame orientation with respect to the gray reference frame with the Euler angles $\{45^\circ, -60^\circ, 90^\circ\}$.

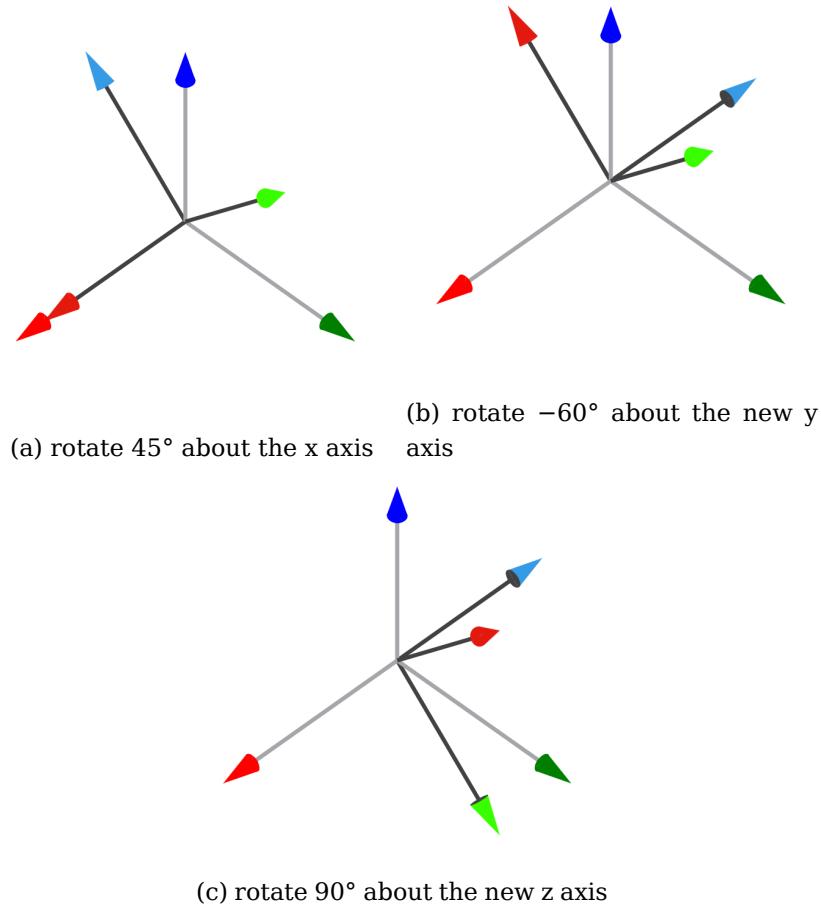


Figure 3: The three consecutive rotations associated with the Euler angles $\{45^\circ, -60^\circ, 90^\circ\}$

Because there are infinitely many sets of Euler angles that define a given orientation, the commands that accept a pose as arguments, accept any numerical value for the three Euler angles (e.g., the set $\{378.34^\circ, -567.32^\circ, 745.03^\circ\}$). However, we output only the equivalent Euler angle set $\{\alpha, \beta, \gamma\}$, for which $-180^\circ \leq \alpha \leq 180^\circ$, $-90^\circ \leq \beta \leq 90^\circ$ and $-180^\circ \leq \gamma \leq 180^\circ$. Furthermore, if you specify the Euler angles $\{\alpha, \pm 90^\circ, \gamma\}$, the controller will always return an equivalent Euler angle set with $\alpha = 0$. *Thus, it is perfectly normal that the Euler angles used to specify an orientation are not the same as the Euler angles returned by the controller, once that orientation has been attained* (see our tutorial on [Euler angles](#)).

Joint positions and last joint turn configuration

The angle associated with a rotational joint i , θ_i , will be referred to as *joint position* (page 365). Since the last joint of the robot (joint 6) 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 of rotation for each joint are engraved on the robot's body. As previously mentioned, all joint positions are zero in [Figure 1](#).

The mechanical limits of the first five joints of the Meca500 are as follows:

$$\begin{aligned}-175^\circ &\leq \theta_1 \leq 175^\circ, \\ -70^\circ &\leq \theta_2 \leq 90^\circ, \\ -135^\circ &\leq \theta_3 \leq 70^\circ, \\ -170^\circ &\leq \theta_4 \leq 170^\circ, \\ -115^\circ &\leq \theta_5 \leq 115^\circ.\end{aligned}$$

Joint 6 has no mechanical limits, but its software limits are ± 100 turns. Finally, we define the integer c_t as the joint 6 [turn configuration parameter](#) (page 366), so that

$$-180^\circ + c_t 360^\circ < \theta_6 \leq 180^\circ + c_t 360^\circ.$$

Joints can be further constrained using the [SetJointLimits](#) (page 206) 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 $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$. 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 365).

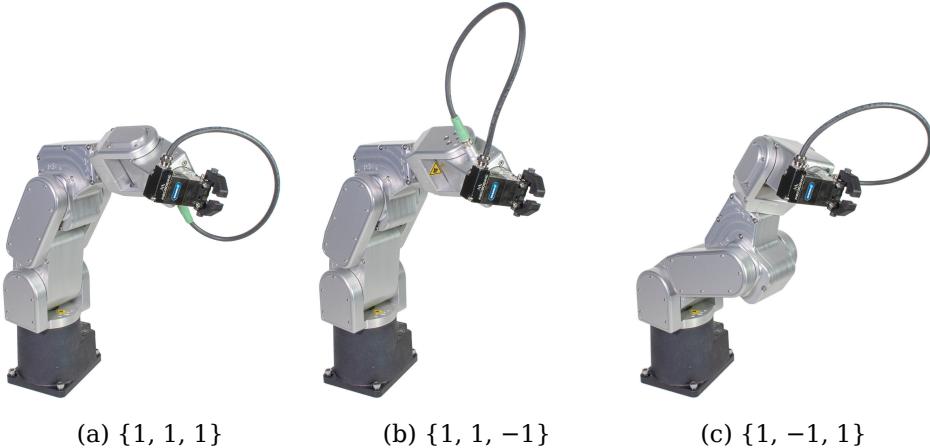
A joint set completely defines the relative poses of each pair of adjacent links, i.e., the [robot posture](#) (page 366). However, the joint sets $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$ and $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6 + c_t 360^\circ\}$, where $-180^\circ < \theta_6 \leq 180^\circ$ and c_t is the turn configuration for joint 6, define 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 365) is the problem of obtaining the robot joint sets that correspond to a desired end-effector pose.



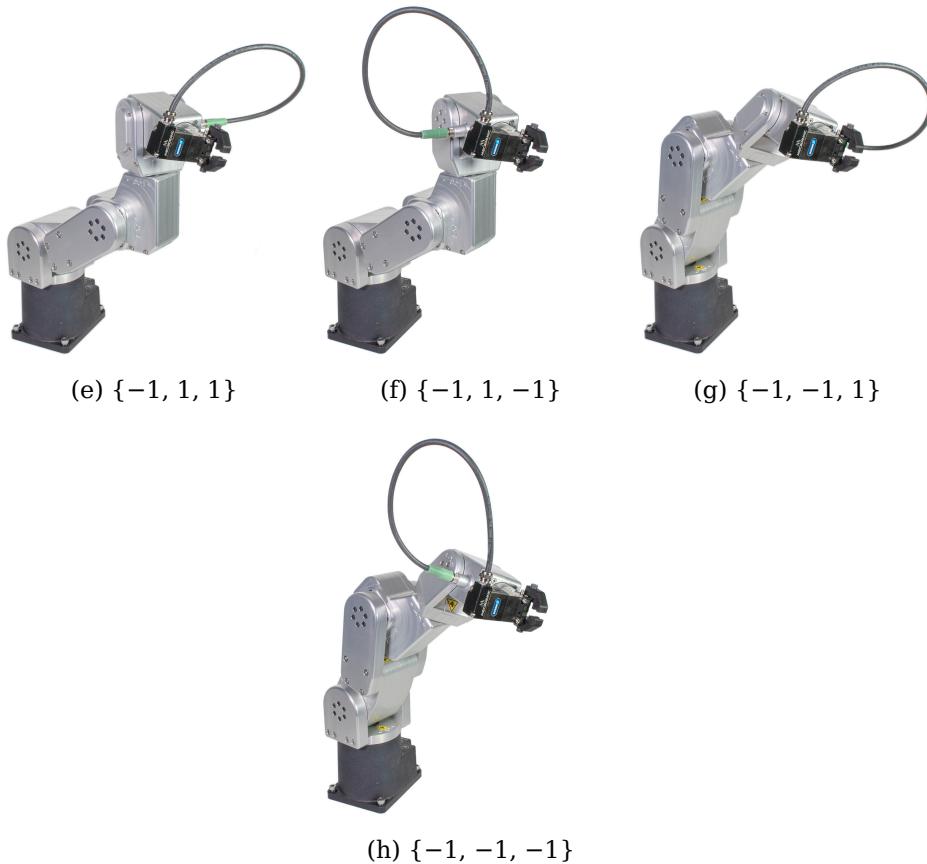


Figure 5: An example showing all eight possible robot postures of the Meca500 for the same end-effector pose

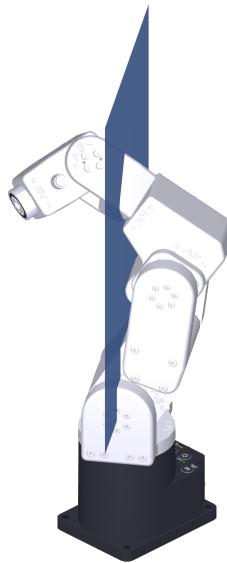
The inverse kinematics of our six-axis robots provide up to eight feasible robot postures for a desired pose of the TRF with respect to the WRF, as shown in Figure 5, and many more joint sets (since if θ_6 is a solution, then $\theta_6 \pm m360^\circ$, where m is an integer, is also a solution). Each of these solutions is associated with a different combination of three binary parameters called the *robot posture configuration parameters* (page 365): c_s , c_e and c_w . Each of these parameters corresponds to a specific geometric condition on the robot posture:

- c_s (shoulder configuration parameter)
 - $c_s = 1$, if the *wrist center* (page 366) (where the axes of joints 4, 5, and 6 intersect) is on the “front” side of the plane passing through the axes of joints 1 and 2 (see Figure 8a).
 - $c_s = -1$, if the wrist center is on the “back” side of this plane (see Figure 8c).
- c_e (elbow configuration parameter)
 - $c_e = 1$, if $\theta_3 > -\arctan(60/19) \approx -72.43^\circ$ (“elbow up” condition, see Figure 8d);
 - $c_e = -1$, if $\theta_3 < -\arctan(60/19) \approx -72.43^\circ$ (“elbow down” condition, see Figure 8f).

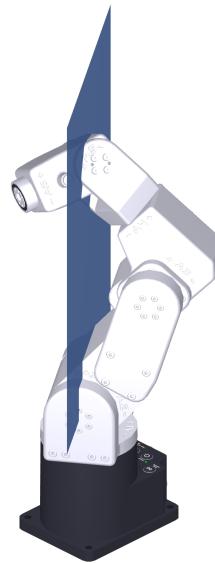
- c_w (wrist configuration parameter)
 - $c_w = 1$, if $\theta_5 > 0^\circ$ ("no flip" condition, see [Figure 8g](#));
 - $c_w = -1$, if $\theta_5 < 0^\circ$ ("flip" condition, see [Figure 8i](#)).

[Figure 5](#) shows an example with all eight possible robot postures, described by the posture configuration parameters $\{c_s, c_e, c_w\}$, for the pose $\{77 \text{ mm}, 210 \text{ mm}, 300 \text{ mm}, -103^\circ, 36^\circ, 175^\circ\}$ of the FRF with respect to the BRF.

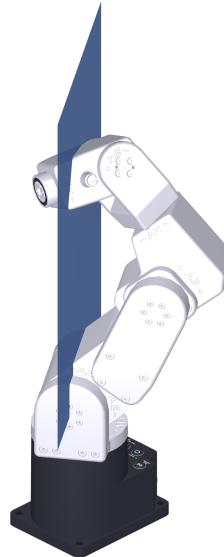
[Figure 8](#) shows an example of each robot posture configuration parameter, and limit conditions, which are called *singularities* (page 366). (We will discuss singularities in the next section.) Note that the popular terms front/back and elbow-up/elbow-down are misleading as they are not relative to the robot base but to specific planes that move when some of the robot joints rotate.



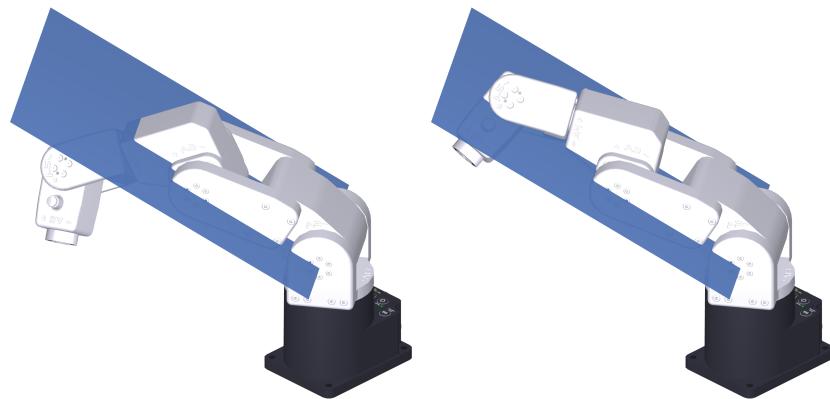
(a) $c_s = 1$, front



(b) shoulder singularity

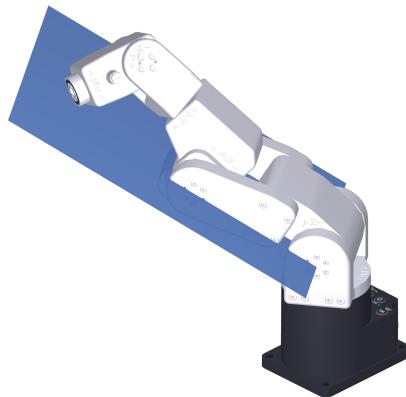


(c) $c_s = -1$, back



(d) $c_e = 1$, elbow up

(e) elbow singularity



(f) $c_e = -1$, elbow down

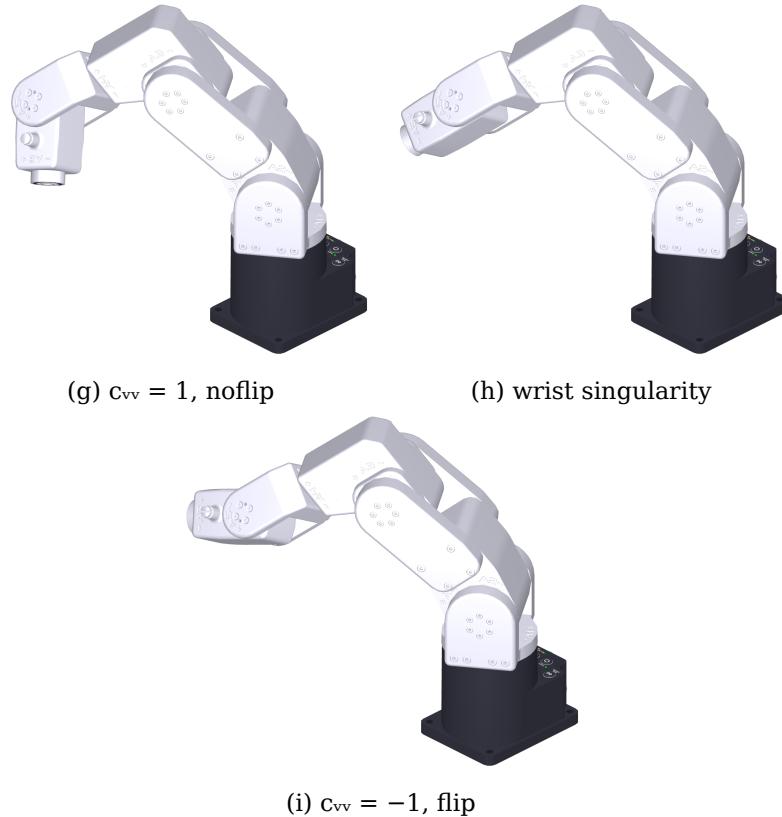


Figure 8: Posture configuration parameters and the three singularity types for the Meca500

The robot calculates the solution to the inverse kinematics that corresponds to the desired posture configuration, $\{c_s, c_e, c_w\}$, defined by the [SetConf](#) (page 160) command. In addition, it solves θ_6 by choosing the angle that corresponds to the desired turn configuration, c_t , defined by the [SetConfTurn](#) (page 162) command. The turn is therefore the last inverse kinematics configuration parameter.

Both the turn configuration and the set of robot posture configuration parameters 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 160) and [SetConfTurn](#) (page 162), [SetAutoConf](#) (page 152) and [SetAutoConfTurn](#) (page 153), 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 152) and [SetAutoConfTurn](#) (page 153), we strongly recommend always specifying the desired values for the configuration parameters with [SetConf](#) (page 160) and [SetConfTurn](#) (page 162). This should be done for every Cartesian motion command (e.g., [MovePose](#) (page 150) and the various [MoveLin*](#) commands) when programming your robot in [online mode programming](#) (page 365).

If you are teaching the [robot position](#) (page 366) 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 276)), but also the definitions of both the TRF and the WRF (using [GetTrf](#) (page 268) and [GetWrf](#) (page 270)). Additionally, you need to capture the corresponding configuration parameters (using [GetRtConf](#) (page 278) and [GetRtConfTurn](#) (page 279)). Then, to ensure accurate execution of the command [MoveLin](#) (page 144) 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_6 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 280). You can later move the robot to that joint set with the [MoveJoints](#) (page 138) 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.

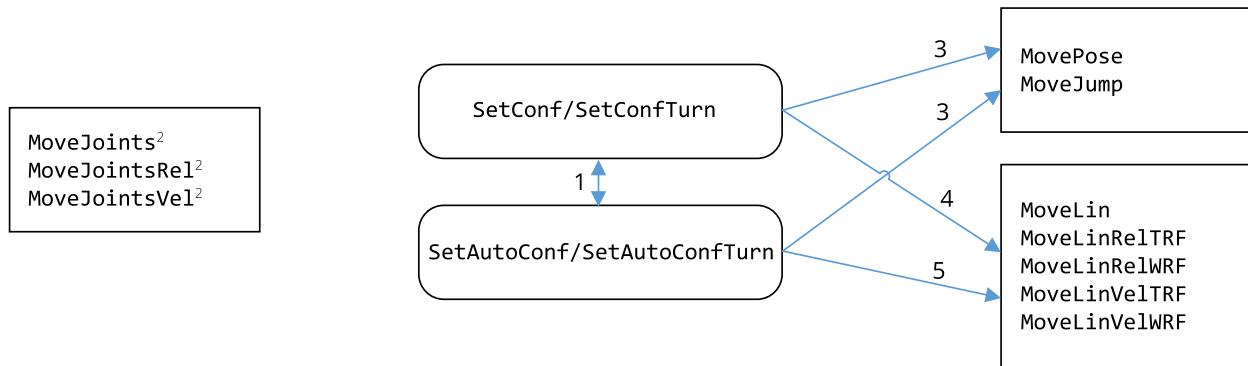


Figure 9: Effect of configuration parameters on robot movement commands

Figure 9 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 160) or [SetConfTurn](#) (page 162), 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 152) or [SetAutoConfTurn\(1\)](#) (page 153), respectively, removes the desired posture or turn configuration, respectively. At any moment, if [SetAutoConf\(0\)](#) (page 152) or [SetAutoConfTurn\(0\)](#) (page 153) 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 138), *MoveJointsRel* (page 140), and *MoveJointsVel* (page 141) 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 152) or *SetAutoConfTurn(0)* (page 153).
3. The command *MovePose* (page 150) 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 150) 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 144) 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.

Finally, note that there is currently no way of specifying only one of the posture configuration parameters and leaving the choice of the others to the robot controller. However, there is an indirect way to specify the elbow and wrist configurations, though this can't be done "on the fly". Indeed, if you prefer to always stick to one of the two possible wrist configurations in the Meca500, you can simply limit the range of joint 5, to either positive or non-negative values, using the command *SetJointLimits* (page 206). Similarly, you can fix the elbow configuration parameter by setting the range of joint 3 to be always smaller or larger than $-\arctan(60/19) \approx -72.43^\circ$.

Workspace and singularities

Users often oversimplify the workspace of a six-axis robot arm as a sphere with a radius equal to the robot's *reach* (page 365) (the maximum distance between the axis of joint 1 and the center of the robot's wrist). However, the actual Cartesian *workspace* (page 366) of a six-axis industrial robot is a six-dimensional entity, encompassing all attainable end-effector poses (refer to our [tutorial on workspace](#), available on our main website). This means the workspace depends on the choice of the Tool Reference Frame (TRF).

Complicating matters further, as discussed in the preceding section, a given end-effector pose can generally correspond to eight distinct robot postures (see [Figure 5](#)). Thus, the Cartesian workspace of a six-axis robot is the union of eight workspace subsets, each corresponding to one of these postures. While these subsets share overlapping regions, there are also areas

exclusive to a single subset — poses that are accessible in only one posture due to joint limits. To fully exploit the robot's workspace, it is often necessary to transition between these subsets. These transitions involve singularities, which can pose challenges when the robot's end-effector is required to follow a specific Cartesian path.

Every six-axis industrial robot arm encounters singularities (refer to our [tutorial on singularities](#)). However, a notable advantage of six-axis robots like ours is that the axes of the last three joints intersect at a single point — the center of the robot's wrist. This design makes these singularities straightforward to describe geometrically (see [Figure 8](#)). As a result, determining whether a robot posture is near a singularity is significantly easier for our robots compared to other designs.

In a singular robot posture, some of the joint set solutions corresponding to the pose of the TRF may coincide, or there may be infinitely many joint sets. The problem with singularities is that at a singular robot posture, the robot's end-effector cannot move in certain directions. This is a physical blockage, not a controller problem. Thus, singularities are one type of workspace boundary (the other type occurs when a joint is at its limit, or when two links interfere mechanically).

Take the Meca500, for example, at its zero robot posture ([Figure 1](#)). At this robot posture, the end-effector cannot be moved laterally (i.e., parallel to the y axis of the BRF); it is physically blocked. In order to move in that direction, it would need to rotate joints 4 and 6 a quarter of turn in opposite directions first. Thus, singularities are not some kind of purely mathematical problem. They represent actual physical limits.

There are three types of singular robot positions, and these correspond to the conditions under which the configuration parameters c_s , c_e , and c_w are not defined. The most common singular robot posture is called a wrist singularity and occurs when $\theta_5 = 0^\circ$ ([Figure 8h](#)). In this singularity, joints 4 and 6 can rotate in opposite directions at equal velocities while the end-effector remains stationary. You will run into this singularity frequently. The second type of singularity is called an elbow singularity ([Figure 8e](#)). It occurs when the arm is fully stretched (i.e., when the wrist center is in one plane with the axes of joints 2 and 3). In the Meca500, this singularity occurs when $\theta_3 = -\arctan(60/19) \approx -72.43^\circ$. You will run into this singularity when you try to reach poses that are too far from the robot base. The third type of singularity is called a shoulder singularity ([Figure 8b](#)). It occurs when the center of the robot's wrist lies on the axis of joint 1. You will run into this singularity when you work too close to the axis of joint 1.

Crossing singularities with linear Cartesian-space movements

Although *singularities can be a nuisance when controlling the robot in Cartesian space and should usually be avoided in production mode*, we have made it possible to cross them to facilitate programming our robots. With the release of firmware 9.1, the Meca500 can start at or pass through wrist and shoulder singularities, while executing any `MoveLin*` command, or end at any singularity while executing a `MoveLin*` or `MovePose` (page 150) command. Furthermore, the passage respects the posture configuration selection settings ([Figure 9](#)). [Figure 10](#) illustrates how this feature makes it possible to follow longer linear paths. In that

figure, we start from an elbow singularity, pass through a wrist singularity, then through a shoulder singularity, and then end at another elbow singularity, all with a single MoveLin* command, and in "AutoConf".

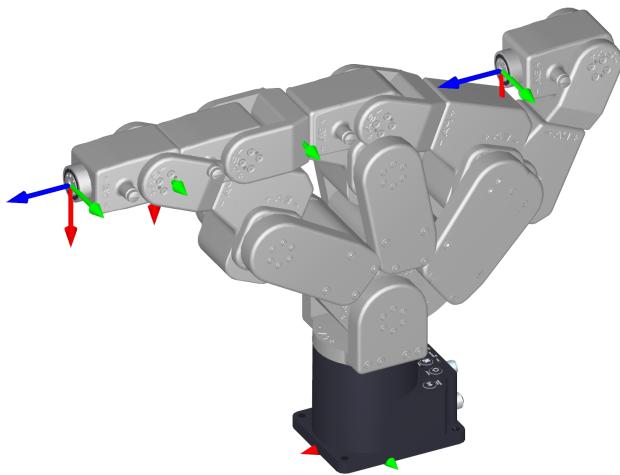


Figure 10: By crossing singularities, the Meca500 can execute longer linear movements

There are two possible situations when crossing a wrist singularity. Consider [Figure 12a](#), where "AutoConf" is enabled, the robot starts from robot position A, passes without any interruption through the singular configuration Z1 (where all joints are at zero degrees) and goes to robot position B, all with a single MoveLin* command. In the process, the robot changes the posture parameter c_w from 1 to -1. However, if you execute [*SetConf\(1,1,1\)*](#) (page 160), then request the robot to move with MoveLin* to the end-effector pose B, starting from robot position A, the robot will refuse the motion, since that would require joint 4 to rotate 180° or -180° when reaching robot position Z1. This is impossible as the range of joint 4 is ±170°.

The following video illustrates the examples described above:

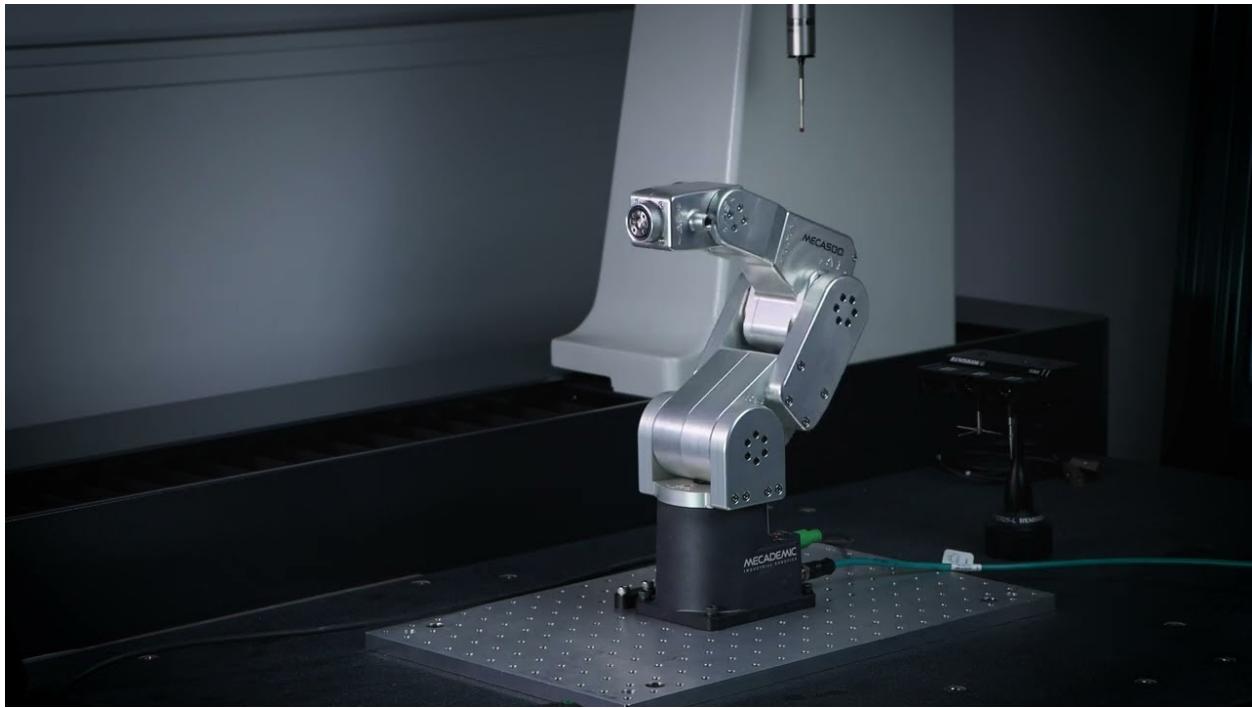
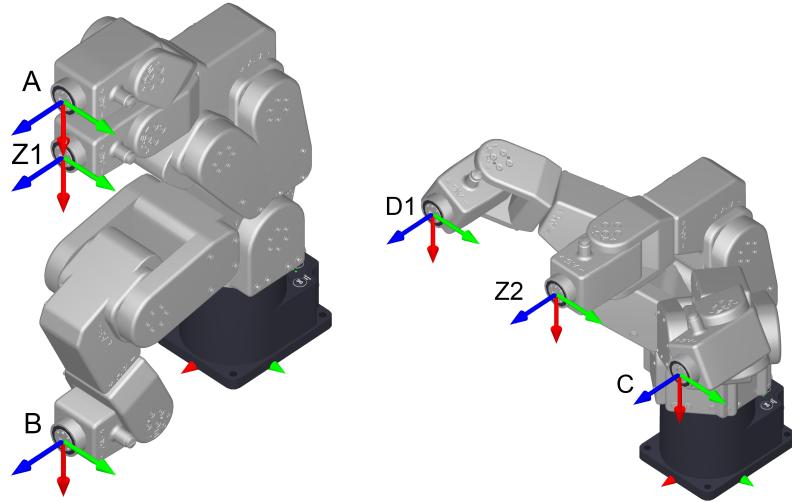
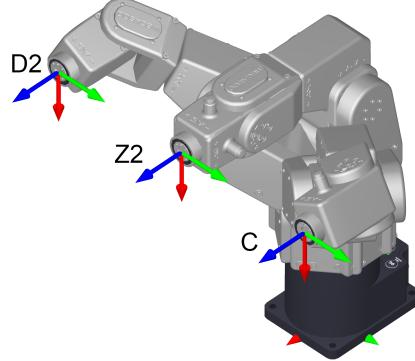


Figure 11: https://youtu.be/MA_tsx0i6DM?rel=0

Similarly, consider Figure 12b, where “AutoConf” is enabled, the robot starts from position C, passes without any interruption through the singular configuration Z2 (where $\theta_1 = \theta_2 = \theta_3 = \theta_5 = 0^\circ$, $\theta_4 = 90^\circ$, $\theta_6 = -90^\circ$) and goes to robot position D1, all with a single *MoveLin* (page 144) command. In the process, the robot changes posture c_w from -1 to 1 . However, as shown in Figure 12c, if you execute *SetConf(1,1,-1)* (page 160), then request the robot to move to the end-effector pose D1, starting from robot position C, the robot will execute the *MoveLin* (page 144) command, but when it reaches configuration Z2, joint 4 will rotate -180° and joint 6 will rotate 180° , at the same time while the end-effector will remain stationary. After that, the robot will continue its linear motion and reach the robot position D2 (which corresponds to the same pose as D1).



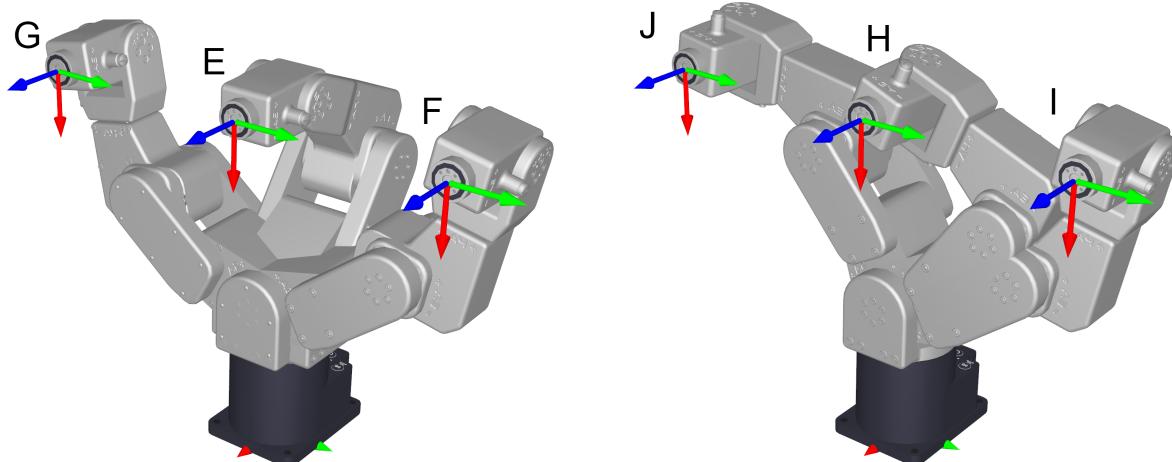
(a) A \leftrightarrow B via Z1, only possible with AutoConf(1)
 (b) C \leftrightarrow D1 via Z2, when using AutoConf(1)



(c) C \leftrightarrow D2 via Z2 and stationary re-orientation, with SetConf(1,1,-1)

Figure 12: Crossing a wrist singularity with *SetAutoConf(1)* (page 152) or with a desired posture configuration

In contrast, since shoulder singularities are much less frequent, yet much more complex to handle, the robot can currently cross them only in “AutoConf”. More precisely, when executing a linear move, the robot will never stop at a shoulder singularity to reorient its joints 1, 4 and 6 while keeping the end-effector stationary. Thus, the motion sequence shown in Figure 13a cannot be executed with a single MoveLin* command, whatever the state of posture configuration selection. However, in “AutoConf”, you can cross a shoulder singularity, as shown in Figure 13b. To experiment with shoulder singularities, simply execute *SetTRF(0,0,-70,0,0,0)* (page 182), to bring the TCP at the wrist center, then *SetWRF(0,0,0,0,0,0)* (page 184), and then bring the TCP to a position where its coordinates x and y are zero.



(a) Impossible sequence with a single MoveLin

(b) I→J via H, with AutoConf(1)

Figure 13: Crossing a shoulder singularity can only be done with *SetAutoConf(1)* (page 152) and implies a change of the posture parameter c_s

Passing exactly through singularities could be beneficial for some applications, but you must fully understand the concept. Otherwise, you might end-up with highly suboptimal robot motions. For example, consider the motion shown in Figure 13b. If you try to follow the same linear path, but one micrometer closer to the z axis of the WRF, joints 4 and 6, or joints 1, 4 and 6, will rotate very fast while the end-effector's speed will be significantly reduced, in a motion similar to what is shown in Figure 13a. Indeed, passing through or close to singularities often leads to longer cycle times, and should be avoided in production mode.

Key concepts for Mecademic robots

Homing

At power-up, the robot knows the approximate angle of each of its joints, with a couple of degrees of uncertainty. Each motor must make one full revolution to accurately find the exact joint angles. This motion is the essential part of a procedure called homing.

During homing, joints too close to a physical limit first rotate slightly away. Then, all joints rotate simultaneously in the positive direction: 3.6° for joints 1, 2, and 3, 7.2° for joint 4, 10.8° for joint 5, and 12° for joint 6. Finally, they return to their initial angles. The entire sequence lasts three seconds. Make sure there is nothing that restricts these joint movements, or the homing process will fail. Homing will also fail if any of the robot joints are outside their user-defined limits ([SetJointLimits](#) (page 206)), if the work zone has been breached ([SetWorkZoneLimits](#) (page 309), [SetWorkZoneCfg](#) (page 307)) or if a collision has been software detected ([SetCollisionCfg](#) (page 305)).

Finally, if your Meca500 is equipped with a MEGP 25* gripper, the robot controller will automatically detect it and home it in parallel with the robot joints, by fully opening, then fully closing the gripper. Make sure there is nothing that restricts the full range of motion of the gripper, except its fingers, while it is being homed.

Once the robot is homed, you do not need to home it again, even if you deactivate it, and then reactivate it, unless you use the optional argument 1, i.e., [ActivateRobot\(1\)](#) (page 187). In Meca500 R4, after an E-Stop has been reset, you do not need to run the homing procedure again, unless the robot is equipped with an MEGP 25* gripper (in that case, only the gripper is homed actually). If you call the homing process, but homing was not needed, the robot will simply ignore the command (though it will still respond with the [2002][Homing done.] message). If homing was needed only for the MEGP 25* gripper, the gripper fingers will move, but not the robot.

Note

The range of the absolute encoder of joint 6 is only $\pm 420^\circ$. *Therefore, you must always rotate joint 6 within that range before deactivating the robot.* Failure to do so may lead to an offset of $\pm 120^\circ$ in joint 6. If this happens, unpower the robot and disconnect your tooling. Then, power up and activate the robot, perform its homing, and zero joint 6. If the screw on the robot's flange is not as in [Figure 1](#), then rotate joint 6 to $+720^\circ$, and deactivate the robot. Next, reactivate it with the command [ActivateRobot\(1\)](#) (page 187), which reinitializes the drives, then home the robot, and zero joint 6 again. Repeat one more time if the problem is not solved.

Note

Before homing is completed, the robot's positioning accuracy is lower than after homing.

As a result:

- The reported real-time position will be less accurate (*GetRtJointPos* (page 280), *GetRtJointVel* (page 282), *GetRtCartPos* (page 276), *GetRtCartVel* (page 277), etc.);
- The robot may not reach the exact requested position when using *Recovery mode* (page 24) to move the robot before homing.

Recovery mode

If the robot is blocked by joint limits (*SetJointLimits* (page 206)), work-zone limits (*SetWorkZoneLimits* (page 309), *SetWorkZoneCfg* (page 307)), collisions (*SetCollisionCfg* (page 305)), or proximity to obstacles, it may be necessary to move it to a safe position (sometimes before performing homing). For these situations, use the *SetRecoveryMode* (page 214) command.

Blending

Industrial robots are most often controlled in *position mode* (page 365), using two groups of commands:

- With *Cartesian-space* (page 364) commands (*MoveLin* (page 144), *MoveLinRelWrf* (page 147), *MoveLinRelTrf* (page 146)), 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 365) commands (*MoveJoints* (page 138), *MoveJointsRel* (page 140), *MovePose* (page 150), *MoveJump* (page 143)), 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 150) 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 14). *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).

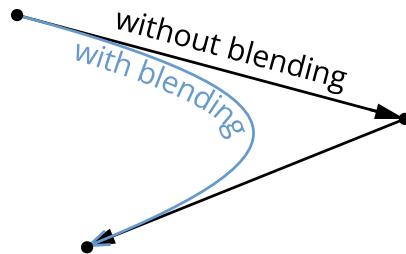


Figure 14: TCP path for two consecutive linear movements, with and without blending

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 154) command.

Furthermore, if blending is enabled, the gripper motion commands ([MoveGripper](#) (page 333), [GripperOpen](#) (page 331), [GripperClose](#) (page 329)) will not cause the robot to stop between two position-mode joint-space commands (blending will occur normally). However, the gripper motion commands will force the robot to stop when used between two position-mode Cartesian-space commands. Once the robot has come to a stop, the gripper's fingers will start moving at the same time as the subsequent Cartesian-space movement. In contrast, the [SetValveState](#) (page 347) command will not cause the robot to stop. Blending will occur normally, and the [SetValveState](#) (page 347) command will be executed at the beginning of the blending path.

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 365). If the robot needs to follow a linear path, the Cartesian-space motion commands *MoveLin* (page 144), *MoveLinRelTrf* (page 146), and *MoveLinRelWrf* (page 147) 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 150), *MoveJoints* (page 138), or *MoveJointsRel* (page 140) 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 168)) 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 15 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 166) or *SetCartLinVel* (page 157) and *SetCartAngVel* (page 156)) or the movement's time duration (*SetMoveDuration* (page 170)). This choice is made using the *SetMoveMode* (page 175) 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 171) 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 366). 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 141), *MoveLinVelTrf* (page 148) or *MoveLinVelWrf* (page 149). Note that the effect from a velocity-mode motion command lasts the time specified in the *SetVelTimeout* (page 183) 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 158), *GripperOpen* (page 331) and *GripperClose* (page 329) 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 168) command. The *SetJointVelLimit* (page 168) command affects the position-mode commands too. See [Figure 15](#) for a complete description of how velocity and acceleration settings affect the two modes.

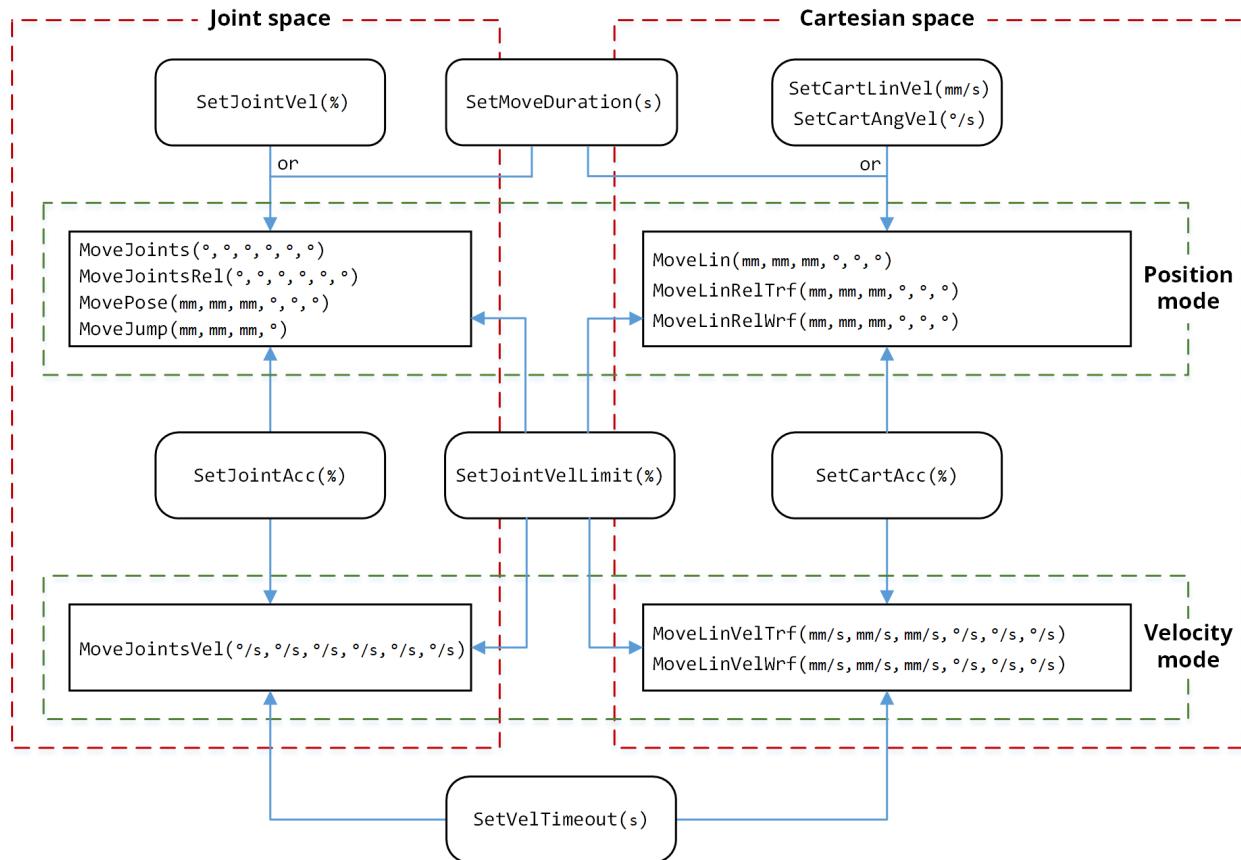


Figure 15: Settings that influence the robot motion in position and velocity modes. Note that *MoveJump* (page 143) is not supported and must not be used in time-based mode.

Note

The instantaneous command *SetTimeScaling* (page 219) affects all velocities, accelerations and even time durations (including the timeout set with *SetVelTimeout* (page 183) and the pause set with the *Delay* (page 137) command).

TCP/IP communication

Mecademic robots must be connected to a computer or 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 this section, or one of the three cyclic protocols, described in the following sections.

When the robot communicates using the TCP/IP protocol, it exchanges null-terminated ASCII strings (an end-line character may replace the null terminator in the TCP stream for convenience).

The default robot IP address is 192.168.0.100

Overview

Control connection

The robot accepts TCP connections on *port 10000*, referred to as the *control port* (page 364). Commands to control the robot and responses from the robot are exchanged over this port. Only one control connection is allowed at a time.

Monitoring connection

The robot also periodically sends data over *TCP port 10001*, referred to as the *monitoring port* (page 365). This connection reports the robot's initial state upon connection, then all subsequent status changes. It also transmits periodic real-time data (e.g., robot position) at the rate specified by the *SetMonitoringInterval* (page 208) command. By default, this includes joint and Cartesian positions. Additional optional data can be enabled with the *SetRealTimeMonitoring* (page 212) command.

Multiple monitoring connections are allowed for convenience.

Note

To avoid desynchronization between the data received from both ports, it is possible to send a copy of the monitoring port data to the control port using the *SetCtrlPortMonitoring* (page 203) command. This allows an application to receive all necessary data (robot responses, state changes, and real-time data) over a single connection on *port 10000*.

Commands syntax

Each command has a defined name and may include arguments inside parentheses. For example:

```
ActivateRobot()
Home()
SetJointVel(50)
MoveJoints(10,20,-10,0,0,45)
```

Note

A dash (-) can be added before the command name to ask the robot to handle the command silently (i.e., without logging it in the robot's detailed event log or the *MecaPortal* event log). Example: *-MoveLin(208,50,40,0,0,90)*

Command categories

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

- *motion commands* (page 365), which are the commands used to construct the robot trajectory (e.g., *Delay* (page 137), *MoveJoints* (page 138), *SetTRF* (page 182), *SetBlending* (page 154)),
- *robot control commands* (page 365), which are commands used to control the robot (e.g., *ActivateRobot* (page 187), *PauseMotion* (page 198), *SetNetworkOptions* (page 209)),
- *data request commands* (page 364), which are commands used to request some data regarding the robot (e.g., *GetTRF* (page 268), *GetBlending* (page 232), *GetJointVel* (page 246)),
- *real-time data request commands* (page 365), which are commands used to request some real-time data regarding the robot (e.g., *GetRtTrf* (page 290), *GetRtCartPos* (page 276), *GetStatusRobot* (page 294)),
- 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 electric grippers and pneumatic 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 365) are placed in a *motion queue* (page 365), 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 364) 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.

Default values

Some command descriptions refer to *default values* (page 364): 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 365): 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.

Performance recommendations

The Meca500 robot has limited CPU capacity and may not handle extremely high rates of commands or monitoring events. If the CPU becomes overloaded, TCP connections may be dropped unexpectedly. To maintain stable communication when sending more than 250 commands per second continuously, follow these recommendations:

- *Prefix frequently sent commands with a dash (-)* It makes the robot process the command silently, i.e., without recording it in the robot's detailed event log or the [MecaPortal](#) event log to reduce CPU overhead.
- *Avoid very short monitoring intervals* (see [SetMonitoringInterval](#) (page 208)) unless necessary. Intervals of 2 ms or less should be used with caution.
- *Minimize the number of simultaneous monitoring connections* unless necessary (e.g., multiple [MecaPortal](#) windows or monitoring-only applications), because each connection requires the robot to send an additional copy of all monitored data.
- Enable user command logging ([LogUserCommands](#) (page 197)) only when needed for debugging, as it adds CPU overhead.

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 <i>MoveJoints</i> (page 138) or <i>MoveJointsRel</i> (page 140) 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 <i>ResetError</i> (page 200) command is sent.
[1012][Linear move is blocked because it requires a reorientation of 180 degrees of the end- effector - Command: '...']	The <i>MoveLin</i> (page 144) or <i>MoveLinRel*</i> 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: '...']	The pose requested in the <i>MoveLin</i> (page 144), <i>MoveLinRel*</i> , <i>MovePose</i> (page 150) or <i>MoveJump</i> (page 143) command is out of reach, with the desired (or with any) configurations. In the case of the <i>MoveLin</i> (page 144) command, this error code is also produced if a pose along the path is out of reach.
[1016][The requested linear move is not possible due to a pose out of reach along the path. - Command: '...']	
[1022][Robot was not saving the program.]	The <i>StopSaving</i> (page 223) 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.]	The command that generated this error cannot be executed because no MEGP 25* gripper was detected
[1040][Command failed.]	General error for various commands.
[1041][No Vbox]	No pneumatic module connected.
[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]	Not available on the Meca500.
[1044][There is no vacuum module present on this robot.]	Not available on the Meca500.
[1550][...]	Variables could not be listed with <i>ListVariables</i> (page 361) for the reason specified in the error message.
[1551][...]	The variable could not be retrieved with <i>GetVariable</i> (page 360) for the reason specified in the error message.
[1552][...]	The variable creation failed with <i>CreateVariable</i> (page 358) for the reason specified in the error message.
[1553][...]	The variable deletion failed with <i>DeleteVariable</i> (page 360) for the reason specified in the error message.

continues on next page

Table 1 – continued from previous page

Message	Explanation
[1554][...]	The variable modification failed with SetVariable (page 362) for the reason specified in the error message.
[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 200) 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 220).
[3025][Gripper error.]	If the gripper was forcing when this message appeared, overheating likely occurred. Let the gripper cool down for a few minutes and send the ResetError (page 200) command. The gripper will stop applying a force; if it was holding a part, the part might fall.
[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.

continues on next page

Table 1 - continued from previous page

Message	Explanation
[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.
[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]	Not available on the Meca500.
[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.

continues on next page

Table 1 - continued from previous page

Message	Explanation
[3050][A power lost error was detected, robot is going to shutdown]	Not available on the Meca500.

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 158) command.

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

Response code	Command
[2000][Motors activated.]	<i>ActivateRobot</i> (page 187)
[2002][Homing done.]	<i>Home</i> (page 195)
[2004][Motors deactivated.]	<i>DeactivateRobot</i> (page 191)
	<i>ResetError</i> (page 200)
[2005][The error was reset.]	
[2006][There was no error to reset.]	
[2007][as, hs, sm, es, pm, eob, eom]	<i>GetStatusRobot</i> (page 294)
[2013][x, y, z, α , β , γ]	<i>GetWrf</i> (page 270)
[2014][x, y, z, α , β , γ]	<i>GetTrf</i> (page 268)
[2015][p]	<i>SetTimeScaling</i> (page 219), <i>GetTimeScaling</i> (page 265)
[2028][e]	<i>GetAutoConf</i> (page 230)
[2029][c _s , c _e , c _w]	<i>GetConf</i> (page 239)
[2031][e]	<i>GetAutoConfTurn</i> (page 231)
[2036][c _t]	<i>GetConfTurn</i> (page 240)
[2042][Motion paused.]	<i>PauseMotion</i> (page 198)
[2043][Motion resumed.]	<i>ResumeMotion</i> (page 201)
[2044][The motion was cleared.]	<i>ClearMotion</i> (page 189)
[2045][The simulation mode is enabled.]	<i>ActivateSim</i> (page 188)
[2046][The simulation mode is disabled.]	<i>DeactivateSim</i> (page 192)
[2047][External tool simulation mode has changed.]	<i>SetExtToolSim</i> (page 335)
	<i>SetRecoveryMode</i> (page 214)
[2049][Robot is in recovery mode]	
[2050][Robot is not in recovery mode]	

continues on next page

Table 2 – continued from previous page

Response code	Command
[2051][Joint velocity/acceleration ... will be limited to ... due to recovery mode]	<i>MoveJointsVel</i> (page 141), <i>MoveLinVelTrf</i> (page 148), <i>SetCartAcc</i> (page 155), <i>MoveLinVelWrf</i> (page 149), <i>SetCartAngVel</i> (page 156), <i>SetCartAcc</i> (page 155), <i>SetJointAcc</i> (page 164), <i>SetJointVel</i> (page 166) <i>SetEom</i> (page 205)
[2052][End of movement is enabled.]	
[2053][End of movement is disabled.]	
	<i>SetEob</i> (page 204)
[2054][End of block is enabled.]	
[2055][End of block is disabled.]	
[2056][b _{id} , e]	<i>GetIoSim</i> (page 316)
[2056][b _{id} , e]	<i>SetIoSim</i> (page 340)
[2060][Start saving program.]	<i>StartSaving</i> (page 221)
[2061][n commands saved.]	<i>StopSaving</i> (page 223)
[2063][Offline program n started.]	<i>StartProgram</i> (page 220) <i>StopSaving</i> (page 223)
[2064][Offline program looping is enabled.]	
[2065][Offline program looping is disabled.]	
[2080][n]	<i>GetCmdPendingCount</i> (page 273)
[2081][vx.x.x]	<i>GetFwVersion</i> (page 242)
[2085][Command successful. ...]	Response to various instantaneous commands
[2088][vx.x.x]	<i>GetExtToolFwVersion</i> (page 311)
[2083][robot's serial number]	<i>GetRobotSerial</i> (page 263)
[2084][Meca500]	<i>GetProductType</i> (page 258)
[2086][vx.x.x]	<i>GetExtToolFwVersion</i> (page 311)
[2090][n, θ _{n,min} , θ _{n,max}]	<i>GetJointLimits</i> (page 244)
[2092][n]	<i>SetJointLimits</i> (page 206) <i>SetJointLimitsCfg</i> (page 207)
[2093][User-defined joint limits enabled.]	
[2093][User-defined joint limits disabled.]	

continues on next page

Table 2 – continued from previous page

Response code	Command
[2094][e]	<i>GetJointLimitsCfg</i> (page 245)
[2095][s]	<i>GetRobotName</i> (page 262)
[2096][Monitoring on control port enabled/disabled]	<i>SetCtrlPortMonitoring</i> (page 203)
[2097][n]	<i>SyncCmdQueue</i> (page 225)
[2113][q ₁ , q ₂ , q ₃ ...]	<i>GetModelJointLimits</i> (page 248)
[2116][t]	<i>GetMonitoringInterval</i> (page 249)
[2117][n ₁ , n ₂ , ...]	<i>GetRealTimeMonitoring</i> (page 260), <i>SetRealTimeMonitoring</i> (page 212)
[2119][n ₁ , n ₂ , n ₃ , n ₄ , n ₅ , n ₆]	<i>GetNetworkOptions</i> (page 255)
[2140][t]	<i>GetRtc</i> (page 292)
[2149][n]	<i>GetCheckpointDiscarded</i> (page 238)
[2150][p]	<i>GetBlending</i> (page 232)
[2151][t]	<i>GetVelTimeout</i> (page 269)
[2152][p]	<i>GetJointVel</i> (page 246)
[2153][p]	<i>GetJointAcc</i> (page 243)
[2154][v]	<i>GetCartLinVel</i> (page 236)
[2155][ω]	<i>GetCartAngVel</i> (page 235)
[2156][n]	<i>GetCartAcc</i> (page 234)
[2157][n]	<i>GetCheckpoint</i> (page 237)
[2158][p]	<i>GetGripperForce</i> (page 313)
[2159][p]	<i>GetGripperVel</i> (page 315)
[2160][l, m]	<i>GetTorqueLimitsCfg</i> (page 267)
[2161][τ ₁ , τ ₂ , τ ₃ , τ ₄ , τ ₅ , τ ₆]	<i>GetTorqueLimits</i> (page 266)
[2162][d _{closed} , d _{open}]	<i>GetGripperForce</i> (page 313)
[2163][l, m]	<i>GetWorkZoneCfg</i> (page 302)
[2164][Workspace configuration set successfully.]	<i>SetWorkZoneCfg</i> (page 307)
[2165][x _{min} , y _{min} , z _{min} , x _{max} , y _{max} , z _{max}]	<i>GetWorkZoneLimits</i> (page 303)
[2166] [Workspace limits set successfully.]	<i>SetWorkZoneLimits</i> (page 309)
[2167][x, y, z, r]	<i>GetToolSphere</i> (page 301)
[2168][Tool sphere set successfully.]	<i>SetToolSphere</i> (page 306)
[2169][p]	<i>GetJointVelLimit</i> (page 247)
[2172][p _h , p _r]	<i>GetVacuumThreshold</i> (page 328)
[2173][t _p]	<i>GetVacuumPurgeDuration</i> (page 327)
[2174][h _{start} , h _{end} , h _{min} , h _{max}]	<i>GetMoveJumpHeight</i> (page 253)
[2175][v _{start} , p _{start} , v _{end} , p _{end}]	<i>GetMoveJumpApproachVel</i> (page 252)
[2176][m]	<i>GetOperationMode</i> (page 274)

continues on next page

Table 2 – continued from previous page

Response code	Command
[2177][l]	<i>ConnectionWatchdog</i> (page 190)
[2178][PStop2 configuration set successfully]	<i>SetPStop2Cfg</i> (page 211)
[2179][l]	<i>GetPStop2Cfg</i> (page 256)
[2181][l]	<i>GetCollisionCfg</i> (page 299)
[2182][v, g ₁ , o _{id,1} , g ₂ , o _{id,2}]	<i>GetCollisionStatus</i> (page 300)
[2183][v, g, o _{id}]	<i>GetWorkZoneStatus</i> (page 304)
[2189][m]	<i>GetMoveMode</i> (page 254)
[2190][s]	<i>GetMoveDurationCfg</i> (page 251)
[2191][t]	<i>GetMoveDuration</i> (page 250)
[2192][t]	<i>GetPayload</i> (page 257)
[2200][t, θ ₁ , θ ₂ , θ ₃ , θ ₄ , θ ₅ , θ ₆]	<i>GetRtTargetJointPos</i> (page 287)
[2201][t, x, y, z, α, β, γ]	<i>GetRtTargetCartPos</i> (page 283)
[2202][t, ω ₁ , ω ₂ , ω ₃ , ω ₄ , ω ₅ , ω ₆]	<i>GetRtTargetJointVel</i> (page 289)
[2203][t, τ ₁ , τ ₂ , τ ₃ , τ ₄ , τ ₅ , τ ₆]	<i>GetRtTargetJointTorq</i> (page 288)
[2204][t, Ẃ, ẏ, ẖ, ω _x , ω _y , ω _z]	<i>GetRtTargetCartVel</i> (page 284)
[2208][t, c _s , c _e , c _w]	<i>GetRtTargetConf</i> (page 285)
[2209][t, c _t]	<i>GetRtTargetConfTurn</i> (page 286)
[2210][t, θ ₁ , θ ₂ , θ ₃ , θ ₄ , θ ₅ , θ ₆]	<i>GetRtJointPos</i> (page 280)
[2211][t, x, y, z, α, β, γ]	<i>GetRtCartPos</i> (page 276)
[2212][t, ω ₁ , ω ₂ , ω ₃ , ω ₄ , ω ₅ , ω ₆]	<i>GetRtJointVel</i> (page 282)
[2213][t, τ ₁ , τ ₂ , τ ₃ , τ ₄ , τ ₅ , τ ₆]	<i>GetRtJointTorq</i> (page 281)
[2214][t, Ẃ, ẏ, ẖ, ω _x , ω _y , ω _z]	<i>GetRtCartVel</i> (page 277)
[2218][t, c _s , c _e , c _w]	<i>GetRtConf</i> (page 278)
[2219][t, c _t]	<i>GetRtConfTurn</i> (page 279)
[2220][t, n, a _x , a _y , a _z]	<i>GetRtAccelerometer</i> (page 275)
[2228][t, x, y, z, α, β, γ]	<i>GetRtWrf</i> (page 291)
[2229][t, x, y, z, α, β, γ]	<i>GetRtTrf</i> (page 290)
[2300][t, simType, phyType, hs, es, oh]	<i>GetRtExtToolStatus</i> (page 317)
[2310][t, v ₁ , v ₂]	<i>GetRtValveState</i> (page 326)
[2320][t, hp, dr, gc, go]	<i>GetRtGripperState</i> (page 320)
[2321][t, p]	<i>GetRtGripperForce</i> (page 318)
[2322][t, d]	<i>GetRtGripperPos</i> (page 319)
[2330][t, b _{id} , present, simMode, errorCode]	<i>GetRtIoStatus</i> (page 322)
[2340][t, b _{id} , p ₁ , p ₂ , p ₃ , p ₄ , p ₅ , p ₆ , p ₇ , p ₈]	<i>GetRtOutputState</i> (page 323)
[2341][t, b _{id} , p ₁ , p ₂ , p ₃ , p ₄ , p ₅ , p ₆ , p ₇ , p ₈]	<i>GetRtInputState</i> (page 321)
[2342][t, v, h, p]	<i>GetRtVacuumState</i> (page 325)
[2343][t, p]	<i>GetRtVacuumPressure</i> (page 324)
[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.

continues on next page

Table 2 – continued from previous page

Response code	Command
[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.
[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 <i>SetMoveDurationCfg</i> (page 171).
[3069][0/1/2]	Response to <i>GetSafetyStopStatus(3069)</i> (page 293).
[3070][0/1/2]	Not available on the Meca500.
[3080][0/1/2]	Not available on the Meca500.
[3081][0/1/2]	Not available on the Meca500.
[3082][0/1/2]	Response to <i>GetSafetyStopStatus(3082)</i> (page 293).
[3083][0/2]	Response to <i>GetSafetyStopStatus(3083)</i> (page 293).
[3084][0/1]	Response to <i>GetSafetyStopStatus(3084)</i> (page 293).
[3085][0/2]	Response to <i>GetSafetyStopStatus(3085)</i> (page 293).

continues on next page

Table 2 – continued from previous page

Response code	Command
[3086][0/1/2]	Response to GetSafetyStopStatus(3086) (page 293).
[3087][0/1/2]	Response to GetSafetyStopStatus(3087) (page 293).

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 208) 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 212) 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 294)
✓			[2015][p]	Same as response of GetTimeScaling (page 265)
			[2026][θ ₁ , θ ₂ , θ ₃ , θ ₄ , θ ₅ , θ ₆]	The same as the response of the legacy command GetJoints
			[2027][x, y, z, α, β, γ]	The same as the response of the legacy command GetPose
			[2044][The motion was cleared.]	Same as response of ClearMotion (page 189)
✓			[2049][Robot is in recovery mode]	Same as response of SetRecoveryMode (page 214)

continues on next page

Table 3 - continued from previous page

chg	opt	prd	Message	Description
✓			[2050][Robot is not in recovery mode]	Same as response of SetRecoveryMode (page 214)
			[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 262)
✓			[2163][l, m]	Same as response of GetWorkZoneCfg (page 302)
✓			[2165][x _{min} , y _{min} , z _{min} , x _{max} , y _{max} , z _{max}]	Same as response of GetWorkZoneLimits (page 303)
✓			[2167][x, y, z, r]	Same as response of GetToolSphere (page 301)
✓			[2181][l]	Same as response of GetCollisionCfg (page 299)
✓			[2182][v, g ₁ , o _{id,1} , g ₂ , o _{id,2}]	Same as response of GetCollisionStatus (page 300)
✓			[2183][v, g, o _{id}]	Same as response of GetWorkZoneStatus (page 304)
✓			[2189][m]	Same as response of GetMoveMode (page 254)
✓	✓		[2200][t, θ ₁ , θ ₂ , θ ₃ θ ₄ , θ ₅ , θ ₆]	Same as response of GetRtTargetJointPos (page 287)
✓	✓		[2201][t, x, y, z, α, β, γ]	Same as response of GetRtTargetCartPos (page 283)
✓	✓		[2202][t, ω ₁ , ω ₂ , ω ₃ , ω ₄ , ω ₅ , ω ₆]	Same as response of GetRtTargetJointVel (page 289)
✓	✓		[2203][t, τ ₁ , τ ₂ , τ ₃ , τ ₄ , τ ₅ , τ ₆]	Same as response of GetRtTargetJointTorq (page 288)
✓	✓		[2204][t, ẋ, ẏ, ẖ, ω _x , ω _y , ω _z]	Same as response of GetRtTargetCartVel (page 284)
			[2208][t, c _s , c _e , c _w]	Same as response of GetRtTargetConf (page 285)
✓			[2209][t, c _t]	Same as response of GetRtTargetConfTurn (page 286)

continues on next page

Table 3 - continued from previous page

chg	opt	prd	Message	Description
✓	✓		[2210][t, θ ₁ , θ ₂ , θ ₃ , θ ₄ , θ ₅ , θ ₆]	Same as response of GetRtJointPos (page 280)
✓	✓		[2211][t, x, y, z, α, β, γ]	Same as response of GetRtCartPos (page 276)
✓	✓		[2212][t, ω ₁ , ω ₂ , ω ₃ , ω ₄ , ω ₅ , ω ₆]	Same as response of GetRtJointVel (page 282)
✓	✓		[2213][t, τ ₁ , τ ₂ , τ ₃ , τ ₄ , τ ₅ , τ ₆]	Same as response of GetRtJointTorq (page 281)
✓	✓		[2214][t, Ẃ, ẏ, ᷗ, ω _x , ω _y , ω _z]	Same as response of GetRtCartVel (page 277)
✓	✓		[2218][t, c _s , c _e , c _w]	Same as response of GetRtConf (page 278)
✓	✓		[2219][t, c _t]	Same as response of GetRtConfTurn (page 279)
✓	✓		[2220][t, n, a _x , a _y , a _z]	Same as response of GetRtAccelerometer (page 275)
✓			[2226][t,hardDecel,linDistance]	When starting or finishing hard deceleration
✓			[2227][t,n]	When latest checkpoint reached changes
✓	✓		[2228][t, x, y, z, α, β, γ]	Same as response of GetRtWrf (page 291)
✓	✓		[2229][t, x, y, z, α, β, γ]	Same as response of GetRtTrf (page 290)
✓			[2230][t]	Same as response of GetRtTrf (page 290)
✓			[2300][t, simType, phyType, hs, es, oh]	Same as response of GetRtExtToolStatus (page 317)
✓			[2310][t, v ₁ , v ₂]	Same as response of GetRtValveState (page 326)
✓			[2320][t, hp, dr, gc, go]	Same as response of GetRtGripperState (page 320)
✓	✓		[2321][t,p]	Same as response of GetRtGripperForce (page 318)
✓	✓		[2322][t, p]	Same as response of GetRtGripperPos (page 319)
✓			[2330][t, b _{id} , present, simMode, errorCode]	Same as response of GetRtIoStatus (page 322)
✓			[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 323)
✓			[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 321)
✓			[2342][t, v, h, p]	Same as response of GetRtVacuumState (page 325)
✓	✓		[2343][t, p]	Same as response of GetRtVacuumPressure (page 324)

continues on next page

Table 3 – continued from previous page

chg	opt	prd	Message	Description
✓			[3000][Connected to ...] 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]	n/a
✓			[3070][0/1/2]	Response to a change in the state of the E-Stop safety stop signal
✓			[3080][0/1/2]	n/a
✓			[3081][0/1/2]	n/a
✓			[3082][0/1/2]	n/a
✓			[3083][0/2]	Occurs after robot is rebooted, and after Reset button is pressed for the first time
✓			[3084][0/1]	n/a
✓			[3085][0/2]	n/a
✓			[3086][0/1/2]	Response to a change in the connection drop safety signal change
✓			[3087][0/1/2]	n/a

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 364) 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 294).

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 200) 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 201) command is received by the robot.

P-Stop 2 and SWStop

As soon as the externally wired SWStop 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 211)). 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 201). The robot will then respond with the messages [2043][Motion resumed.] and [3032][0].

E-Stop and P-Stop 1

Currently, the Meca500 cannot differentiate between the E-STOP button on the power supply, an externally wired E-Stop (pins E-Stop on the D-SUB connector), or an externally wired protective stop (pins P-Stop 1), as explained in the robot's user manual.

In revision 3 of the Meca500, the E-Stop completely shuts down the robot. In revision 4, when the E-Stop is activated, the robot decelerates to a full stop, *power to the motors and the EOAT connected to the robot 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 (and the EOAT), 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 187) command. If the Meca500 R4 was already homed, re-homing is not required, except if an MEGP 25* gripper was connected to the robot's tool I/O port.

Enabling device

The Meca500 is not designed for use with an enabling device.

Operation mode switch

The Meca500 does not have an operation mode selector.

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 190) 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 201) command, to which the robot will respond with the message [3081][0].

Supply voltage fluctuation

This section does not apply to the Meca500.

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 and the P-Stop 1.

Standstill fault

This section does not apply to the Meca500.

Minor error

This section does not apply to the Meca500.

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 209) 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 309)) that are executed immediately by the robot;
- Queued commands (e.g., [*MoveJoints*](#) (page 138)) that are queued and executed one after another;
- Velocity-mode commands (e.g., [*MoveJointsVel*](#) (page 141)) 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 309)).

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 309)) execute immediately. The robot confirms completion by updating its *MoveID* field (Section 5) to match yours;

- Motion commands (e.g., [MoveJoints](#) (page 138)) 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 141), [MoveLinVelWrf](#) (page 149), and [MoveLinVelTrf](#) (page 148)), 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 138)), 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: *Meca500_vX.X.X.X.eds*
- PROFINET: *GSDML-V2.42-Mecademic-meca500-XXXXXXXX.xml*
- EtherCAT: *Meca500_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 57): Controls the general robot state (activation, simulation mode, recovery mode, etc.);
- *Motion control* (page 60): Controls robot motion (pausing, resuming, sending motion commands, etc.);
- *Motion-related commands* (page 63): ID and arguments of the command to execute;
- *Host time* (page 67): Synchronizes the robot's time with the host time;
- *Brake control* (page 69): Controls the robot's brakes when deactivated;
- *Dynamic data configuration* (page 71): 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
<i>DeactivateRobo</i>	Bool	0:0	1	1600h:1	<p>Deactivates the robot (see <i>DeactivateRobot</i> (page 191)) when set to 1.</p> <p>Deactivation is confirmed by the <i>Activated</i> status bit (Section 5).</p>
<i>ActivateRobot</i>	Bool	0:1	1	1600h:2	<p>Activates the robot (see <i>ActivateRobot</i> (page 187)) when set to 1, but only if the <i>Deactivate</i> bit is 0.</p> <p>Activation is confirmed by the <i>Activated</i> status bit (Section 5).</p>
<i>Home</i>	Bool	0:2	1	1600h:3	<p>Homes the robot (see <i>Home</i> (page 195)) when set to 1 (if the robot is activated but not yet homed).</p> <p>Homing is confirmed by the <i>Homed</i> status bit (Section 5).</p>
<i>ResetError</i>	Bool	0:3	1	1600h:4	<p>Resets the error (see <i>ResetError</i> (page 200)) when set to 1.</p> <p>The reset is confirmed when <i>ErrorCode</i> becomes 0 (Section 5).</p>

continues on next page

Table 4 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>ActivateSim</i>	Bool	0:4	1	1600h:5	<p>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 188).</p> <p>The simulation mode status is confirmed by the <i>SimActivated</i> 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 <i>SetSimModeCfg</i> (page 218).</p>
<i>EnableRecovery</i>	Bool	0:5	1	1600h:6	<p>Enables (set to 1) or disables (set to 0) recovery mode (see SetRecoveryMode (page 214)).</p> <p>The recovery mode state is confirmed by the <i>RecoveryMode</i> status bit (Section 5).</p>
<i>DisableEtherCA</i> (Reserved)	Bool	0:6	1	1600h:7	Disables the EtherCAT protocol when set to 1.
		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
<i>MoveID</i>	Integer	4	16	1601h:1	<p>A user-defined number. Changing it triggers the command specified in <i>MotionCommandID</i> to be added to the motion queue.</p> <p>Reception of the command is confirmed by the <i>MoveID</i> status bit (Section 5).</p> <p>See Section 5 for details.</p>
<i>SetPoint</i>	Bool	6:0	1	1601h:2	<p>Must be set to 1 for commands to be sent to the robot.</p> <p>See Section 5 for details.</p>
<i>PauseMotion</i>	Bool	6:1	1	1601h:3	<p>Pauses robot motion without clearing commands in the queue (<i>PauseMotion</i> (page 198)).</p> <p>Pause is confirmed by the <i>Paused</i> status bit (Section 5).</p>
<i>ClearMotion</i>	Bool	6:2	1	1601h:4	<p>Clears the motion queue and pauses the robot (<i>ClearMotion</i> (page 189)).</p> <p>Clear is confirmed by the <i>Cleared</i> status bit (Section 5).</p>

continues on next page

Table 5 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>ResumeMotion</i>	Bool	6:3	1	1601h:5	<p>A rising edge (value changed from 0 to 1) resumes robot motion (<i>ResumeMotion</i> (page 201)) if the following conditions are met:</p> <ul style="list-style-type: none"> - <i>PauseMotion</i> and <i>ClearMotion</i> are cleared; - No safety stop signals are active. <p>Resuming also clears resettable safety stops (such as P-Stop 2 or <i>enabling device released</i> safety stop signals).</p> <p>It also clears collision and work zone events.</p> <p>Motion resume is confirmed when the <i>Paused</i> status bit is cleared (Section 5).</p>
<i>UseVariables</i>	Bool	6:4	1	1601h:6	<p>When set, the robot interprets float arguments (Section 5) as the cyclic ID of variables to use as function arguments.</p> <p>See <i>Managing variables with cyclic protocols</i> (page 356) for details.</p>
(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
<i>MotionCommandI</i>	Integer	8	32	1602h:1	<p>The ID of the motion-related command to execute.</p> <p>See Table 8 for command IDs and Using motion-related commands (page 53) for more information.</p>

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>Argument 1</i>	Real	12	32	1602h:2	First argument of the motion-related command, if applicable.
<i>Argument 2</i>	Real	16	32	1602h:3	First argument of the motion-related command, if applicable.
<i>Argument 3</i>	Real	20	32	1602h:4	First argument of the motion-related command, if applicable.
<i>Argument 4</i>	Real	24	32	1602h:5	First argument of the motion-related command, if applicable.
<i>Argument 5</i>	Real	28	32	1602h:6	First argument of the motion-related command, if applicable.
<i>Argument 6</i>	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	<i>MoveJoints</i> (page 138) [†]
2	<i>MovePose</i> (page 150) [†]
3	<i>MoveLin</i> (page 144) [†]
4	<i>MoveLinRelTrf</i> (page 146) [†]
5	<i>MoveLinRelWrf</i> (page 147) [†]
6	<i>Delay</i> (page 137) [†]
7	<i>SetBlending</i> (page 154) [†]
8	<i>SetJointVel</i> (page 166) [†]
9	<i>SetJointAcc</i> (page 164) [†]
10	<i>SetCartAngVel</i> (page 156) [†]
11	<i>SetCartLinVel</i> (page 157) [†]
12	<i>SetCartAcc</i> (page 155) [†]
13	<i>SetTrf</i> (page 182) [†]
14	<i>SetWrf</i> (page 184) [†]
15	<i>SetConf</i> (page 160)
16	<i>SetAutoConf</i> (page 152) [†]
17	<i>SetCheckpoint</i> (page 158) [†]
18	Gripper action: argument 1 is 0 for <i>GripperClose</i> (page 329) and 1 for <i>GripperOpen</i> (page 331)
19	<i>SetGripperVel</i> (page 339) [†]
20	<i>SetGripperForce</i> (page 336) [†]
21	<i>MoveJointsVel</i> (page 141) [†]
22	<i>MoveLinVelWrf</i> (page 149) [†]
23	<i>MoveLinVelTrf</i> (page 148) [†]
24	<i>SetVelTimeout</i> (page 183) [†]
25	<i>SetConfTurn</i> (page 162) [†]
26	<i>SetAutoConfTurn</i> (page 153) [†]
27	<i>SetTorqueLimits</i> (page 178) [†]
28	<i>SetTorqueLimitsCfg</i> (page 180) [†]
29	<i>MoveJointsRel</i> (page 140) [†]
30	<i>SetValveState</i> (page 347) [†]
31	<i>SetGripperRange</i> (page 337) [†]
32	<i>MoveGripper</i> (page 333) [†]
33	<i>SetJointVelLimit</i> (page 168) [†]
34	<i>SetOutputState</i> (page 341) (Not available on this robot)
35	<i>SetOutputState_Immediate</i> (page 342) (Not available on this robot)
36	<i>SetIoSim</i> (page 340) (Not available on this robot)
37	<i>VacuumGrip</i> (page 348) (Not available on this robot)

continues on next page

Table 8 – continued from previous page

ID	Description
38	<i>VacuumGrip_Immediate</i> (page 349) (Not available on this robot)
39	<i>VacuumRelease</i> (page 350) (Not available on this robot)
40	<i>VacuumRelease_Immediate</i> (page 351) (Not available on this robot)
41	<i>SetVacuumThreshold</i> (page 345) (Not available on this robot)
42	<i>SetVacuumThreshold_Immediate</i> (page 346) (Not available on this robot)
43	<i>SetVacuumPurgeDuration</i> (page 343) (Not available on this robot)
44	<i>SetVacuumPurgeDuration_Immediate</i> (page 344) (Not available on this robot)
45	<i>MoveJump</i> (page 143) (Not available on this robot)
46	<i>SetMoveJumpHeight</i> (page 174) (Not available on this robot)
47	<i>SetMoveJumpApproachVel</i> (page 173) (Not available on this robot)
48	<i>SetTimeScaling</i> (page 219) [†]
49	<i>SetMoveMode</i> (page 175) [†]
50	<i>SetMoveDurationCfg</i> (page 171) [†]
51	<i>SetMoveDuration</i> (page 170) [†]
60	<i>SetPayload</i> (page 176) [†]
100	<i>StartProgram</i> (page 220) [†]
150	<i>SetJointLimits</i> (page 206) [†]
151	<i>SetJointLimitsCfg</i> (page 207) [†]
152	<i>SetWorkZoneCfg</i> (page 307) [†]
153	<i>SetWorkZoneLimits</i> (page 309) [†]
154	<i>SetCollisionCfg</i> (page 305) [†]
155	<i>SetToolSphere</i> (page 306) [†]
156	<i>SetCalibrationCfg</i> (page 202) [†]
200	<i>RebootRobot</i> (page 199)
1002	Clear robot homing, forcing drive reinitialization on the next activation, similar to <i>ActivateRobot(1)</i> (page 187)
10,000 to 19,999	Set a robot variable (see <i>Setting a variable</i> (page 356))

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

Host time

The *HostTime* section in the cyclic output synchronizes the robot's date/time with the host's (see *SetRtc* (page 217)).

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>HostTime</i>	Integer	36	32	1610h:1	<p>Current time in seconds since UNIX epoch (00:00:00 UTC, January 1, 1970).</p> <p>If non-zero, the robot updates its time to this value (same as <i>SetRtc</i> (page 217)).</p> <p>This ensures accurate timestamps in robot logs, as the robot resets its time on reboot.</p>

Brake control

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

The robot has brakes on joints 1, 2, and 3.

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](#).

Danger

Disable brakes with caution; without brakes, all links will collapse downward. The brakes control fields will be removed in the upcoming firmware release.

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>BrakesControlA</i>	Bool	40:0	1	1611h:1	<p>Must be set to 1 to allow brakes control through cyclic data.</p> <p>This bit ensures that the brakes are not inadvertently disengaged if cyclic data sent to the robot contains all zeros.</p>
<i>BrakesEngaged</i>	Bool	40:1	1	1611h:2	<p>If set to 1, the brakes are engaged.</p> <p>If 0, the brakes are disengaged, and the robot may fall under the effects of gravity.</p> <p>This bit is ignored if the <i>BrakesControlAllowed</i> bit is cleared or if the robot is activated.</p>
(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
<i>DynamicDataTyp</i>	Integer	44:0	32	1620h:1	Dynamic data type for index #1 (see Table 12).
<i>DynamicDataTyp</i>	Integer	48:0	32	1621h:1	Dynamic data type for index #2 (see Table 12).
<i>DynamicDataTyp</i>	Integer	52:0	32	1622h:1	Dynamic data type for index #3 (see Table 12).
<i>DynamicDataTyp</i>	Integer	56:0	32	1623h:1	Dynamic data type for index #4 (see Table 12).

Table 12: List of *DynamicTypeID* values

ID	Description
0	<p><i>Automatic.</i></p> <p>The robot will automatically choose a dynamic data type and change it every cycle, going through all of them in around-robin manner.</p> <p>This is the easiest way for the host to receive all possible values periodically.</p>
1	<p><i>Firmware version</i> (GetFwVersion (page 242)).</p> <p>Values: [major version, minor version, patch version, build number].</p>
2	<p><i>Product type</i> (GetProductType (page 258)).</p> <p>Values: [product type] where 3 = Meca500 R3, 4 = Meca500 R4, 20 = MCS500 R1.</p>
3	<p><i>Serial number</i> (GetRobotSerial (page 263)).</p> <p>Values: [serial number as a 32 bits float, serial number as a 32 bits unsigned integer].</p> <p>Please use the integer value. The float version may not properly represent large serial numbers</p> <p>and remains available only for backward compatibility.</p>
4-10	Reserved.
11	<p><i>Joint limits configuration</i> (GetJointLimitsCfg (page 245)).</p> <p>Values: [1/0].</p>
12	<p><i>Model joint limits</i> (GetModelJointLimits (page 248)), for joints 1, 2, and 3.</p> <p>Values: [$q_{1,\min}$, $q_{2,\min}$, $q_{3,\min}$, $q_{1,\max}$, $q_{2,\max}$, $q_{3,\max}$], in $^{\circ}$.</p>

continues on next page

Table 12 – continued from previous page

ID	Description
13	<p><i>Model joint limits</i> (GetModelJointLimits (page 248)), for joints 4, 5, and 6.</p> <p>Values: [$q_{4,\min}$, $q_{5,\min}$, $q_{6,\min}$, $q_{4,\max}$, $q_{5,\max}$, $q_{6,\max}$], in $^{\circ}$.</p>
14	<p><i>Effective joint limits</i> (GetJointLimits (page 244)), for joints 1, 2, and 3.</p> <p>Values: [$q_{1,\min}$, $q_{2,\min}$, $q_{3,\min}$, $q_{1,\max}$, $q_{2,\max}$, $q_{3,\max}$], in $^{\circ}$.</p>
15	<p><i>Effective joint limits</i> (GetJointLimits (page 244)), for joints 4, 5, and 6.</p> <p>Values: [$q_{4,\min}$, $q_{5,\min}$, $q_{6,\min}$, $q_{4,\max}$, $q_{5,\max}$, $q_{6,\max}$], in $^{\circ}$.</p>
17	<p><i>Work zone configuration</i> (GetWorkZoneCfg (page 302)).</p> <p>Values: [work zone limits severity, work zone limits detection mode].</p>
18	<p><i>Work zone limits</i> (GetWorkZoneLimits (page 303)).</p> <p>Values: [x_{\min}, y_{\min}, z_{\min}, x_{\max}, y_{\max}, z_{\max}], in mm.</p>
19	<p><i>Tool sphere</i> (GetToolSphere (page 301)).</p> <p>Values: [x, y, z, r], in mm.</p>
20	<p><i>Conf</i> and <i>Conf turn</i> (GetConf (page 239), GetConfTurn (page 240), GetAutoConf (page 230), GetAutoConfTurn (page 231)).</p> <p>Values: [shoulder $-1/1/\text{NaN}$, elbow $-1/1/\text{NaN}$, wrist $-1/1/\text{NaN}$, last joint turn or NaN].</p> <p>NaN indicates auto-conf or auto-conf-turn.</p>
21	<p><i>Motion queue parameters</i> (GetBlending (page 232), GetVelTimeout (page 269)).</p> <p>Values: [blending ratio percent, velocity timeout in seconds].</p>

continues on next page

Table 12 – continued from previous page

ID	Description
22	<p><i>Motion queue velocities and accelerations</i> (GetJointVel (page 246), GetJointAcc (page 243), GetCartLinVel (page 236), GetCartAngVel (page 235), GetCartAcc (page 234), GetJointVelLimit (page 247)).</p> <p>Values: [joint velocity, joint acceleration, Cartesian linear velocity, Cartesian angular velocity, Cartesian acceleration, joint velocity limit], in percent.</p>
23	<p><i>Gripper parameters</i> (GetGripperForce (page 313), GetGripperVel (page 315), GetGripperRange (page 314)).</p> <p>Values: [gripper force, gripper velocity, fingers opening corresponding to closed state, fingers opening corresponding to open state].</p> <p>Arguments 1 and 2 are in percentage, while arguments 3 and 4 are in mm.</p>
24	<p><i>Torque limits configuration</i> (GetTorqueLimitsCfg (page 267)).</p> <p>Values: [severity, detection mode].</p>
25	<p><i>Torque limits</i> (GetTorqueLimits (page 266)).</p> <p>Values: [motor 1 limit, motor 2 limit, ...], in percent.</p>
26	<p><i>Vacuum configuration</i> (GetVacuumThreshold (page 328), GetVacuumPurgeDuration (page 327)).</p> <p>Values: [holdThreshold, releaseThreshold, purgeDuration].</p> <p>Arguments 1 and 2 are in kPa, argument 3 is in seconds.</p>
27	<i>Move jump height</i> (Not available on this robot).
28	<i>Move jump approach velocity</i> (Not available on this robot).
29	<p><i>Move mode configuration</i> (GetMoveMode (page 254), GetMoveDurationCfg (page 251), GetMoveDuration (page 250)).</p> <p>Values: [move mode, severity, duration].</p>

continues on next page

Table 12 – continued from previous page

ID	Description
30	<p><i>Robot calibration status</i> (GetCalibrationCfg (page 233), GetRobotCalibrated (page 261)).</p> <p>Values: [calibrationEnabled, calibrated].</p>
31	<p><i>Robot payload</i> (GetPayload (page 257)).</p> <p>Values: [m, c_x, c_y, c_z], in kg or mm.</p>
32	<p><i>Target real-time joint velocity</i> (GetRtTargetJointVel (page 289)).</p> <p>Values: [ω₁, ω₂, ...], in °/s.</p>
33	<p><i>Target real-time joint torque</i> (GetRtTargetJointTorq (page 288)).</p> <p>Values: [motor 1 torque, motor 2 torque, ...], in percent.</p>
34	<p><i>Target real-time Cartesian velocity</i> (GetRtTargetCartVel (page 284)).</p> <p>Values: [dot x, dot y, dot z, ω_x, ω_y, ω_z], in mm/s or °/s.</p>
36	<p><i>Collision configuration</i> (GetCollisionCfg (page 299)).</p> <p>Values: [collision severity level].</p>
37	<p><i>Collision status</i> (GetCollisionStatus (page 300)).</p> <p>Values: [collision boolean state, group of colliding object 1, ID of colliding object 1, group of colliding object 2, ID of colliding object 2].</p>
38	<p><i>Work zone status</i> (GetWorkZoneStatus (page 304)).</p> <p>Values: [work zone breach Boolean state, group of object in breach, ID of object in breach].</p>

continues on next page

Table 12 – continued from previous page

ID	Description
40	<p><i>Actual joint position</i> (GetRtJointPos (page 280)).</p> <p>Values: $[q_1, q_2, q_3, \dots]$. Unit is $^{\circ}$.</p>
41	<p><i>Actual end-effector pose</i> (GetRtCartPos (page 276)).</p> <p>Values: $[x, y, z, \alpha, \beta, \gamma]$. Units are mm or $^{\circ}$.</p>
42	<p><i>Actual joint velocity</i> (GetRtJointVel (page 282)).</p> <p>Values: $[\omega_1, \omega_2, \dots]$, in $^{\circ}/s$.</p>
43	<p><i>Actual joint torque</i> (GetRtJointTorq (page 281)).</p> <p>Values: [joint 1 torque, joint 2 torque, ...], in percent.</p>
44	<p><i>Actual Cartesian velocity</i> (GetRtCartVel (page 277)).</p> <p>Values: $[\dot{x}, \dot{y}, \dot{z}, \omega_x, \omega_y, \omega_z]$, in mm/s or $^{\circ}/s$.</p>
45	<p><i>Actual conf and conf turn</i> (GetRtConf (page 278), GetRtConfTurn (page 279)).</p> <p>Values: [shoulder -1/0/1, elbow -1/0/1, wrist -1/0/1, last joint turn].</p>
46	<p><i>Accelerometer</i> (GetRtAccelerometer (page 275)).</p> <p>Values: $[a_x, a_y, a_z]$, in 1/16,000 of G.</p>
52	<p><i>External tool status</i> (GetRtExtToolStatus (page 317)).</p> <p>Values: [type, homing done, error state, overheated].</p>

continues on next page

Table 12 – continued from previous page

ID	Description
53	<p><i>EOAT status.</i></p> <p>In the case of a MEGP25* gripper: GetRtGripperState (page 320), GetRtGripperForce (page 318), and GetRtGripperPos (page 319).</p> <p>Values: [holding part, desired fingers opening reached, gripper closed, gripper open, gripper force, fingers opening].</p> <p>In the case of the MPM500 pneumatic module: GetRtValveState (page 326).</p> <p>Values: [valve 1 state, valve 2 state].</p>
54	<p><i>Time scaling</i> (GetTimeScaling (page 265)).</p> <p>Values: [p], in percent.</p>
61	<p><i>Configured joint limits</i> (GetJointLimits (page 244)), for joints 1, 2, and 3 (ignored if joint limits are disabled).</p> <p>Values: [q_{1,min}, q_{2,min}, q_{3,min}, q_{1,max}, q_{2,max}, q_{3,max}], in °.</p>
62	<p><i>Configured joint limits</i> (GetJointLimits (page 244)), for joints 4, 5, and 6 (ignored if joint limits are disabled).</p> <p>Values: [q_{4,min}, q_{5,min}, q_{6,min}, q_{4,max}, q_{5,max}, q_{6,max}], in °.</p>
72	Not available on this robot.
73	Not available on this robot.
10,000 to 19,999	A <i>DynamicTypeID</i> in the range 10,000 to 19,999 reports the values (up to six floats) of the robot variable whose cyclic ID matches this <i>DynamicTypeID</i> . See Commands for managing variables (beta) (page 352) for details on assigning a cyclic ID to a variable.

Note

Each *DynamicTypeID* 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.

Note

No dynamic data IDs are defined for information that is already available in other cyclic data fields, such as [*Target joint set*](#) (page 87), [*Target end-effector pose*](#) (page 89), [*Target configuration*](#) (page 91), [*Target WRF*](#) (page 94), etc. See [*Cyclic input format*](#) (page 80) 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 80): General robot state (e.g., activation, simulation mode, recovery mode, etc.);
- *Motion status* (page 82): Robot motion status (e.g., paused state, motion queue status, and other motion-related conditions);
- *Target joint set* (page 87): Real-time calculated joint positions (*GetRtTargetJointPos* (page 287));
- *Target end-effector pose* (page 89): Real-time calculated Cartesian position (*GetRtTargetCartPos* (page 283));
- *Target configuration* (page 91): Real-time calculated shoulder, elbow, wrist, and turn configuration (*GetRtTargetConf* (page 285) and *GetRtTargetConfTurn* (page 286));
- *Target WRF* (page 94): World reference frame used in the real-time calculated position (*GetRtWrf* (page 291), *GetWrf* (page 270));
- *Target TRF* (page 96): Tool reference frame used in the real-time calculated position (*GetRtTrf* (page 290), *GetTrf* (page 268));
- *Robot timestamp* (page 98): Precise monotonic robot timestamp associated with this cyclic data (*GetRtc* (page 292));
- *Safety status* (page 100): Safety-related information (e.g., safety signals, power supply states, operating mode);
- *Dynamic data* (page 103): 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 294)).

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>Busy</i>	Bool	0:0	1	1A00h:2	True only while the robot is being activated, homed, or deactivated.
<i>Activated</i>	Bool	0:1	1	1A00h:3	Indicates whether the motors are on (powered).
<i>Homed</i>	Bool	0:2	1	1A00h:4	Indicates whether the robot is homed and ready to receive motion commands.
<i>SimActivated</i>	Bool	0:3	1	1A00h:5	Indicates whether the robot simulation mode is activated.
<i>BrakesEngaged</i>	Bool	0:4	1	1A00h:6	Indicates whether the brakes are engaged.
<i>RecoveryMode</i>	Bool	0:5	1	1A00h:7	Indicates whether the robot recovery mode is activated.
<i>EStop</i> (deprecated)	Bool	0:6	1	1A00h:8	Indicates whether the emergency stop safety signal is activated. Deprecated; use <i>EStop</i> bit from <i>SafetyStatus</i> instead.
<i>CollisionStatus</i>	Bool	0:7	1	1A00h:9	Indicates whether the robot has detected an imminent collision (<i>SetCollisionCfg</i> (page 305)).
<i>WorkZoneStatus</i>	Bool	1:0	1	1A00h:10	Indicates whether the robot has detected a work zone breach (<i>SetWorkZoneCfg</i> (page 307), <i>SetWorkZoneLimits</i> (page 309)).
<i>MonitoringMode</i>	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.
<i>ErrorCode</i>	Integer	2	16	1A00h:1	Indicates the error code (see <i>Table 1</i> and <i>Table 3</i>) 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 294)).

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>CheckpointReac</i>	Integer	4	16	1A01h:1	<p>Indicates the last checkpoint number reached (GetCheckpoint (page 237)).</p> <p>The value remains the same until another checkpoint is reached.</p>
<i>CheckpointDisc</i>	Integer	6	16	1A01h:2	<p>Indicates the last checkpoint number discarded (GetCheckpointDiscarded (page 238)).</p> <p>The value remains unchanged until another checkpoint is discarded.</p>
<i>MoveID</i>	Integer	8	16	1A01h:3	<p>Acknowledges the <i>MoveID</i> of the last command the robot received (Motion control (page 60)).</p> <p>For details, refer to Using motion-related commands (page 53).</p>
<i>FifoSpace</i>	Integer	10	16	1A01h:4	<p>The number of commands that can be added to the robot's motion queue at any time (the maximum is 13,000).</p> <p>If 0 (too many commands sent), subsequent commands will be ignored.</p>

continues on next page

Table 14 - continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>Paused</i>	Bool	12:0	1	1A01h:6	<p>Indicates whether motion is paused.</p> <p>This bit stays set (and the robot remains paused) until motion is resumed with <i>Motion control</i> (page 60) bit <i>ResumeMotion</i>.</p>
<i>EOB</i>	Bool	12:1	1	1A01h:7	<p>The End of Block (<i>EOB</i>) bit is set when the robot is not moving and there are no motion commands left in the queue.</p> <p>Note that the <i>EOB</i> bit may occasionally be set before all commands are completed due to network or processing delays.</p> <p>Therefore, don't rely on this flag to determine when all movements have finished. Use a checkpoint instead (<i>SetCheckpoint</i> (page 158)).</p>
<i>EOM</i>	Bool	12:2	1	1A01h:8	<p>The End of Motion (<i>EOM</i>) bit is set when the robot is not moving.</p> <p>Note that the <i>EOM</i> bit may occasionally be set between two consecutive motion commands.</p> <p>Therefore, do not rely on this flag to determine when all movements have finished. Use a checkpoint instead (<i>SetCheckpoint</i> (page 158)).</p>

continues on next page

Table 14 - continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>Cleared</i>	Bool	12:3	1	1A01h:9	<p>Indicates whether the motion queue is cleared. If the queue is cleared, the robot is not moving.</p> <p>This bit remains set (and the robot remains paused) when the motion queue is cleared due to the <i>ClearMotion</i> control bit, robot deactivation, or a safety signal.</p> <p>It remains set until motion is resumed with <i>Motion control</i> (page 60) bit <i>ResumeMotion</i> (if the robot is still activated) or the robot is reactivated (if it was deactivated).</p>
<i>PStop2</i> (deprecated)	Bool	12:4	1	1A01h:10	<p>Indicates whether the SWStop safety signal is set.</p> <p>Deprecated; use the <i>PStop2</i> bit from <i>SafetyStatus</i> instead.</p>
<i>ExcessiveTorque</i>	Bool	12:5	1	1A01h:11	<p>Indicates whether a joint torque is exceeding the corresponding user-defined torque limit (<i>SetTorqueLimits</i> (page 178), <i>GetTorqueLimitsStatus</i> (page 295)).</p>
(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
<i>OfflineProgram</i>	Integer	14	16	1A01h:5	ID of the offline program currently running; 0 if none (<i>StartProgram</i> (page 220)).

Target joint set

The *TargetJointSet* section in the cyclic input reports the robot's real-time calculated joint position (similar to [GetRtTargetJointPos](#) (page 287)).

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

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

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 283)).

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>X coordinate</i>	Real	40	32	1A03h:1	X coordinate of the origin of the TRF with respect to the WRF, in mm.
<i>Y coordinate</i>	Real	44	32	1A03h:2	Y coordinate of the origin of the TRF with respect to the WRF, in mm.
<i>Z coordinate</i>	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 $^{\circ}$.
β angle	Real	56	32	1A03h:5	β Euler angle representing the orientation of the TRF with respect to the WRF, in $^{\circ}$.
γ angle	Real	60	32	1A03h:6	γ Euler angle representing the orientation of the TRF with respect to the WRF, in $^{\circ}$.

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 285), *GetRtTargetConfTurn* (page 286)). For more details, see [Section 3](#).

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>Shoulder</i>	Integer	64	8	1A08h:1	<p>Real-time shoulder posture configuration corresponding to the calculated joint position.</p> <p>The value is typically -1 or 1 but may also be 0 when the robot is near the shoulder singularity.</p> <p>See Section 3.</p>
<i>Elbow</i>	Integer	65	8	1A08h:2	<p>Real-time elbow posture configuration corresponding to the calculated joint position.</p> <p>The value is typically -1 or 1 but may also be 0 when the robot is near the elbow singularity.</p> <p>See Section 3.</p>
<i>Wrist</i>	Integer	66	8	1A08h:3	<p>Real-time wrist posture configuration corresponding to the calculated joint position.</p> <p>The value is typically -1 or 1 but may also be 0 when the robot is near the wrist singularity.</p> <p>See Section 3.</p>

continues on next page

Table 17 - continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>Turn</i>	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 89)), similar to the *GetRtWrf* (page 291) and *GetWrf* (page 270) command.

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>X coordinate</i>	Real	68	32	1A09h:1	X coordinate of the origin of the WRF with respect to the BRF, in mm.
<i>Y coordinate</i>	Real	72	32	1A09h:2	Y coordinate of the origin of the WRF with respect to the BRF, in mm.
<i>Z coordinate</i>	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 $^{\circ}$.
β angle	Real	84	32	1A09h:5	β Euler angle representing the orientation of the WRF with respect to the BRF, in $^{\circ}$.
γ angle	Real	88	32	1A09h:6	γ Euler angle representing the orientation of the WRF with respect to the BRF, in $^{\circ}$.

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 89)), similar to the *GetRtTrf* (page 290) and *GetTrf* (page 268) commands.

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>X coordinate</i>	Real	92	32	1A0Ah:1	X coordinate of the origin of the TRF with respect to the FRF, in mm.
<i>Y coordinate</i>	Real	96	32	1A0Ah:2	Y coordinate of the origin of the TRF with respect to the FRF, in mm.
<i>Z coordinate</i>	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 $^{\circ}$.
β angle	Real	108	32	1A0Ah:5	β Euler angle representing the orientation of the TRF with respect to the FRF, in $^{\circ}$.
γ angle	Real	112	32	1A0Ah:6	γ Euler angle representing the orientation of the TRF with respect to the FRF, in $^{\circ}$.

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 292)).

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

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

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 47).

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

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>EStop</i>	Bool	128:0	1	1A11h:1	E-Stop safety stop signal state [†]
<i>PStop1</i>	Bool	128:1	1	1A11h:2	Not supported on the Meca500
<i>PStop2</i>	Bool	128:2	1	1A11h:3	P-Stop 2 safety stop signal state [†]
(Reserved)		128:3	1		Reserved for future use.
<i>OperationModeC</i>	Bool	128:4	1	1A11h:5	Not supported on the Meca500
<i>EnablingDevice</i>	Bool	128:5	1	1A11h:6	Not supported on the Meca500
<i>VoltageFluctua</i>	Bool	128:6	1	1A11h:7	Not supported on the Meca500
<i>Reboot</i>	Bool	128:7	1	1A11h:8	Not supported on the Meca500
<i>RedundancyFaul</i>	Bool	129:0	1	1A11h:9	Not supported on the Meca500
<i>StandstillFaul</i>	Bool	129:1	1	1A11h:10	Not supported on the Meca500
<i>ConnectionDrop</i>	Bool	129:2	1	1A11h:11	TCP/IP connection dropped safety stop signal state [†]
<i>MinorError</i>	Bool	129:3	1	1A11h:12	Not supported on the Meca500
(Reserved)		129:4	20		Reserved for future use.
<i>EStopResettabl</i>	Bool	132:0	1	1A11h:33	(Only on Meca500 R4) E-Stop safety stop signal ready to be reset (Reset button)
<i>PStop1Resettabl</i>	Bool	132:1	1	1A11h:34	Not supported on the Meca500
<i>PStop2Resettabl</i>	Bool	132:2	1	1A11h:35	P-Stop 2 safety stop signal ready to be reset (with <i>ResumeMotion</i>)
(Reserved)		132:3	1		Reserved for future use.
<i>OperationModeC</i>	Bool	132:4	1	1A11h:37	Not supported on the Meca500
<i>EnablingDevice</i>	Bool	132:5	1	1A11h:38	Not supported on the Meca500
<i>VoltageFluctua</i>	Bool	132:6	1	1A11h:39	Not supported on the Meca500
<i>RebootResettabl</i>	Bool	132:7	1	1A11h:40	Not supported on the Meca500
<i>RedundancyFaul</i>	Bool	133:0	1	1A11h:41	Always 0. A redundancy fault requires rebooting the robot; it cannot be reset.
<i>StandstillFaul</i>	Bool	133:1	1	1A11h:42	Not supported on the Meca500
<i>ConnectionDrop</i>	Bool	133:2	1	1A11h:43	Connection dropped safety stop signal ready to be reset (with <i>ResumeMotion</i>)

continues on next page

Table 21 - continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>MinorErrorRese</i> (Reserved)	Bool	133:3	1	1A11h:44	Not supported on the Meca500
		133:4	20		Reserved for future use.
<i>OperationMode</i>	Integer	136	8	1A11h:65	Not supported on the Meca500
<i>ResetReady</i>	Bool	137:0	1	1A11h:66	(Only on Meca500 R4) 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
<i>VMotorOn</i> (Reserved)	Bool	137:1	1	1A11h:67	Robot motors powered or not
		137:2	6		Reserved for future use
<i>PsuInputs_Esto</i>	Bool	138:0	1	1A11h:74	Not supported on the Meca500
<i>PsuInputs_PSto</i>	Bool	138:1	1	1A11h:75	Not supported on the Meca500
<i>PsuInputs_PSto</i>	Bool	138:2	1	1A11h:76	Not supported on the Meca500
<i>PsuInputs_Rese</i>	Bool	138:3	1	1A11h:77	Not supported on the Meca500
<i>PsuInputs_Rese</i>	Bool	138:4	1	1A11h:78	Not supported on the Meca500
<i>PsuInputs_Enab</i> (Reserved)	Bool	138:5	1	1A11h:79	Not supported on the Meca500
		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
<i>DynamicType</i>	Integer	140	32	1A20h:1	Dynamic data type (see Table 12 for available values).
<i>ValueIdx_0</i>	Real	144	32	1A20h:2	Value index 0 (the meaning depends on the <i>DynamicType</i> , see Table 12).
<i>ValueIdx_1</i>	Real	148	32	1A20h:3	Value index 1 (the meaning depends on the <i>DynamicType</i> , see Table 12).
<i>ValueIdx_2</i>	Real	152	32	1A20h:4	Value index 2 (the meaning depends on the <i>DynamicType</i> , see Table 12).
<i>ValueIdx_3</i>	Real	156	32	1A20h:5	Value index 3 (the meaning depends on the <i>DynamicType</i> , see Table 12).
<i>ValueIdx_4</i>	Real	160	32	1A20h:6	Value index 4 (the meaning depends on the <i>DynamicType</i> , see Table 12).

continues on next page

Table 22 - continued from previous page

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

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

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

continues on next page

Table 23 - continued from previous page

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

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

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

continues on next page

Table 24 - continued from previous page

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

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

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

continues on next page

Table 25 - continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
<i>ValueIdx_5</i>	Real	248	32	1A23h:7	Value index 5 (the meaning depends on the <i>DynamicType</i> , 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](#) (Meca500_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 [The configuration menu](#).

Alternatively, you can use the [SwitchToEtherCAT](#) (page 224) command, which can be entered in the [The code editor panel](#) or sent to the robot via the [TCP/IP API](#) (page 29).

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](#).

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

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 113) 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 Meca500 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](#) (Meca500_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 [The configuration menu](#).

Alternatively, you can use the [*EnableEtherNetIp\(\)*](#) (page 193) command, which can be entered in the [The code editor panel](#) or sent to the robot via the [*TCP/IP API*](#) (page 29).

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 224) 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 119) 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 Meca500 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 128).

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 be enabled using the *Network configuration* panel in the [The configuration menu](#).

Alternatively, you can use the [*EnableProfinet\(\)*](#) (page 194) command, which can be entered in the [The code editor panel](#) or sent to the robot via the [*TCP/IP API*](#) (page 29).

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 224) 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-meca500-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 128) 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 366): A simplified log containing user-friendly traces of major events (e.g., robot activation, movement, E-Stop activation).
- *Robot log* (page 366): A more detailed version of the user log, intended primarily for the support team.
- *Detailed event log* (page 364): 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 196) and *LogUserCommands* (page 197).

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 137)). 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 158) (a response is then sent when the checkpoint has been reached).

The robot sends an end-of-movement message ([EOM](#) (page 364), code 3004) whenever it has stopped moving for at least 1 ms, if this option is activated with [SetEom](#) (page 205). 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 364), 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 138), [SetTrf](#) (page 182), [MovePose](#) (page 150) and [Delay](#) (page 137) 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 158) 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 138)
- [MoveJointsRel](#) (page 140)
- [MovePose](#) (page 150)
- [SetJointAcc](#) (page 164)
- [SetJointVel](#) (page 166)

Cartesian-space, position-mode movement commands

- *MoveLin* (page 144)
- *MoveLinRelTrf* (page 146)
- *MoveLinRelWrf* (page 147)
- *SetCartAcc* (page 155)
- *SetCartAngVel* (page 156)
- *SetCartLinVel* (page 157)

Velocity-mode movement commands

- *MoveJointsVel* (page 141)
- *MoveLinVelTrf* (page 148)
- *MoveLinVelWrf* (page 149)
- *SetVelTimeout* (page 183)

Robot posture and turn configuration commands

- *SetAutoConf* (page 152)
- *SetAutoConfTurn* (page 153)
- *SetConf* (page 160)
- *SetConfTurn* (page 162)

Other motion commands

- *Delay* (page 137)
- *SetBlending* (page 154)
- *SetCheckpoint* (page 158)
- *SetJointVelLimit* (page 168)
- *SetMoveDuration* (page 170)
- *SetMoveDurationCfg* (page 171)
- *SetMoveMode* (page 175)
- *SetPayload* (page 176)
- *SetTorqueLimits* (page 178)
- *SetTorqueLimitsCfg* (page 180)
- *SetTrf* (page 182)
- *SetWrf* (page 184)

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 137) command and stops temporarily. (In contrast, the [PauseMotion](#) (page 198) 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [Delay](#) (page 137) 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 166) and [SetJointVelLimit](#) (page 168). 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 166) and [SetJointVelLimit](#) (page 168). 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 16). Finally, with [MoveJoints](#) (page 138), the robot can cross singularities without any problem.

Syntax

`MoveJoints($\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$)`

Arguments

- the target position of each joint, in degrees.

The default ranges for the robot joints are given in [Technical specifications](#) of the robot's user manual. Note that these ranges can be further limited with the command [SetJointLimits](#) (page 206). The target joints position must be within the allowable joint limits or else the command will not be executed.

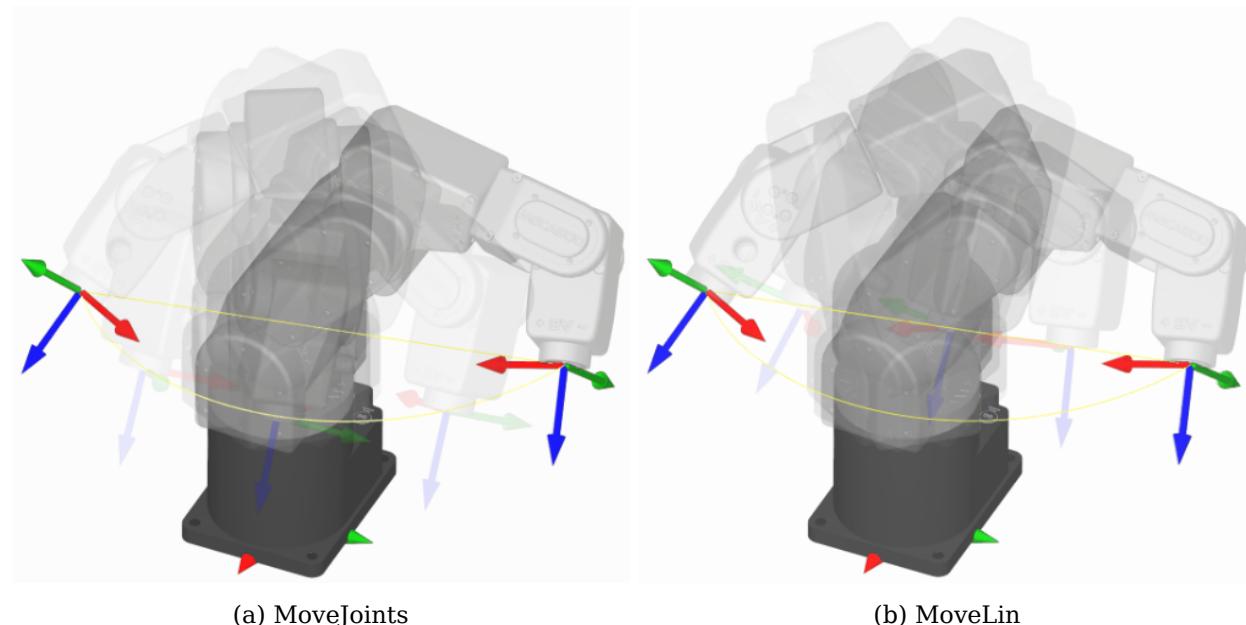


Figure 16: An example showing the difference between a path that is linear in joint space (often referred to as a point-to-point motion) and one that is linear in Cartesian space (the TCP follows a line)

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveJoints* (page 138) command is represented by *MotionCommandID* 1. See [Section 5](#) for more details.

MoveJointsRel

This command has the exact behavior as the [MoveJoints](#) (page 138) 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 287).

Syntax

MoveJointsRel($\Delta\theta_1, \Delta\theta_2, \Delta\theta_3, \Delta\theta_4, \Delta\theta_5, \Delta\theta_6$)

Arguments

- the desired relative displacement of each joint, in degrees.

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [MoveJointsRel](#) (page 140) 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 183), unless a subsequent [MoveJointsVel](#) (page 141) command is issued. Unlike position-mode motion commands, the [MoveJointsVel](#) (page 141) 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 141) command.

Syntax

`MoveJointsVel($\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6$)`

Arguments

- the desired velocity of each joint, in °/s.

The value of each of the arguments can be positive, negative or zero.

The maximum joint velocities are given in [Technical specifications](#) of the robot's user manual.

Note

The specified desired joint velocities are modified proportionally by the joint velocity override factor set by [SetJointVelLimit\(p\)](#) (page 168), when $p < 100$. In the Meca500 R4, p can be greater than 100, but there will be a distortion, since not all joints can rotate faster than their top rated velocities (e.g., joints 1 and 2 can rotate up to 150% faster, but joint 3 only 125%). Thus, if $p = 100$, you can still send the command [MoveJointsVel\(225,225,225,350,350,500\)](#) (page 141), but the robot joints will rotate only at the maximum velocities of the Meca500 R3. In contrast, if $p = 150$, and you send that same command, all joints will rotate at the requested rates (i.e., the maximum joint velocities for the Meca500 R4).

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveJointsVel* (page 141) command is represented by *MotionCommandID* 21. See Section 5 for more details.

MoveJump

This command is available only on our SCARA robots.

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 16b](#). 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, α , β , γ)

Arguments

- x, y, z: the target position for the TRF with respect to the WRF, in mm;
- α , β , γ : Euler angles representing the target orientation of the TRF with respect to the WRF, in degrees.

Further details

With this command, normally, the initial and final robot postures have to be in the same configuration, $\{c_s, c_e, c_w\}$. Only in some very peculiar cases, where the path passes exactly through a shoulder or wrist singularity, and when the automatic posture configuration selection is enabled with [SetAutoConf\(1\)](#) (page 152), a change in c_s or c_w , respectively, is possible (see [Section 3](#)).

If you specify a desired turn configuration, the [MoveLin](#) (page 144) 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 144) command if the required end-effector reorientation is exactly 180° , because there could be two possible paths.

Use the [MoveLin](#) (page 144) 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 138) or [MovePose](#) (page 150) 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 366), 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.]

- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveLin* (page 144) command is represented by *MotionCommandID* 3. See Section 5 for more details.

MoveLinRelTrf

This command has the same behavior as the [MoveLin](#) (page 144) command, but allows a desired pose to be specified relative to the current pose of the TRF. Thus, the arguments x, y, z, α , β , and γ 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 146) command).

As with the [MoveLin](#) (page 144) command, if the complete motion cannot be performed, it will not even start and an error will be generated.

Syntax

MoveLinRelTrf(x, y, z, α , β , γ)

Arguments

- x, y, z: the position coordinates, in mm;
- α , β , γ : Euler angles, 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [MoveLinRelTrf](#) (page 146) command is represented by *MotionCommandID* 4. See [Section 5](#) for more details.

MoveLinRelWrf

This command is similar to the [MoveLinRelTrf](#) (page 146) 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, α , β , γ)

Arguments

- x, y, z: the position coordinates, in mm;
- α , β , γ : Euler angles, 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [MoveLinRelWrf](#) (page 147) 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 168)). 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 183), unless a subsequent [MoveLinVelTrf](#) (page 148) or [MoveLinVelWrf](#) (page 149) command is issued. Additionally, the motion will stop if a [PauseMotion](#) (page 198) 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($\dot{x}, \dot{y}, \dot{z}, \omega_x, \omega_y, \omega_z$)

Arguments

- $\dot{x}, \dot{y}, \dot{z}$: the components of the linear velocity of the TCP expressed in the TRF, in mm/s;
- $\omega_x, \omega_y, \omega_z$: the components of the angular velocity of the TRF expressed in the TRF, in $^{\circ}/s$.

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [MoveLinVelTrf](#) (page 148) 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 168)). 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 183), unless a subsequent [MoveLinVelTrf](#) (page 148) or [MoveLinVelWrf](#) (page 149) command is issued. Additionally, the motion will stop if a [PauseMotion](#) (page 198) 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(\dot{x} , \dot{y} , \dot{z} , ω_x , ω_y , ω_z)

Arguments

- \dot{x} , \dot{y} , \dot{z} : the components of the linear velocity of the TCP with respect to the WRF, in mm/s;
- ω_x , ω_y , ω_z : the components of 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [MoveLinVelWrf](#) (page 149) 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 138) 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 166) and [SetJointVelLimit](#) (page 168) 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, α , β , γ)

Arguments

- x, y, z: the target position for the TRF with respect to the WRF, in mm;
- α , β , γ : Euler angles representing the target orientation of the TRF 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 138) 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [MovePose](#) (page 150) 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 150) 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.

In the case of MoveLin* commands, enabling the automatic posture configuration selection allows the change of configuration, but only if the path happens to pass exactly through a wrist or shoulder singularity.

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, the controller will automatically choose one of the boundary configurations. For example, if you execute [SetAutoConf\(0\)](#) (page 152) while the robot is at the joint set {0,0,0,0,0,0}, the new desired configuration will be {1,1,1}. Finally, the automatic robot configuration selection is also disabled as soon as the robot receives the command [SetConf](#) (page 160).

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [SetAutoConf](#) (page 152) 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 150) command and all MoveLin* commands. When the automatic turn selection is enabled, and a [MovePose](#) (page 150) 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 153) 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 162).

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetAutoConfTurn* (page 153) 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 138), [MoveJointsRel](#) (page 140), [MovePose](#) (page 150) and [MoveJump](#) (page 143), or between consecutive movements with the position-mode Cartesian-space commands [MoveLin](#) (page 144), [MoveLinRelTrf](#) (page 146) and [MoveLinRelTrf](#) (page 146). For example, there will never be blending between the trajectories of a [MovePose](#) (page 150) command followed by a [MoveLin](#) (page 144) 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 deacceleration periods, controlled by [SetJointAcc](#) (page 164), [SetCartAcc](#) (page 155) and [SetJointVelLimit](#) (page 168).

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [SetBlending](#) (page 154) 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 15](#)). 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 176) 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 180)).

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [*SetCartAcc*](#) (page 155) 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 144), [MoveLinRelTrf](#) (page 146) and [MoveLinRelWrf](#) (page 147) commands. It has impact on these movement commands only if the move mode is velocity-based (see [SetMoveMode](#) (page 175)).

Syntax

SetCartAngAcc(ω)

Arguments

- ω : TRF angular velocity limit, in °/s, ranging from 0.001 to 1,000.

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 168) command and the linear velocity limit set by the [SetCartLinVel](#) (page 157) 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [SetCartAngVel](#) (page 156) 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 144), [MoveLinRelTrf](#) (page 146) and [MoveLinRelWrf](#) (page 147) commands. It has impact on these movement commands only if the move mode is velocity-based (see [SetMoveMode](#) (page 175)).

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 168) command and the linear velocity limit set by the [SetCartAngVel](#) (page 156) 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [SetCartLinVel](#) (page 157) 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 158), then other motion commands, you will be able to know the exact moment when the motion command sent just before the *SetCheckpoint* (page 158) 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 366), 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.]
- [1006][The robot is not homed.]

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

Cyclic protocols

In cyclic protocols, the *SetCheckpoint* (page 158) 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 150) and MoveLin* commands (see [Section 3](#) and [Section 3](#)). When a desired posture configuration is set, a [MovePose](#) (page 150) 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 150) or MoveLin* command, by using the [SetAutoConf](#) (page 152) command. Using [SetConf](#) (page 160) automatically disables the automatic posture configuration selection.

Syntax

SetConf(c_s, c_e, c_w)

Arguments

- c_s : shoulder configuration parameter, either -1 or 1.
- c_e : elbow configuration parameter, either -1 or 1.
- c_w : wrist configuration parameter, either -1 or 1.

Default Values

Automatic posture configuration selection is enabled by default (see [SetAutoConf](#) (page 152)); when the robot starts, there is no default desired posture configuration. The desired posture configuration must be specified using the [SetConf](#) (page 160) command or the [SetAutoConf\(0\)](#) (page 152) 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetConf* (page 160) 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 150) and MoveLin* commands (see [Section 3](#) and [Section 3](#)). When c_t is set, a [MovePose](#) (page 150) 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 150) or MoveLin* command, by using the [SetAutoConf](#) (page 152) command. Using [SetConfTurn](#) (page 162) automatically disables the automatic turn configuration selection.

Syntax

SetConfTurn(c_t)

Arguments

- c_t : turn configuration, an integer between -100 and 100 .

The turn configuration parameter defines the desired range for joint 6, according to the following inequality: $-180^\circ + c_t 360^\circ < \theta_6 \leq 180^\circ + c_t 360^\circ$.

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 162) or to execute the command [SetAutoConfTurn\(0\)](#) (page 153). 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 range of the last joint appropriately using the [SetJointLimits](#) (page 206) 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 162), with the appropriate turn configuration as argument.

If using a cable-less end-effector, then the automatic turn configuration should never be disabled. However, remember to always bring joint 6 within the $\pm 420^\circ$ range before powering the robot off (recall [Section 3](#)).

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetConfTurn* (page 162) 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 15](#)). 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 150.

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 176) 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 180)).

The argument of this command is exceptionally limited to 150. This is because in firmware 8, a scaling was applied so that if this argument is kept at 100, most joint-space movements are feasible even at full payload. More precisely, if you are upgrading the firmware of your Meca500 from firmware 7 and you want to keep the same joint accelerations, you need to multiply the arguments of your [SetJointAcc](#) (page 164) commands by the factor 1.43.

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetJointAcc* (page 164) 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 150), [MoveJoints](#) (page 138), and [MoveJointsRel](#) (page 140) commands. It has impact on these movement commands only if the move mode is velocity-based (see [SetMoveMode](#) (page 175)).

Syntax

SetJointVel(p)

Arguments

- p: percentage of R3 top rated joint velocities, ranging from 0.001 to 100, for Meca500 R3, and to 150, for R4.

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 168) if p_o < p. Also, it is not possible to limit the velocity of only one joint. With [SetJointVel](#) (page 166) and [SetJointVelLimit](#) (page 168), the maximum velocities of all joints are reduced proportionally.

In the case of the Meca500 R4, for backward compatibility, p can be greater than 100, up to 150, and the maximum velocity of each joint can be increased

- up to 225°/s for joints 1 and 2 (i.e., up to 150%);
- up to 225°/s for joint 3 (i.e., up to 125%);
- up to 350°/s for joints 4 and 5 (i.e., up to 117%);
- up to 500°/s for joint 6 (i.e., the joint velocity cannot exceed its top rated velocity).

Thus, for example, if p = 140 (and p_o > p), the velocity of joints 1 and 2 will be limited to min(150*1.4, 225) = 210°/s, the velocity of joint 3 will be limited to min(180*1.4, 225) = 225°/s, etc.

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 366), 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.]

- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetJointVel* (page 166) command is represented by *MotionCommandID* 8. See [Section 5](#) for more details.

SetJointVelLimit

The *SetJointVelLimit* (page 168) overrides the default joint velocity limits. Unlike the *SetJointVel* (page 166) command, this command affects the movements generated by *all* Move* commands (even the MoveLinVel* ones).

Syntax

SetJointVelLimit(p_o)

Arguments

- p_o : percentage of R3 top rated joint velocities, ranging from 0.001 to 100, for Meca500 R3, and to 150, for R4.

Default values

By default, $p_o = 100$.

Further details

As of firmware 10.3, when the argument of *SetJointVelLimit* (page 168) is less than 100, the robot will optimize its joint accelerations in the case of slower movements.

As already mentioned in the description of the *SetJointVel* (page 166) command, in both revisions of Meca500, the top rated velocity of joints 1 and 2 is 150°/s, of joint 3 is 180°/s, of joints 4 and 5 is 300°/s, and of joint 6 is 500°/s. In the case of the R4, the maximum velocity of each joint can be increased up to 225°/s for joints 1, 2, and 3, and up to 350°/s for joints 4 and 5. The velocity of joint 6 cannot be increased over its top rated limit of 500°/s. Thus, for example, if $p_o = 140$, the velocity of joints 1 and 2 will be limited to $\min(150*1.4, 225) = 210$ °/s, the velocity of joint 3 will be limited to $\min(180*1.4, 225) = 225$ °/s, etc., during a *MoveLin* (page 144) motion.

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetJointVelLimit* (page 168) 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 175), the motion-queue [SetMoveDuration](#) (page 170) sets the desired duration for the movement resulting from every subsequent position-mode command. 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 171) 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [SetMoveDuration](#) (page 170) 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 170) 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 168)). 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 219)) and recovery mode ([Section 3](#))

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetMoveDurationCfg* (page 171) command is represented by *MotionCommandID* 50. See Section 5 for more details.

SetMoveJumpApproachVel

This command is available only on our SCARA robots.

Cyclic protocols

In cyclic protocols, the *SetMoveJumpApproachVel* (page 173) command is represented by *MotionCommandID* 47. See Section 5 for more details.

SetMoveJumpHeight

This command is only available on our SCARA robots.

Cyclic protocols

In cyclic protocols, the *SetMoveJumpHeight* (page 174) 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 144), [MoveJoints](#) (page 138), and [MovePose](#) (page 150)) 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 175).

The command [SetMoveDurationCfg](#) (page 171) specifies what happens when a move command cannot meet the desired duration set by the [SetMoveDuration](#) (page 170) command, in time-based move mode.

Syntax

SetMoveMode(m)

Arguments

- m: submode, where
 - 0 selects the velocity-based submode, meaning the commands [SetJointVel](#) (page 166), [SetCartLinVel](#) (page 157), and [SetCartAngVel](#) (page 156) affect all subsequent position-mode movement commands.
 - 1 selects the time-based submode, meaning the command [SetMoveDuration](#) (page 170) affects all subsequent position-mode movement commands.

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [SetMoveMode](#) (page 175) 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 workpiece 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 180)).

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetPayload* (page 176) 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 180) is 0 or 1, the absolute values of the actual motor torques (*GetRtJointTorq* (page 281)), also reported as percentages of the maximum allowable torques, are compared to the respective thresholds.
- When the second argument of *SetTorqueLimitsCfg* (page 180) is 2, which is the default setting as of firmware 11.1, the absolute value of the difference between the actual (*GetRtJointTorq* (page 281)) and calculated motor torque (*GetRtTargetJointTorq* (page 288)) of each joint is compared with the respective threshold. Thus, in this case, the arguments of *SetTorqueLimits* (page 178) 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 180).

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 281)) is estimated by measuring the current in the corresponding drive. The calculated torque (*GetRtTargetJointTorq* (page 288)) is obtained from the dynamic model of the robot.

Syntax

SetTorqueLimits($\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6$)

Arguments

- τ_i : torque threshold represented by a percentage of the maximum allowable torque that can be applied at motor i , where $i = 1, 2, \dots, 6$ ranging from 0.001 to 100.

Default values

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

Further details

Unlike the *SetJointLimits* (page 206) commands, the *SetTorqueLimits* (page 178) command can only be applied after the robot has been homed. 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 178) 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetTorqueLimits* (page 178) 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 178) 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 178),
 - 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 178).

With the option m = 0, you must use either very low accelerations ([SetJointAcc](#) (page 164)) or very high torque limits ([SetTorqueLimits](#) (page 178)).

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 178) 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 176) 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetTorqueLimitsCfg* (page 180) 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, α , β , γ)

Arguments

- x, y, z: the coordinates of the origin of the TRF with respect to the FRF, in mm;
- α , β , γ : Euler angles representing the orientation of the TRF 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetTrf* (page 182) 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 141), [MoveLinVelTrf](#) (page 148), or [MoveLinVelWrF](#) (page 149)). If no subsequent velocity-mode motion command is received within this period, all joint speeds will automatically be set to zero. The [SetVelTimeout](#) (page 183) command serves as a safety precaution and should be used accordingly. Note that the velocity-mode timeout is influenced by the [SetTimeScaling](#) (page 219) 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 164) or [SetCartAcc](#) (page 155) 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the [SetVelTimeout](#) (page 183) 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, α , β , γ)

Arguments

- x, y, z: the coordinates of the origin of the WRF with respect to the BRF, in mm;
- α , β , γ : Euler angles representing the orientation of the WRF 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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetWrf* (page 184) 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 189)
- *PauseMotion* (page 198)
- *ResumeMotion* (page 201)
- *ResetError* (page 200)
- *SetCalibrationCfg* (page 202)
- *SetEob* (page 204)
- *SetEom* (page 205)
- *SetPStop2Cfg* (page 211)
- *SetTimeScaling* (page 219)

Robot status related commands

- *ActivateRobot* (page 187)
- *DeactivateRobot* (page 191)
- *Home* (page 195)
- *RebootRobot* (page 199)
- *SetRecoveryMode* (page 214)

Simulation commands

- *ActivateSim* (page 188)
- *DeactivateSim* (page 192)

Utilities commands

- *SetRealTimeMonitoring* (page 212)
- *SetRobotName* (page 216)
- *SetRtc* (page 217)
- *SetMonitoringInterval* (page 208)
- *SyncCmdQueue* (page 225)
- *LogTrace* (page 196)
- *LogUserCommands* (page 197)
- *TcpDump* (page 226)
- *TcpDumpStop* (page 227)

Program execution commands

- *StartProgram* (page 220)
- *StartSaving* (page 221)
- *StopSaving* (page 223)
- *SetOfflineProgramLoop* (page 210)

Network commands

- *ConnectionWatchdog* (page 190)
- *EnableEtherNetIp* (page 193)
- *EnableProfinet* (page 194)
- *SwitchToEtherCat* (page 224)
- *SetNetworkOptions* (page 209)
- *SetCtrlPortMonitoring* (page 203)

Joint limits commands

- *SetJointLimits* (page 206)
- *SetJointLimitsCfg* (page 207)

ActivateRobot

This command activates all motors (as well as the EOAT connected to the tool I/O port) 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. Homing is then required again.

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 187) command is mapped to the *ActivateRobot* bit in the *RobotControl* data. See [Table 4](#) for more details.

The equivalent of *ActivateRobot(1)* (page 187) in cyclic protocols (to force drives reinitialization) is to send *MotionCommandID* 1002 (“clear homing”) before activating the robot. See [Section 5](#) 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 188) and [DeactivateSim](#) (page 192) 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 218));
- m: integer specifying the simulation mode type as
 - 0, disabled (equivalent to using the command [DeactivateSim](#) (page 192)),
 - 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 188) 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 198) command (i.e., by decelerating). The rest of the trajectory is deleted. The command [ResumeMotion](#) (page 201) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the [ClearMotion](#) (page 189) 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 190) 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 190).

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 (as well as the EOAT connected to the tool I/O port) and engages the brakes on the robot joints. You must deactivate the robot in order to use certain commands (e.g., [SetJointLimits](#) (page 206), [SetNetworkOptions](#) (page 209)).

Syntax

DeactivateRobot()

Responses

- [2004][Motors deactivated.]

Further details

If you deactivate a Meca500 that was already homed, and then reactivate it, you do not need to home it again, unless it has an MEGP 25* gripper installed. In the latter case, however, the homing process is performed only for the gripper, and so the robot does not move. You also need to home the robot again if you reactivated it with [ActivateRobot\(1\)](#) (page 187).

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 191) 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 192) 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 starts the robot and MEGP 25* gripper homing process (Section 3). While homing, it is critical to remove any obstacles that could hinder the robot and gripper movements.

Syntax

Home()

Responses

- [2002][Homing done.]
- [1032][Homing failed because joints are outside limits.]
- [1014][Homing failed.]

The first response (2002) is sent if homing was completed successfully. The second response (1032) is sent if the homing procedure failed because it was started while a robot joint was outside its user defined limits. The last response (1014) is sent if the homing failed for other reasons.

Usage restrictions

This command can only be executed when the robot is activated or already homed (in which case it will return the response and do nothing).

Note

Before homing is completed, the robot's positioning accuracy is lower than after homing. As a result:

- The reported real-time position will be less accurate (*GetRtJointPos* (page 280), *GetRtJointVel* (page 282), *GetRtCartPos* (page 276), *GetRtCartVel* (page 277), etc.);
- The robot may not reach the exact requested position when using *Recovery mode* (page 24) to move the robot before homing.

Cyclic protocols

In cyclic protocols, the *Home* (page 195) command is mapped to the *Home* bit in the *RobotControl* data. See [Table 4](#) for more details.

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 144) command is in progress when the [PauseMotion](#) (page 198) 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 276) or [GetRtJointPos](#) (page 280) commands.

The [PauseMotion](#) (page 198) command pauses the robot's motion without deleting the remaining trajectory, allowing it to be resumed with the [ResumeMotion](#) (page 201) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the [PauseMotion](#) (page 198) 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 199) 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 366).

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 200).

Cyclic protocols

In cyclic protocols, the *ResetError* (page 200) 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 198) command.
- Due to a torque overload configured in pause motion mode (see [SetTorqueLimitsCfg](#) (page 180)).
- By the SWStop (Meca500 R4), 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 198) or the robot was deactivated and then reactivated.

The [ResumeMotion](#) (page 201) command must also be sent after the [ClearMotion](#) (page 189) command. However, the robot will remain stationary until another motion command is received or retrieved from the motion queue. Additionally, the [ResumeMotion](#) (page 201) command must be sent after the [ResetError](#) (page 200) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the [ResumeMotion](#) (page 201) 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 261) 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]`

Further details

Note that after enabling calibration, and if it had been performed by Mecademic, the robot will move to slightly different (and more accurate) end-effector poses when executing Cartesian-space motion command. However, the resulting motion from [MoveJoints](#) (page 138) and [MoveJointsRel](#) (page 140) will remain unchanged.

Also, when moving a calibrated robot using Cartesian-space commands, *never move close to wrist and shoulder singularities, as the robot may behave unexpectedly*.

Usage restrictions

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

Cyclic protocols

In cyclic protocols, the [SetCalibrationCfg](#) (page 202) 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 203) 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 204) 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 158) 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 138), [MoveJointsRel](#) (page 140), [MovePose](#) (page 150), [MoveJump](#) (page 143), [MoveLin](#) (page 144), [MoveLinRelTrf](#) (page 146), [MoveLinRelWrf](#) (page 147), [PauseMotion](#) (page 198) and [ClearMotion](#) (page 189) commands, as well as after the [SetCartAcc](#) (page 155) and [SetJointAcc](#) (page 164) 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 144), [MoveLinRelTrf](#) (page 146), [MoveLinRelWrf](#) (page 147)) or two consecutive joint-space commands ([MoveJoints](#) (page 138), [MovePose](#) (page 150), [MoveJump](#) (page 143)).

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 207). 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;
- q_{n,max}: upper joint limit, in degrees.

 **Note**

The configured limits must allow each of the six joints at least 25° of motion range.

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 206) 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 207).

Responses

- [2092][n]

Usage restrictions

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

Cyclic protocols

In cyclic protocols, the [SetJointLimits](#) (page 206) 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 206) 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 207) 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 212) 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 immediately after the *StartSaving(1)* (page 221) command to specify whether the saved program should execute once or run indefinitely when the Start/Stop button on the robot's base is pressed. It applies only to program number 1 and only when starting a program using the Start/Stop button, not when using the *StartProgram* (page 220) command.

Syntax

SetOfflineProgramLoop(e)

Arguments

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

Default values

By default, looping is disabled.

Responses

- [1022][Robot was not saving the program.]

This command does not generate an immediate response. A message indicating whether loop execution was enabled or disabled is shown only when a program is being saved. However, if the command is sent while no program is being saved, the above message is returned.

Usage restrictions

This command can only be executed when a program is being saved using *StartSaving* (page 221).

Cyclic protocols

This command is not available in cyclic protocols.

SetPStop2Cfg

This command is used to set the behavior of the robot when the SWStop signal in the Meca500 R4 is activated.

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 201).

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 208) command. The [*SetRealTimeMonitoring*](#) (page 212) 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 212) command even if the robot is not activated.

Syntax

SetRealTimeMonitoring(*n₁*, *n₂*, ...)

Arguments

- *n₁,n₂*: a list of number codes or names, as follows
 - 2200 or *TargetJointPos*, for the response of the [*GetRtTargetJointPos*](#) (page 287) command;
 - 2201 or *TargetCartPos*, for the response of the [*GetRtTargetCartPos*](#) (page 283) command;
 - 2202 or *TargetJointVel*, for the response of the [*GetRtTargetJointVel*](#) (page 289) command;
 - 2203 or *TargetJointTorq*, for the response of the [*GetRtTargetJointTorq*](#) (page 288) command;
 - 2204 or *TargetCartVel*, for the response of the [*GetRtTargetCartVel*](#) (page 284) command;
 - 2210 or *JointPos*, for the response of the [*GetRtJointPos*](#) (page 280) command;
 - 2211 or *CartPos*, for the response of the [*GetRtCartPos*](#) (page 276) command;
 - 2212 or *JointVel*, for the response of the [*GetRtJointVel*](#) (page 282) command;
 - 2213 or *JointTorq*, for the response of the [*GetRtJointTorq*](#) (page 281) command;
 - 2214 or *CartVel*, for the response of the [*GetRtCartVel*](#) (page 277) command;
 - 2218 or *Conf*, for the response of the [*GetRtConf*](#) (page 278) command (sent only when changed);
 - 2219 or *ConfTurn*, for the response of the [*GetRtConfTurn*](#) (page 279) command (sent only when changed);

- 2220 or Accel, for the response of the *GetRtAccelerometer* (page 275) command;
- 2321 or GripperForce, for the response of the *GetRtGripperForce* (page 318) command;
- 2322 or GripperPos, for the response of the *GetRtGripperPos* (page 319) command;
- 2343 or VacuumPressure, for the response of the *GetRtVacuumPressure* (page 324) command;
- All, to enable all of the above responses.

Default values

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

Responses

- [2117][n₁, n₂ ...]
 - n₁, n₂ ...: a list of response codes.

Additional details

The *SetRealTimeMonitoring* (page 212) command does not have a cumulative effect; if you execute the command *SetRealTimeMonitoring(All)* (page 212) and then the command *SetRealTimeMonitoring(TargetCartPos)* (page 212) or the command *SetRealTimeMonitoring(2201)* (page 212), 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

Overview

The [SetRecoveryMode](#) (page 214) command temporarily relaxes certain safety restrictions, allowing limited motion when the robot would otherwise be blocked due to errors or limit violations. It is intended to safely reposition the robot (even before performing homing) or to recover from fault conditions.

Bypass limit conditions

Recovery mode allows moving the robot when its joints are outside user-defined limits, outside a work-zone limit, in self-collision, or when there is a torque overload.

Moving before homing

Recovery mode also allows moving the robot without performing homing (by calling [ActivateRobot](#) (page 187) but not [Home](#) (page 195)). This can be useful to move it to a safe position where there is enough clearance for homing movements.

Before homing is completed, the robot's positioning accuracy is lower than after homing. As a result:

- The reported real-time position will be less accurate ([GetRtJointPos](#) (page 280), [GetRtJointVel](#) (page 282), [GetRtCartPos](#) (page 276), [GetRtCartVel](#) (page 277), etc.);
- The robot may not reach the exact requested position when using [Recovery mode](#) (page 24) to move the robot before homing.

Similarly, controlling the MEGP 25* grippers or the MPM500 pneumatic module is possible, but with restrictions:

- Gripping force and velocity are limited;
- The [MoveGripper](#) (page 333) command cannot be used, only [GripperOpen](#) (page 331) and [GripperClose](#) (page 329) can be used.

Limited velocity

When recovery mode is enabled, joint and Cartesian velocities and accelerations are significantly reduced for safety reasons.

Syntax

SetRecoveryMode(e)

Arguments

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

Default values

By default, recovery mode is disabled.

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 214) 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 216) 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 262).

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 m500.

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 217) 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 217) 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 188) and [DeactivateSim](#) (page 192) commands (these commands can only be executed when the robot is deactivated).

The [SetSimModeCfg](#) (page 218) command configures the default simulation mode type (fast or normal) enabled when [ActivateSim](#) (page 188) 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 218).

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 137)). 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 219) 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.

Alternately, pressing the Start/Stop button on the robot's base will start program named "1", if such a program exists, and execute it the number of times defined by the *SetOfflineProgramLoop* (page 210) command.

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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the *StartProgram* (page 220) 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 365) that can later be played using the *StartProgram* (page 220) 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 221) and *StopSaving* (page 223) commands.

Note

We recommend using the *The code editor panel* to edit, save, and delete offline programs, as it is easier and more flexible than using *StartSaving* (page 221) / *StopSaving* (page 223).

Note

The robot will execute but not record request commands (Get*). If the robot receives a change of state command (*Home* (page 195), *PauseMotion* (page 198), *SetEom* (page 205), etc.) while recording, it will abort saving the program. Finally, only program 1 can be executed using the Start/Pause button on the Meca500's base.

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 [The code editor panel](#) to edit, save, and delete offline programs, as it is easier and more flexible than using [StartSaving](#) (page 221) / [StopSaving](#) (page 223).

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 57) 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. If 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 294) command allows the application to know if a received robot status (code 2007) is the response of the [*GetStatusRobot*](#) (page 294) 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 226) 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 166) and [GetJointVel](#) (page 246)) 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 258), or after a firmware upgrade, as in the case of [GetFwVersion](#) (page 242)).

Contrary to motion commands, *robot control commands are executed immediately*, i.e., are instantaneous. Therefore, if you send a [SetTrf](#) (page 182) command, then a [MovePose](#) (page 150) command, then another [SetTrf](#) (page 182) command, and immediately after that a [GetTrf](#) (page 268) command, you will get the arguments of the first [SetTrf](#) (page 182) command.

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

- [GetAutoConf](#) (page 230)
- [GetAutoConfTurn](#) (page 231)
- [GetBlending](#) (page 232)
- [GetCalibrationCfg](#) (page 233)
- [GetCartAcc](#) (page 234)
- [GetCartAngVel](#) (page 235)
- [GetCartLinVel](#) (page 236)
- [GetCheckpoint](#) (page 237)
- [GetCheckpointDiscarded](#) (page 238)
- [GetConf](#) (page 239)
- [GetConfTurn](#) (page 240)
- [GetEthernetIpEnabled](#) (page 241)
- [GetJointAcc](#) (page 243)
- [GetJointLimits](#) (page 244)
- [GetJointLimitsCfg](#) (page 245)
- [GetJointVel](#) (page 246)
- [GetJointVelLimit](#) (page 247)
- [GetMonitoringInterval](#) (page 249)
- [GetMoveDuration](#) (page 250)

- *GetMoveDurationCfg* (page 251)
- *GetMoveMode* (page 254)
- *GetNetworkOptions* (page 255)
- *GetPayload* (page 257)
- *GetProfinetEnabled* (page 259)
- *GetPStop2Cfg* (page 256)
- *GetRealTimeMonitoring* (page 260)
- *GetRobotName* (page 262)
- *GetSimModeCfg* (page 264)
- *GetTimeScaling* (page 265)
- *GetTorqueLimits* (page 266)
- *GetTorqueLimitsCfg* (page 267)
- *GetTrf* (page 268)
- *GetVelTimeout* (page 269)
- *GetWrf* (page 270)

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

- *GetFwVersion* (page 242)
- *GetModelJointLimits* (page 248)
- *GetProductType* (page 258)
- *GetRobotCalibrated* (page 261)
- *GetRobotSerial* (page 263)

A few other data request commands exist, but these are presented in the sections *Work zone supervision and collision prevention commands* (page 296) (e.g., *GetToolSphere* (page 301)), *Commands for optional accessories* (page 310) and *Commands for managing variables (beta)* (page 352) (e.g., *GetVariable* (page 360)).

GetAutoConf

This command returns the state of the automatic posture configuration selection, which can be influenced by the *SetAutoConf* (page 152) and *SetConf* (page 160) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetAutoConf* (page 230) is represented by *DynamicTypeID* 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 153) and *SetConfTurn* (page 162) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetAutoConfTurn* (page 231) is represented by *DynamicTypeID* 20. See [Section 5](#) for more details.

GetBlending

This command returns the blending percentage, which is set using the *SetBlending* (page 154) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetBlending* (page 232) is represented by *DynamicTypeID* 21. See [Section 5](#) for more details.

GetCalibrationCfg

This command returns the state of the optional robot calibration, configured using the [*SetCalibrationCfg*](#) (page 202) 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 233) is represented by *DynamicTypeID* 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 155) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetCartAcc* (page 234) is represented by *DynamicTypeID* 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 156) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetCartAngVel](#) (page 235) is represented by *DynamicTypeID* 22. See [Section 5](#) for more details.

GetCartLinVel

This command returns the desired Tool Center Point (TCP) velocity limit, configured using the *SetCartLinVel* (page 157) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetCartLinVel* (page 236) is represented by *DynamicTypeID* 22. See [Section 5](#) for more details.

GetCheckpoint

This command returns the argument of the last executed *SetCheckpoint* (page 158).

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 237) 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 158)).

Checkpoint can be discarded by *ClearMotion* (page 189), *DeactivateRobot* (page 191), 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 238) 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 150) or [MoveLin*](#) command in the motion queue. This configuration is either explicitly specified using the [SetConf](#) (page 160) command or automatically assigned when the [SetAutoConf\(0\)](#) (page 152) command is executed.

Syntax

GetConf()

Responses

- [2029][c_s, c_e, c_w]
 - c_s : shoulder configuration parameter, either -1 or 1^\dagger ;
 - c_e : elbow configuration parameter, either -1 or 1^\dagger ;
 - c_w : wrist configuration parameter, either -1 or 1^\dagger .

[†] If automatic posture configuration selection is enabled, each parameter's 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetConf](#) (page 239) is represented by [DynamicTypeID](#) 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 150) or [MoveLin*](#) command in the motion queue. This is either the turn configuration explicitly specified using the [SetConfTurn](#) (page 162) command or the one automatically assigned when the [SetAutoConfTurn\(0\)](#) (page 153) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetConfTurn](#) (page 240) is represented by *DynamicTypeID* 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 193) command for more details.

Syntax

GetEthernetIpEnabled()

Responses

- [2073][e]
 - e: 0, 1 or 2 as defined in the description of the *EnableEtherNetIp* (page 193) 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 242) is represented by *DynamicTypeID* 1. See Section 5 for more details.

GetJointAcc

This command returns the desired joint accelerations reduction factor, set using the *SetJointAcc* (page 164) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetJointAcc* (page 243) is represented by *DynamicTypeID* 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 206) command and enabled using the [SetJointLimitsCfg](#) (page 207) 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;
 - $q_{n,\max}$: upper joint limit, in degrees.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command [GetJointLimits](#) (page 244) is represented by *DynamicTypeID* 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 207).

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 245) is represented by *DynamicTypeID* 11. See [Section 5](#) for more details.

GetJointVel

This command returns the desired joint velocities reduction factor, set using the [SetJointVel](#) (page 166) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetJointVel](#) (page 246) is represented by *DynamicTypeID* 22. See [Section 5](#) for more details.

GetJointVelLimit

This command returns the desired joint velocities override, set using the [SetJointVelLimit](#) (page 168) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetJointVelLimit](#) (page 247) is represented by *DynamicTypeID* 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 6;
 - q_{n,min}: lower joint limit, in degrees;
 - q_{n,max}: upper joint limit, in degrees.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetModelJointLimits* (page 248) is represented by *DynamicTypeID* 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 170) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetMoveDuration* (page 250) is represented by *DynamicTypeID* 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 170) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetMoveDurationCfg* (page 251) is represented by *DynamicTypeID* 29. See [Section 5](#) for more details.

GetMoveJumpApproachVel

This command is available only on our SCARA robots.

Cyclic protocols

In cyclic protocols, the commands *GetMoveJumpApproachVel* (page 252) is represented by *DynamicTypeID* 28. See [Section 5](#) for more details.

GetMoveJumpHeight

This command is available only on our SCARA robots.

Cyclic protocols

In cyclic protocols, the commands *GetMoveJumpHeight* (page 253) is represented by *DynamicTypeID* 27. See [Section 5](#) for more details.

GetMoveMode

This command returns the default move mode set by the [SetMoveMode](#) (page 175) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetMoveMode](#) (page 254) is represented by *DynamicTypeID* 29. See [Section 5](#) for more details.

GetNetworkOptions

This command returns the parameters affecting the network connection.

Syntax

GetNetworkOptions()

Responses

- [2119][n₁, n₂, n₃, n₄, n₅, n₆]
 - n₁: number of successive keep-alive TCP packets that can be lost before the TCP connection is closed, where n₁ is an integer number ranging from 0 to 60;
 - n₂, n₃, n₄, n₅, n₆: 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 211) 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 176) 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 364), in millimeters.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the commands *GetPayload* (page 257) is represented by *DynamicTypeID* 31. See Section 5 for more details.

GetProductType

This command returns the type (model) of the product.

Syntax

GetProductType()

Responses

- [2084][Meca500]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the commands *GetProductType* (page 258) is represented by *DynamicTypeID* 2. See Section 5 for more details.

GetProfinetEnabled

This command returns the state of the PROFINET protocol. See the description of the [EnableProfinet](#) (page 194) command for more details.

Syntax

GetProfinetEnabled()

Responses

- [2077][e]
 - e: 0, 1 or 2 as defined in the description of the [EnableProfinet](#) (page 194) 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 212) command.

Syntax

GetRealTimeMonitoring()

Responses

- [2117][n₁, n₂, ...]

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 261) is represented by *DynamicTypeID* 30. See [Section 5](#) for more details.

GetRobotName

This command returns the robot's name, set with the command [SetRobotName](#) (page 216). 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 263) is represented by *DynamicTypeID* 3. See Section 5 for more details.

GetSimModeCfg

This command returns the default simulation mode set using the [*SetSimModeCfg*](#) (page 218) 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 219) 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 265) is represented by *DynamicTypeID* 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 178).

Syntax

GetTorqueLimits()

Responses

- [2161][$\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6$]
 - τ_i : percentage of the maximum allowable torque that can be applied at motor i , where $i = 1, 2, \dots, 6$ ranging from 0.001 to 100.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the commands *GetTorqueLimits* (page 266) is represented by *DynamicTypeID* 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 178). This desired behavior is configured using the [SetTorqueLimitsCfg](#) (page 180) command.

Syntax

GetTorqueLimitsCfg()

Responses

- [2160][l, m]
 - l: an integer defining the torque limit event severity (see [SetTorqueLimitsCfg](#) (page 180));
 - m: an integer defining the detection mode (see [SetTorqueLimitsCfg](#) (page 180)).

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the commands [GetTorqueLimitsCfg](#) (page 267) is represented by *DynamicTypeID* 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 182) command.

Syntax

GetTrf()

Responses

- [2014][x, y, z, α , β , γ]
 - x, y, z: the coordinates of the origin of the TRF with respect to the FRF, in mm;
 -
 - α , β , γ : Euler angles representing the orientation of the TRF with respect to the FRF, in degrees.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetTrf](#) (page 268) 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 183) 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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetVelTimeout](#) (page 269) is represented by *DynamicTypeID* 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 184) command.

Syntax

GetWrf()

Responses

- [2014][x, y, z, α , β , γ]
 - x, y, z: the coordinates of the origin of the WRF with respect to the BRF, in mm;
 - α , β , γ : Euler angles representing the orientation of the WRF with respect to the BRF, in degrees.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command *GetWrf* (page 270) 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 281), [GetRtJointPos](#) (page 280), [GetRtCartPos](#) (page 276), etc.
- **Real-time target data** These commands return data based on targets calculated by the trajectory planner. Examples include [GetRtTargetJointPos](#) (page 287), [GetRtTargetCartPos](#) (page 283), etc.

For instance, if the robot is active and stationary, the [GetRtTargetJointPos](#) (page 287) 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 may oscillate by $\pm 0.001^\circ$ around the desired angles. If you execute the [GetRtJointPos](#) (page 280) 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 280)) and the target positions ([GetRtTargetJointPos](#) (page 287)) 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:

- [GetCmdPendingCount](#) (page 273)
- [GetRtAccelerometer](#) (page 275)
- [GetRtCartPos](#) (page 276)
- [GetRtCartVel](#) (page 277)
- [GetRtConf](#) (page 278)
- [GetRtConfTurn](#) (page 279)
- [GetRtJointPos](#) (page 280)
- [GetRtJointTorq](#) (page 281)
- [GetRtJointVel](#) (page 282)
- [GetRtTargetCartPos](#) (page 283)

- [*GetRtTargetCartVel*](#) (page 284)
- [*GetRtTargetConf*](#) (page 285)
- [*GetRtTargetConfTurn*](#) (page 286)
- [*GetRtTargetJointPos*](#) (page 287)
- [*GetRtTargetJointTorq*](#) (page 288)
- [*GetRtTargetJointVel*](#) (page 289)
- [*GetRtTrf*](#) (page 290)
- [*GetRtWrf*](#) (page 291)
- [*GetRtc*](#) (page 292)
- [*GetSafetyStopStatus*](#) (page 293)
- [*GetStatusRobot*](#) (page 294)
- [*GetTorqueLimitsStatus*](#) (page 295)

A few other real-time data request commands exist, but these are presented in the sections *Work zone supervision and collision prevention commands* (page 296) ([*GetCollisionStatus*](#) (page 300), [*GetWorkZoneStatus*](#) (page 304)) and *Commands for optional accessories* (page 310).

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 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

This command is not available in cyclic protocols.

GetOperationMode

This command is not available in the Meca500.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetOperationMode* (page 274) is represented by the *OperationMode* field of the *SafetyStatus* section (see [Section 5](#)).

GetRtAccelerometer

An accelerometer is embedded in link 5 of the Meca500 (i.e., the body with the I/O port). It reports the acceleration of link 5 with respect to the WRF in the range $\pm 32,000$, which corresponds to $\pm 2g$. If the robot is not moving and is installed upright on a stationary horizontal surface, [GetRtAccelerometer\(5\)](#) (page 275) will return roughly $\{0,0,-16000\}$, no matter what the joint set. In other words, in stationary conditions, you can essentially think of the accelerometer as if it were embedded in the base of the robot.

Syntax

GetRtAccelerometer(n)

Arguments

- n: link number, currently must be 5.

Responses

- [2220][t, n, a_x, a_y, a_z]
 - t: timestamp in microseconds;
 - n: link number, currently 5;
 - a_x, a_y, a_z: acceleration in link 5, measured with respect to the WRF, in units such that 16,000 is equivalent to 9.81 m/s² (i.e., 1g).

 **Note**

Data from this accelerometer should not be used for precise measurements.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command [GetRtAccelerometer](#) (page 275) is represented by *DynamicTypeID* 46. See [Section 5](#) for more details.

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, α , β , γ]
 - t: timestamp in microseconds;
 - x, y, z: the coordinates of the origin of the TRF with respect to the WRF, in mm;
 -
 - α , β , γ : Euler angles representing the orientation of the TRF 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 276) is represented by *DynamicTypeID* 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_x , ω_y , 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_x , ω_y , ω_z : components of the angular velocity vector of the TRF with respect to the WRF, in °/s.

The current TCP speed with respect to the WRF is therefore $(\dot{x}^2 + \dot{y}^2 + \dot{z}^2)^{1/2}$, and the current angular speed of the end-effector with respect to the WRF is $(\omega_x^2 + \omega_y^2 + \omega_z^2)^{1/2}$. Note that *the components of the angular velocity vector are not the time derivatives of the Euler angles*.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command `GetRtCartVel` (page 277) is represented by `DynamicTypeID` 44. See [Section 5](#) for more details.

GetRtConf

Unlike the *GetConf* (page 239) command, which returns the desired posture configuration parameters, the *GetRtConf* (page 278) command returns the current posture configuration parameters, calculated from real-time data provided by the joint encoders. Additionally, the *GetRtConf* (page 278) command includes a timestamp in its response.

Syntax

GetRtConf()

Responses

- [2218][c_s, c_e, c_w]
 - c_s : shoulder configuration parameter, -1, 1, or 0^\dagger ;
 - c_e : elbow configuration parameter, -1, 1, or 0^\dagger ;
 - c_w : wrist configuration parameter, -1, 1, or 0^\dagger .

[†] 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 278) is represented by *DynamicTypeID* 45. See Section 5 for more details.

GetRtConfTurn

Contrary to [GetConfTurn](#) (page 240), which returns the desired turn configuration parameter, [GetRtConfTurn](#) (page 279) 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 279) 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 279) is represented by *DynamicTypeID* 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, $\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$]
 - t: timestamp in microseconds;
 - θ_i : the angle of joint i, in degrees;

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtJointPos* (page 280) is represented by *DynamicTypeID* 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, $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6$]
 - t: timestamp in microseconds;
 - τ_i : the torque of motor i as a signed percentage of the maximum allowable torque (i = 1, 2, ..., 6).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtJointTorq* (page 281) is represented by *DynamicTypeID* 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, $\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6$]
 - t: timestamp in microseconds;
 - ω_i : the rate of change of joint i, in °/s (i = 1, 2, ..., 6).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtJointVel* (page 282) is represented by *DynamicTypeID* 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, α , β , γ]
 - t: timestamp in microseconds;
 - x, y, z: the coordinates of the origin of the TRF with respect to the WRF, in mm;
 -
 - α , β , γ : Euler angles representing the orientation of the TRF 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 283) 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_x , ω_y , 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_x , ω_y , ω_z : components of the angular velocity vector 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 284) is represented by *DynamicTypeID* 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^\dagger ;
 - c_e : elbow configuration parameter, -1, 1, or 0^\dagger ;
 - c_w : wrist configuration parameter, -1, 1, or 0^\dagger .

[†] 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 285) 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 286) is represented by the *TargetConfiguration* section (see Section 5).

GetRtTargetJointPos

This command returns the current target joint set.

Syntax

GetRtTargetJointPos()

Responses

- [2200][t, $\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$]
 - t: timestamp in microseconds;
 - θ_i : the angle of joint i, in degrees;

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 287) is represented by the *TargetJointSet* section (see [Section 5](#)).

GetRtTargetJointTorq

This command returns the current target (calculated) motor torques.

Syntax

GetRtTargetJointTorq()

Responses

- [2203][t, $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6$]
 - t: timestamp in microseconds;
 - τ_i : the torque of motor i as a signed percentage of the maximum allowable torque (i = 1, 2, ..., 6).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTargetJointTorq* (page 288) is represented by *DynamicTypeID* 33. See [Section 5](#) for more details.

GetRtTargetJointVel

This command returns the current target joint velocities.

Syntax

GetRtTargetJointVel()

Responses

- [2202][t, $\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6$]
 - t: timestamp in microseconds;
 - ω_i : the rate of change of joint i, in °/s (i = 1, 2, ..., 6).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTargetJointVel* (page 289) is represented by *DynamicTypeID* 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 182) command. It returns exactly the same pose as the *GetTrf* (page 268) command, but the response code is different, and a timestamp precedes the pose data.

Syntax

GetRtTrf()

Responses

- [2229][x, y, z, α , β , γ]
 - t: timestamp in microseconds;
 - x, y, z: the coordinates of the origin of the TRF with respect to the FRF, in mm;
 -
 - α , β , γ : Euler angles representing the orientation of the TRF 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 290) 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 184) command. It returns exactly the same pose as the [GetWrf](#) (page 270) command, but the response code is different, and a timestamp precedes the pose data.

Syntax

GetRtWrf()

Responses

- [2228][x, y, z, α , β , γ]
 - t: timestamp in microseconds;
 - x, y, z: the coordinates of the origin of the WRF with respect to the BRF, in mm;
 -
 - α , β , γ : Euler angles representing the orientation of the WRF 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 291) is represented by the *TargetWrf* section (see [Section 5](#)).

GetRtc

This command returns the current Epoch Time in seconds, set using the [SetRtc](#) (page 217), 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 292) 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;
 - 3070, for the state of the E-Stop safety stop signal;
 - 3083, for the state of the safety stop signal associated with robot reboot or reset signal;
 - 3086, for the state of the safety stop signal associated with a connection drop.

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 198) or a [ClearMotion](#) (page 189) 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 294) 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 295) is represented by the *ExcessiveTorque* section (see [Section 5](#)).

Work zone supervision and collision prevention commands

In addition to using the [SetJointLimits](#) (page 206) and [SetJointLimitsCfg](#) (page 207) commands to further constrain the robot's joint limits, you can also define a work zone with the [SetWorkZoneLimits](#) (page 309) command (see [Figure 17](#)). 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 306) command.

You can then use the [SetWorkZoneCfg](#) (page 307) 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 305) 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 17](#) illustrates the objects currently supervised. The base STL model also includes part of the cables coming from the base (not shown).

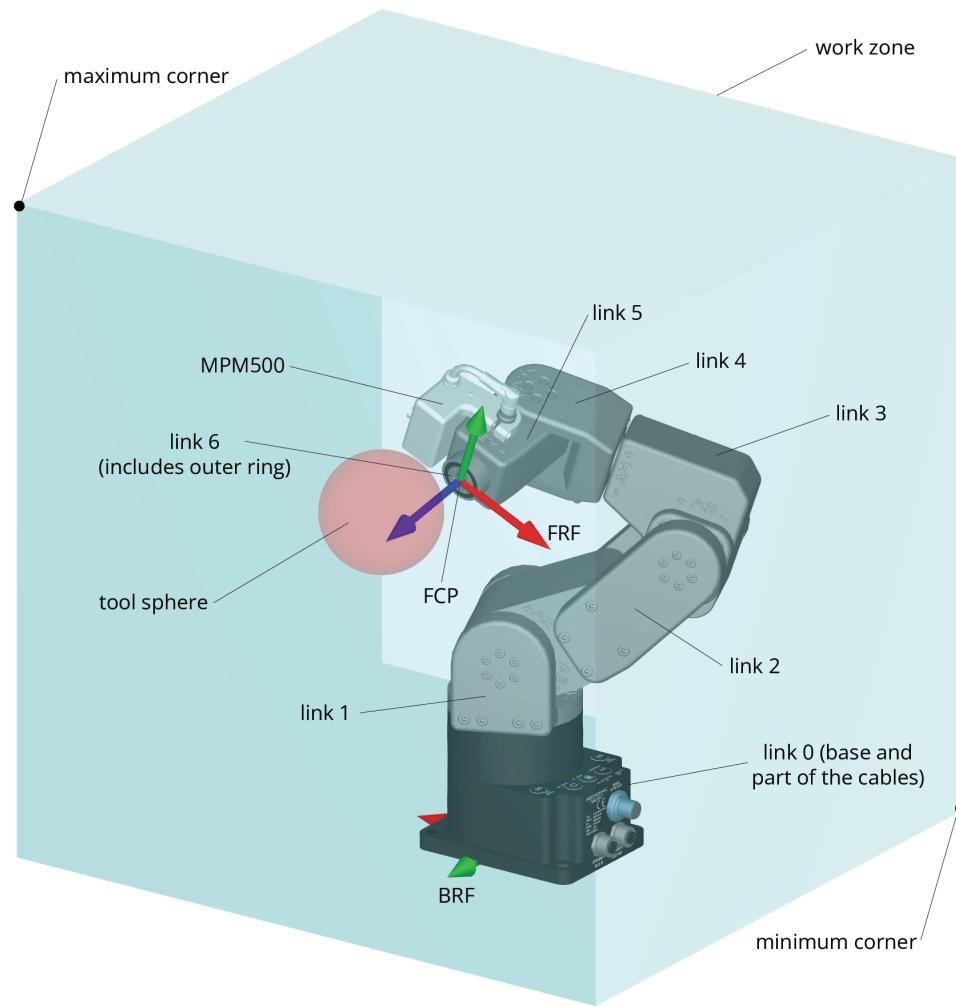


Figure 17: Objects tested in the work zone supervision and collision prevention feature

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 300)
- [GetWorkZoneStatus](#) (page 304)
- [SetCollisionCfg](#) (page 305) / [GetCollisionCfg](#) (page 299)

- *SetToolSphere* (page 306) / *GetToolSphere* (page 301)
- *SetWorkZoneCfg* (page 307) / *GetWorkZoneCfg* (page 302)
- *SetWorkZoneLimits* (page 309) / *GetWorkZoneLimits* (page 303)

GetCollisionCfg

This command returns the severity level set with the *SetCollisionCfg* (page 305) 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 299) command is represented by *DynamicTypeID* 36. See [Section 5](#) for more details.

GetCollisionStatus

This command returns the current collision status (refer to [Figure 17](#)).

Syntax

GetCollisionStatus

Responses

- [2182][v, g₁, o_{id,1}, g₂, o_{id,2}]
 - v: collision state (1 or 0[†]),
 - g₁, g₂: 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, 10,000 for MPM500, .
- [†] If v = 0, g₁ = g₂ = 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 300) command is represented by *DynamicTypeID* 37. See [Section 5](#) for more details.

GetToolSphere

This command returns the current definition of the tool sphere, set with the *SetToolSphere* (page 306) 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 301) command is represented by *DynamicTypeID* 19. See [Section 5](#) for more details.

GetWorkZoneCfg

This command returns the current work zone configuration, set with the [*SetWorkZoneCfg*](#) (page 307) 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 302) command is represented by *DynamicTypeID* 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 309) 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 303) command is represented by *DynamicTypeID* 18. See [Section 5](#) for more details.

GetWorkZoneStatus

This command returns the current work zone violation status (refer to Figure 17).

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, 10,000 for MPM500, .
- [†] 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 304) command is represented by *DynamicTypeID* 38. See Section 5 for more details.

SetCollisionCfg

This command specifies the event severity for the collision supervision (robot links, tool sphere, and MPM500 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 305) 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 309) command can then be supervised, as defined by the [SetWorkZoneCfg](#) (page 307) and [SetCollisionCfg](#) (page 305) 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 Meca500 does not check interferences between the tool sphere and the robot flange, the tool sphere and link 5 (the one with the I/O port and the “-A6+” engraving), and the tool sphere and the MPM500 module. Note that, if you set your tool sphere too large, e.g., with [SetToolSphere\(0,0,060\)](#) (page 306), it will always interfere with link 4, i.e., the yoke one with the “-A5+” engraving.

Usage restrictions

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

Cyclic protocols

In cyclic protocols, the [SetToolSphere](#) (page 306) 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 306) command, and the MPM500 module if detected on the Meca500);
 - 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 307) command is represented by *MotionCommandID* 152. See Section 5 for more details.

SetWorkZoneLimits

This command 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 than 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 309).

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 309) 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 accessories for your robot: electric grippers (MEGP 25*) and pneumatic module (MPM500). Some of the commands in this section are queued, others are instantaneous (Get*, [SetExtToolSim](#) (page 335), and *_Immediate).

The following is the complete list of commands used for the electric grippers (MEGP 25*) and pneumatic module (MPM500):

- [GetExtToolFwVersion](#) (page 311)
- [GetRtExtToolStatus](#) (page 317)
- [GetRtGripperForce](#) (page 318)
- [GetRtGripperPos](#) (page 319)
- [GetRtGripperState](#) (page 320)
- [GetRtValveState](#) (page 326)
- [GripperClose](#) (page 329)
- [GripperOpen](#) (page 331)
- [MoveGripper](#) (page 333)
- [SetExtToolSim](#) (page 335) / [GetExtToolSim](#) (page 312)
- [SetGripperForce](#) (page 336) / [GetGripperForce](#) (page 313)
- [SetGripperRange](#) (page 337) / [GetGripperRange](#) (page 314)
- [SetGripperVel](#) (page 339) / [GetGripperVel](#) (page 315)
- [SetValveState](#) (page 347)

GetExtToolFwVersion

This instantaneous command returns the firmware version of Meca500's EOAT connected to its tool I/O port. The Meca500 must be activated. If during the activation, the robot detects that the firmware version of the EOAT is older than the firmware version of the robot, the [3039] response will be given, and the activation process will fail. If no EOAT is detected, the x's in the [2086] message will be zeros.

Syntax

GetExtToolFwVersion()

Responses

- [2086][vx.x.x]
- [3039][External tool firmware must be updated.]

If no external tool is connected, the response will be [2086][v0.0.0].

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetExtToolSim

This instantaneous command returns the emulation mode of the EOAT that can be connected to the tool port of the robot, normally set with the *SetExtToolSim* (page 335).

Syntax

GetExtToolSim()

Responses

- [2047][m]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetGripperForce

This queued command returns the grip force set by the [SetGripperForce](#) (page 336) command.

Syntax

GetGripperForce()

Responses

- [2158][p]
 - p: grip force limit, as signed percentage of the maximum grip force (~40 N).

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetGripperForce](#) (page 313) is represented by *DynamicTypeID* 23. See [Section 5](#) for more details.

GetGripperRange

This queued command returns the gripper range set by the [SetGripperRange](#) (page 337) command.

Syntax

GetGripperRange()

Responses

- [2162][d_{closed} , d_{open}]
 - d_{closed} : fingers opening that should correspond to closed state, in mm;
 - d_{open} : fingers opening that should correspond to open state, in mm.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetGripperRange](#) (page 314) is represented by *DynamicTypeID* 23. See [Section 5](#) for more details.

GetGripperVel

This queued command returns the gripper velocity set by the [SetGripperVel](#) (page 339) command.

Syntax

`GetGripperVel()`

Responses

- [2159][p]
 - p: percentage of maximum finger velocity (~50 mm/s).

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 366). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1006][The robot is not homed.]

Cyclic protocols

In cyclic protocols, the command [GetGripperVel](#) (page 315) is represented by *DynamicTypeID* 23. See [Section 5](#) for more details.

GetLoSim

This command is available only on the MCS500 robot.

GetRtExtToolStatus

This instantaneous command returns the general status of the external tool connected to the I/O port of the Meca500, preceded with a timestamp. For additional status information, use the commands [GetRtGripperState](#) (page 320) or [GetRtValveState](#) (page 326).

Syntax

GetRtExtToolStatus()

Responses

- [2300][t, simType, phyType, hs, es, oh]
 - t: timestamp in microseconds;
 - simType: simulated external tool type (0 for none, 10 for MEGP 25E gripper, 11 for MEGP 25LS gripper, 20 for MPM500 pneumatic module);
 - phyType: physical external tool type mounted on the Meca500 (0 for none, 10 for MEGP 25E gripper, 11 for MEGP 25LS gripper, 20 for MPM500 pneumatic module);
 - hs: homing state (0 for homing not performed, 1 for homing performed);
 - es: error state (0 for absence of error, 1 for presence of error);
 - oh: overheat (0 if there is no overheat, 1 if the gripper is in overheat).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command [GetRtExtToolStatus](#) (page 317) is represented by *DynamicTypeID* 52. See [Section 5](#) for more details.

Cyclic protocols

This command is not available in cyclic protocols.

GetRtGripperForce

This instantaneous command returns the currently applied grip force of the MEGP 25* grippers, preceded by a timestamp.

Syntax

GetRtGripperForce()

Responses

- [2321][t, p]
 - t: timestamp in microseconds;
 - p: currently applied grip force, as signed percentage of the maximum grip force (~40 N).

A positive grip force means the jaws are forcing outwards, while a negative grip force means the jaws are forcing towards each other.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtGripperForce* (page 318) is represented by *DynamicTypeID* 53. See [Section 5](#) for more details.

GetRtGripperPos

This instantaneous command returns the current fingers opening (also referred to as gripper position) of the MEGP 25* grippers (see [MoveGripper](#) (page 333)), preceded with a timestamp.

Syntax

GetRtGripperPos()

Responses

- [2322][t, d]
 - t: timestamp in microseconds;
 - d: fingers opening, in mm.

Note

You can use this command to perform rough measurements on a part. However, you would need to use short, rigid, precisely machined, and properly installed fingers. These fingers will also have to be designed in such a way that the part is automatically aligned. For example, you can measure the diameter of a cylindrical vial, once you lift the vial. Even in such perfect conditions, you can still obtain measurement errors of as much as 0.5 mm.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command [GetRtGripperPos](#) (page 319) is represented by *DynamicTypeID* 53. See [Section 5](#) for more details.

GetRtGripperState

This instantaneous command returns the current state of the MEGP 25* grippers connected to the I/O port of the Meca500, preceded with a timestamp.

Syntax

GetRtGripperState()

Responses

- [2320][t, hp, dr, gc, go]
 - t: timestamp in microseconds;
 - hp: holding part (0 if the gripper is not forcing, 1 otherwise);
 - dr: desired fingers opening reached (0 after a *MoveGripper* (page 333), *GripperClose* (page 329) or *GripperOpen* (page 331) command was executed and until the desired fingers opening was reached, 1 otherwise);
 - gc: gripper closed (1 if the current fingers opening is equal to or smaller than the fingers opening detected during homing or defined with the *SetGripperRange* (page 337) command as the one corresponding to the closed position, 0 otherwise);
 - go: gripper open (1 if the current fingers opening is equal to or greater than the fingers opening detected during homing or defined with the *SetGripperRange* (page 337) command as the one corresponding to the open position, 0 otherwise).

Usage restrictions

This command can be executed in any robot state.

If no gripper is connected, the robot will report the following:

- [1038][No gripper connected.]

Cyclic protocols

In cyclic protocols, the command *GetRtGripperState* (page 320) is represented by *DynamicTypeID* 53. See [Section 5](#) for more details.

GetRtInputState

This command is available only on the MCS500 robot.

GetRtIoStatus

This command is available only on the MCS500 robot.

GetRtOutputState

This command is available only on the MCS500 robot.

GetRtVacuumPressure

This command is available only on the MCS500 robot.

GetRtVacuumState

This command is available only on the MCS500 robot.

GetRtValveState

This instantaneous command returns the current state of the MPM500 pneumatic module connected to the I/O port of the Meca500, preceded with a timestamp.

Syntax

GetRtValveState()

Responses

- [2310][t, v₁, v₂]
 - t: timestamp in microseconds;
 - v₁: state of valve 1 (0 if closed, 1 if open);
 - v₂: state of valve 2 (0 if closed, 1 if open).

Usage restrictions

This command can be executed in any robot state.

If no pneumatic module is connected, the robot will report the following:

- [1041][No pneumatic module connected.]

Cyclic protocols

In cyclic protocols, the command *GetRtValveState* (page 326) is represented by *DynamicTypeID* 53. See [Section 5](#) for more details.

GetVacuumPurgeDuration

This command is available only on the MCS500 robot.

GetVacuumThreshold

This command is available only on the MCS500 robot.

GripperClose

This queued command is used to close the MEGP 25E or MEGP 25LS grippers. The gripper will move its fingers together until the grip force reaches 40 N. You can reduce this maximum grip force using the [SetGripperForce](#) (page 336) command. You can also control the speed of the gripper with the [SetGripperVel](#) (page 339) command.

By default, [GripperClose](#) (page 329) command closes the gripper fingers until resistance is met. However, a maximum closing distance can be set using the [SetGripperRange](#) (page 337) command.

You can use this command with the MPM500 pneumatic module too. See the [SetValveState](#) (page 347) command for more details.

Syntax

GripperClose()

Note

The gripper commands [GripperOpen](#) (page 331), [GripperClose](#) (page 329), and [MoveGripper](#) (page 333) are queued like other motion commands but, their execution runs in parallel with robot motion:

- They start at the beginning of the blending (or deceleration) phase of the preceding motion command in the queue.
- They continue executing in parallel with any subsequent motion commands, which begin immediately as the gripper command starts.

To coordinate with motion commands:

- To ensure a gripper command starts only after the robot has come to a complete stop, insert a [Delay](#) (page 137) or use [SetCheckpoint](#) (page 158) and wait for it before posting the gripper command.
- To ensure the robot remains stationary until the gripper command completes, insert a [Delay](#) (page 137) or use [GetRtGripperState](#) (page 320) to detect when the gripper is holding or has released the part.

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 366), 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.]
- [1006][The robot is not homed.]

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

If no gripper is connected, the robot will report the following:

- [1038][No gripper connected.]

Cyclic protocols

In cyclic protocols, the commands *GripperClose* (page 329) and *GripperOpen* (page 331) are represented by *MotionCommandID* 18. See [Section 5](#) for more details.

GripperOpen

This queued command is used to open the MEGP 25E or MEGP 25LS grippers. The gripper will move its fingers apart until the grip force reaches 40 N. You can reduce this maximum grip force using the [SetGripperForce](#) (page 336) command. You can also control the speed of the gripper with the [SetGripperVel](#) (page 339) command.

By default, [GripperClose](#) (page 329) command opens the gripper fingers until resistance is met. However, a maximum opening distance can be set using the [SetGripperRange](#) (page 337) command.

You can use this command with the MPM500 pneumatic module too. See the [SetValveState](#) (page 347) command for more details.

Syntax

GripperOpen()

Note

The gripper commands [GripperOpen](#) (page 331), [GripperClose](#) (page 329), and [MoveGripper](#) (page 333) are queued like other motion commands but, their execution runs in parallel with robot motion:

- They start at the beginning of the blending (or deceleration) phase of the preceding motion command in the queue.
- They continue executing in parallel with any subsequent motion commands, which begin immediately as the gripper command starts.

To coordinate with motion commands:

- To ensure a gripper command starts only after the robot has come to a complete stop, insert a [Delay](#) (page 137) or use [SetCheckpoint](#) (page 158) and wait for it before posting the gripper command.
- To ensure the robot remains stationary until the gripper command completes, insert a [Delay](#) (page 137) or use [GetRtGripperState](#) (page 320) to detect when the gripper is holding or has released the part.

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 366), 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.]
- [1006][The robot is not homed.]

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

If no gripper is connected, the robot will report the following:

- [1038][No gripper connected.]

Cyclic protocols

In cyclic protocols, the commands *GripperClose* (page 329) and *GripperOpen* (page 331) are represented by *MotionCommandID* 18. See [Section 5](#) for more details.

MoveGripper

The MEGP 25* grippers are equipped with incremental encoders, so it is impossible to directly measure the absolute positions of the gripper jaws. Thus, during the homing of the robot, the gripper is also homed by completely closing and then opening its fingers, until resistance is met in each direction. The maximum fingers opening is detected and is a positive number not larger than 6 mm (MEGP 25E) or 48 mm (MEGP 25LS). Most importantly, the fingers opening, a non-negative distance, is defined as the sum of the distances traveled by each jaw from their fully-closed positions detected during homing.

The *MoveGripper* (page 333) command makes the gripper fingers move towards the specified fingers opening.

Syntax

MoveGripper(d)

Arguments

- d: desired fingers opening, a non-negative value in mm, from 0 to the maximum fingers opening detected during homing.

Further details

Unlike other position-mode Move* commands, the *MoveGripper* (page 333) command does not return an error if the desired finger opening is blocked by an object. The fingers will continue applying force toward the desired opening using the force set by the *SetGripperForce* (page 336) command, and the “holding part” gripper status will be true (see *GetRtGripperState* (page 320)).

If the object is removed, the fingers will move to the desired fingers opening.

You can adjust the grip force with *SetGripperForce* (page 336) and control the gripper speed with *SetGripperVel* (page 339).

Note

The gripper commands *GripperOpen* (page 331), *GripperClose* (page 329), and *MoveGripper* (page 333) are queued like other motion commands but, their execution runs in parallel with robot motion:

- They start at the beginning of the blending (or deceleration) phase of the preceding motion command in the queue.
- They continue executing in parallel with any subsequent motion commands, which begin immediately as the gripper command starts.

To coordinate with motion commands:

- To ensure a gripper command starts only after the robot has come to a complete stop, insert a *Delay* (page 137) or use *SetCheckpoint* (page 158) and wait for it before posting the gripper command.
- To ensure the robot remains stationary until the gripper command completes, insert a *Delay* (page 137) or use *GetRtGripperState* (page 320) to detect when the gripper is holding or has released the part.

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

If no gripper is connected, the robot will report the following:

- [1038][No gripper connected.]

Note

Unlike the *GripperOpen* (page 331) and *GripperClose* (page 329) commands, *MoveGripper* (page 333) is not available when the robot is activated in recovery mode (and not homed) as gripper's available range is detected during homing.

Cyclic protocols

In cyclic protocols, the *MoveGripper* (page 333) command is represented by *MotionCommandID* 32. See Section 5 for more details.

SetExtToolSim

This instantaneous command enables the emulation of one of our three EOAT (two electric grippers and a pneumatic module), in the case of the Meca500. The emulation mode is also automatically enabled or disabled with the [ActivateSim](#) (page 188) or [DeactivateSim](#) (page 192) commands. You can emulate any of our Meca500 EOAT, even if you have another of these three already installed on the Meca500.

Syntax

SetExtToolSim(m)

Arguments

- m: tool model, where:
 - 0 disables the simulation of an external tool,
 - 1 emulates the current external tool,
 - 10 emulates the MEGP 25E gripper,
 - 11 emulates the MEGP 25LS gripper,
 - 20 emulates the MPM500 pneumatic module.

Default values

By default, when m = 1 (current tool type) and no tool is connected, the MEGP 25E gripper is emulated.

Responses

- [2047][m]

Usage restrictions

This command can be executed in any robot state.

However, changing the simulated tool to a different type (ex: gripper vs pneumatic module) is only possible when the robot is deactivated.

Cyclic protocols

This command is not available in cyclic protocols.

SetGripperForce

This queued command limits the grip force of Mecademic grippers.

Syntax

SetGripperForce(p)

Arguments

- p: percentage of maximum grip force (~40 N), ranging from 5 to 100.

Default values

By default, the grip force limit is 50%.

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetGripperForce* (page 336) command is represented by *MotionCommandID* 20. See Section 5 for more details.

SetGripperRange

This queued command sets the closed and open states of the gripper and is used mainly to redefine the actions of the *GripperClose* (page 329) and *GripperOpen* (page 331) commands, respectively.

The *SetGripperRange* (page 337) command is useful for the MEGP 25LS gripper. For example, if you are manipulating parts that require fingers opening between 10 mm and 20 mm, but the allowable range of the gripper as detected during the homing is 48 mm, it would be more efficient to redefine the actions of the *GripperClose* (page 329) and *GripperOpen* (page 331) commands by calling *SetGripperRange(8,22)* (page 337), or else the fingers will move more than necessary, increasing your cycle time.

The *SetGripperRange* (page 337) command does not limit the accessible range of the gripper, in contrast to the *SetJointLimits* (page 206) command, which limits the range of a joint. For example, if during homing, the robot detected that the range for the finger opening was [0, 15], and then you sent *SetGripperRange(8,13)* (page 337), you can still open the gripper more with *MoveGripper(14)* (page 333). However, using the commands *GripperOpen* (page 331) and *GripperClose* (page 329) will be equivalent to using the commands *MoveGripper(8)* (page 333) and *MoveGripper(13)* (page 333), respectively. Furthermore, when the fingers opening is 8 mm (or less) or 13 mm (or more), the state of the gripper will be “gripper open” or “gripper close”, respectively (see *GetRtGripperState* (page 320)).

Syntax

SetGripperRange(d_{closed}, d_{open})

Arguments

- d_{closed}: fingers opening that should correspond to closed state, in mm;
- d_{open}: fingers opening that should correspond to open state, in mm.

Default values

By default, the gripper closed and open states are those detected during the homing of the gripper, i.e., d_{closed} = 0 and d_{open} ≤ 6, in the case of the MEGP 25E gripper, or d_{open} ≤ 48, in the case of the MEGP 25LS gripper. To go back to these default values, use *SetGripperRange(0,0)* (page 337).

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 366), 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.]

- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetGripperRange* (page 337) command is represented by *MotionCommandID* 31. See Section 5 for more details.

SetGripperVel

This queued command limits the velocity of the gripper fingers (with respect to the gripper).

Syntax

SetGripperVel(p)

Arguments

- p: percentage of maximum finger velocity (~50 mm/s), ranging from 5 to 100.[†]

[†] If the gripper force is set to 100% using the *SetGripperForce* (page 336) command, it is possible to exceptionally increase the argument p up to 200. However, it should be noted that doing so will result in reduced accuracy of the force control on the gripper fingers.

Default values

By default, the finger velocity limit is 40%.

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetGripperVel* (page 339) command is represented by *MotionCommandID* 19. See Section 5 for more details.

SetIoSim

This command is available only on the MCS500 robot.

SetOutputState

This command is available only on the MCS500 robot.

SetOutputState_Immediate

This command is available only on the MCS500 robot.

SetVacuumPurgeDuration

This command is available only on the MCS500 robot.

SetVacuumPurgeDuration_Immediate

This command is available only on the MCS500 robot.

SetVacuumThreshold

This command is available only on the MCS500 robot.

SetVacuumThreshold_Immediate

This command is available only on the MCS500 robot.

SetValveState

This queued command is used to control each of the two valves in the MPM500 pneumatic module independently.

Syntax

SetValveState(v₁, v₂)

Arguments

- v₁: open (1), close (0), or keep unchanged (-1 or *) valve 1;
- v₂: open (1), close (0), or keep unchanged (-1 or *) valve 2.

Default values

Both valves are closed by default, i.e., at power-up, and are automatically closed when the robot is deactivated.

Further details

Since the MPM500 is often used with pneumatic grippers, you can also use the command *GripperOpen* (page 331) instead of *SetValveState(1,0)* (page 347), and *GripperClose* (page 329) instead of *SetValveState(0,1)* (page 347). However, note that these commands do not have the same effect on blending (see Section 3).

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 366), 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.]
- [1006][The robot is not homed.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetValveState* (page 347) command is represented by *MotionCommandID* 30. See Section 5 for more details.

VacuumGrip

This command is available only on the MCS500 robot.

VacuumGrip_Immediate

This command is available only on the MCS500 robot.

VacuumRelease

This command is available only on the MCS500 robot.

VacuumRelease

This command is available only on the MCS500 robot.

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 358)
- [DeleteVariable](#) (page 360)
- [GetVariable](#) (page 360)
- [ListVariables](#) (page 361)
- [SetVariable](#) (page 362)

The following provides a summary of important details regarding variables in the robot:

- **Persistence** Variables are saved on the robot and persist after rebooting.
- **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 358). Command [SetVariable](#) (page 362) 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 [The configuration menu](#) of the MecaPortal operating manual.

Managing variables in robot programs (TCP API)

In robot programs, you can use [CreateVariable](#) (page 358), [DeleteVariable](#) (page 360) or [SetVariable](#) (page 362) commands to manage your variables.

We suggest to create a program responsible for variables creation (using [CreateVariable](#) (page 358)) 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 176)

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 358)) 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.

Reading a variable

All variables with an assigned cyclic ID are reported in the cyclic dynamic data, using their cyclic ID as the *DynamicTypeID*. See [Dynamic data configuration](#) (page 71) for details on reading dynamic data —including variables— via 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 150));
- 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.

⚠ Warning

Although the Meca500 robot's EtherCAT stack supports setting variables, it does not support referencing variables as arguments for motion commands. As an alternative, you can create a robot program that uses the variable and call this program from EtherCAT (see [StartProgram](#) (page 220), called using command ID 100, see [Table 8](#)).

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 150) 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 150) command as follows:

- Set motion command with ID 2 ([*MovePose*](#) (page 150))
- 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 150) 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 150)

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 352).

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 358) 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 358) 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 358) will fail with error [1552].
- `override` (optional, 0 by default):
 - Specifies how [CreateVariable](#) (page 358) 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 356).

DeleteVariable (beta feature)

This command deletes a variable from the robot.

For more information, see [Commands for managing variables \(beta\)](#) (page 352).

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 356).

GetVariable (beta feature)

This command returns the value of a robot variable.

For more information, see [Commands for managing variables \(beta\)](#) (page 352).

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 356).

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 352).

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 356).

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 358) command.

For more information, see [Commands for managing variables \(beta\)](#) (page 352).

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 362) 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 356).

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 six-dimensional space defined by the position (x, y, z) and orientation (α , β , 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 268), [GetBlending](#) (page 232), [GetJointVel](#) (page 246)). 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 155)). 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).

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 204).

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 205).

error mode: The robot goes into error mode when it encounters an error while executing a command or a hardware problem (see [Table 1](#)).

Euler angles: A set of three angles, $\{\alpha, \beta, \gamma\}$, used to define an orientation in space. We use the mobile (intrinsic) XYZ convention. See [Section 3](#) of the Programming manual for more details.

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 specific joint.

joint set: The set of all joint positions.

joint space: The six-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 137), [MoveJoints](#) (page 138), [SetTRF](#) (page 182), [SetBlending](#) (page 154)). 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 216)), 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 set of two-value (-1 or 1) parameters c_s , c_e , and c_w that normally defines each of the eight possible robot postures for a given pose of the robot's end-effector.

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 center of the robot's wrist.

real-time data request commands: Commands used to request some real-time data regarding the current status of robot (e.g., [GetRtTrf](#) (page 290), [GetRtCartPos](#) (page 276), [GetStatusRobot](#) (page 294)).

robot control commands: Commands used to immediately control the robot, (e.g., [ActivateRobot](#) (page 187), [PauseMotion](#) (page 198), [SetNetworkOptions](#) (page 209)). 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 and homed, or alternatively when [Recovery mode](#) (page 24) is enabled while the robot is activated but not homed.

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.g., 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 $(-180^\circ, 180^\circ]$.

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).

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.

wrist center: the point where the axes of joints 4, 5, and 6 intersect.