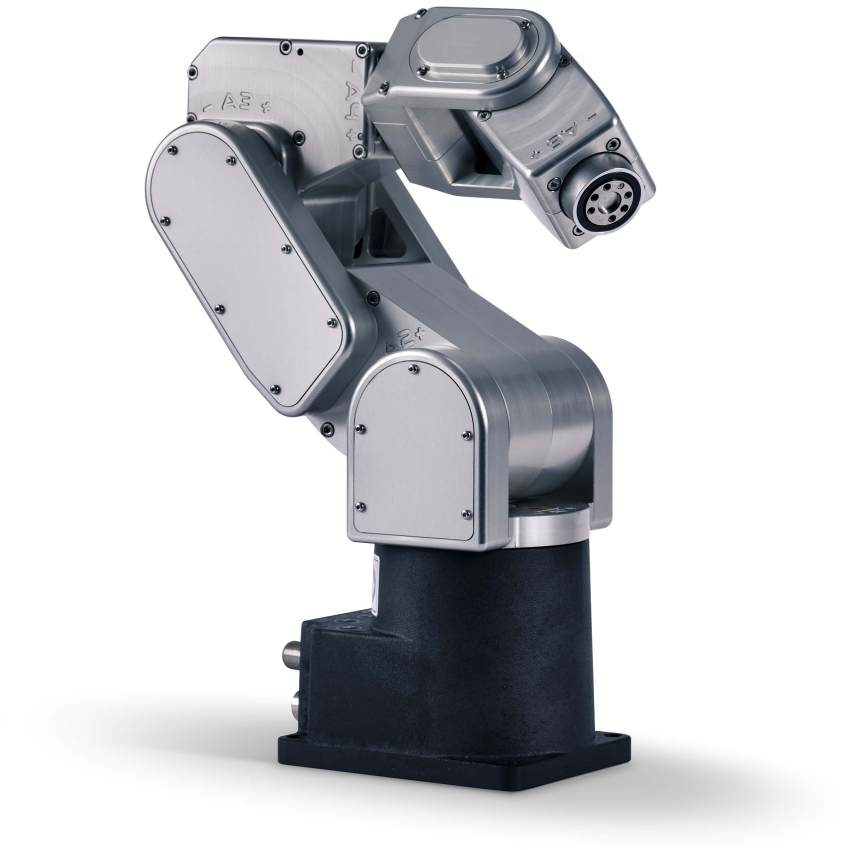
Programming Manual



Meca500 (R3)

Robot Firmware: 8.4

Document Revision: A

July 20, 2021

Contents

About this manual

This programming manual describes the key theoretical concepts related to industrial robots and the three methods that can be used for communicating with our robots from an Ethernet-enabled computing device (IPC, PLC, PC, Mac, Raspberry Pi, etc.): using either TCP/IP, EtherCAT or EtherNet/IP protocols. To maximize flexibility, instead of offering a proprietary robot programming language, we provide a set of robot-related instructions. You can therefore use any modern programming language that can run on your computing device. In addition, we currently offer two packages, one for ROS and another for Python, and we'll soon add a C++ API on our GitHub account.

The default communication method used by our robots is over TCP/IP and consists of a set of text-based motion and request commands to be sent to one of our robots, as well as a set of text messages sent back by the robot. Therefore, in the following chapter, we will refer to some of these motion commands in explaining the key theoretical concepts related to industrial robots. Furthermore, the communication methods based on the EtherCAT protocol and the EtherNet/IP protocols and described in Chapters 4 and 5, respectively, are essentially translations of our TCP/IP method. Thus, in these two chapters, we do not describe again each concept but simply refer to Chapter 2.

In other words, even if you intend to use EtherCAT or EtherNet/IP only, you must read every single page of this manual. However, before reading this programming manual, you must first read the User Manual for Meca500 R3 (FW8.4), available on our web site.

1 Basic theory and definitions

At Mecademic, we are particularly attentive to and concerned about technical accuracy, detail, and consistency, and use terminology that is not always standard. You must, therefore, read this section very carefully, even if you have prior experience with robot arms.

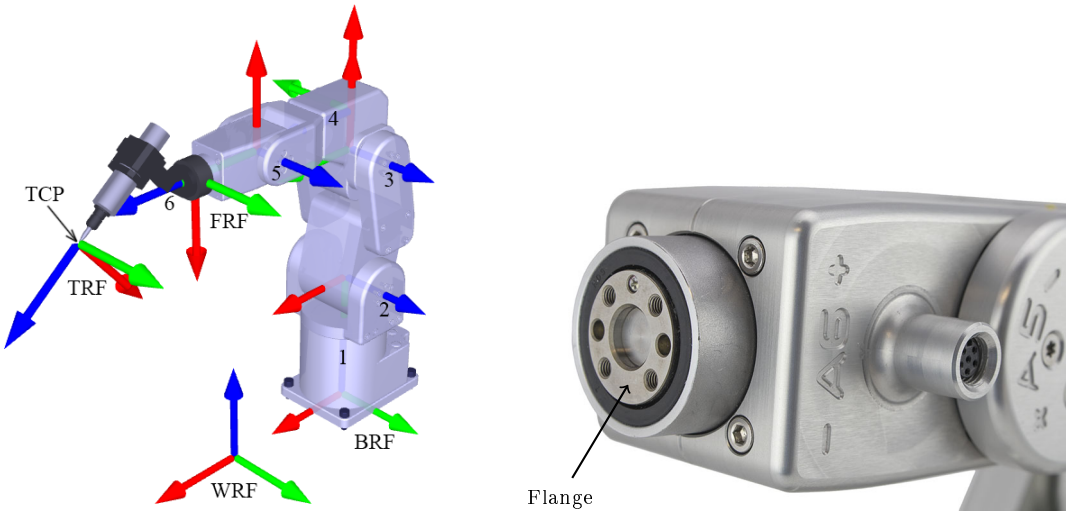
1.1 Definitions and conventions

1.1.1 Units

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

1.1.2 Joint numbering

The joints of the Meca500 are numbered in ascending order, starting from the base (Fig. 1a).



(a) robot with all joints at zero degrees (b) robot's flange with joint 6 at zero degrees

Figure 1: (a) Joint numbering and reference frames for the Meca500 and (b) its flange

1.1.3 Reference frames

At Mecademic, we use right-handed Cartesian coordinate systems (*reference frames*). The reference frames (according to the original Denavit and Hartenberg convention) that we use are shown in Fig. 1a, but you only need to be familiar with four of them. (The x axes are in red, the y axes are in green, and the z axes are in blue.) These four reference frames and the key terms related to them are:

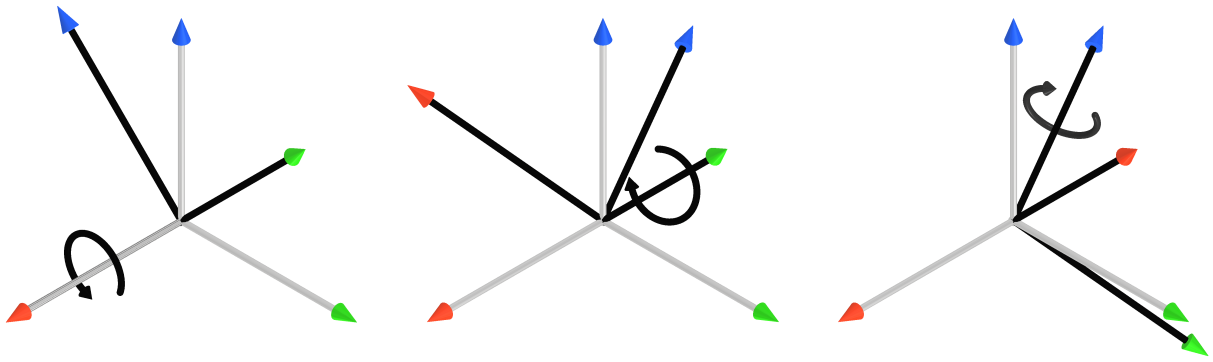
- *BRF*: *base reference frame*. Static reference frame fixed to the robot base. Its z axis coincides with the axis of joint 1 and points upwards, its origin lies on the bottom of the robot base, and its x axis is normal to the base front edge and points forward.
- *WRF*: *world reference frame*. The main static reference frame. By default, it coincides with the BRF. It can be defined with respect to the BRF with the SetWRF command.
- *FRF*: *flange reference frame*. Mobile reference frame fixed to the robot's *flange* (Fig. 1b). Its z axis coincides with the axis of joint 6, and points outwards. Its origin lies on the surface of the robot's flange. Finally, when all joints are at zero, the y axis of the FRF has the same direction as the y axis of the BRF.
- *TRF*: *tool reference frame*. 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 command.
- *TCP*: *tool center point*. Origin of the TRF. (Not to be confused with the Transmission Control Protocol acronym, which is also used in this document.)

1.1.4 Pose and Euler angles

Some of Meca500's commands take a *pose* (position and orientation of one reference frame with respect to another) as an input. In these commands, and in Meca500's web interface, a pose consists of a Cartesian position, $\{x, y, z\}$, and an orientation specified in *Euler angles*, $\{\alpha, \beta, \gamma\}$, according to the mobile XYZ convention (also referred to as $R_x R_y R_z$). 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 degrees, then about its y axis by β degrees, and finally about its z axis by γ degrees, the final orientation of frame F_m will be the same as that of frame F_1.

An example of specifying orientation using the mobile XYZ Euler angle convention is shown in Fig. 2. In the third image of this figure, the orientation of the black reference frame with respect to the gray reference frame is represented with the Euler angles $\{45°, -60°, 90°\}$.

It is crucial to understand that there are infinitely many Euler angles that correspond to a given orientation. For your convenience, the various motion commands that take a pose as arguments accept any numerical values for the three Euler angles (e.g., the set $\{378.34°, -567.32°, 745.03°\}$). However, we output only the equivalent Euler angle set $\{\alpha, \beta, \gamma\}$, for which $-180° \leq \alpha \leq 180°$, $-90° \leq \beta \leq 90°$ and $-180° \leq \gamma \leq 180°$. Furthermore, if you specify the Euler angles $\{\alpha, \pm 90°, \gamma\}$, the controller will always return an equivalent Euler angles set in which $\alpha = 0$. Thus, it is perfectly normal that the Euler angles that you have 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, on our web site).



(a) rotate 45° about the x axis (b) rotate $-60°$ about the new y axis (c) rotate 90° about the new z axis

Figure 2: The three consecutive rotations associated with the Euler angles $\{45°, -60°, 90°\}$

Finally, as we will see in Section 1.2.1, note that the pose of the end-effector alone does not define unequivocally the required joint angles.

1.1.5 Joint angles and joint 6 turn configuration

The angle associated with joint i ($i = 1, 2, ..., 6$), θ_i, will be referred to as *joint angle* i. Since joint 6 can rotate more than one revolution, you should think of a joint angle as a motor angle, rather than as the angle between two consecutive robot links.

A joint angle is measured about the z axis associated with the given joint using the right-hand rule. Note that the direction of rotation for each joint is engraved on the robot's body. All joint angles are zero in the robot shown in Fig. 1a. Note, however, that unless you attach an end-effector with cabling to the robot's flange, there is no way of telling the value of θ_6 just by observing the robot. For example, in Fig. 1b, θ_6 might as well be equal to 360°.

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

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

Joint 6 has no mechanical limits, but its software limits are ±100 turns. Finally, let us define the integer c_t as the axis 6 *turn configuration*, such that $-180° + c_t 360° < \theta_6 \leq 180° + c_t 360°$.

Note that you can further constrain the joint limits with the command SetJointLimits.

1.1.6 Joint set and robot posture

As we will explain later, for a desired location of the robot end-effector with respect to the robot base, there are several possible solutions for the values of the joint angles, i.e., several possible sets $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$. Thus, the simplest way to describe how the robot is postured, is by giving its set of joint angles. We will refer to this set at the *joint set*, and occasionally as *joint position*.

For example, in Fig. 1a, the joint set is $\{0°, 0°, 0°, 0°, 0°, 0°\}$, although, it could have been $\{0°, 0°, 0°, 0°, 0°, 360°\}$, and you wouldn't be able to tell the difference from the outside.

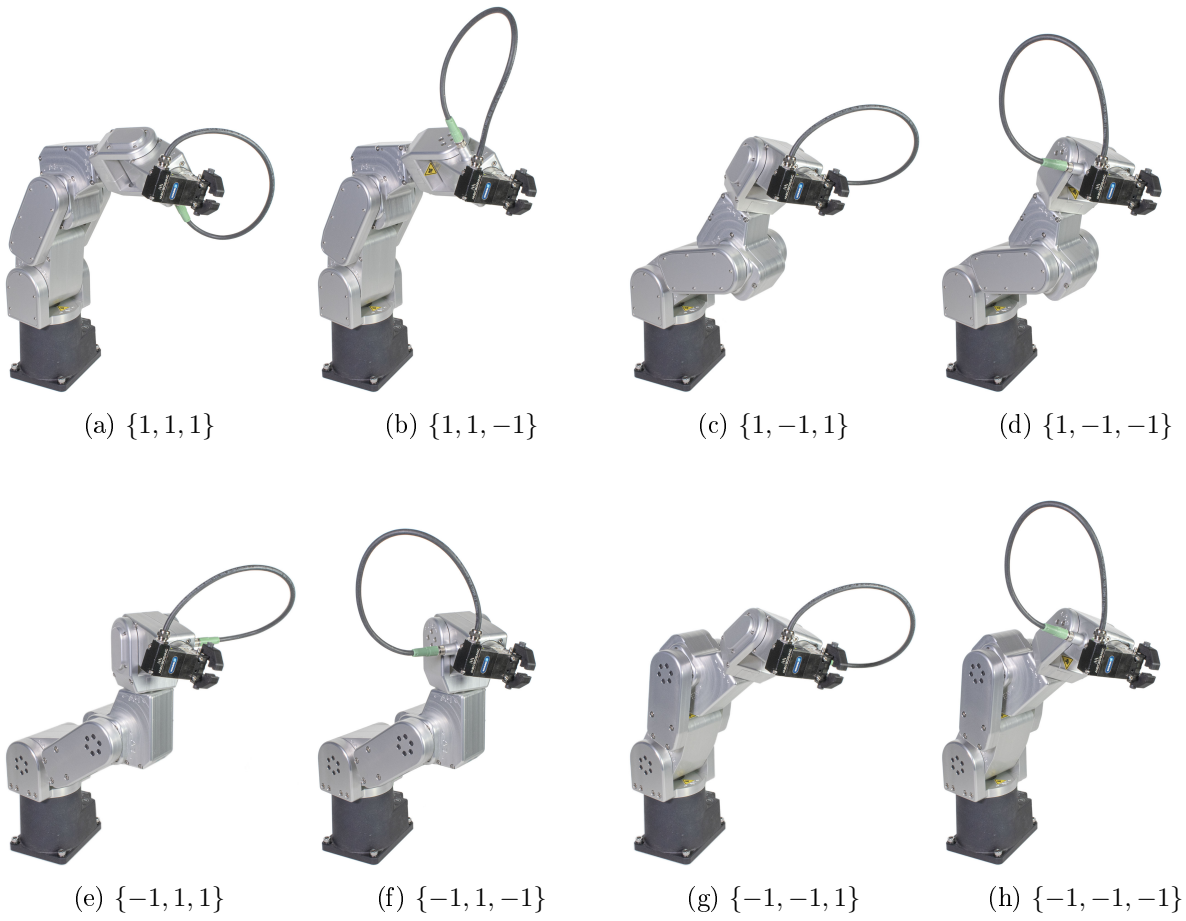
A joint set defines completely the relative poses, i.e., the "arrangement," of the seven robot links (a six-axis robot arm is typically composed of a series of seven links, starting with the base and ending with the end-effector). We will call this arrangement the *robot posture*. Thus, the joint sets $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$ and $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6 + c_t 360°\}$, where $-180° < \theta_6 \leq 180°$ and c_t is the axis 6 turn configuration, correspond to the same robot posture. Therefore, a joint set has the same information as a robot posture AND an axis 6 turn configuration.

1.2 Configurations, singularities and workspace

1.2.1 Inverse kinematic solutions and configuration parameters

Like virtually all six-axis industrial robot arms available on the market, Meca500's inverse kinematics generally provide up to eight feasible robot postures for a desired pose of the TRF with respect to the WRF (Fig. 3), and many more joint sets (since if θ_6 is a solution, then $\theta_6 \pm n360°$, where n is an integer, is also a solution). Each of these solutions is associated with one of eight *robot posture configurations*, defined by three parameters: 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* (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 Fig. 4a). The condition $c_s = 1$ is often referred to as "front".
  - $c_s = -1$, if the wrist center is on the "back" side of this plane (see Fig. 4c).
- c_e (elbow configuration parameter):
  - $c_e = 1$, if $\theta_3 > -\arctan(60/19) \approx -72.43°$ ("elbow up" condition, see Fig. 4d);
  - $c_e = -1$, if $\theta_3 < -\arctan(60/19) \approx -72.43°$ ("elbow down" condition, see Fig. 4f).
- c_w (wrist configuration parameter):
  - $c_w = 1$, if $\theta_5 > 0°$ ("no flip" condition, see Fig. 4g);
  - $c_w = -1$, if $\theta_5 < 0°$ ("flip" condition, see Fig. 4i).



(a) $\{1, 1, 1\}$ (b) $\{1, 1, -1\}$ (c) $\{1, -1, 1\}$ (d) $\{1, -1, -1\}$

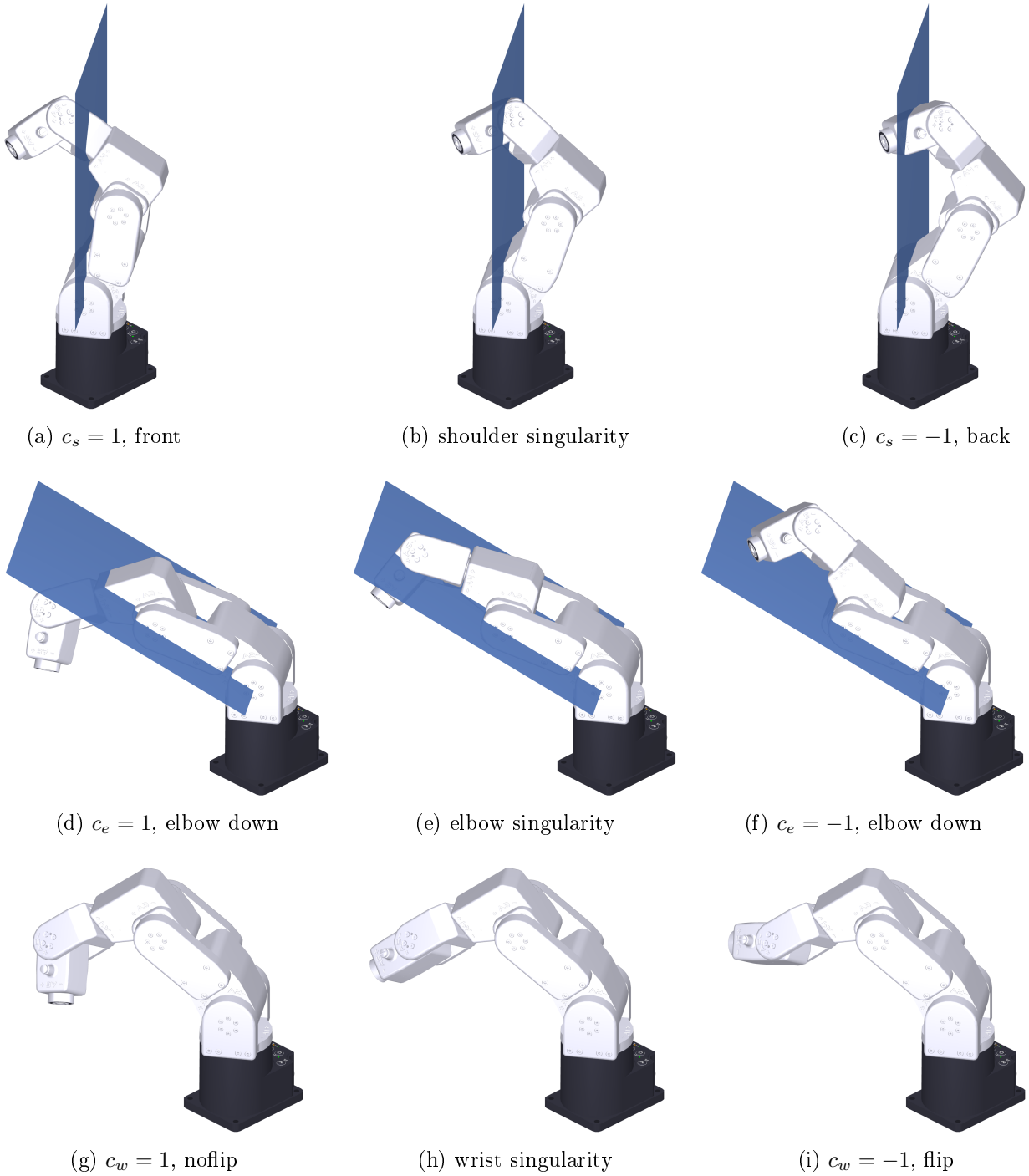
(e) $\{-1, 1, 1\}$ (f) $\{-1, 1, -1\}$ (g) $\{-1, -1, 1\}$ (h) $\{-1, -1, -1\}$

Figure 3: An example showing all eight possible robot postures, described by the robot posture configuration parameters $\{c_s, c_e, c_w\}$, for the pose {77 mm, 210 mm, 300 mm, −103°, 36°, 175°} of the FRF with respect to the BRF

Figure 4 shows an example of each robot posture configuration parameter, as well as of the limit conditions, which are called *singularities*. 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.

When we solve the inverse kinematics, we select the solution that corresponds to your desired robot posture configuration, $\{c_s, c_e, c_w\}$, defined by the command SetConf, but we also have to select the value for θ_6 that corresponds to your desired turn configuration, c_t (an integer in the range ±100), defined by the command SetConfTurn. 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



(a) $c_s = 1$, front (b) shoulder singularity (c) $c_s = -1$, back

(d) $c_e = 1$, elbow down (e) elbow singularity (f) $c_e = -1$, elbow down

(g) $c_w = 1$, noflip (h) wrist singularity (i) $c_w = -1$, flip

Figure 4: Inverse kinematic configuration parameters and the three types of singularities

turn and robot posture configuration parameters, the main being the fact that the change of turn does not involve singularities. This is why we use different commands (SetConf and SetConfTurn, SetAutoConf and SetAutoConfTurn, etc.).

While we do offer you the possibility of calculating the optimal inverse kinematic solution (commands SetAutoConf and SetAutoConfTurn), we highly recommend that you always specify the desired values for the configurations parameters (with the commands SetConf and SetConfTurn) for every Cartesian-coordinates motion command (i.e., MovePose and the various MoveLin* commands), at least when programming your robot in *online mode*.

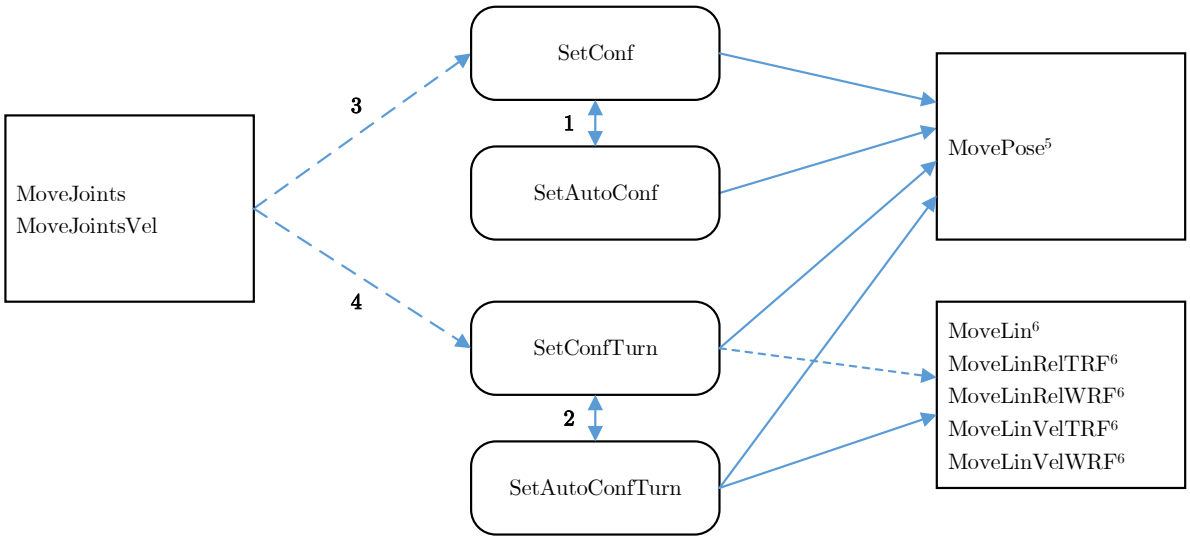
Thus, if you are teaching the *robot position* and want that later its end-effector moves to the current pose along a linear path, you need to record not only the current pose of the TRF w.r.t. the WRF (by retrieving it with GetRtCartPos), but also the definitions of the TRF and the WRF (with GetTRF and GetWRF), and finally the corresponding configuration parameters (with GetRtConf and GetRtConfTurn). Then, when you later want to approach this robot position with MoveLin from a starting robot position, you need to make sure the robot is already in the same robot posture configuration and that θ_6 is no more than half a revolution away from the desired value. If, however, you do not need the robot's TCP to follow a linear trajectory, then you should better get the current joint values only (using GetRtJointPos) and later go to that robot position using the command MoveJoints, thus not having to record and then specify the configuration parameters.

1.2.2 Automatic configuration selection

Using the automatic configuration selection should only be used once you understand how this selection is done, and mainly in *off-line mode*. In other words, the automatic configuration selection should be enabled only if the desired pose of the TRF with respect to the WRF (commands MovePose or MoveLin) or with respect to the current pose of the TRF (commands MoveLinRelTRF and MoveLinRelWRF) was calculated.

Figure 5 illustrates how the automatic and manual configuration selections work. Firstly (see notes 1 and 2), setting a desired robot posture configuration (SetConf) disables the automatic robot posture configuration selection, which is set by default. Inversely, enabling the automatic robot posture configuration selection, with SetAutoConf(1), removes the desired robot posture configuration, if such was set. Similarly, setting a desired turn configuration (SetConfTurn) disables the automatic turn configuration selection. Inversely, enabling the automatic turn configuration selection, with SetAutoConfTurn(1), removes the desired turn configuration, if such was set.

Secondly (see notes 3 and 4), even if you have set desired configuration parameters or enabled the automatic configuration selection, you can always use the MoveJoints and MoveJointsVel commands and these may override your choice. Indeed, once you have moved the robot with any of these two commands, the desired robot posture configuration is automatically set to the one corresponding to the resulting robot position. Similarly, the desired



Notes:

1. SetConf(c_s,c_e,c_w) disables the AutoConf setting, while SetAutoConf(1) disables the desired robot posture configuration setting.
2. SetConfTurn(c_t) disables the AutoConfTurn setting, while SetAutoConfTurn(1) disables the desired turn setting.
3. Both MoveJoints and MoveJointsVel update the desired robot posture configuration with the one corresponding to the robot position after the execution of any of these two commands, unless AutoConf was already on.
4. Both MoveJoints and MoveJointsVel update the desired turn configuration with the one corresponding to the value of θ_6 after the execution of any of these two commands, unless AutoConfTurn was already on.
5. Joint 6 (or even joints 1 and 4) can rotate more than 180° with a single MovePose command.
6. No joint can rotate more than 180° with a single MoveLin* command, so SetConfTurn is only validated.

Figure 5: The relationship between the inverse kinematics configuration parameters and the different motion commands

turn configuration is automatically set to the one corresponding to the resulting robot position, but only if automatic turn selection was not enabled.

Enabling automatic robot posture configuration selection or specifying a desired robot posture configuration has effect only on the command MovePose. Indeed, the robot cannot change its robot posture configuration during a MoveLin*, since it cannot cross a singularity. An automatic robot posture configuration selection allows the robot controller to select the robot position (among many possible), corresponding to the desired pose, that is fastest to reach. In contrast, the selection of the turn configuration has effect on both the MovePose and MoveLin* commands. If automatic turn selection is enabled, the robot will always rotate joint 6 at most 180°.

Note that it could be perfectly logical to set a desired robot posture configuration, but enable the automatic turn configuration. For example, if your end-effector is Schunk's Adheso cableless gripping system, there is no reason to prefer one turn over another (e.g., to prefer $\theta_6 = 17°$ over $\theta_6 = 377°$).

Finally, there is no limit on the required joint rotations (note 5 in Fig. 5) when using the MovePose command (as long as these rotations are withing the joint limits, of course). In contrast, no joint can rotate more than 180° with a single MoveLin* command (note 6). Thus, specifying the desired turn configuration does not does guarantee that the robot will execute the motion command. For example, if $\theta_6 = -1°$ in the starting robot position, then you sent the command SetTurn(1), and then the command MoveLin, the robot will report an error, whatever the desired pose, because joint 6 would have to rotate more than 180°, which, while physically possible, has been disabled by us.

1.2.3 Workspace and singularities

Many users mistakenly oversimplify the workspace of a six-axis robot arm as a sphere of radius equal to the *reach* of the robot (the maximum distance between the axis of joint 1 and the center of the robot's wrist). The truth is that the Cartesian *workspace* of a six-axis industrial robot is a six-dimensional entity: the set of all attainable end-effector poses (see our tutorial on workspace, available on our web site). Therefore, the workspace of a robot depends on the choice of TRF. Worse yet, as we saw in the preceding section, for a given end-effector pose, we can generally have eight different robot postures (Fig. 3). Thus, the Cartesian workspace of a six-axis robot arm is the combination of eight workspace subsets, one for each the eight robot posture configurations. These eight workspace subsets have common parts, but there are also parts that belong to only one subset (i.e, there are end-effector poses accessible with only one configuration, because of joint limits). Therefore, in order to make optimal use of all attainable end-effector poses, the robot must often pass from one subset to the other. These passages involve so-called *singularities* and are problematic when the robot's end-effector is to follow a specific Cartesian path.

Any six-axis industrial robot arm has singularities (see our tutorial on singularities, available on our web site). However, the advantage of robot arms like the Meca500, where the axes of the last three joints intersect at one point (the center of the robot's wrist), is that these singularities are very easy to describe geometrically (see Fig. 4). In other words, it is very easy to know whether a robot posture is close to singularity in the case of the Meca500.

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).

For example, consider the Meca500 at its zero robot posture (Fig. 1a). 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. Thus, singularities are not some kind of purely mathematical problem. They represent actual physical limits. That said, because of the way a robot controller is programmed, at a singular robot posture (or at a robot posture that is very close to a singularity), the robot cannot be moved in any direction using a Cartesian-space motion command (MoveLin, MoveLinRelTRF, MoveLinRelWRF, MoveLinVelTRF, or MoveLinVelWRF).

There are three types of singular robots positions, and these correspond to the conditions under which the configuration parameters c_1, c_3 and c_5 are not defined. The most common singular robot posture is called *wrist singularity* and occurs when $\theta_5 = 0°$ (Fig. 4h). 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 very frequently. The second type of singularity is called *elbow singularity* (Fig. 4e). 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 $\theta_3 = -\arctan(60/19) \approx -72.43°$. 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 *shoulder singularity* (Fig. 4b). 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.

As already mentioned, you can never pass through a singularity using a Cartesian-space motion command. However, you will have no problem with singularities when using the command MoveJoints, and to a certain extent, the command MovePose. Finally, you need to know that when using a Cartesian-space command, problems occur not only when crossing a singularity, but also when passing too close to a singularity. When passing close to a wrist or shoulder singularity, some joints will move very fast (i.e., 4 and 6, in the case of a wrist singularity, and 1, 4 and 6, in the case of a shoulder singularity), even though the TCP speed is very low. Thus, you must avoid moving in the vicinity of singularities when using Cartesian-space motion commands.

1.3 Key concepts related to Mecademic robots

1.3.1 Homing

At power-up, the Meca500 knows the approximate angle of each of its joints, with a couple of degrees of uncertainty. To find the exact joint angles with high accuracy, each motor must make one full revolution. This motion is the essential part of a procedure called *homing*.

During homing, all joints rotate simultaneously. Each of joints 1, 2 and 3 rotates 3.6°, joints 4 and 5 rotate 7.2° each, and joint 6 rotates 12°. Then, all joints rotate back to their initial angles. The whole sequence lasts three seconds. Make sure there is nothing that restricts the above-mentioned joint movements, or else the homing process will fail. Homing will also fail if any of the robot joints are outside its user-defined limits (SetJointLimits).

Finally, if your robot is equipped with Mecademic's gripper (MEGP 25E), the robot controller will automatically detect it, and the homing procedure will end with a homing of the gripper. The gripper will fully open, then fully close. Make sure there is nothing that restricts the full 6-mm range of motion of the gripper, while the latter is being homed.

NOTICE

The range of the absolute encoder of joint 6 is only ±420°. Therefore, you must always rotate joint 6 within that range before deactivating the robot. Failure to do so may lead to an offset of ±120° in joint 6. If this happens, unpower the robot and disconnect your tooling. Then, power up the robot, activate it, home it, and zero joint 6. If the screw on the robot's flange is not as in Fig. 1b, then rotate joint 6 to +720°, and deactivate the robot. Next, reactivate it, home it and zero joint 6 again. Repeat one more time if the problem is not solved.

1.3.2 Blending

All multi-purpose industrial robots function in a similar manner when it comes to moving around in *position mode*. You either ask the robot to move its end-effector to a certain pose, with a *Cartesian-space* command, or its joints to a certain joint set, with a *joint-space* command. When your target is a joint set, you have no control over the path that the robot's end-effector will follow. When the target is a pose, you can either leave it to the robot to choose the path or require that the TCP follows a linear path. Thus, if you need to follow a complex curve (as in a gluing application), you need to decompose your curve into multiple linear segments. Then, instead of having the robot stop at the end of each segment and make a sharp change in direction, you can blend these segments using what we call *blending*. You can think of blending as the action of taking a rounded shortcut.

Blending allows the trajectory planner to keep the velocity of the robot's end-effector as smoothly changing as possible between two position-mode joint-space movements (MoveJoints, MovePose) or two position-mode Cartesian-space movements (MoveLin, MoveLinRelWRF, MoveLinRelTRF). When blending is activated, the trajectory planner will transition between the two paths using a blended curve (Fig. 6). The higher the TCP speed, the more rounded the transition will be. You cannot control directly the radius of the blending. Also,

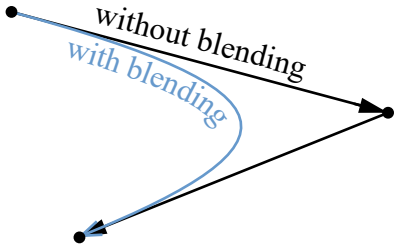


Figure 6: TCP path for two consecutive linear movements, with and without blending

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

1.3.3 Position and velocity modes

As already discussed in Section 1.3.2 on blending, the conventional way of having an industrial robot move is by requesting that its end-effector moves to a desired pose or that its joints rotate to a desired joint set. This basic control method is called *position mode*. If, in addition to specifying a desired pose for the robot's end-effector, you wish that the robot's TCP follows a linear path, then you must use the Cartesian-space motion commands MoveLin, MoveLinRelTRF and MoveLinRelWRF. If you simply wish the robot's end-effector to get to a certain pose or the robot's joints to rotate to a certain joint set, then you should use the joint-space motion commands MovePose or MoveJoints, respectively.

In position mode, with Cartesian-space motion commands, you can specify the maximum linear and angular velocities and the maximum accelerations of the end-effector. However, you cannot set a limit on the joint velocities and 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 and with maximum acceleration. Similarly, with joint-space motion commands, you can only specify the maximum velocity and acceleration of the joints. However, it is impossible to limit neither the velocity nor the acceleration of the robot's end-effector. Figure 7 summarizes the possible settings for the velocity and acceleration in position mode.

We also offer a second method for controlling the Meca500, by defining either its end-effector velocity or its joint velocities. We call this alternative, robot control method the *velocity mode*. Velocity mode is aimed at advanced applications such as force control, dynamic path corrections, or telemanipulation. For example, the jogging feature in Meca500's web interface is implemented using the velocity-mode commands.

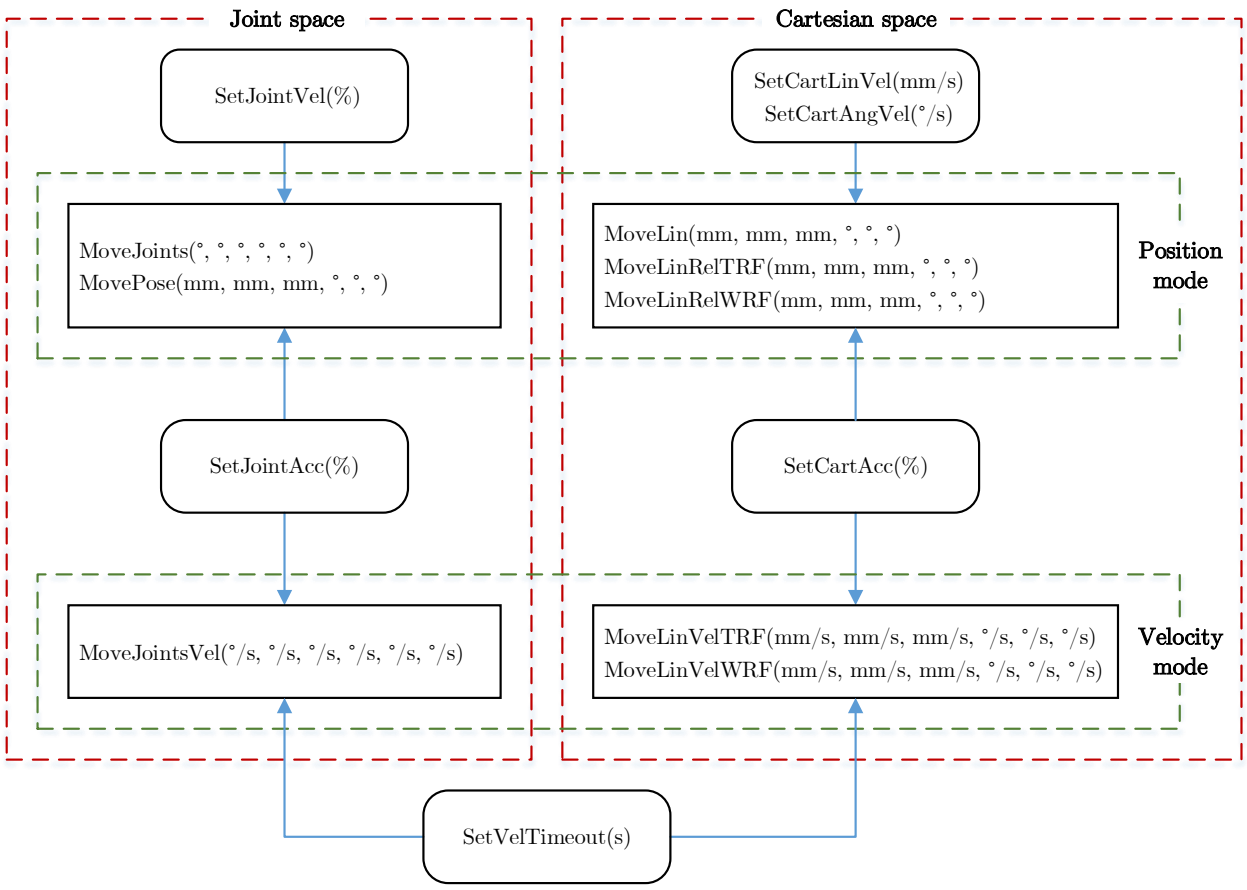


Figure 7: Settings that influence the robot motion in position and velocity modes

To control the robot in velocity mode, simply send one of the three velocity-mode motion commands: MoveJointsVel, MoveLinVelTRF or MoveLinVelWRF. Note, however, that the effect from the velocity-mode motion command will last no more than the time specified in the SetVelTimeout command or less. Normally, this timeout must be very small (its default value is 0.05 s, and its maximum value 1 s) and you should keep sending velocity-mode motion commands to the robot for as long as you wish to control it in velocity mode. Thus, you can simply send position-mode and velocity-mode motion commands 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 the commands SetCheckpoint, GripperOpen and GripperClose.

It is important to note, however, that there is a significant difference in the behavior of position- and velocity-mode motion commands. 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 will have to be reset. In velocity mode, if the robot runs into a singularity or a joint limit, it will simply stop without raising an error.

Finally, in velocity mode, you control directly the velocity of the robot's end-effector or the velocities of the robot joints. There is no command that can further limit these velocities (velocity override). Also, the command SetJointAcc limits the accelerations of the joints only for MoveJointsVel while the command SetCartAcc limits the acceleration of the end-effector only for MoveLinVelTRF and MoveLinVelWRF (see Figure 7).

2 Communicating over TCP/IP

To operate the Meca500, the robot must be connected to a computer or to a PLC over Ethernet. Commands may be sent through Mecademic's web interface or through a custom computer program using either the TCP/IP protocol, which is detailed in the remainder of this section, or EtherCAT, which is explained in the next section. In the case of TCP/IP, the Meca500 communicates using null-terminated ASCII strings. The robot default IP address is 192.168.0.100, and its default TCP port is 10000, referred to as the *control port*. Commands to the robot and messages from the robot are sent over the control port. Additionally, after homing, the robot will periodically send data over TCP port 10001, referred to as the *monitoring port*, at the rate specified by the SetMonitoringInterval command. This data includes the joint set and TRF pose, as well as other data but only when it changes, and finally other optional data enabled with the SetRealTimeMonitoring command (see Section 2.5.4).

When using the TCP/IP protocol, the Meca500 can interpret two types of instructions: *motion commands* and *request commands*. Every command must end with the ASCII NUL character (\0) or end-of-line character (\n). Commands are not case-sensitive.

In the description of some commands, we will refer to *default values*. These are essentially variables that are initialized every time the robot is activated. In contrast, certain parameter values are *persistants*. They do 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.

2.1 Motion commands

Motion commands are used to construct a trajectory for the robot. When the Meca500 receives a motion command, it places it in a *motion queue*. Generally, the command will be run once all preceding commands have been executed.

In the following subsections, the motion commands are presented in alphabetical order. Most motion commands have arguments, but not all of these arguments have default values (e.g., the argument for the command Delay). The arguments for most motion commands are IEEE-754 floating-point numbers, separated—if more than one—with commas and, optionally, spaces.

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

In addition, whenever the robot has stopped moving, it can send the "[3004][End of movement.]" (*EOM*) message, if this option is activated with SetEOM. For example, if

blending is enabled, and you send three MoveJoints commands, the robot will send an EOM message only after all three MoveJoints commands have been executed and the robot has come to a complete stop.

Furthermore, by default, the robot sends an "[3012][End of block.]" (*EOB*) message, every time the robot has stopped moving AND its motion queue has been emptied. The EOB message can be deactivated with SetEOB. Thus, for example, if both EOM and EOB messages are enabled, and you send immediately one after the other a MoveJoints, a SetTRF, a MovePose and a Delay command, as soon as the robot has stopped, it will send an EOM message, and then as soon as the delay has lapsed, an EOB message.

Once again, note that the EOB and EOM messages should not be used as means for detection that a sequence of motions commands has been executed. Because of communication delays (albeit extremely small) you may receive an EOB message while the robot has just started executing another motion command. Using the SetCheckpoint command is the best way to follow the sequence of execution of commands.

Finally, motion commands can generate errors, which are accompanied by various error messages, explained in Section 2.5.1.

2.1.1 Delay(t)

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 command and stops temporarily. (In contrast, the PauseMotion command interrupts the motion as soon as received by the robot.)

Arguments

- t: desired pause duration in seconds.

2.1.2 GripperOpen/GripperClose

These two commands are used to open or close Mecademic's optional MEGP 25E gripper. The gripper will move its fingers apart or closer until the grip force reaches 40 N. You can reduce this maximum grip force with the SetGripperForce command. In addition, you can control the speed of the gripper with the SetGripperVel command.

It is very important to understand that the GripperOpen and GripperClose commands have the same behavior as a robot motion command, being executed only after the preceding motion command has been completed. Currently, however, if a robot motion command is sent after the GripperOpen or GripperClose command, the robot will start executing the motion command without waiting for the gripper to finish its action. You must therefore send a Delay command after GripperOpen and GripperClose commands.

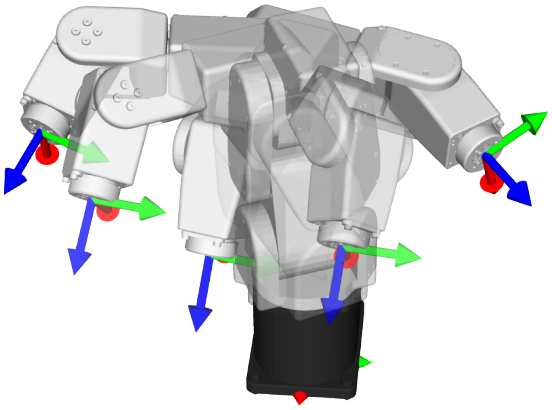


Figure 8: End-effector motion when using the MoveJoints or MovePose commands

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

This command makes the robot rotate simultaneously its joints to the specified joint set. All joint rotations start and stop at the same time. The path that the robot takes is linear in the joint space, but nonlinear in the Cartesian space. Therefore, the TCP trajectory is not easily predictable (Fig. 8). Finally, with MoveJoints, the robot can cross singularities.

Arguments

- θ_i: the (admissible) angle of joint i, where $i = 1, 2, ..., 6$, in degrees. The admissible default ranges for the joint angles are as follows:

  $-175° \leq \theta_1 \leq 175°$,
  $-70° \leq \theta_2 \leq 90°$,
  $-135° \leq \theta_3 \leq 70°$,
  $-170° \leq \theta_4 \leq 170°$,
  $-115° \leq \theta_5 \leq 115°$,
  $-36,000° \leq \theta_6 \leq 36,000°$.

  These ranges can be further limited with the command SetJointLimits.

2.1.4 MoveJointsVel($\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6$)

This command makes the robot rotate simultaneously its joints with the specified joint velocities. All joint rotations start and stop at the same time. The path that the robot takes is linear in the joint space, but nonlinear in the Cartesian space. Therefore, the TCP path is not easily predictable (Fig. 8). With MoveJointsVel, the robot can cross singularities without any problem.

Arguments

- θ_i: the velocity of joint i, where $i = 1, 2, ..., 6$, in $^\circ/s$. The admissible ranges for the joint velocities are as follows:

$$-150^\circ/\text{s} \leq \dot{\theta}_1 \leq 150^\circ/\text{s},$$
$$-150^\circ/\text{s} \leq \dot{\theta}_2 \leq 150^\circ/\text{s},$$
$$-180^\circ/\text{s} \leq \dot{\theta}_3 \leq 180^\circ/\text{s},$$
$$-300^\circ/\text{s} \leq \dot{\theta}_4 \leq 300^\circ/\text{s},$$
$$-300^\circ/\text{s} \leq \dot{\theta}_5 \leq 300^\circ/\text{s},$$
$$-500^\circ/\text{s} \leq \dot{\theta}_6 \leq 500^\circ/\text{s}.$$

Note that the robot will decelerate to a full stop after a period defined by the command SetVelTimeout, unless another MoveJointsVel command is sent. Also, bear in mind that the MoveJointsVel command, unlike position-mode motion commands, generates no motion errors when a joint limit is reached. The robot simply stops slightly before the limit.

2.1.5 MoveLin$(x, y, z, \alpha, \beta, \gamma)$

This command makes the robot move its end-effector, so that its TRF ends up at a desired pose with respect to the WRF while the TCP moves along a linear path in Cartesian space, as illustrated in Fig. 9. If the final (desired) orientation of the TRF is different from the initial orientation, the orientation will be modified along the path using LERP interpolation.

Using this command, the robot cannot move to or through a singular robot posture or one that is too close to being singular. With this command, the initial and final robot postures have to be in the same configuration, $\{c_s, c_e, c_w\}$. If the complete motion cannot be performed due to singularities or joint limits, it will normally not even start, and an error will be generated.

In contrast, this command accounts for the desired turn configuration. Thus, for example, if you enable the automatic selection of the turn configuration, then you execute a MoveLin command from a robot position where $\theta_6 = -92^\circ$ towards a pose for which $\theta_6 = 93^\circ \pm n360^\circ$ are possible solutions, the solution that will be chosen for the final robot position will be that for which $\theta_6 = -267^\circ$ (i.e., joint 6 will rotate only 175°).

If, however, a motion requires that a joint rotates more than 180°, it will not be executed. In the example above, if a desired turn configuration of 0 was set, the MoveLin command will be rejected, since the final value for θ_6 would have to be 93°, but that would require that joint 6 rotates more than 180°.

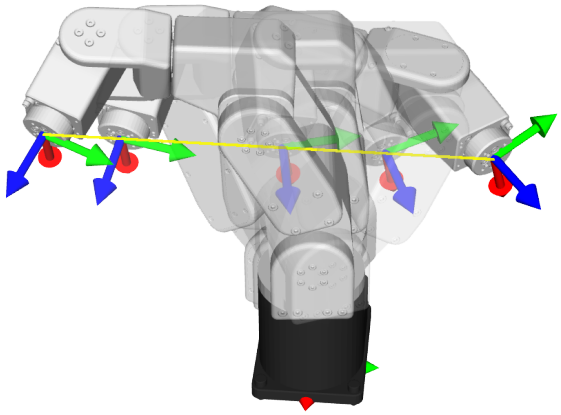


Figure 9: The TCP path when using the MoveLin command

Arguments

- x, y, and z: the coordinates of the origin of the TRF w.r.t. the WRF, in mm;
- α, β, and γ: the Euler angles representing the orientation of the TRF w.r.t. the WRF, in degrees.

2.1.6 MoveLinRelTRF$(x, y, z, \alpha, \beta, \gamma)$

This command is similar to the MoveLin 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, 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 command).

As with the MoveLin command, if the complete motion cannot be performed due to singularities or joint limits, it will normally not even start and an error will be generated. These joint limits include the desired turn configuration and the fact that no joint can rotate more than 180° (in a single linear displacement).

Arguments

- x, y, and z: the position coordinates, in mm;
- α, β, and γ: the Euler angles, in degrees.

2.1.7 MoveLinRelWRF$(x, y, z, \alpha, \beta, \gamma)$

This command is similar to the MoveLinRelTRF 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.

As with the MoveLin command, if the complete motion cannot be performed due to singularities or joint limits, it will normally not even start and an error will be generated. These joint limits include the desired turn configuration and the fact that no joint can rotate more than 180° (in a single linear displacement).

Arguments

- x, y, and z: the position coordinates, in mm;
- α, β, and γ: the Euler angles, in degrees,

2.1.8 MoveLinVelTRF($\dot{x}, \dot{y}, \dot{z}, \omega_x, \omega_y, \omega_z$)

This command makes the robot move its TRF with the specified Cartesian velocity, defined with respect to the TRF.

Arguments

- $\dot{x}$, $\dot{y}$, and $\dot{z}$: the components of the linear velocity of the TCP w.r.t. the TRF, in mm/s, ranging from −1000 mm/s to 1000 mm/s;
- ω_x, ω_y, ω_z: the components of the angular velocity of the TRF w.r.t. the TRF, in $°/s$, ranging from −300°/s to 300°/s.

Note that the robot will decelerate to a complete stop after a period of time defined by the SetVelTimeout command, unless another MoveLinVelTRF or a MoveLinVelWRF command is sent and, of course, unless a PauseMotion command is sent or some motion limit is encountered. Also, bear in mind that this command, unlike position-mode motion commands, generates no motion errors when a joint limit (including the desired turn configuration) or a singularity is reached. The robot simply stops slightly before the limit. If you want joint 6 to be able to change turns, you need to use execute SetAutoConfTurn(1) first.

2.1.9 MoveLinVelWRF($\dot{x}, \dot{y}, \dot{z}, \omega_x, \omega_y, \omega_z$)

This command makes the robot move its TRF with the specified Cartesian velocity, defined with respect to the WRF.

Arguments

- $\dot{x}$, $\dot{y}$, and $\dot{z}$: the components of the linear velocity of the TCP w.r.t. the WRF, in mm/s, ranging from −1000 mm/s to 1000 mm/s;
- ω_x, ω_y, ω_z: the components of the angular velocity of the TRF w.r.t. the WRF, in $°/s$, ranging from −300°/s to 300°/s.

Note that the robot will decelerate to a complete stop after a period of time defined by the SetVelTimeout command, unless another MoveLinVelWRF or a MoveLinVelTRF command

is sent and, of course, unless a PauseMotion command is sent or some motion limit is encountered. Also, bear in mind that this command, unlike position-mode motion commands, generates no motion errors when a joint limit (including the desired turn configuration) or a singularity is reached. The robot simply stops slightly before the limit. If you want joint 6 to be able to change turns, you need to use execute SetAutoConfTurn(1) first.

2.1.10 MovePose($x, y, z, \alpha, \beta, \gamma$)

This command makes the robot move its TRF to a specific pose with respect to the WRF. Essentially, the robot controller calculates all possible joint sets corresponding to the desired pose, except those corresponding to a singular robot posture. Then, it either chooses the joint set that corresponds to the desired robot posture and turn configurations or the one that is fastest to reach. Finally, it executes internally a MoveJoints command with the chosen joint set.

Thus, all joint rotations start and stop at the same time. The path the robot takes is linear in the joint-space, but nonlinear in Cartesian space. Therefore, the path the TCP will follow to its final destination is not easily predictable, as illustrated in Fig. 8.

Using this command, the robot can cross a singularity or start from a singular robot posture, as long as SetAutoConf is enabled. However, the robot cannot go to a singular robot posture using MovePose. For example, assuming that the TRF coincides with the FRF, you cannot execute the command MovePose(190,0,308,0,90,0), since this pose corresponds only to singular robot posture (e.g., the joint set {0,0,0,0,0,0}). You can only use the MovePose command with a pose that corresponds to at least one non-singular robot posture.

As with the MoveJoints command, if the complete motion cannot be performed due to joint limits, it will normally not even start, and an error will be generated.

Arguments

- x, y, and z: the coordinates of the origin of the TRF w.r.t. the WRF, in mm;
- α, β, and γ: the Euler angles representing the orientation of the TRF w.r.t. the WRF, in degrees.

2.1.11 SetAutoConf(e)

This command enables or disables the automatic robot posture configuration selection and has effect only on the MovePose command. This automatic selection, in conjunction with the turn configuration selection (recall Section 1.2.1), allows the controller to choose the "closest" joint set corresponding to the desired pose (the arguments of the MovePose command).

Arguments

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

Default values

SetAutoConf is enabled by default. If you disable it, the new desired inverse kinematic configuration will be the one corresponding to the current robot posture, i.e., the one after all preceding motion commands have been completed. Note, however, that if you disable the automatic robot configuration selection in a singular robot posture, the controller will automatically choose one of the two, four or eight boundary configurations. For example, if you execute SetAutoConf(0) 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 (provided, of course, that the command is free of syntax errors).

2.1.12 SetAutoConfTurn(e)

This command enables or disables the automatic turn selection for joint 6 (recall Section 1.2.1). It has effect on the MovePose command, as well as on all MoveLin* commands.

Arguments

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

Default values

SetAutoConfTurn is disabled by default (for legacy reasons). If you enable it, move the robot, then 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 (provided, of course, that the command is free of syntax errors).

2.1.13 SetBlending(p)

This command enables/disables the robot's blending feature (recall Section 1.3.2). Note that the commands MoveLin, MovePose, and MoveJoints will only send "[3004][End of movement.]" responses when the robot comes to a complete stop and the command SetEOM(1) was used. Therefore, enabling blending may suppress these responses.

Also, note that there is blending only between consecutive movements with the commands MoveJoints and MovePose, or between consecutive movements with the commands MoveLin,

MoveLinRelTRF and MoveLinRelTRF. In other words, there will never be blending between the trajectories of a MovePose command followed by a MoveLin command.

Arguments

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

Default values

Blending is enabled at 100% by default.

2.1.14 SetCartAcc(p)

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

Arguments

– p: percentage of maximum acceleration of the eTRF, ranging from 0.001% to 600%.

Default values

The default end-effector acceleration limit is 50%.

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.

2.1.15 SetCartAngVel(ω)

This command limits the angular velocity of the robot TRF with respect to its WRF. It only affects the movements generated by the MoveLin, MoveLinRelTRF and MoveLinRelWRF commands.

Arguments

– ω: TRF angular velocity limit, ranging from 0.001°/s to 300°/s.

Default values

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

2.1.16 SetCartLinVel(v)

This command limits the Cartesian linear velocity of the robot's TRF with respect to its WRF. It only affects the movements generated by the MoveLin, MoveLinRelTRF and MoveLinRelWRF commands.

Arguments

- v: TCP velocity limit, ranging from 0.001 mm/s to 1000 mm/s.

Default values

The default TCP velocity limit is 150 mm/s.

2.1.17 SetCheckpoint(n)

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, then other motion commands, you will be able to know the exact moment when the motion command sent just before the SetCheckpoint command was completed. At that precise moment, the robot will send you back the response [3030][n], where n is a positive integer number defined by you. If blending was activated, the checkpoint response will be sent somewhere along the blending. Finally, note that you can use the same checkpoint number multiple times.

Using a checkpoint is the only reliable way to know whether a particular motion sequence was completed. Do not rely on the EOM or EOB messages as they may be received well before the completion of a motion or a motion sequence (or, of course, not at all, if these messages were not enabled).

Arguments

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

Responses

[3030][n]

2.1.18 SetConf(c_s, c_e, c_w)

This command sets the desired robot posture configuration to be observed in the MovePose command.

The robot posture configuration (see Fig. 4) can be automatically selected by using the SetAutoConf command. Using SetConf automatically disables the automatic robot posture configuration selection.

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 robot posture configuration selection is enabled by default (see SetAutoConf), so when you start the robot, there is no default desired robot posture configuration.

2.1.19 SetConfTurn(c_t)

This command sets the desired turn configuration for joint 6, which can rotate ±100 revolutions. This command is useful only if you have a wired end-effector with sufficiently long cables to allow joint 6 to rotate more than ±180°. If, for example, you use our MEGP 25E gripper, you should simply limit joint 6 to ±180° with the command SetJointLimits and then use either SetAutoConfTurn(1) or SetAutoConf(0). If, instead, you use a cableless end-effector, then you should never disable the automatic turn configuration.

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 equation: $-180° + c_t 360° < \theta_6 \leq 180° + c_t 360°$.

Default values

For legacy reasons, automatic turn configuration selection is disabled by default (see the command SetAutoConfTurn), and when you start the robot, the default turn configuration is the one corresponding to the initial value of θ_6 (i.e., after homing).

2.1.20 SetGripperForce(p)

This command limits the grip force of the MEGP 25E gripper.

Arguments

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

Default values

By default, the grip force limit is 50%.

2.1.21 SetGripperVel(p)

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

Arguments

– p: percentage of maximum finger velocity (∼100 mm/s), ranging from 5% to 100%.

Default values

By default, the finger velocity limit is 50%.

2.1.22 SetJointAcc(p)

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

Arguments

– p: percentage of maximum acceleration of the joints, ranging from 0.001% to 150%.

Default values

The default joint acceleration limit is 100%.

Note that 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 from firmware 7 and you want to keep the same joint accelerations, you need to multiply the arguments of your SetJointAcc commands by the factor 1.43.

2.1.23 SetJointVel(p)

This command limits the angular velocities of the robot joints. It affects the movements generated by the MovePose and MoveJoints commands.

Arguments

– p: percentage of maximum joint velocities, ranging from 0.001% to 100%.

Default values

By default, the limit is set to 25%.

It is not possible to limit the velocity of only one joint. With this command, the maximum velocities of all joints are limited proportionally. The maximum velocity of each joint will be reduced to a percentage p of its top velocity. The top 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.

2.1.24 SetTorqueLimits($p_1, p_2, p_3, p_4, p_5, p_6$)

This command sets the thresholds for the torques applied to each joint, as percentages of the maximum allowable torques that can be applied at each joint. When a torque limit is exceeded, a customizable event is created. The event behavior can be set by the command SetTorqueLimitsCfg.

This command is intended only for improving the chances of protecting your robot, its end-effector, and the surrounded equipment, in the case of a collision. The torque in each joint is estimated by measuring the current in the corresponding drive.

Unlike the SetJointLimits commands, the SetTorqueLimits 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. However, recall that SetJointLimits is a motion command. It will therefore be inserted in the motion queue and not necessarily executed immediately.

Arguments

- p_i: percentage of the maximum allowable torque that can be applied at joint i, where $i = 1, 2, ..., 6,$, ranging from 0.001% to 100%.

Default values

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

2.1.25 SetTorqueLimitsCfg(s, m)

This command set the behavior of the robot when a joint torque exceeds the threshold set by the SetTorqueLimits command. It also sets the filtering type used for accurate detection.

Arguments

- s: integer defining the torque limit event severity as

  0, no action;

  1, trace warning;

  2, pause motion;

  3, clear motion;

  4, error.

- m: integer defining the detection mode as

  0, always detect;

  1, skip detection during acceleration/deceleration and blending.

Default values

By default, the event severity is set to 0, and the detection mode to 1.

2.1.26 SetTRF($x, y, z, \alpha, \beta, \gamma$)

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.

Arguments

- x, y, and z: the coordinates of the origin of the TRF w.r.t. the FRF, in mm;
- α, β, and γ: the Euler angles representing the orientation of the TRF w.r.t. the FRF, in degrees.

Default values

By default, the TRF coincides with the FRF.

2.1.27 SetVelTimeout(t)

This command sets the timeout after a velocity-mode motion command (MoveJointsVel, MoveLinVelTRF, or MoveLinVelWRF), after which all joint speeds will be set to zero unless another velocity-mode motion command is received. The SetVelTimeout command should be regarded simply as a safety precaution.

Arguments

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

Default values

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

2.1.28 SetWRF($x, y, z, \alpha, \beta, \gamma$)

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.

Arguments

- x, y, and z: the coordinates of the origin of the WRF w.r.t. the BRF, in mm;
- α, β, and γ: the Euler angles representing the orientation of the WRF w.r.t. the BRF, in degrees.

2.2 Request commands of general type

Contrary to motion commands, request commands are executed immediately and all return a specific response. For clarity, we have divided request commands into three groups. The majority of the requests commands in this section are the most important request commands and serve mainly to control the status of the robot (e.g., activate and home the robot) and to configure the robot.

In the following subsections, the request commands of general type are presented in alphabetical order.

2.2.1 ActivateRobot

This command activates all motors and disables the brakes on joints 1, 2, and 3. It must be sent before homing is started. This command only works if the robot is idle.

Responses

[2000][Motors activated.]
[2001][Motors already activated.]

The first response is generated if the robot was not active, while the second one is generated if the robot was already active.

2.2.2 ActivateSim/DeactivateSim

The Meca500 supports a simulation mode in which all of the robot's hardware functions normally, but none of the motors move. 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 and DeactivateSim commands.

Responses

[2045][The simulation mode is enabled.]
[2046][The simulation mode is disabled.]

2.2.3 ClearMotion

This command stops the robot movement in the same fashion as the PauseMotion command (i.e., by decelerating). However, if the robot is stopped in the middle of a trajectory, the rest of the trajectory is deleted. As is the case with PauseMotion, you need to send the command ResumeMotion to make the robot ready to execute new motion commands.

Responses

[2044][The motion was cleared.]
[3004][End of movement.]

2.2.4 DeactivateRobot

This command disables all motors and engages the brakes on joints 1, 2, and 3. It must not be sent while the robot is moving. Deactivating the robot while in motion could damage the joints. This command should be run before powering down the robot.

When this command is executed, the robot loses its homing. The homing process must be repeated after reactivating the robot.

Responses

[2004][Motors deactivated.]

2.2.5 BrakesOn/BrakesOff

These commands engages or disengages the brakes of joints 1, 2 and 3, if and only if the robot is powered but deactivated. When the brakes are released, the robot will fall down.

Responses

[2010][All brakes set.]
[2008][All brakes released.]

2.2.6 EnableEtherNetIP(e)

This command enables or disables EtherNetIP.

Arguments

– e: enable (1) or disable (0) EtherNet/IP.

Default values

EtherNet/IP is enabled when the robot is shipped from Mecademic. However, changes in this setting have a persistent effect (remain even after a powering the robot off).

2.2.7 GetFwVersion

This command returns the version of the firmware installed on the robot.

Responses

[2081][vx.x.x]

2.2.8 GetModelJointLimits(n)

This command returns the default joint limits, i.e., those presented in Section 1.1.5.

Arguments

- n: joint number, an integer ranging from 1 to 6.

Responses

[2113][$n, \theta_{n,min}, \theta_{n,max}$]

- n: joint number, an integer ranging from 1 to 6;
- $\theta_{n,min}$: lower joint limit, in degrees;
- $\theta_{n,max}$: upper joint limit, in degrees.

2.2.9 GetProductType

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

Responses

[2084][Meca500]

2.2.10 GetRobotSerial

This command returns the serial number of the robot, for robots manufactured recently. For all other robots, the serial number can only be found on the back of the robot's base.

Responses

[2083][robot's serial number]

2.2.11 Home

This command starts the robot and gripper homing process (Section 1.3.1). While homing, it is critical to remove any obstacles that could hinder the robot and gripper movements. This command takes about three seconds to execute.

Responses

[2002][Homing done.]
[2003][Homing already done.]
[1032][Homing failed because joints are outside limits.]
[1014][Homing failed.]

The first response is sent if homing was completed successfully, while the second one is sent if the robot is already homed. The third response is sent if the homing procedure failed because it was started while a robot joint was outside its user-defined limits. The last response is sent if the homing failed for other reasons.

2.2.12 LogTrace(s)

This command inserts a comment into the robot's log. It is useful for debugging, allowing you to show our support team where exactly a certain event occurs.

Arguments

- s: a text string (the comment), preferably enclosed in quotation marks.

Responses

[2085][Command successful: '...'.]

2.2.13 PauseMotion

This command stops the robot movement. The command is executed as soon as received (within approximately 5 ms from it being sent, depending on your network configuration), but the robot stops by decelerating, and not by engaging the brakes. For example, if a MoveLin command is currently being executed when the PauseMotion command is received, the robot TCP will stop somewhere along the linear trajectory. If you want to know where exactly did the robot stop, you can use the GetRtCartPos or GetRtJointPos commands.

Strictly speaking, the PauseMotion command pauses the robot motion; the rest of the trajectory is not deleted and can be resumed with the ResumeMotion command. The PauseMotion command is useful if you develop your own HMI and need to implement a pause button. It can also be useful if you suddenly have a problem with your tool (e.g., while the robot is applying an adhesive, the reservoir becomes empty).

The PauseMotion command normally generates the following two responses. The first one is always sent, whereas the second one is sent only if the robot was moving when it received the PauseMotion command.

Finally, if a motion error occurs while the robot is at pause (e.g., if another moving body hits the robot), the motion is cleared and can no longer be resumed.

Responses

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

2.2.14 ResetError

This command resets the robot error status. The command can generate one of the following two responses. The first response is generated if the robot was indeed in an error mode, while the second one is sent if the robot was not in error mode.

Responses

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

2.2.15 ResetPStop

As described in the User Manual of the Meca500 R3, you can connect one Stop Category 2 protective stop (P-Stop 2) to the robot's power supply. When you apply voltage to the terminals of P-Stop 2, the robot is put in protective stop (Stop Category 2) immediately, and the message [3032][1] is returned. To exit the protective stop, you must first remove the voltage from the P-Stop 2 terminals. Then, you must send the command ResetPStop, which resets the protective stop and generates the message [3032][0].

Responses

[3032][e]
where $e = 1$ if voltage is still applied to the P-Stop 2 terminals, and $e = 0$ otherwise.

2.2.16 ResumeMotion

This command resumes the robot movement, if it was previously paused with the command PauseMotion. More precisely, the robot end-effector resumes the rest of the trajectory from the pose where it was brought to a stop (after deceleration), unless an error occurred after the PauseMotion or the robot was deactivated and then reactivated.

Note that it is not possible to pause the motion along a trajectory, have the end-effector move away, then have it come back, and finally resume the trajectory. If you send motion commands while the robot is paused, they will simply be placed in the queue.

The ResumeMotion command must also be sent after the command ClearMotion. However, in the latter case, the robot will not move until another motion command is received (or retrieved from the motion queue). Finally, the ResumeMotion command must also be sent after the command ResetError.

Responses

[2043][Motion resumed.]

2.2.17 SetEOB(e)

When the robot completes a motion command or a block of motion commands, it can send the message "[3012][End of block.]". This means that there are no more motion commands in the queue and the robot velocity is zero. The user could enable or disable this message by sending the SetEOB command.

Arguments

- e: enable (1) or disable (0) 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.]

2.2.18 SetEOM(e)

The robot can also send the message "[3004][End of movement.]" as soon as the robot velocity becomes zero. This can happen after the commands MoveJoints, MovePose, MoveLin, MoveLinRelTRF, MoveLinRelWRF, PauseMotion and ClearMotion commands, as well as after the SetCartAcc and SetJointAcc commands. Recall, however, that if blending is enabled (even only partially), then there would be no end-of-movement message between two consecutive Cartesian-space commands (MoveLin, MoveLinRelTRF, MoveLinRelWRF) or two consecutive joint-space commands (MoveJoints, MovePose).

Arguments

- e: enable (1) or disable (0) 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.]

2.2.19 SetJointLimits(n, $\theta_{n,min}$, $\theta_{n,max}$)

This command redefines the lower and upper limits of a robot joint. It can only be executed while the robot is powered up, but not activated. Furthermore, for these user-defined joint

limits to be taken into account, you must execute the command SetJointLimitsCfg(1), also while the robot is powered up, but not activated. Obviously, the new joint limits must be within the default joint limits (recall Section 1.1.5). Finally, note these user-defined joint limits remain active even after you power down the robot.

Arguments

- n: joint number, an integer ranging from 1 to 6;
- $\theta_{n,min}$: lower joint limit, in degrees;
- $\theta_{n,max}$: upper joint limit, in degrees.

Responses

[2092][n]

2.2.20 SetJointLimitsCfg(e)

This command enables or disables the user-defined limits set by the SetJointLimits command. It can only be executed while the robot is powered up, but not activated. If the user-defined limits are disabled, the default joint limits become active. However, the user-defined limits remain memorized, even after a power down, so you can always re-enable them.

For example, imagine that you try to home the robot, but one of the wrist joints has been inadvertently rotated outside its activated, user-defined limits. The homing will fail, so you will have to deactivate the robot, disable the user-defined limits, reactivate the robot, home it, then bring it to a robot position that is within the user-defined limits, then deactivate the robot, re-enable the user-defined limits, and finally home the robot again.

Arguments

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

Responses

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

2.2.21 SetMonitoringInterval(t)

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 and Section 2.5.4, for more details).

Arguments

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

Default values

By default, the monitoring time interval is 0.015 s.

2.2.22 SetNetworkOptions($n_1, n_2, n_3, n_4, n_5, n_6$)

This command is used to set persistent parameters affecting the network connection. The command can only be executed while the robot is powered but not activated. The newly set parameter values will take effect only after a robot reboot.

Arguments

- n_1: number of successive keepalive TCP packets that can be lost before the TCP connection is closed, where n_1 is an integer number ranging from 0 to 43,200;
- n_2, n_3, n_4, n_5, n_6: currently not used.

Default values

By default, $n_1 = 3$.

2.2.23 SetOfflineProgramLoop(e)

This command is used to define whether the program that is to be saved must later be executed a single time or infinitely many times.

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. It is only when saving a program that a message indicates whether loop execution was enabled or disabled. However, if the command is sent while no program is being saved, the above message is returned.

2.2.24 SetRTC(t)

Since our robots do not have batteries, when a Meca500 is powered on, the internal clock starts at the date at which the robot image was built. Each time you connect to the robot via the web interface, the internal clock of the robot is automatically adjusted to UTC. You

must, therefore, start your programs outside the web interface with the SetRTC command, if you want all timestamps in your log files to be with respect to UTC.

Arguments

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

2.2.25 StartProgram(n)

To start the program that has been previously saved in the robot memory, the robot must be activated prior to using the StartProgram command. The number of times the program will be executed is defined with the SetOfflineProgramLoop command. Note that you can either use this command, or simply press the Start/Stop button on the robot base (provided that no one is connected to the robot). However, pressing the Start/Stop button on the robot base will only start program 1.

Arguments

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

Responses

[2063][Offline program n started.]
[3017][No offline program saved.]

2.2.26 StartSaving(n)

The Meca500 is equipped with several membrane buttons on its base, one of which is a Start/Pause button. These buttons can be used to run a simple program (possibly in loop), when no external device is connected to the robot. For example, this feature is particularly useful for running demos. To distinguish a program that will reside in the memory of the robot controller, from any other programs that will be sent to the robot and executed line by line, the program will be referred to as an *offline program*.

To save such an offline program, you need to use the StartSaving and StopSaving commands. Note that the program will remain in the robot internal memory even after disconnecting the power. To clear a specific program, simply overwrite a new program by using the StartSaving command with the same argument.

The StartSaving command starts the recording of all subsequent motion commands. Once the robot receives this command, it will generate the first of the two responses given bellow and start waiting for the command(s) to save. If the robot receives a Get* command

(GetBlending, GetRtCartPos, etc.), it will execute it but not save it to the offline program, and will generate the second response. Finally, it the robot receives a command that changes the state of the robot (BrakesOn, Home, PauseMotion, SetEOM, etc.), it will generate the third response and abort the saving of the program.

You can save up to 500 different programs, each with up to 13,000 motion commands, by specifying the program number in the argument of the StartSaving command. Only program 1, however, can be executed through the Start/Pause button on the robot base. Thus, you can think of all the other "offline programs" as procedures that you can call in your main program or even within your other offline programs with the command StartProgram.

Arguments

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

Responses

[2060][Start saving program.]
[1023][Ignoring command for offline mode. - Command: '...']
[1031][Program saving aborted after receiving illegal command. - Command: '...']

2.2.27 StopSaving

This command will make the controller save the program and end the saving process. Normally, two responses will be generated: the first and the second or third of the three responses given below. Finally, if you send this command while the robot is not saving a program, the fourth response will be returned.

Responses

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

2.2.28 SwitchToEtherCAT

This command will disable the Ethernet TCP/IP and EtherNet/IP protocols and enable EtherCAT (Section 4). You can disable EtherCAT once you start to use it (Section 4.2.13) or by performing a factory reset (keeping the power button on the robot's base pressed during restart).

2.2.29 TCPDump(n)

This command starts an Ethernet capture (pcap format) on the robot, for the specified duration). The Ethernet capture will be part of the logs archive.

Arguments

– n: duration in seconds.

Responses

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

2.2.30 TCPDumpStop

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

Responses

[3036][TCP dump capture stopped.]

2.3 Request commands of type user-defined data

The request commands in this section generally return (on TCP port 10000) the values of parameters that have been already configured with a Set* command (or the default values). There are a few exceptions though. For example the command GetConf may return the parameters set with the command SetConf but it may also return the current "desired" robot posture parameters or even {*,*,*}.

In the previous paragraph, by "already configured," we meant that the Set* command has already been executed, not simply sent. Recall that motion commands that are sent to the robot are put in a motion queue and executed on a FIFO basis. Thus, if you send a SetTRF command, then a MovePose command, then another SetTRF command, and immediately after that a GetTRF command, you will get the arguments of the first SetTRF command.

In the following subsections, the request commands of type user-defined are presented in alphabetical order. For every Get* command in this section, there is a corresponding Set* command.

2.3.1 GetAutoConf

This command returns the state of the automatic robot posture configuration selection.

Responses

[2028][e]

- e: enabled (1) or disabled (0).

2.3.2 GetAutoConfTurn

This command returns the state of the automatic turn configuration selection.

Responses

[2031][e]

- e: enabled (1) or disabled (0).

2.3.3 GetBlending

This command returns the blending percentage, normally set by the SetBlending command.

Responses

[2150][p]

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

2.3.4 GetCartAcc

This command returns the desired limit of the acceleration of the TRF w.r.t. the WRF, normally set by the command SetCartAcc.

Responses

[2156][p]

- p: percentage of maximum acceleration of the TRF.

2.3.5 GetCartAngVel

This command returns the desired limit of the angular velocity of the TRF w.r.t. the WRF, normally set by the command SetCartAngVel.

Responses

[2155][ω]

- ω: TRF angular velocity limits, in °/s.

2.3.6 GetCartLinVel

This command returns the desired TCP velocity limit, normally set by SetCartLinVel.

Responses

[2154][v]

– v: TCP velocity limit, in mm/s.

2.3.7 GetCheckpoint

This command returns the argument of the last executed SetCheckpoint.

Responses

[2156][p]

– n: checkpoint number.

2.3.8 GetConf

This command returns the "desired" robot posture configuration (see Fig. 4), or more precisely, the robot posture configuration that will be applied to the next MovePose command in the motion queue. Recall that this is either the robot posture configuration that you have specified with the command SetConf or the current robot posture configuration.

Responses

[2029][c_s, c_e, c_w]

– c_s: shoulder robot posture configuration parameter, either -1 or $1^\dagger$;

– c_e: elbow robot posture configuration parameter, either -1 or $1^\dagger$;

– c_w: wrist robot posture configuration parameter, either -1 or $1^\dagger$;

† if automatic robot posture configuration selection is enabled, the value of each parameter is an asterisk, i.e., the response is [2029][*,*,*].

2.3.9 GetConfTurn

This command returns the "desired" turn configuration (see Fig. 4), or more precisely, the turn configuration that will be applied to the next MovePose or MoveLin* command in the motion queue. Recall that this is either the turn configuration that you have specified with the command SetConfTurn or the current turn configuration.

Responses

[2036][c_t]

– c_t: turn configuration parameter, an integer from -100 to $100^\dagger$;

† if automatic turn configuration selection is enabled, the value returned is an asterisk, i.e., the response is [2036][*].

2.3.10 GetGripperForce

This command returns the percentage of maximum grip force for the MEGP 25E gripper. This percentage is normally set by the SetGripperForce command.

Responses

[2158][p]

- p: percentage of maximum grip force.

2.3.11 GetGripperVel

This command returns the percentage of maximum finger velocity for the MEGP 25E gripper. This percentage is normally set by the SetGripperVel command.

Responses

[2159][p]

- p: percentage of maximum velocity of the gripper fingers.

2.3.12 GetJointAcc

This command returns the desired joint accelerations reduction factor, normally set by the command SetJointAcc.

Responses

[2153][p]

- p: percentage of maximum joint accelerations.

2.3.13 GetJointLimits(n)

This command returns the current effective joint limits, i.e., the default joint limits or the user-defined limits if applied (SetJointLimits) and enabled (SetJointLimitsCfg).

Arguments

- n: joint number, an integer ranging from 1 to 6.

Responses

[2090][$n, \theta_{n,min}, \theta_{n,max}$]

- n: joint number, an integer ranging from 1 to 6;
- $\theta_{n,min}$: lower joint limit, in degrees;
- $\theta_{n,max}$: upper joint limit, in degrees.

2.3.14 GetJointLimitsCfg

This command returns the status of the user-enabled joint limits, normally set by the command SetJointLimitsCfg.

Responses

[2094][e]

- e: status, 1 for enabled, 0 for disabled.

2.3.15 GetJointVel

This command returns the desired joint velocity reduction factor, normally set by the command SetJointVel.

Responses

[2152][p]

- p: percentage of maximum joint velocities.

2.3.16 GetMonitoringInterval

This command returns the time interval at which real-time feedback from the robot is sent from the robot over TCP port 10001.

Responses

[2116][t]

- t: time interval in seconds.

2.3.17 GetNetworkOptions

This command returns the parameters affecting the network connection.

Responses

[2119][$n_1, n_2, n_3, n_4, n_5, n_6$]

- n_1: number of successive keepalive TCP packets that can be lost before the TCP connection is closed, where n_1 is an integer number ranging from 0 to 43,200
- n_2, n_3, n_4, n_5, n_6: currently not used.

2.3.18 GetRealTimeMonitoring

This command returns the numerical codes of the responses that have been enabled with the SetRealTimeMonitoring command.

Responses

[2117][$n_1, n_2, ...$]

2.3.19 GetTorqueLimits

This command returns the desired joint torque thresholds, normally set by the command SetTorqueLimits.

Responses

[2161][$p_1, p_2, p_3, p_4, p_5, p_6$]

- p_i: percentage of the maximum allowable torque that can be applied at joint i, where i = 1, 2, ..., 6.

2.3.20 GetTorqueLimitsCfg

This command returns the desired behavior of the robot, when a joint torques exceeds the thresholds set by the SetTorqueLimits. This desired bahevior is normally set by the command SetTorqueLimitsCfg.

Responses

[2160][s, m]

- s: an integer defining the torque limit event severity (see SetTorqueLimitsCfg);
- m: an integer defining the detection mode (see SetTorqueLimitsCfg).

2.3.21 GetTRF

This command returns the current definition of the TRF w.r.t. the FRF, normally set by the SetTRF command.

Responses

[2014][$x, y, z, \alpha, \beta, \gamma$]

- x, y, and z: the coordinates of the origin of the TRF w.r.t. the FRF, in mm;
- α, β, and γ: the Euler angles representing the orientation of the TRF w.r.t. the FRF, in degrees.

2.3.22 GetVelTimeout

This command returns the timeout for velocity-mode motion commands, normally set by the SetVelTimeout command.

Responses

[2151][t]

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

2.3.23 GetWRF

This command returns the current definition of the WRF w.r.t. the BRF, normally set by the SetWRF command.

Responses

[2013][$x, y, z, \alpha, \beta, \gamma$]

- x, y, and z: the coordinates of the origin of the WRF w.r.t. the BRF, in mm;
- α, β, and γ: the Euler angles representing the orientation of the WRF w.r.t. the BRF, in degrees.

2.4 Request commands of type real-time data

The request commands in this subsection return real-time data pertaining to the current status of the robot. The obvious example is the current joint set, but there is a command that also returns the current length of the motion queue, and another that returns the current status of the torque limits, for example.

There are two types of robot positioning real-time data. The first set of commands return data according to real-time measurements by the robot's sensors. The command GetRtJointTorq returns the current joint torques, as measured by the motor currents, the command GetRtAccelerometer returns the acceleration in link 5, as measured by the accelerometer embedded in link 5, and the command GetRtJointPos returns the current joint set, as measured by the joint encoders. Subsequently, the commands GetRtCartPos, GetRtJointVel, GetRtCartVel, GetRtConf and GetRtConfTurn return data as calculated from the real-time joint encoder measurements.

The second set of commands return real-time targets as calculated by our trajectory planner: GetRtTargetCartPos, GetRtTargetJointVel, GetRtTargetCartVel, GetRtTargetConf and GetRtTargetConfTurn. For example, if the robot is active and homed, but not moving, the GetRtTargetJointPos command will always return the same joint set, as long as the robot remains stationary. In reality, the robot is never perfectly still as the drives are constantly controlling the motors. In the case of the Meca500, this means that the joints oscillate approximately $\pm 0.001°$ about the desired joint angles. In other words, if you execute

two times in a row the command GetRtJointPos while the robot is "not moving", you will see that the joint values differ by as much as a couple of micro-degrees.

In a more extreme situation, if a high force is applied to the robot, you will see larger differences between the real joint set (GetRtJointPos) and the desired one (GetRtTargetJointPos). The differences become even larger during rapid motions at high payloads and at a collision.

Each of the GetRt* commands returns a response that starts with a timestamp, measured in micro-seconds. The GetRtTargetCartPos and GetRtTargetJointPos return the same data as the deprecated commands GetPose and GetJoints respectively, except for the timestamp.

All of the commands in this section return responses on TCP port 10000, except for the SetRealTimeMonitoring command, which triggers a continuous flux of responses over TCP port 10001.

2.4.1 GetCmdPendingCount

This command returns the number of motion commands that are currently in the motion queue.

Responses

[2080][n]

2.4.2 GetJoints

This deprecated command returns the current target joint set. Use GetRtTargetJointPos instead.

Responses

[2026][$\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$]

- θ_i: the angle of joint i, in degrees, where $i = 1, 2, ..., 6$.

2.4.3 GetPose

This deprecated command returns the current target pose of the robot TRF with respect to the WRF. Use GetRtTargetCartPos instead.

Responses

[2027][$x, y, z, \alpha, \beta, \gamma$]

- x, y, and z: the coordinates of the origin of the TRF w.r.t. the WRF, in mm;
- α, β, and γ: the Euler angles representing the orientation of the TRF w.r.t. the WRF, in degrees.

2.4.4 GetRtAccelerometer(n)

An accelerometer is embedded in link 5 of the Meca500, i.e., the body with the I/O port (just before joint 6). It reports the acceleration of link 5 w.r.t. the WRF in the range $\pm 32,000$, which corresponds to $\pm$2g. Thus, if the robot is not moving and is installed upwards on a stationary horizontal surface, GetRtAccelerometer(5) will return approximately $\{0,0,-16000\}$, no matter what the joint set. In other words, in stationary conditions, you can essentially think as if the accelerometer is embedded in the base of the robot.

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, and in units such that 16,000 is equivalent to 9.81 m/s^2 (i.e., 1g).

Note that that data from this accelerometer is not highly accurate and should not be used for precise measurements.

2.4.5 GetRTC

This command returns the current Epoch Time in seconds, normally set by SetRTC, 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.

Responses

[2140][t]

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

2.4.6 GetRtCartPos

Contrary to the legacy command GetPose, the GetRtCartPos command returns the pose of the TRF w.r.t. the WRF, as calculated from the current joint set read by the joint encoders, rather than as calculated from the current *target* joint set. In addition, it returns a timestamp.

Responses

[2211][$t, x, y, z, \alpha, \beta, \gamma$]

- t: timestamp in microseconds;
- x, y, and z: the coordinates of the origin of the TRF w.r.t. the WRF, in mm;
- α, β, and γ: the Euler angles representing the orientation of the TRF w.r.t. the WRF, in degrees.

2.4.7 GetRtCartVel

This command returns the current Cartesian velocity vector of the TRF w.r.t. the WRF, as calculated from the real-time data coming from the joint encoders.

Responses

[2214][$t, \dot{x}, \dot{y}, \dot{z}, \omega_x, \omega_y, \omega_z$]

- t: timestamp in microseconds;
- $\dot{x}, \dot{y}, \dot{z}$: components of the linear velocity vector of the TCP w.r.t. the WRF, in mm/s;
- $\omega_x, \omega_y, \omega_z$: components of the angular velocity vector of the TRF w.r.t. the WRF, in °/s.

The current *TCP speed* w.r.t. the WRF is therefore $\sqrt{\dot{x}^2 + \dot{y}^2 + \dot{z}^2}$, and the current *angular speed* of the end-effector w.r.t. the WRF is $\sqrt{\omega_x^2 + \omega_y^2 + \omega_z^2}$. Note that the components of the angular velocity vector are not the time derivatives of the Euler angles.

2.4.8 GetRtConf

Contrary to the command GetConf which returns the desired robot posture configuration parameters, the GetRtConf returns the current robot posture configuration parameters, as calculated from the real-time data coming from the joint encoders. In addition, the GetRtConf command returns a timestamp.

Responses

[2208][t, c_s, c_e, c_w]

- t: timestamp in microseconds;
- c_s: shoulder robot posture configuration parameter, either -1 or $1^\dagger$;
- c_e: elbow robot posture configuration parameter, either -1 or $1^\dagger$;
- c_w: wrist robot posture configuration parameter, either -1 or $1^\dagger$;

† at the corresponding singularity, we return 1.

2.4.9 GetRtConfTurn

Contrary to the command GetConfTurn which returns the desired turn configuration parameter, the GetRtConfTurn returns the current turn configuration parameter, as calculated from the real-time data coming from the joint encoder of joint 6. In addition, the GetRtConfTurn command returns a timestamp.

Responses

[2209][t, c_t]

- t: timestamp in microseconds;
- c_t: turn configuration parameter, an integer between -100 and 100.

2.4.10 GetRtJointPos

Contrary to the legacy command GetJoints, the GetRtJointPos command returns the current joint set read by the joint encoders, rather than the current target joint set. In addition, it returns a timestamp.

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, where $i = 1, 2, ..., 6$.

2.4.11 GetRtJointTorq

This command returns the current joint torques.

Responses

[2213][$t, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6$]

- t: timestamp in microseconds;
- τ_i: the torque of joint i as a signed percentage of the maximum allowable torque, where $i = 1, 2, ..., 6$.

2.4.12 GetRtJointVel

This command returns the current joint velocities, as calculated by differentiating the data coming from the joint encoders.

Responses

[2212][$t, \dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6$]

- t: timestamp in microseconds;
- $\dot{\theta}_i$: the rate of change of joint i, in °/s, where $i = 1, 2, ..., 6$.

2.4.13 GetRtTargetCartPos

This command returns the current target pose of the TRF w.r.t. the WRF, rather than the pose as calculated from real-time data from the joint encoders. It returns the same data as the legacy GetPose command, except for the additional timestamp.

Responses

[2201][$t, x, y, z, \alpha, \beta, \gamma$]

- t: timestamp in microseconds;
- x, y, and z: the coordinates of the origin of the TRF w.r.t. the WRF, in mm;
- α, β, and γ: the Euler angles representing the orientation of the TRF w.r.t. the WRF, in degrees.

2.4.14 GetRtTargetCartVel

This command returns the current target Cartesian velocity vector of the TRF w.r.t. the WRF.

Responses

[2204][$t, \dot{x}, \dot{y}, \dot{z}, \omega_x, \omega_y, \omega_z$]

- t: timestamp in microseconds;
- $\dot{x}, \dot{y}, \dot{z}$: components of the linear velocity vector of the TCP w.r.t. the WRF, in mm/s;
- $\omega_x, \omega_y, \omega_z$: components of the angular velocity vector of the TRF w.r.t. the WRF, in °/s.

2.4.15 GetRtTargetConf

This command returns the robot posture configuration parameters calculated from the current target joint set.

Responses

[2208][t, c_s, c_e, c_w]

- t: timestamp in microseconds;
- c_s: shoulder robot posture configuration parameter, either -1 or $1^{\dagger}$;
- c_e: elbow robot posture configuration parameter, either -1 or $1^{\dagger}$;
- c_w: wrist robot posture configuration parameter, either -1 or $1^{\dagger}$;

† at the corresponding singularity, we return 1.

2.4.16 GetRtTargetConfTurn

This command returns the turn configuration parameters calculated from the current target joint value for joint 6.

Responses

[2209][t, c_t]

- t: timestamp in microseconds;
- c_t: turn configuration parameter, an integer between -100 and 100.

2.4.17 GetRtTargetJointPos

This command returns the current target joint set. It returns the same data as the legacy GetJoints commands, except for the additional timestamp.

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, where $i = 1, 2, ..., 6$.

2.4.18 GetRtTargetJointVel

This command returns the current target joint velocities.

Responses

[2202][$t, \dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6$]

- t: timestamp in microseconds;
- $\dot{\theta}_i$: the rate of change of joint i, in $^{\circ}$/s, where $i = 1, 2, ..., 6$.

2.4.19 GetStatusGripper

This command returns the gripper's status.

Responses

[2079][ge, hs, ph, lr, es, fo]

- ge: gripper enabled, i.e., present (0 for disabled, 1 for enabled);
- hs: homing state (0 for homing not performed, 1 for homing performed);
- ph: holding part (0 if the gripper does not hold a part, 1 otherwise);
- lr: limit reached (0 if the fingers are not fully open or closed, 1 otherwise);
- 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).

2.4.20 GetStatusRobot

This command returns the status of the robot.

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 (1 if simulation mode is enabled, 0 otherwise);
- 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 idle and motion queue is empty, 0 otherwise);
- eom: end of movement status (1 if robot is idle, 0 if robot is moving).

Note that $pm = 1$ if and only if a PauseMotion or a ClearMotion was sent, or if the robot is in error mode.

2.4.21 GetTorqueLimitsStatus

This command returns the status of the torque limits (whether a torque limit is currently exceeded).

Responses

[3028][s]

- s: status (0 if no detection, 1 if a torque limit was exceeded).

2.4.22 SetRealTimeMonitoring($n_1, n_2, ...$)

As already mentioned in Section 2, after the robot is homed, TCP port 10001 (i.e., the monitoring port) transmits periodically the robot's joint set and TRF pose, as well as other data (see Section 2.5.4), at the rate specified by the SetMonitoringInterval command.

You can enable the transmission of various other real-time data over the monitoring port, with the difference that they are preceded by a monotonic timestamp in microseconds (see SetRTC. You can choose the additional data to be sent over TCP port 10001 with the command SetRealTimeMonitoring, the arguments of which are a list of numerical codes or alphabetical names. You can send this command even if the robot is not activated. Essentially, you can get exactly the same responses that you can get with the GetRt* and GetRtTarget* commands, but on the monitoring port, instead of on the control port, and every monitoring interval, rather than only when requested.

Arguments

- $n_1, n_2, ...$: a list of number codes or names, as follows

  2200 or *TargetJointPos*, for the response of the GetRtTargetJointPos command;

  2201 or *TargetCartPos*, for the response of the GetRtTargetCartPos command;

  2204 or *TargetCartVel*, for the response of the GetRtTargetCartVel command;

  2208 or *TargetConf*, for the response of the GetRtTargetConf command;

  2209 or *TargetConfTurn*, for the response of the GetRtTargetConfTurn command;

  2210 or *JointPos*, for the response of the GetRtJointPos command;

  2211 or *CartPos*, for the response of the GetRtCartPos command;

  2212 or *JointVel*, for the response of the GetRtJointVel command;

  2213 or *JointTorq*, for the response of the GetRtJointTorq command;

  2214 or *CartVel*, for the response of the GetRtCartVel command;

  2218 or *Conf*, for the response of the GetRtConf command;

  2219 or *ConfTurn*, for the response of the GetRtConfTurn command;

  2220 or *Accel*, for the response of the GetRtAccelerometer command;

  All, enables all of the above responses.

Default values

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

Responses

[2117][$n_1, n_2, ...$]

– $n_1, n_2, ...$: a list of response codes.

Finally, the SetRealTimeMonitoring command does not have an accumulative effect. In other words, if you execute the command SetRealTimeMonitoring(All) and then the command SetRealTimeMonitoring(2201), you will only enable the message 2201.

More details about the monitoring port are presented in Section 2.5.4.

2.5 Responses and messages

The Meca500 sends responses and messages over its control port when it encounters an error, when it receives a request command or certain motion commands, and when its status changes. All responses from the Meca500 consist of an ASCII string in the following format:

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

The four-digit code indicates the type of response:

- [1000] to [1999]: Error message due to a command;
- [2000] to [2999]: Response to a command, or pose and joint set feedback;
- [3000] to [3999]: Status update message or general error.

The second part of a command error message [1xxx] or a status update message [3xxx] will always be a description text. The second part of a command response [2xxx] may be a description text or a set of comma-separated return values, depending on the command.

All text descriptions are intended to communicate information to the user and are subject to change without notice. For example, the description "Homing failed" may eventually be replaced by "Homing has failed." Therefore, you must rely only on the four-digit code of such messages. Any change in the codes or in the format of the comma-separated return values will always be documented in the firmware upgrade manual. Finally, return values are either integers or IEEE-754 floating-point numbers with three decimal places.

2.5.1 Command error messages

When the Meca500 encounters an error while executing a command, it goes into error mode. Then, all pending commands are canceled, the robot stops and ignores subsequent commands until it receives a ResetError command. Table 1 lists all command error messages.

Table 1: List of command error messages

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.
[1006][The robot is not homed.]	The robot must be homed.
[1007][Joint over limit. - Command: '...']	The robot cannot reach the joint set or pose requested because of its joint limits.
[1011][The robot is in error.]	A command has been sent but the robot is in error mode and cannot process it until a ResetError command is sent.
[1012][Singularity detected.]	The MoveLin command sent requires that the robot pass through a singularity, or the MovePose command sent requires that the robot goes to a singular robot posture.
[1013][Activation failed.]	Activation failed. Try again.
[1014][Homing failed.]	Homing procedure failed. Try again.
[1016][Pose out of reach.]	The pose requested in the MoveLin or MovePose commands cannot be reached by the robot (even if the robot had no joint limits).
[1017][Communication failed. - Command: '...']	Problem with communication.
[1018]['\0' missing. - Command: '...']	Missing NULL character at the end of a command.
[1020][Brakes cannot be released.]	Something is wrong. The brakes cannot be engaged. Try again.
[1021][Deactivation failed. - Command: '...']	Something is wrong. The deactivation failed. Try again.
[1022][Robot was not saving the program.]	The command StopSaving 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.

Table 1: List of command error messages (continued)

Message	Explanation
[1024][Mastering needed. - Command: '...']	Somehow, mastering was lost. Contact Mecademic.
[1025][Impossible to reset the error. Please, power-cycle the robot.]	Turn off the robot, then turn it back on in order to reset the error.
[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.
[1028][Network error.]	Error on the network. Resend the command.
[1029][Offline program full. Maximum program size is 13,000 commands. Saving stopped.]	Memory full.
[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.
[1032][Homing failed because joints are outside limits.]	Homing cannot be done, because the current joint set is outside the user-defined joint limits.
[1038][No gripper connected.]	No gripper was detected.
[1040][Command failed.]	General error for various commands.

2.5.2 Command responses

Recall that, unlike request commands, motion commands do not generate directly any (non-error) response, other than the optional EOB and EOM messages (see Section 2.1 for details) and the message eventually generated by the SetCheckpoint command. Table 2 presents a summary of all request commands and the possible non-error responses for each of them.

Table 2: List of request commands and the corresponding possible responses

Command	Possible response(s)
ActivateRobot	[2000][Motors activated.] [2001][Motors already activated.]
ActivateSim	[2045][The simulation mode is enabled.]
BrakesOff	[2008][All brakes released.]
BrakesOn	[2010][All brakes set.]
ClearMotion	[2044][The motion was cleared.]
DeactivateRobot	[2004][Motors deactivated.]

Table 2: List of request commands and the corresponding possible responses (continued)

Command	**Possible response(s)**
DeactivateSim	[2046][The simulation mode is disabled.]
GetAutoConf	[2028][e]
GetAutoConfTurn	[2031][e]
GetBlending	[2150][p]
GetCartAcc	[2156][p]
GetCartAngVel	[2155][ω]
GetCartLinVel	[2154][v]
GetCheckpoint	[2157][n]
GetCmdPendingCount	[2080][n]
GetConf	[2029][c_s, c_e, c_w]
GetConfTurn	[2036][c_t]
GetFwVersion	[2081][vx.x.x]
GetGripperForce	[2158][p]
GetGripperVel	[2159][p]
GetJointAcc	[2153][p]
GetJointLimits	[2090][$n, \theta_{n,min}, \theta_{n,max}$]
GetJointLimitsCfg	[2094][e]
GetJoints	[2026][$\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$]
GetJointVel	[2152][p]
GetMonitoringInterval	[2116][t]
GetNetworkOptions	[2119][$n_1, n_2, n_3, n_4, n_5, n_6$]
GetPose	[2027][$x, y, z, \alpha, \beta, \gamma$]
GetProductType	[2084][Meca500]
GetRealTimeMonitoring	[2117][$n_1, n_2, ...$]
GetRobotSerial	[2083][robot's serial number]
GetRtAccelerometer	[2220][t, n, a_x, a_y, a_z]
GetRTC	[2140][t]
GetRtCartPos	[2211][$t, x, y, z, \alpha, \beta, \gamma$]
GetRtCartVel	[2214][$t, \dot{x}, \dot{y}, \dot{z}, \omega_x, \omega_y, \omega_z$]
GetRtConf	[2208][t, c_s, c_e, c_w]
GetRtConfTurn	[2209][t, c_t]
GetRtJointPos	[2210][$t, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$]
GetRtJointTorq	[2213][$t, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6$]
GetRtJointVel	[2212][$t, \dot{\theta_1}, \dot{\theta_2}, \dot{\theta_3}, \dot{\theta_4}, \dot{\theta_5}, \dot{\theta_6}$]
GetRtTargetCartPos	[2201][$t, x, y, z, \alpha, \beta, \gamma$]
GetRtTargetCartVel	[2214][$t, \dot{x}, \dot{y}, \dot{z}, \omega_x, \omega_y, \omega_z$]

Table 2: List of request commands and the corresponding possible responses (continued)

Command	**Possible response(s)**
GetRtTargetConf	[2208][t, c_s, c_e, c_w]
GetRtTargetConfTurn	[2209][t, c_t]
GetRtTargetJointPos	[2200][$t, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$]
GetRtTargetJointVel	[2202][$t, \dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6$]
GetStatusGripper	[2079][ge, hs, ph, lr, es, fo]
GetStatusRobot	[2007][$as, hs, sm, es, pm, eob, eom$]
GetTorqueLimits	[2161][$p_1, p_2, p_3, p_4, p_5, p_6$]
GetTorqueLimitsCfg	[2160][s, m]
GetTRF	[2014][$x, y, z, \alpha, \beta, \gamma$]
GetVelTimeout	[2151][t]
GetWRF	[2013][$x, y, z, \alpha, \beta, \gamma$]
Home	[2002][Homing done.] [2003][Homing already done.]
PauseMotion	[2042][Motion paused.] [3004][End of movement.]
ResetError	[2005][The error was reset.] [2006][There was no error to reset.]
ResetPStop	[3032][e]
ResumeMotion	[2043][Motion resumed.]
SetEOB	[2054][End of block is enabled.] [2055][End of block is disabled.]
SetEOM	[2052][End of movement is enabled.] [2053][End of movement is disabled.]
SetJointLimits	[2092][n]
SetJointLimitsCfg	[2093][User-defined joint limits enabled] [2093][User-defined joint limits disabled]
SetRealTimeMonitoring	[2117][$n_1, n_2, ...$]
SetOfflineProgramLoop	[1022][Robot was not saving the program.]
StartProgram	[2063][Offline program n started.] [3004][End of movement.] [3012][End of block.]
StartSaving	[2060][Start saving program.] [1023][Ignoring command for offline mode. - Command: '...'] [1031][Program saving aborted after receiving illegal command. - Command: '...']

Table 2: List of request commands and the corresponding possible responses (continued)

Command	Possible response(s)
StopSaving	[2061][n commands saved.] [2064][Offline program looping is enabled.] [2065][Offline program looping is disabled.]
TCPDump	[3035][TCP dump capture started for n seconds.] [3036][TCP dump capture stopped.]
TCPDumpStop	[3036][TCP dump capture stopped.]

2.5.3 Status messages

Status messages generally occur without any specific action from the network client. These could contain general status (e.g., when the robot has ended its motion) or error messages (e.g., error of motion due to a collision). Table 3 lists all possible status messages.

Table 3: List of all status messages

Message	Explanation
[3000][Connected to Meca500 x_x_x.x.x.]	Confirms connection to robot.
[3001][Another user is already connected, closing connection.]	Another user is already connected to the Meca500. 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 NUL character. Most probably caused by a missing NUL character
[3004][End of movement.]	The robot has stopped moving.
[3005][Error of motion.]	Motion error. Most probably caused by a collision or an overload. Correct the situation and send the ResetError command. If the motion error persists, try power-cycling the robot.
[3009][Robot initialization failed due to an internal error. Restart the robot.]	Error in robot startup procedure. Contact Mecademic if restarting the Meca500 did not resolve the issue.
[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.

Table 3: List of all status messages (continued)

Message	Explanation
[3014][Problem with saved program, save a new program.]	There was a problem saving the program.
[3016][Ignoring command while in offline mode.]	A non-motion command was sent while executing a program and was ignored.
[3017][No offline program saved.]	There is no program in memory.
[3018][Loop ended. Restarting the program.]	The offline program is being restarted.
[3025][Gripper error.]	If the gripper was forcing when this message appeared, most probably an overheating occurred. Let the gripper cool down for a few minutes and send the ResetError command. Note that the gripper will stop applying a force, so 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 Mecademic.
[3028][s]	A torque limit was exceeded.
[3030][n]	Checkpoint n was reached.
[3032][e]	P-Stop 2 enabled ($e = 1$) or cleared ($e = 0$).
[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.

2.5.4 Messages over the monitoring port

The Meca500 is configured to send immediate robot feedback over TCP port 10001. Several kinds of feedback messages are sent over this port, in this order:

- [2016][$\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$], i.e., joint set,
- [2027][$x, y, z, \alpha, \beta, \gamma$], i.e., the pose of the TRF w.t.r. the WRF,
- [2029][c_s, c_e, c_w], i.e., the posture configuration (only when it changes),
- [2036][c_t], i.e., the turn configuration (only when it changes),
- [2007][$as, hs, sm, es, pm, eob, eom$], i.e., the robot status (only when it changes),
- [2079][ge, hs, ph, lr, es, fo], i.e., the MEGP 25E gripper status (only when a gripper is installed and its status changes),

- other optional data if enabled by SetRealTimeMonitoring,
- [2030][t].

By default, these feedback messages are sent every 15 ms. You can set the time interval between subsequent feedback messages with the command SetMonitoringInterval. Note that each two of the above ASCII messages are separated by a single null-character and that there are no white spaces in any of these messages.

Note that the optional messages enabled by SetRealTimeMonitoring(2200,2201,2208,2209) are redundant, as they present the same data as the messages 2016, 2027, 2036 and 2036, that we have kept for legacy reasons.

Finally, here is an example of the messages sent over TCP port 10001 in one interval (for clarity, the null-characters have been replaced by line breaks):

Listing 1: Example of the feedback sent over TCP port 10001

```
[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]
[2029][-1,-1,1]
[2208][58675156984,-1,-1,1]
[2209][58675156984,0]
[2230][58675156984]
```

3 Communicating over cyclic protocols

3.1 Overview

The Meca500 can also be controlled using the EtherCAT and EtherNet/IP protocols. These two protocols are described in the next two chapters, but while inherently different, we use them in a very similar way. Therefore, we will present the concepts that are common to both protocols in this section, instead of repeating them twice.

3.1.1 Cyclic data

With EtherCAT and EtherNet/IP protocols, the Meca500 is controlled using *cyclic data* exchanges. Through changes in the cyclic data, a PLC will be able to activate, configure and move the robot, as well as monitor the robot. The cyclic data payload format is almost identical in these two protocols. The following explanations and data fields apply to both protocols.

3.2 Types of robot commands

The following types of commands can be sent to the robot using cyclic data.

3.2.1 Status change commands

Some fields (bits) of the cyclic data directly control the robot status, namely:

- pause motion;
- clear motion;
- simulation mode.

A change in the cyclic value of these fields will cause the corresponding status change on the robot. The corresponding status bit in the cyclic data from the robot will then confirm when robot status has changed.

Do not assume that robot state has changed based on some cycle count or time delay. Always check the corresponding confirmation bit in the cyclic data from the robot.

3.2.2 Triggered actions

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

- activate robot;
- deactivate robot;
- home robot;
- reset error;
- reset PStop.

These status bits should be set to 1 to trigger the corresponding action and cleared (reset to 0) only once the action has been completed. Completion of the action is confirmed by the corresponding bit in the cyclic data from the robot.

Do not assume that the action has been successfully triggered based on some cycle count or time delay. Always check the corresponding confirmation bit in the cyclic data from the robot. Clearing the status bit before that confirmation may prevent the action from being performed.

3.2.3 Motion commands

Most commands related to the robot movement are posted to the robot's *motion queue*. The robot will try to execute these commands completely, one after the other, as explained earlier in this manual (Section 2.1). If you send two position-mode motion commands, one immediately after the other, the second command will not be executed until the first one was completed. In contrast, if you send two velocity-mode motion commands, one immediately after the other, the second one will cancel the execution of the first one, as soon as received.

3.3 Sending motion commands to the robot

Motion commands are sent via three cyclic data fields and the command arguments.

3.3.1 CommandID

We have assigned a unique number to each of the available motion commands (see Table 7). By entering this number in the CommandID field, you are specifying the motion command that is to be sent to the robot.

3.3.2 MoveID and SetPoint

With the combination of two fields, MoveID and SetPoint, we are able to send either cyclic motion commands (i.e., executed at every cycle) or non-cyclic motion commands (i.e., commands that are added to the motion queue).

The SetPoint is a bit that enables or disables the robot's reception of motion commands from the cyclic data. When this bit is cleared, the robot ignores the CommandID and the MoveID fields. The MoveID field determines if commands are cyclic (MoveID is 0) or non-cyclic (MoveID is not 0, one new command being queued every time the MoveID value is changed).

Always wait for the robot to acknowledge the current MoveID before changing the cyclic data (MoveID, CommandID or the motion command arguments). Otherwise, a motion command may be lost, or the arguments may not match the desired CommandID.

Always change the MoveID after updating CommandID and the corresponding arguments, otherwise the robot may receive a mix of old and new CommandID and arguments.

3.3.3 Adding non-cyclic motion commands to the motion queue (position mode)

Non-cyclic motion commands (MoveJoints, MovePose, MoveLin, Delay, SetJointVel, SetConf, etc.) are added to the motion queue and processed later (once previous commands have been completed). They are sent by changing the MoveID field to a different non-zero integer value (while SetPoint is 1).

When MoveID is changed, the motion command defined in the CommandID field will be added to the motion queue. Once the robot receives a new motion command, it will acknowledge it by updating its own MoveID field to match your MoveID value.

The following sequence must be followed:

- Initially (at application startup), clear both the MoveID and SetPoint fields.
- Then, to add a motion command to the robot's motion queue,
  - set the CommandID to the value corresponding to the desired command,
  - enter the desired values for the command arguments,
  - change MoveID to a different non-zero integer value,
  - set SetPoint to 1.
- To stop the robot immediately, set the PauseMotion bit or the ClearMotion bit.
- Otherwise, the robot will automatically stop moving once the motion queue is empty.

Remember that the MoveID and CommandID fields, as well as the command arguments must not be changed until the robot acknowledges the previous motion command, by returning the corresponding MoveID in its cyclic data.

3.3.4 Sending cyclic motion commands (velocity mode)

The only cyclic motion commands are the three velocity mode commands: MoveJointsVel, MoveLinVelWRF, MoveLinVelTRF. They can be sent every cycle, with MoveID kept at 0 and SetPoint set to 1.

The following sequence must be followed:

- Initially (at application startup), clear both the MoveID and SetPoint fields.
- To start moving the robot,
  - set CommandID to the ID corresponding to the desired velocity mode command,
  - enter the desired values for the command six arguments.
  - set SetPoint to 1.
- To change the velocity at any time (at every cycle, if needed), simply change the six arguments of the command.
- To stop the robot, you must reset SetPoint to 0.

Using position mode Command IDs in cyclic mode (i.e., with MoveID set to 0, and SetPoint set to 1) will quickly fill up the motion queue with copies of the same command, one per cycle, which is certainly not the desired result.

3.4 Cyclic data that can be sent to the robot

The EtherCAT and EtherNet/IP cyclic data contains the following fields for data that can be sent to the robot. See Sections 4 and 5 for detailed protocol-specific information about each field.

3.4.1 Robot control

Table 4 lists the five fields that control the status of the robot.

Table 4: Robot control fields

Field	Type	Description
Deactivate	Bool (action)	Deactivates the robot when set to 1.
Activate	Bool (action)	Activates the robot when set to 1 (and Deactivate bit is 0).
Home	Bool (action)	Homes the robot when set to 1 (if the robot is activated but not homed).
Reset Error	Bool (action)	Resets the error when set to 1.
Simulation Mode	Bool (state)	Enables (when set to 1) or disables (when reset to 0) the simulation mode (if the robot is not activated).

3.4.2 Motion control

Table 5 lists the five fields that control the motion of the robot.

Table 5: Motion control fields

Field	Type	Description
MoveID	Integer	A user-defined number, the change of which triggers the addition of the command specified in CommandID to the motion queue.
SetPoint	Bool (state)	Has to be set to 1 for motion commands to be sent to the robot.
Pause Motion	Bool (state)	Puts the robot in pause without clearing the commands in the queue. Motion is resumed once both the Pause Motion and Clear Motion bits are reset to 0.
Clear Motion	Bool (action/state)	Clears the motion queue and puts the robot in pause. Motion is resumed once both the Pause Motion and the Clear Motion bits are reset to 0.
Reset PStop	Bool (state)	Resets the protective stop 2.

3.4.3 Motion parameters

The motion parameters include the CommandID and six arguments for the motion command. These are illustrated in Table 6. The list of available CommandID values is given in Table 7.

Table 6: Motion parameters

Field	Type	Description
CommandID	Integer	Motion command ID (see Table 7).
Argument 1	Real	the first argument of the motion command, if applicable, as described in Section 2.
Argument 2	Real	the second argument of the motion command, if applicable, as described in Section 2.
Argument 3	Real	the third argument of the motion command, if applicable, as described in Section 2.
Argument 4	Real	the fourth argument of the motion command, if applicable, as described in Section 2.
Argument 5	Real	the fifth argument of the motion command, if applicable, as described in Section 2.
Argument 6	Real	the sixth argument of the motion command, if applicable, as described in Section 2.

Table 7: List of CommandID numbers

ID	Description
0	No movement: all six arguments are ignored.
1	MoveJoints, all six arguments are in degrees.
2	MovePose, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
3	MoveLin, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
4	MoveLinRelTRF, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
5	MoveLinRelWRF, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
6	Delay, argument 1 is the pause in seconds.
7	SetBlending, argument 1 is the percentage of blending, from 0 or 100.
8	SetJointVel, argument 1 is the percentage of maximum joint velocities, from 0.001 to 100.
9	SetJointAcc, argument 1 is the percentage of maximum joint accelerations, from 0.001 to 150.
10	SetCartAngVel, argument 1 is the Cartesian angular velocity limit, from 0.001 to 300, measured in °/s.
11	SetCartLinVel, argument 1 is the linear velocity limit for the TCP, from 0.001 to 1,000, measured in mm/s.
12	SetCartAcc, argument 1 is the percentage of maximum Cartesian accelerations, ranging from 0.001 to 100.
13	SetTRF, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
14	SetWRF, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.

Table 7: List of CommandID number (continued)

ID	Description
15	SetConf, arguments 1, 2, and 3 are −1 or 1.
16	SetAutoConf, argument 1 is 0 or 1.
17	SetCheckpoint, argument 1 in an integer number, ranging from 1 to 8,000.
18	Gripper, argument 1 is 0 for GripperClose, and 1 for GripperOpen.
19	SetGripperVel, argument 1 is the percentage of maximum finger velocity (100 mm/s), from 5 to 100.
20	SetGripperForce, argument 1 is the percentage of maximum grip force (40 N), from 5 to 100.
21	MoveJointsVel, all six arguments are in °/s.
22	MoveLinVelWRF, arguments 1, 2, 3 are in mm/s and 4, 5, 6 are in °/s.
23	MoveLinVelTRF, arguments 1, 2, 3 are in mm/s and 4, 5, 6 are in °/s.
24	SetVelTimeout, argument 1 is in seconds.
25	SetConfTurn, argument 1 is an integer number, ranging from −100 to 100.
26	SetAutoConfTurn, argument 1 is 0 or 1.
27	SetTorqueLimits, all six arguments are percentage of maximum joint torque, from 0.001 to 100.
28	SetTorqueLimitsCfg, argument 1 is 0 or 1.
100	StartProgram (EtherNet/IP only), argument 1 is the number of the offline program to start, from 1 to 500.

3.5 Cyclic data received from the robot

The robot sends data at every cycle. It reports the status of the robot and the status of the motion, as described in Tables 8 and 9, respectively. It also reports the status of the gripper, as explained in Table 10. In addition, the robot sends feedback about its target end-effector pose (the same data as reported by the TCP/IP command GetRtTargetCartPos), the target inverse kinematics configuration (same as the combination of GetRtTargetConf and GetRtTargetConfTurn), the target joint set (same as GetRtTargetJointPos), and the target joint velocities (same as GetRtTargetJointVel). Finally, the robot reports that actual joint torques (same as GetRtJointTorq) and and the acceleration in link 5 of the robot (same as GetRtAccelerometer).

Currently, when using EtherCAT and EtherNet/IP, the robot does not report the target end-effector Cartesian velocity, nor does it return the data reported by the commands GetRtCartPos, GetRtCartVel, GetRtConf, GetRtConfTurn, GetRtJointPos, GetRtJointVel.

Table 8: Robot status

Field	Type	Description
Error	Integer	Indicates the error number (see Tables 1 and 3) or 0, if there is no error.
Busy	Bool	True only while the robot is being activated, homed or deactivated.
Activated	Bool	Indicates whether the motors are on (powered).
Home	Bool	Indicates whether the robot is homed and ready to receive motion commands.
SimActivated	Bool	Indicates whether the robot simulation mode is activated.

Table 9: Motion status

Field	Type	Description
Checkpoint ID	Integer	Indicates the last checkpoint number reached (the value stays unchanged until another checkpoint number is reached). See Section 2.1.17 for a detailed description of checkpoints.
MoveID	Integer	Acknowledges the MoveID of the last motion command queued for execution.
FIFO space	Integer	The number of commands that can be added to the robot's motion queue at any given time (the maximum is 13,000). If 0 (too many commands sent), subsequent commands will be ignored.
Paused	Bool	Indicates whether the motion is paused. This bit will remain set (and robot will remain paused) as long as motion control bits Pause or Clear Motion remain set. Motion will resume once both motion control Pause and Clear Motion bits become 0.
EOB	Bool	The End of Block (EOB) bit is true only when the robot is not moving and there is no motion command left in the motion queue. Note that the EOB bit may be raised before all sent commands have been completed, due to network or processing delays. Therefore, do not rely on this flag to be informed when a sequence of movements has been completed (use a checkpoint instead).
EOM	Bool	The End Of Motion (OOM) bit is true if the robot is not moving. Note that the EOM bit may be raised between two consecutive motion commands. Therefore, do nott rely on this flag to be informed when a sequence of movements has been completed (use a checkpoint instead).

Table 9: Motion status (continued)

Field	Type	Description
Cleared	Bool	Indicates whether the motion queue is cleared. If the queue is cleared, the robot is not moving. This bit will remain true (and robot will remain paused) as long as the motion control bit Clear Motion remains set. Motion will resume once both motion control Pause and Clear Motion bits become 0.
PStop 2	Bool	Indicates whether the protective stop (Cat. 2) is set.
Excessive Torque	Bool	Indicates whether a joint torque is exceeding the corresponding user-defined torque limit.
Offline program ID	Integer	EtherNet/IP only. ID of the offline program currently running (0 if none).

Table 10: Gripper status

Field	Type	Description
Detected	Bool	True if the gripper is enabled and detected.
Homed	Bool	True if the gripper is homed.
Holding part	Bool	True if the gripper is holding a part (i.e., the gripper is forcing but is not fully open or closed).
Limit reached	Bool	True if the fingers are fully open or closed. Note that these conditions correspond to the limits detected during the homing of the gripper.
Overload	Bool	True if the gripper is overheated.

4 Communicating over EtherCAT

EtherCAT is an open real-time Ethernet protocol originally developed by Beckhoff Automation. When communicating with the Meca500 over EtherCAT, you can obtain guaranteed response times of 1 ms (as of firmware 8.4). Furthermore, you no longer need to parse strings as when using the TCP/IP protocol.

4.1 Overview

4.1.1 Connection types

If using EtherCAT, you can connect several Meca500 robots in different network topologies, including line, star, tree, or ring, since each robot has a unique node address. This enables targeted access to a specific robot even if your network topology changes.

4.1.2 ESI file

The EtherCAT Slave Information (ESI) XML file for the Meca500 robot can be found in the zip file that contains your robot's firmware update. These zip files are available in the Downloads section of our web site.

4.1.3 Enabling EtherCAT

The default communication protocol of the robot is the Ethernet TCP/IP protocol. The latter is the protocol needed for jogging the robot through its web interface. To switch to the EtherCAT communication protocol, you must connect to the robot via the TCP/IP protocol first from an external client (e.g., a PC). If you are already connected, you must deactivate the robot (by sending the command DeactivateRobot). Then, you must simply send the SwitchToEtherCAT command.

As soon as the robot receives this command, the Ethernet TCP/IP connection LED (i.e., #1 or #2 in Fig. 10) will go off, then turn back on. This means that the robot is now in EtherCAT mode. Now, it is possible to connect to an EtherCAT master. Note, however, than until EtherCAT is disabled, TCP/IP or EtherNet/IP communication is not possible (e.g., you cannot use the robot's web interface). To disable EtherCAT, use the Communication mode SDO (Section 4.2.13) or perform a factory reset (by keeping the power button on the robot's base pressed for a few seconds during restart).

4.1.4 LEDs

When EtherCAT communication is enabled, the three LEDs on the outer edge of the robot's base (Fig. 10) communicate the state of the EtherCAT connection, as summarized in Table 11.

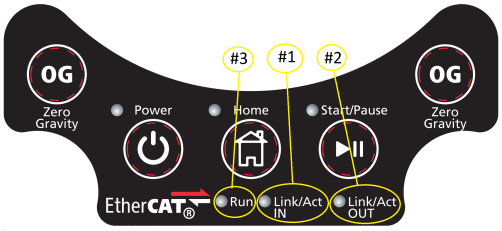


Figure 10: EtherCAT LEDs

Table 11: Meanings of the EtherCAT related LEDs

LED	Name	LED State	EtherCAT state
#1	IN port link	On	Link is active but there is no activity
		Blinking	Link is active and there is activity
		Off	Link is inactive
#2	OUT port link	On	Link is active but there is no activity
		Blinking	Link is active and there is activity
		Off	Link is inactive
#3	Run	On	Operational
		Blinking	Pre-Operational
		Single flash	Safe-Operational
		Off	Init

4.2 Object dictionary

This section describes all objects available for interacting with the Meca500. Please refer to Chapter 3 for a description of these objects and their fields. The current section simply defines how these objects are mapped to EtherCAT cyclic Process Data Object (PDO). There are also two EtherCAT-specific Service Data Objects (SDO), presented in the last two subsections.

In the tables of this section, SI stands for subindex, and "O. code" for "Object code".

4.2.1 Robot control

This object controls the robot's initialization and simulation. Table 12 describes the object's five indices.

Table 12: Robot control object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
7200h		Record		Robot control				RO	none
	1	Variable	BOOL	Deactivate	0	0	1	RW	1600h:1
	2	Variable	BOOL	Activate	0	0	1	RW	1600h:2
	3	Variable	BOOL	Home	0	0	1	RW	1600h:3
	4	Variable	BOOL	Reset error	0	0	1	RW	1600h:4
	5	Variable	BOOL	Sim mode	0	0	1	RW	1600h:5

4.2.2 Motion control

This object controls the actual robot movement. Table 13 describes the object's five indices.

Table 13: Motion control object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
7310h		Record		Motion control				RO	none
	1	Variable	UINT	Move ID	0	0	65,535	RW	1601h:1
	2	Variable	BOOL	SetPoint	0	0	1	RW	1601h:2
	3	Variable	BOOL	Pause	0	0	1	RW	1601h:3
	4	Variable	BOOL	Clear move	0	0	1	RW	1601h:4
	5	Variable	BOOL	ResetPStop	0	0	1	RW	1601h:5

4.2.3 Movement

The movement object is a pair of indices and regroups the main motion commands. The first index is the ID number indicating the motion command, while the second index has six subindices corresponding to the arguments of the motion command, as described in Table 14.

Table 14: Movement object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
7305h		Variable	UDINT	Motion command	0	0	24	RO	1602h.1
7306h		Array		Arguments				RO	none
	1	Variable	⋆		†	†	†	RW	1602h.2
	2	Variable	⋆		†	†	†	RW	1602h.3
	3	Variable	⋆		†	†	†	RW	1602h.4
	4	Variable	⋆		†	†	†	RW	1602h.5
	5	Variable	⋆		†	†	†	RW	1602h.6
	6	Variable	⋆		†	†	†	RW	1602h.7

⋆ REAL or BOOL, depending on the value of index 7305h.
† depending on the value of index 7305h (refer to Table 7).

4.2.4 Robot status

The structure of the robot status object is described in Table 15.

Table 15: Robot status object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6010h		Record		Robot status				RO	none
	1	Variable	UINT	Error	n/a	0	65,535	RO	1A00h.1
	2	Variable	BOOL	Busy	n/a	0	1	RO	1A00h.2
	3	Variable	BOOL	Activated	n/a	0	1	RO	1A00h.3
	4	Variable	BOOL	Homed	n/a	0	1	RO	1A00h.4
	5	Variable	BOOL	SimActivated	n/a	0	1	RO	1A00h.5

4.2.5 Motion status

The structure of the motion status object is described in Table 16.

Table 16: Motion status object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6015h		Record		Motion status				RO	none
	1	Variable	UDINT	Checkpoint	n/a	0	8,000	RO	1A01h.1
	2	Variable	UINT	Move ID	n/a	0	65,535	RO	1A01h.2
	3	Variable	UINT	FIFO space	n/a	0	13,000	RO	1A01h.3
	4	Variable	BOOL	Paused	n/a	0	1	RO	1A01h.4
	5	Variable	BOOL	EOB	n/a	0	1	RO	1A01h.5
	6	Variable	BOOL	EOM	n/a	0	1	RO	1A01h.6
	7	Variable	BOOL	Cleared	n/a	0	1	RO	1A01h.7
	8	Variable	BOOL	PStop	n/a	0	1	RO	1A01h.8
	9	Variable	BOOL	Excessive Torque	n/a	0	1	RO	1A01h.9

4.2.6 Gripper status

The structure of the gripper status object is described in Table 17.

Table 17: Gripper status object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6045h		Array		Gripper status				RO	none
	1		BOOL	Detected	n/a	0	1	RO	1A07h.1
	2		BOOL	Homed	n/a	0	1	RO	1A07h.2
	3		BOOL	Holding part	n/a	0	1	RO	1A07h.3
	4		BOOL	Limit reached	n/a	0	1	RO	1A07h.4
	5		BOOL	Error	n/a	0	1	RO	1A07h.5
	6		BOOL	Overload	n/a	0	1	RO	1A07h.6

4.2.7 Joint set

The structure of the joint set object is described in Table 18. The data is the same as that returned by TCP/IP command GetRtTargetJointPos.

Table 18: Joint set object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6030h		Array		Joint set				RO	none
	1		REAL		n/a	−175	175	RO	1A02h.1
	2		REAL		n/a	−70	90	RO	1A02h.2
	3		REAL		n/a	−135	70	RO	1A02h.3
	4		REAL		n/a	−170	170	RO	1A02h.4
	5		REAL		n/a	−115	115	RO	1A02h.5
	6		REAL		n/a	−36,000	36,000	RO	1A02h.6

4.2.8 End-effector pose

The structure of the end-effector pose object is described in Table 19. The data is the same as that returned by TCP/IP command GetRtTargetCartPos.

Table 19: End-effector pose object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6031h		Array		End-effector pose				RO	none
	1		REAL		n/a	−3.4E38	3.4E38	RO	1A03h.1
	2		REAL		n/a	−3.4E38	3.4E38	RO	1A03h.2
	3		REAL		n/a	−3.4E38	3.4E38	RO	1A03h.3
	4		REAL		n/a	−3.4E38	3.4E38	RO	1A03h.4
	5		REAL		n/a	−3.4E38	3.4E38	RO	1A03h.5
	6		REAL		n/a	−3.4E38	3.4E38	RO	1A03h.6

4.2.9 Configuration

The structure of the configuration object is described in Table 20. The data is the same as that returned by the combination of the TCP/IP commands GetRtTargetConf and GetRtTargetConfTurn.

Table 20: Configuration object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6046h		Array		Configuration				RO	none
	1		INT8	c_s	n/a	−1	1	RO	1A08h.1
	2		INT8	c_e	n/a	−1	1	RO	1A08h.2
	3		INT8	c_w	n/a	−1	1	RO	1A08h.3
	4		INT8	c_t	n/a	−100	100	RO	1A08h.4

4.2.10 Joint velocities

The structure of the joint velocities object is described in Table 21). The data is the same as that returned by the TCP/IP command GetRtTargetJointVel.

Table 21: Joint velocities object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6032h		Array		Joint velocities				RO	none
	1		REAL		n/a	−150	150	RO	1A04h.1
	2		REAL		n/a	−150	150	RO	1A04h.2
	3		REAL		n/a	−180	180	RO	1A04h.3
	4		REAL		n/a	−300	300	RO	1A04h.4
	5		REAL		n/a	−300	300	RO	1A04h.5
	6		REAL		n/a	−500	500	RO	1A04h.6

4.2.11 Torque ratios

The structure of the torque ratios object is described in Table 22. The data is the same as that returned by the TCP/IP command GetRtJointTorq.

Table 22: Torque ratios object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6033h		Array		Torque ratios				RO	none
	1		REAL		n/a	−100	100	RO	1A05h.1
	2		REAL		n/a	−100	100	RO	1A05h.2
	3		REAL		n/a	−100	100	RO	1A05h.3
	4		REAL		n/a	−100	100	RO	1A05h.4
	5		REAL		n/a	−100	100	RO	1A05h.5
	6		REAL		n/a	−100	100	RO	1A05h.6

4.2.12 Accelerometer

The structure of the accelerometer object is described in Table 23. The data is the same as that returned by the TCP/IP command GetRtJointTorq(5).

Table 23: Accelerometer object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6040h		Array		Accelerometer				RO	none
	1		DINT	X	n/a	−32,000	32,000	RO	1A06h.1
	2		DINT	Y	n/a	−32,000	32,000	RO	1A06h.2
	3		DINT	Z	n/a	−32,000	32,000	RO	1A06h.3

4.2.13 Communication mode (SDO)

When EtherCAT is enabled, subindex 1 of this SDO is equal to 2 (see Table 24). In EtherCAT, you must be connected to the robot via the Ethernet port ECAT IN. Currently, you cannot change the communication mode for port ECAT OUT and therefore subindex 2 of this SDO is ignored. Regardless of the value of subindex 2, the communication mode of port ECAT OUT will be the same as that of port ECAT IN. To switch both ports to TCP/IP, change the value of subindex 1 to 1.

Table 24: Communication mode SDO

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
8000h		Record		Commun. mode				RO	n/a
	1	Variable	USINT	Port In	1	1	2	RW	n/a
	2	Variable	USINT	Port Out	1	1	2	RW	n/a

4.2.14 Brakes (SDO)

The Brakes SDO controls the brakes of joints 1, 2 and 3 (Table 25). When the robot is activated, the brakes are always disengaged. As soon as the robot is deactivated, the brakes are automatically engaged. When the robot is powered on but deactivated, you can disengage the brakes by setting the Brakes SDO to 0. Do this with great caution, because the robot will fall down under the effects of gravity. You can reengage the brakes by setting the Brakes SDO to 1 or by powering off the robot.

The value of the Brakes SDO is ignored when the robot is activated. However, be careful when deactivating the robot, because if the value of the Brakes SDO was 0 during the deactivation, the brakes will not be engaged and the robot will fall down.

Table 25: Brakes SDO

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
8010h		Variable	USINT	Brakes	1	0	1	RW	n/a

4.3 PDO mapping

The process data objects (PDOs) provide the interface to the application objects. The 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 Meca500) to the EtherCAT master.

In the previous section, we listed the PDOs that are assigned to some of the objects. Here, in Tables 26 and 27, we list all PDOs and recall the objects they are assigned to.

Table 26: RxPDOs

PDO	Object(s)	Name	Note
1600h	7200h	Robot control	see Table 12
1601h	7310h	Motion control	see Table 13
1602h	7305h, 7306h	Movement	see Table 14

Table 27: TxPDOs

PDO	Object	Name	Note
1A00h	6010h	Robot status	see Table 15
1A01h	6015h	Motion status	see Table 16
1A02h	6030h	Angular position	see Table 18
1A03h	6031h	Cartesian position	see Table 19
1A04h	6032h	Angular velocities	see Table 21
1A05h	6033h	Torque ratios	see Table 22
1A06h	6040h	Accelerometer	see Table 23
1A07h	6045h	Gripper status	see Table 17
1A08h	6046h	Configuration	see Table 20

5 Communicating over EtherNet/IP

Certified by ODVA, the Meca500 is compatible with the EtherNet/IP protocol. A common industry standard, it can be used with many different PLC brands. Tested to work at 10 ms, faster times are also possible. The Meca500 typically uses implicit (cyclic) messaging.

Refer to our Support Center for specific PLC examples.

5.1 Overview

5.1.1 Connection types

When using EtherNet/IP, you can connect several Meca500 robots in the same way as with TCP/IP. Either Ethernet port on the base of the robot can be used. Meca500 robots can be either daisy-chained together or connected in a star pattern. The two ports on the Meca500 act as a switch in EtherNet/IP mode.

5.1.2 EDS file

The Electronic Data Sheet (EDS) file for the Meca500 robot can be found in the zip file that contains your robot's firmware update. These zip files are available in the Downloads section of our web site.

5.1.3 Forward open exclusivity

The Meca500 robot will allow only one "forward open" connection that controls the output assembly. If another controller attempts to establish a "forward open" connection when the robot is already being a controller, the connection will be refused. If a controller attempts to establish a "forward open" connection while a user is connected to the robot through the web interface, the connection will be refused too. Similarly, if a controller has established a "forward open" connection with the robot, you can no longer operate the robot through the web interface. However, you can use the web interface for real-time monitoring.

5.1.4 Enabling EtherNet/IP

To enable the EtherNet/IP communication protocol, you must connect to the robot via the TCP/IP protocol first from an external client (e.g., a PC). If you are already connected, you must deactivate the robot (by sending the command DeactivateRobot). Then, you must simply send the EnableEtherNetIP(1) command. This is a persistent command, so it only

needs to be set once. To disable EtherNet/IP, you need to send the EnableEtherNetIP(0) command.

Note that EtherNet/IP can be left permanently enabled as it does not prevent using the TCP/IP protocol, unlike EtherCAT and the SwitchToEtherCAT command. Of course, that does not mean that you can have both a web interface connection on a PC and an EtherNet/IP connection on a PLC at the same time; the robot will accept only one master at the time. The appropriate error code (TCP/IP or EtherNet/IP) is returned if the connection is refused because another master already controls the robot (error [3001] in TCP/IP and extended status error 0x106, Ownership Conflict, in EtherNet/IP).

5.2 Output Tag Assembly

The Output Tag Assembly has an Instance of 150 with a size 44-byte array, as detailed in Table 28. Please refer to Chapter 3 for a description of the different objects and their fields. The following subsections simply define how these objects are mapped to EtherNet/IP output tag assembly (as also described in the EDS file).

Table 28: Output tag assembly

Bytes	Data Type	Name	Description
0–3	DWORD	Robot Control	Controls the robot's initialization and simulation
4–5	UINT	MoveID	User-defined ID of the command currently being sent
6–7	WORD	Motion Control	Controls the actual robot movement.
8–11	UDINT	Movement	Defines the type of command that is being sent
12–15	REAL	Arg. 1 for Mov.	Specifies argument 1 of the motion command
16–19	REAL	Arg. 2 for Mov.	Specifies argument 2 of the motion command
20–23	REAL	Arg. 3 for Mov.	Specifies argument 3 of the motion command
24–27	REAL	Arg. 4 for Mov.	Specifies argument 4 of the motion command
28–31	REAL	Arg. 5 for Mov.	Specifies argument 5 of the motion command
32–35	REAL	Arg. 6 for Mov.	Specifies argument 6 of the motion command
36–39	DINT	Host Time	Number of seconds since Epoch Time.
40–43	DWORD	Brake Control	Controls the brakes on joints 1, 2 and 3

5.2.1 Robot control tag

The robot control tag controls the robot's initialization and simulation. Table 29 describes the bits of this tag.

Table 29: Robot control tag

Bytes	Data Type	Bits 5–31	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0–3	DWORD	Unused	Sim. mode	Reset error	Home	Activate	Deactivate

5.2.2 MoveID tag

The MoveID tag (Table 40) contains the distinct user-defined ID number associated with each motion command sent to the robot.

Table 30: MoveID tag

Bytes	Data Type	Name	Minimum	Maximum
4–5	UINT	MoveID	0	65,535

5.2.3 Motion control tag

This tag controls the actual robot movement. Table 31 describes the tag's bits.

Table 31: Motion control tag

Bytes	Data Type	Bits 4–15	Bit 3	Bit 2	Bit 1	Bit 0
6–7	WORD	Unused	Reset PStop	Clear Motion	Pause Motion	Set Point

5.2.4 Motion command group of tags

This group of tags will define the type of motion command that is being sent to the robot and the arguments of the respective command. The motion command tag (Table 32) contains the ID of the motion command (see Table 7). The motion command argument tags contain the arguments of the motion command (Table 33).

Table 32: Motion command tag

Bytes	Data Type	Name	Possible values
8–11	UDINT	Motion command ID	0, 1, 2, ..., 24 or 100

Table 33: Motion command arguments tags

Bytes	Data Type	Name
12–15	Real	Motion command argument 1
16–19	Real	Motion command argument 2
20–23	Real	Motion command argument 3
24–27	Real	Motion command argument 4
28–31	Real	Motion command argument 5
32–35	Real	Motion command argument 6

5.2.5 Host time tag

This tag (Table 34) sets the robot's host time (real-time-clock) to a specified number of seconds since 00:00:00 UTC January 1, 1970. This is equivalent to the TCP/IP command SetRTC. The tag is ignored if its value is 0.

Table 34: Host time tag

Bytes	Data Type	Name	Minimum	Maximum
36–39	DINT	Host Time	0	$2^{32} - 1$

5.2.6 Brakes control tag

The Brakes control tag actives or deactivates the brakes of joints 1, 2 and 3 (all three at the same time). When the robot is activated, the brakes are always disengaged. As soon as the robot is deactivated the brakes control tag will be applied. When the robot is powered on but deactivated you can disengage the brakes by using this tag. Do this with great caution, because the robot might fall down under the effects of gravity.

The Brakes control tag is described in Table 35. The description for bits 0 and 1 are as follows:

0. Enable Brake Control: If set to 1, the brakes control is enabled. Robot must be deactivated for brakes control. This bit will be ignored if the robot is activated. The purpose of this bit is to ensure that the brakes don't get inadvertently disabled if the output assembly contains all zeroes.
1. Engage Brakes: If set to 1, the brakes are engaged. If reset to 0, the brakes are disengaged and the robot might fall down under the effects of gravity. Therefore, make sure to first assign the desired value to this bit, and then set the Enable Brake Control bit to 1.

Table 35: Brakes control tag

Bytes	Data Type	Bits 2–31	Bit 1	Bit 0
40-43	DWORD	Unused	Engage Brakes	Enable Brake Control

5.3 Input Tag Assembly

The Input Tag Assembly has an Instance of 100 with a size 168-byte array, as detailed in Table 36. Please refer to Section 3.5 for a description of the different objects and their fields. The following subsections simply define how these objects are mapped to EtherNet/IP input tag assembly (as also described in the EDS file).

Table 36: Input tag assembly

Bytes	Data Type	Data	Description
0–1	WORD	Robot status	State of the robot (Busy, Activated, Home, Sim)
2–3	UINT	Error code	The code of the last error
4–7	UDIN	Checkpoint	Last checkpoint number that was reached
8–9	UINT	MoveID	ID of the last motion command
10–11	UINT	FIFO space	Space left in the FIFO buffer
12–13	WORD	Motion status	Actual state of the robot's movement
14–15	UINT	Offline program ID	ID of the offline program currently running.
16–19	REAL	Position of J1	θ_1
20–23	REAL	Position of J2	θ_2
24–27	REAL	Position of J3	θ_3
28–31	REAL	Position of J4	θ_4
32–35	REAL	Position of J5	θ_5
36–39	REAL	Position of J6	θ_6
40–43	REAL	Coordinate x	Coordinate x of TRF w.r.t. WRF
44–47	REAL	Coordinate y	Coordinate y of TRF w.r.t. WRF
48–51	REAL	Coordinate z	Coordinate z of TRF w.r.t. WRF
52–55	REAL	Euler angle alpha	Euler angle α of TRF w.r.t. WRF
56–59	REAL	Euler angle beta	Euler angle β of TRF w.r.t. WRF
60–63	REAL	Euler angle gamma	Euler angle γ of TRF w.r.t. WRF
64–67	REAL	Velocity of J1	Velocity of joint 1
68–71	REAL	Velocity of J2	Velocity of joint 2
72–75	REAL	Velocity of J3	Velocity of joint 3
76–79	REAL	Velocity of J4	Velocity of joint 4
80–83	REAL	Velocity of J5	Velocity of joint 5

Table 36: Input tag assembly (continued)

Bytes	Data Type	Data	Description
84–87	REAL	Velocity of J6	Velocity of joint 6
88–91	REAL	Torque ratio of J1	Torque ratio of joint 1
92–95	REAL	Torque ratio of J2	Torque ratio of joint 2
96–99	REAL	Torque ratio of J3	Torque ratio of joint 3
100–103	REAL	Torque ratio of J4	Torque ratio of joint 4
104–107	REAL	Torque ratio of J5	Torque ratio of joint 5
108–111	REAL	Torque ratio of J6	Torque ratio of joint 6
112–115	REAL	Accelerometer x	Accelerometer data along x axis
116–119	REAL	Accelerometer y	Accelerometer data along y axis
120–123	REAL	Accelerometer z	Accelerometer data along z axis
124–127	DWORD	Gripper status	Gripper status
128	SINT	Conf. parameter cs	Configuration parameter c_s
129	SINT	Conf. parameter ce	Configuration parameter c_e
130	SINT	Conf. parameter cw	Configuration parameter c_w
131	SINT	Conf. parameter ct	Configuration parameter c_t
132-135	BYTES	Reserved bytes	Not used
136-143	BYTES	Global timestamp	Free-running timestamp, in microseconds
144-151	BYTES	Last update timestamp	Last position update in output assembly
152-155	BYTES	Output change count	Number of times that output tag assembly data changes were detected by the robot
156-167	BYTES	Reserved bytes	Not used

5.3.1 Robot status tag

The structure of the robot status tag is described in Table 37.

Table 37: Robot status tag

Bytes	Data Type	Bits 5–15	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0–1	WORD	Unused	BrakesEngaged	Sim. mode	Homed	Activated	Busy

5.3.2 Error code tag

The structure of the error code tag is described in Table 38.

Table 38: Error code tag

Bytes	Data Type	Name	Minimum	Maximum
2–3	UINT	Error Code	0	65,535

5.3.3 Checkpoint tag

The structure of the checkpoint tag is described in Table 39.

Table 39: Checkpoint tag

Bytes	Data Type	Name	Minimum	Maximum
4–7	UDINT	Checkpoint	0	8,000

5.3.4 MoveID tag

The structure of the MoveID tag is described in Table 40.

Table 40: MoveID tag

Bytes	Data Type	Name	Minimum	Maximum
8–9	REAL	MoveID	0	65,535

5.3.5 FIFO space tag

The structure of the FIFO space tag is described in Table 41.

Table 41: FIFO space tag

Bytes	Data Type	Name	Minimum	Maximum
10–11	UINT	FIFO space	0	13,000

5.3.6 Motion status tag

The structure of the motion status tag is described in Table 42.

Table 42: Motion status tag

Bytes	Data Type	Bits 6–15	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
12–13	WORD	Unused	Exc. Torque	PStop 2	Cleared	EOM	EOB	Paused

5.3.7 Offline program ID

This tag indicates the ID of the offline program currently running (Table 43).

Table 43: Offline program tag

Bytes	Data Type	Name	Minimum	Maximum
14–15	UINT	Offline program ID	1	500

5.3.8 Joint set

The structure of the joint set tag is described in Table 44. The data is the same as that returned by TCP/IP command GetRtTargetJointPos.

Table 44: Joint set tag

Bytes	Data Type	Name	Minimum	Maximum
16–19	REAL	Position of J1	−175.000	175.000
20–23	REAL	Position of J2	−70.000	90.000
24–27	REAL	Position of J3	−135.000	70.000
28–31	REAL	Position of J4	−170.000	170.000
32–35	REAL	Position of J5	−115.000	115.000
36–39	REAL	Position of J6	−36,000.000	36,000.000

5.3.9 End-effector pose

The structure of the end-effector pose tag is described in Table 45. The data is the same as that returned by TCP/IP command GetRtTargetCartPos.

Table 45: End-effector pose tag assembly

Bytes	Data Type	Name
40–43	REAL	Coordinate x
44–47	REAL	Coordinate y
48–51	REAL	Coordinate z
52–55	REAL	Euler angle alpha
56–59	REAL	Euler angle beta
60–63	REAL	Euler angle gamma

5.3.10 Joint velocities

The structure of the joint velocities tag is described in Table 46. The data is the same as that returned by the TCP/IP command GetRtTargetJointVel.

Table 46: Joint velocities tag

Bytes	Data Type	Name	Minimum	Maximum
64–67	REAL	Velocity of J1	−150.000	150.000
68–71	REAL	Velocity of J2	−150.000	150.000
72–75	REAL	Velocity of J3	−180.000	180.000
76–79	REAL	Velocity of J4	−300.000	300.000
80–83	REAL	Velocity of J5	−300.000	300.000
84–87	REAL	Velocity of J6	−500.000	500.000

5.3.11 Joint torque ratios

The structure of the torque ratios tag is described in Table 47. The data is the same as that returned by the TCP/IP command GetRtJointTorq.

Table 47: Joint torque ratio tag

Bytes	Data Type	Name	Minimum	Maximum
88–91	REAL	Torque ratio of J1	−100.000	100.000
92–95	REAL	Torque ratio of J2	−100.000	100.000
96–99	REAL	Torque ratio of J3	−100.000	100.000
100–103	REAL	Torque ratio of J4	−100.000	100.000
104–107	REAL	Torque ratio of J5	−100.000	100.000
108–111	REAL	Torque ratio of J6	−100.000	100.000

5.3.12 Accelerometer

The structure of the accelerometer tag is described in Table 48. The data is the same as that returned by the TCP/IP command GetRtJointTorq(5).

Table 48: Accelerometer tag

Bytes	Data Type	Name	Minimum	Maximum
112–115	REAL	Accelerometer x	−32.000	32.000
116–119	REAL	Accelerometer y	−32.000	32.000
120–123	REAL	Accelerometer z	−32.000	32.000

5.3.13 Gripper status tag

TThe structure of the gripper status object is described in Table 49.

Table 49: Gripper status tag

Bytes	Data Type	Bits 5–15	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
124–127	DWORD	Unused	Overl.	Limit reach.	Holding part	Homed	Detected

5.3.14 Configuration

The structure of the configuration tag is described in Table 50. The data is the same as that returned by the TCP/IP commands GetRtTargetConf and GetRtTargetConfTurn.

Table 50: Robot configuration tags

Bytes	Data Type	Name	Minumum	Maximum
128	SINT	Configuration parameter cs	−1	1
129	SINT	Configuration parameter ce	−1	1
130	SINT	Configuration parameter cw	−1	1
131	SINT	Configuration parameter ct	−100	100

5.3.15 Reserved bytes

Bytes 132 through 135, and 156 through 167, are not used currently.

Mecademic Inc.
1300 Saint-Patrick St
Montreal QC H3K 1A4
CANADA
www.mecademic.com