mecademic

Programming Manual

Meca500 (R3)

Robot Firmware: 8.1

Document Revision: A

June 15, 2020

Contents

This page is intentionally left blank

1 About this manual

This programming manual describes the key theoretical concepts related to industrial robots and the two 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 the TCP/IP or the EtherCAT protocol. 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, on our GitHub account.

The default communication method used by our robots is over TCP/IP and consists of a set of motion and requests commands to be sent to one of our robots, as well as a set of messages sent back by the robot. Therefore, in the following section, we will refer to some of these motion commands in explaining the key theoretical concepts related to industrial robots. Furthermore, the communication method based on the EtherCAT protocol and described in Section 4 is essentially a translation of our TCP/IP method. Thus, in Section 4, we do not describe again each concept but simply refer to Section 3.

In other words, even if you intend to use the EtherCAT protocol only, you must read every single page of this manual. However, before reading this programming manual, you must first read the Meca500 user manual.

2 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.

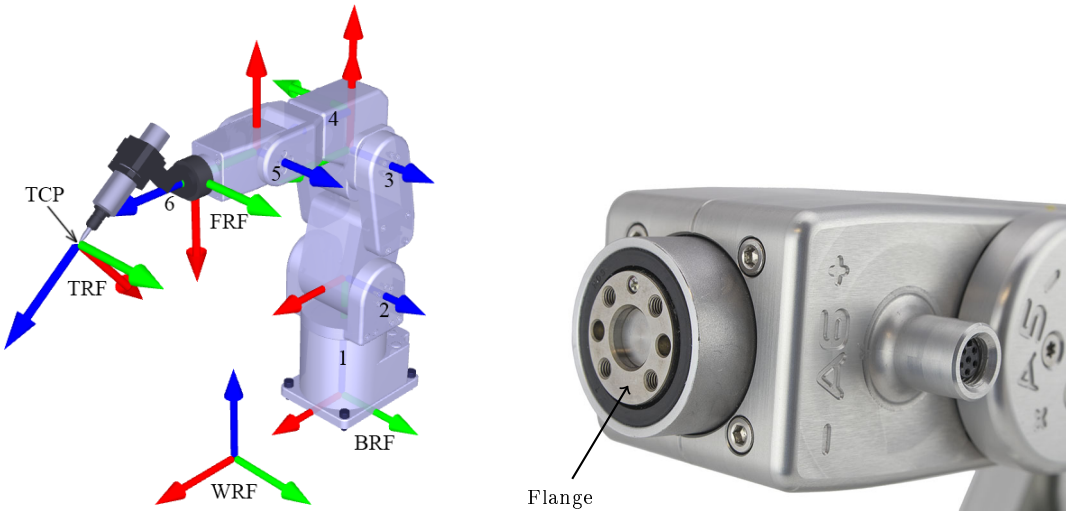
2.1 Definitions and conventions

2.1.1 Units

We use the International System of Units (SI), but for angles. Distances that are displayed to or given by the user are in millimeters (mm), angles in degrees (°) and time is in seconds (s).

2.1.2 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

2.1.3 Reference frames

At Mecademic, we use right-handed Cartesian coordinate systems (*reference frames*). The reference frames 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, while its origin lies on the bottom of the robot base. The x axis of the BRF is perpendicular to the front edge of the robot base and points forward. The BRF cannot be reconfigured.
- *WRF*: *World Reference Frame*. The robot main static reference frame. By default, it coincides with the BRF. It can be reconfigured with respect to (w.r.t.) the BRF using 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 reconfigured with respect to the FRF using 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.)

2.1.4 Joint angles and joint 6 revolution number

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:

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

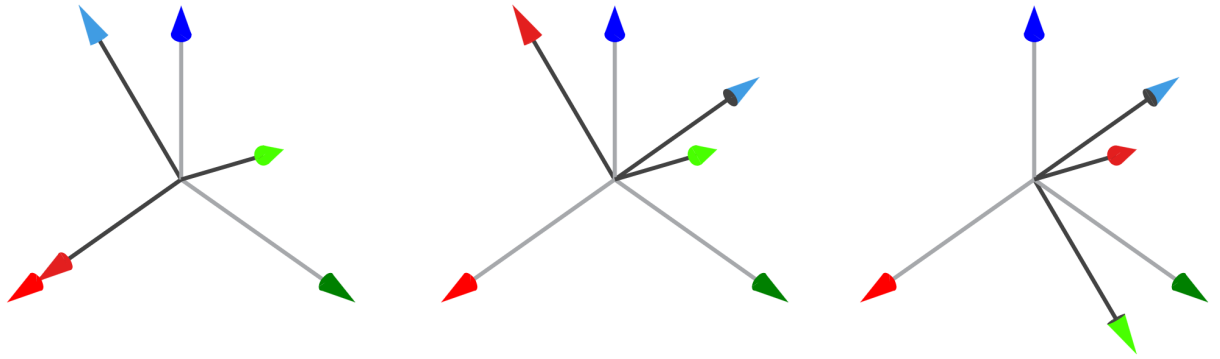
Joint 6 has no mechanical limits, but its software limits are ±100 revolutions. Note, however, that the normal working range of joint 6 is $-180^\circ \leq \theta_6 \leq 180^\circ$, as will be explained in Section 2.2.3. This closed interval will be referred to as a ±180° range. Finally, let us define the integer r as the *axis 6 revolution number*, such that $-180^\circ + r360^\circ < \theta_6 \leq 180^\circ + r360^\circ$.

2.1.5 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 placed, is by giving its set of joint angles. We will refer to this set at the *joint set*.

For example, in Fig. 1a, the joint set is $\{0^\circ, 0^\circ, 0^\circ, 0^\circ, 0^\circ, 0^\circ\}$, although, it could have been $\{0^\circ, 0^\circ, 0^\circ, 0^\circ, 0^\circ, 360^\circ\}$, 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 $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6 + r360^\circ\}$, where $-180^\circ < \theta_6 \leq 180^\circ$ and r is the axis 6 revolution number, correspond to the same robot posture. Therefore, a joint set has the same information as a robot posture AND an axis 6 revolution number.



(a) rotation of 45° about the x axis

(b) rotation of $-90°$ about the new y axis

(c) rotation of 90° about the new z axis

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

2.1.6 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 $\{45°, -90°, 90°\}$.

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).

Finally, as we will see later, note that the pose of the end-effector alone does not define unequivocally how the robot is placed.

2.2 Key concepts

2.2.1 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 very high accuracy, each motor must make one full revolution. This motion is the essential part of a procedure called *homing*.

During homing, if a joint is too close to one of its limits, it rotates slightly away. Then, all joints rotate simultaneously. Specifically, 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.

Finally, if your robot is equipped with Mecademic's gripper (MEGP 25), 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.

2.2.2 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 constant 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. 3). The higher the TCP speed, the more rounded the transition will be. You cannot control directly the radius of the blending. Also, 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.

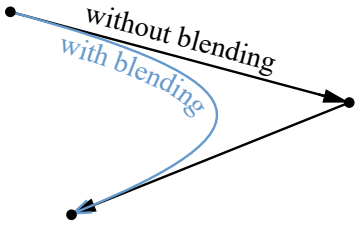
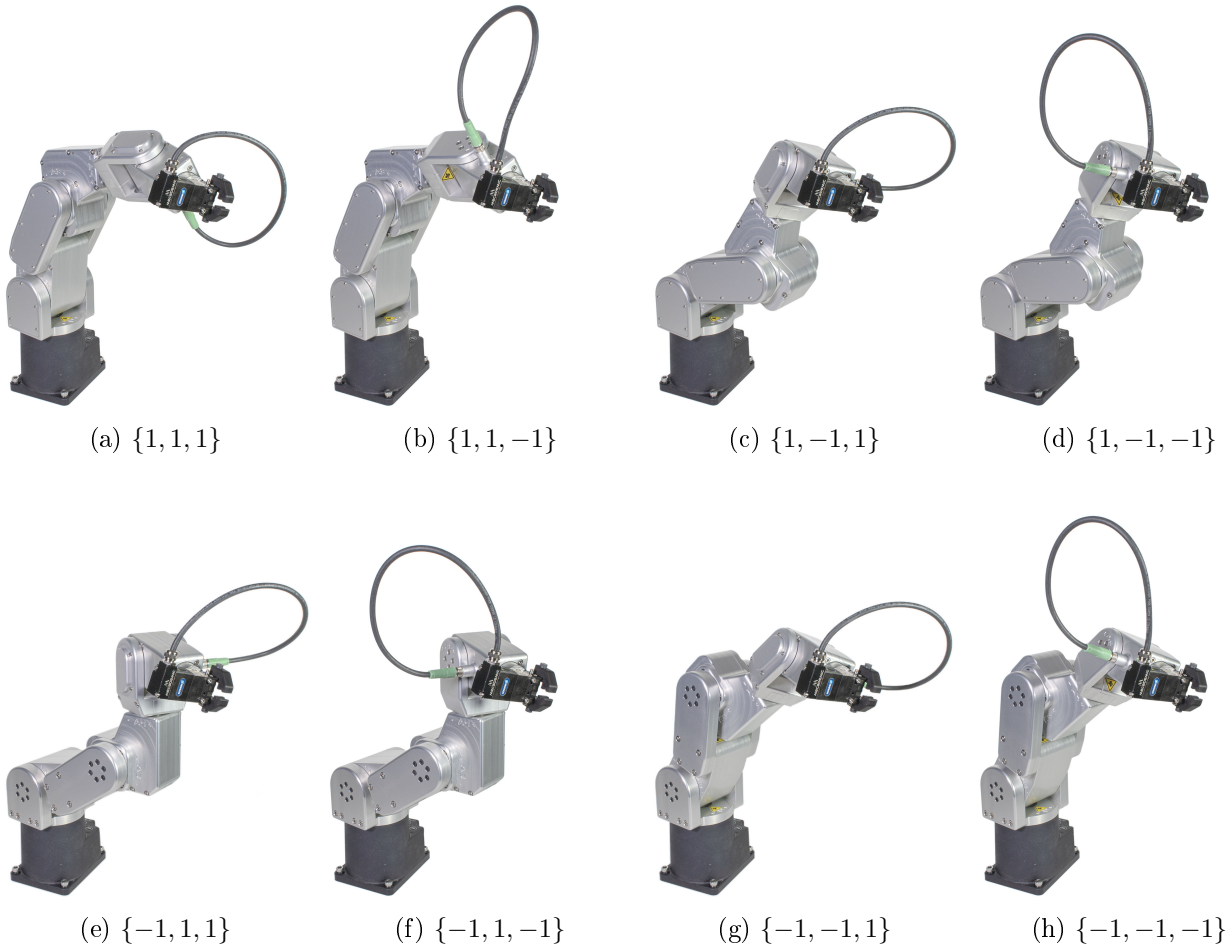


Figure 3: TCP path for two consecutive linear movements, with and without blending

2.2.3 Inverse kinematic configuration

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. 4), and many more joints 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 so-called configuration types, or *configurations*, defined by three parameters: c_1, c_3 and c_5. Each parameter corresponds to a specific geometric condition on the robot posture:

- c_1 :
  - $c_1 = 1$, if the *wrist center* (where the axes of joints 4, 5 and 6 intersect) is on the positive side of the yz plane of the frame associated with joint 2 (see Fig. 1a). This frame is obtained by shifting the BRF upwards and rotating it about the axis of joint 1 at θ_1 degrees. (The condition $c_1 = 1$ is often referred to as "front".)
  - $c_1 = -1$, if the wrist center is on the negative side of this plane ("back" condition).
- c_3:
  - $c_3 = 1$, if $\theta_3 > -\arctan(60/19) \approx -72.43°$ ("elbow up" condition)
  - $c_3 = -1$, if $\theta_3 < -\arctan(60/19) \approx -72.43°$ ("elbow down" condition)
- c_5:
  - $c_5 = 1$, if $\theta_5 > 0°$ ("no flip" condition)
  - $c_5 = -1$, if $\theta_5 < 0°$ ("flip" condition)



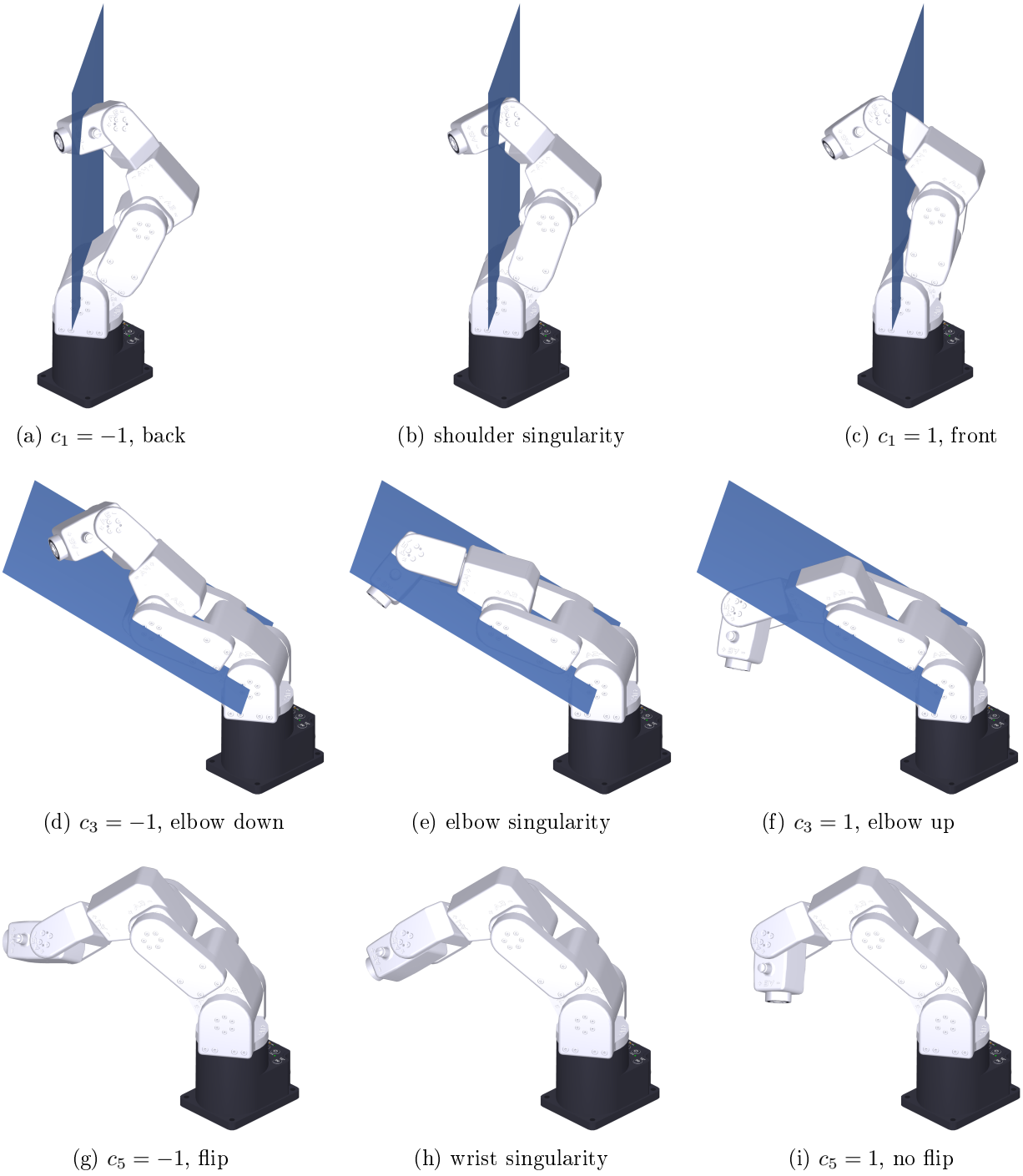
(a) $\{1, 1, 1\}$ (b) $\{1, 1, -1\}$ (c) $\{1, -1, 1\}$ (d) $\{1, -1, -1\}$

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

Figure 4: An example showing all eight possible configurations $\{c_1, c_3, c_5\}$ for the pose {77 mm, 210 mm, 300 mm, −103°, 36°, 175°} of the FRF with respect to the BRF

Figure 5 shows an example of each inverse kinematics 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.

Note that when we solve the inverse kinematics, we only find solutions for θ_6 that are in the range ±180°. Thus, if you want to "teach" unequivocally how the robot is placed (e.g., after jogging the robot), you can choose between two possible approaches. One approach is to save directly the current joint set by retrieving it first with GetJoints. In this way, you don't have to worry about configurations and singularities and you will record the exact value for θ_6, which could be outside the range ±180°. However, you won't be able to move the robot to this joint set by forcing the TCP to follow a straight line. For example, if the



(a) $c_1 = -1$, back (b) shoulder singularity (c) $c_1 = 1$, front

(d) $c_3 = -1$, elbow down (e) elbow singularity (f) $c_3 = 1$, elbow up

(g) $c_5 = -1$, flip (h) wrist singularity (i) $c_5 = 1$, no flip

Figure 5: Inverse kinematic configuration parameters and the three types of singularities

recorded joint set corresponds to the robot holding a pin inserted in a hole, you won't be able to retrieve the pin from the hole.

Thus, if you want to follow a linear path to a desired non-singular robot posture, you must follow a different approach. First, θ_6 must be inside the range $\pm 180°$. Then, you must

record the corresponding pose of the TRF (by retrieving it with GetPose) with respect to the WRF, but also the definitions of TRF and WRF and finally the corresponding configuration (by retrieving it with GetConf). Then, when you later want to approach this robot posture with MoveLin from a starting robot posture, you need to make sure the robot is already in this configuration. For example, you can set the desired configuration (using SetConf) and use MovePose to get to the starting robot posture. Then, you can use MoveLin to get to the target robot posture. Of course, you also need to use the same TRF and WRF.

In many cases, however, you simply want to move the robot end-effector to a given pose, and do not care about the path followed by the end-effector. In these situations, you can enable the automatic configuration selection feature (using SetAutoConf) and not worry about configurations. (In fact, this feature is activated by default.) If you simply want the TRF to move to a certain pose you can use the command MovePose and the robot will automatically select the optimal configuration for this pose (the one that corresponds to the robot posture that is fastest to reach, but in which $-180° \leq \theta_6 \leq 180°$).

Note, however, that when using a Cartesian-space motion command, it is impossible to change a configuration. (The command SetAutoConf has effect only on MovePose.) This is because a configuration change can only be achieved by crossing a singularity. In general, such a passage is physically impossible when having the TCP follow a specific trajectory.

2.2.4 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. 4). Thus, the Cartesian workspace of a six-axis robot arm is the combination of eight workspace subsets, one for each the eight configuration types. 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. 5). 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. 5h). 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. 5e). 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. 5b). 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.

2.2.5 Position and velocity modes

As already discussed in Section 2.2.2 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 6 summarizes the possible settings for the velocity and acceleration in position mode.

As of firmware 8.0.0, we offer a second method for controlling the Meca500, by defining either its end-effector velocity or its joint velocities. We call this alternative 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.

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

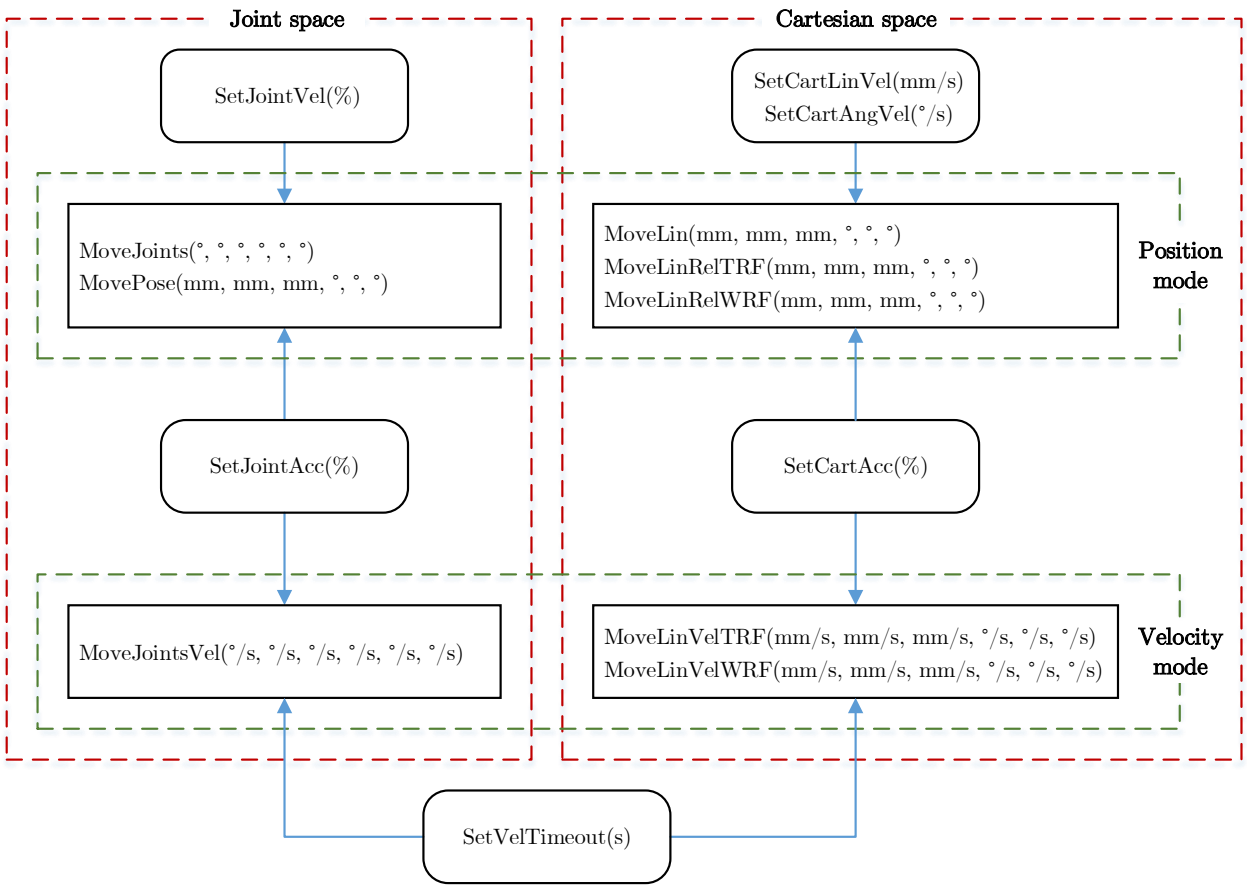


Figure 6: Settings that influence the robot motion in position and velocity modes

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 6).

3 Communicating over TCP/IP

To operate the Meca500, the robot must be connected to a computer or 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, which are transmitted over TCP/IP. The

robot default IP address is 192.168.0.100, and its default TCP/IP port is 10,000, 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 repeatedly send its joint set and TRF pose over TCP/IP port 10,001, referred to as the *monitoring port*, at the rate specified by the SetMonitoringInterval command. The robot status too will be sent over the monitoring port, but only when it changes. Note that in future firmware releases, other status updates will also be sent over the monitoring port (e.g., the gripper status).

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, commonly denoted as \0. Commands are not case-sensitive.

3.1 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 queue. Generally, the command will be run once all preceding commands have been executed. The arguments for all motion commands, except SetAutoConf, SetConf, and SetCheckpoint, are IEEE-754 floating-point numbers, separated—if more than one—with commas and, optionally, spaces.

In the following subsections, the motion commands are presented in alphabetical order. All motion commands have arguments, but not all have default values (e.g., the command Delay). Also, all motion commands generate an "[3012][End of block.]" message, by default, but this message can be deactivated with SetEOB. Furthermore, most motion commands may also generate an "[3004][End of movement.]" message, if this option is activated with SetEOM. Finally, motion commands can generate errors, explained in Section 3.3.1.

3.1.1 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

3.1.2 GripperOpen/GripperClose

These two commands are used to open or close Mecademic's optional MEGP 25 gripper. The gripper will move its fingers apart or closer until the grip force reaches 40 N. You can

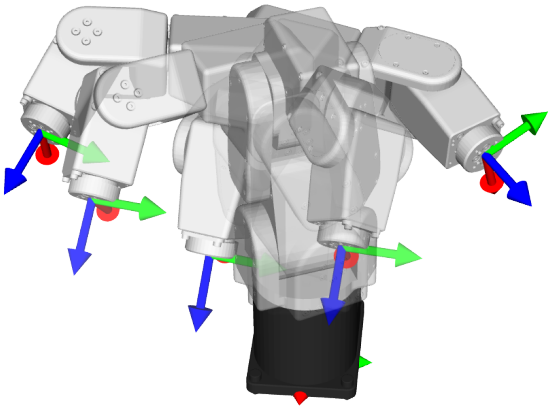


Figure 7: End-effector motion when using the MoveJoints or MovePose commands

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.

3.1.3 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. 7). 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 ranges for the joint angles are as follows:

  $-175° \leq \theta_1 \leq 175°$,

  $-70° \leq \theta_2 \leq 90°$,

  $-135° \leq \theta_3 \leq 70°$,

  $-170° \leq \theta_4 \leq 170°$,

  $-115° \leq \theta_5 \leq 115°$,

  $-36,000° \leq \theta_6 \leq 36,000°$.

Note that MoveJoints and MoveJointsVel are the only commands that can make joint 6 move outside its normal range of ±180°. Furthermore, no other motion command can make the robot move if joint 6 is outside its normal range of ±180°.

3.1.4 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. 7). 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.

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

This command makes the robot move its TRF to a specific pose with respect to the WRF while keeping the TCP along a linear path in Cartesian space, as illustrated in Fig. 8. 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. This command cannot automatically change the robot configuration. Furthermore, note that when executing this command, θ_6 must already be in the the range ±180° and will remain in that range during the execution of the command.

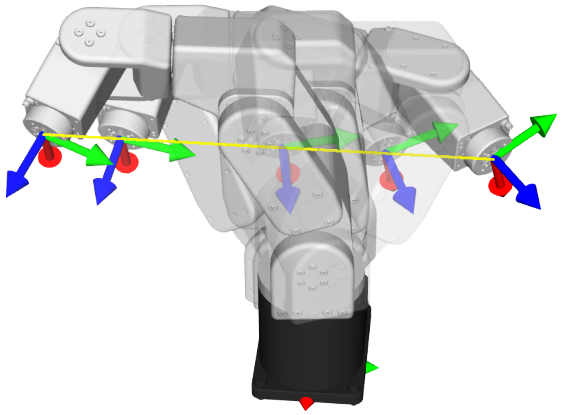


Figure 8: The TCP path when using the MoveLin command

Finally, note that if the motion requires that a joint rotates more than 180°, it will not be executed (unless you are upgrading from firmware 7). Furthermore, 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.

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

3.1.6 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 fact that joint 6 must be in the range $\pm 180°$ at all times and 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

3.1.7 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 not even start. These joint limits include the fact that joint 6 must be in the range ±180° at all times and 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

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

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

Arguments

- $\dot{x}$, $\dot{y}$, and $\dot{z}$: the components of the instantaneous 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 instantaneous angular velocity of the TRF w.r.t. the TRF, in $^\circ/s$, 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 ±180° limit on joint 6) or a singularity is reached. The robot simply stops slightly before the limit.

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

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

Arguments

- $\dot{x}$, $\dot{y}$, and $\dot{z}$: the components of the instantaneous 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 instantaneous angular velocity of the TRF w.r.t. the WRF, in $^\circ/s$, 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 $\pm 180^\circ$ limit on joint 6) or a singularity is reached. The robot simply stops slightly before the limit.

3.1.10 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 and for which θ_6 is in the range $\pm 180^\circ$, except those corresponding to a singular robot posture. Then, it either chooses the joint set that corresponds to the desired configuration or the one that is fastest to reach (see SetConf and SetAutoConf). 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. 7.

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 singularities or joint limits, it will normally not even start, and an error will be generated.

Finally, note that this command will make the robot move to a joint set (corresponding to the desired pose) for which θ_6 is in the range $\pm 180^\circ$. Finally, note that this command will be executed only if θ_6 is already in the range $\pm 180^\circ$.

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

3.1.11 SetAutoConf(e)

This command enables or disables the automatic robot configuration selection and has effect only on the MovePose command. This automatic selection allows the controller to choose the "closest" joint set corresponding to the desired pose (the arguments of the MovePose command) and for which θ_6 is in the range $\pm 180°$ (recall Section 2.2.3).

Arguments

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

3.1.12 SetBlending(p)

This command enables/disables the robot's blending feature (recall Section 2.2.2). 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.

3.1.13 SetCartAcc(p)

This command limits the Cartesian acceleration (both the linear and the angular) of the end-effector during movements resulting from Cartesian-space commands (see Fig. 6). 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 end-effector, 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.

3.1.14 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.

3.1.15 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 1,000 mm/s

Default values

The default TCP velocity limit is 150 mm/s.

3.1.16 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]

3.1.17 SetConf(c_1, c_3, c_5)

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

The robot inverse kinematic configuration (see Fig. 5) can be automatically selected by using the SetAutoConf command. Using SetConf automatically disables the automatic configuration selection.

Arguments

- c_1: first inverse kinematics configuration parameter, either -1 or 1
- c_3: second inverse kinematics configuration parameter, either -1 or 1
- c_5: third inverse kinematics configuration parameter, either -1 or 1

3.1.18 SetGripperForce(p)

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

Arguments

- p: percentage of maximum grip force ($\sim$40 N), ranging from 10% to 100%

Default values

By default, the grip force limit is 40%.

3.1.19 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 10% to 100%

Default values

By default, the finger velocity limit is 40%.

3.1.20 SetJointAcc(p)

This command limits the acceleration of the joints during movements resulting from joint-space commands (see Fig. 6). 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.

3.1.21 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.

3.1.22 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.

3.1.23 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

3.1.24 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.

3.2 Request commands

Request commands are used mainly to get status updates from the robot. Once received, the robot will immediately execute the command (e.g., return the requested information). For example, if you send a MoveJoints command, followed by a GetPose command, you will not get the final pose of the TRF but an intermediate one, very close to the starting pose. To get the final pose, you need to make sure that the robot has completed its movement, before sending the GetPose. While some of these request commands make the robot move or stop (e.g., StartProgram or PauseMotion), request commands do not have a lasting effect on subsequent motions.

In the following subsections, the request commands are presented in alphabetical order. Most request commands do not have arguments. Note that the commands PauseMotion ClearMotion, and ResumeMotion have direct effect on the movement of the robot, but they are considered request commands, since they are executed as soon as received. Finally, the format of the robot response is summarized in Section 3.3.2.

3.2.1 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.

3.2.2 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.]

3.2.3 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.]

3.2.4 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.]

3.2.5 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.

Responses

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

3.2.6 GetConf

This command returns the robot current inverse kinematic configuration (see Fig. 5), not the desired one, if any, set by the command SetConf.

Responses

[2029][c_1, c_3, c_5]

- c_1: first inverse kinematic configuration parameter, either -1 or 1
- c_3: second inverse kinematic configuration parameter, either -1 or 1

- c_5: third inverse kinematic configuration parameter, either -1 or 1

Note that, currently, if you request the inverse kinematic configuration while the robot is in a robot singularity, the controller will automatically choose one of the two, four or eight boundary configurations. For example, if you execute GetConf while the robot is at the joint set {0,0,0,0,0,0}, the robot will return the configuration {1,1,1}.

3.2.7 GetFwVersion

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

Responses

[2081][v8.x.x]

3.2.8 GetJoints

This command returns the robot joint angles in degrees.

Responses

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

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

Note that the values for θ_6 returned are in the range $[-36,000°, 36,000°]$.

3.2.9 GetPose

This command returns the current pose of the robot TRF with respect to the WRF.

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.

3.2.10 GetProductType

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

Responses

[2084][Meca500]

3.2.11 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)

3.2.12 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 moving, 0 if robot is idle)

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

3.2.13 Home

This command starts the robot and gripper homing process (Section 2.2.1). While homing, it is critical to remove any obstacles that could hinder the robot and gripper movements. This command takes about four seconds to execute.

Responses

[2002][Homing done.]
[2003][Homing already done.]
[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.

3.2.14 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 GetPose or GetJoints 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.]

3.2.15 ResetError

As described in the User Manual of the Meca500,

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.]

3.2.16 ResetPStop

As described in the user manual of the Meca500, 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.

3.2.17 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.]

3.2.18 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.]

3.2.19 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.]

3.2.20 SetMonitoringInterval(t)

This command is used to set the time interval at which the current pose of the TRF with respect to the WRF and the current joint set are sent over TCP/IP port 10,001.

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.

3.2.21 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 idle TCP packets that can be lost before the TCP connection is closed, where n 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, $n_1 = 3$.

3.2.22 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.

3.2.23 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 started.]
[3017][No offline program saved.]

3.2.24 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 recoding 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 (GetJoints, GetPose, 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: '...']

3.2.25 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 bellow. 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.]

3.2.26 SwitchToEtherCAT

This command will disable the Ethernet TCP/IP protocol and enable EtherCAT (see Section 4).

3.3 Responses

The Meca500 sends responses (also referred to as messages) over its control port when it encounters an error, when it receives a request command, 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.

3.3.1 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.

Table 1: (continued)

Message	Explanation
[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.
[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.
[1038][No gripper connected.]	No gripper was detected.

3.3.2 Command responses

Table 2 presents a summary of all motion and request commands and the possible non-error responses for each of them. Note that many of the commands do not generate specific command responses and may generate only an end-of-block and/or end-of-movement messages.

Table 2: List of possible responses to a command

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.]
DeactivateSim	[2046][The simulation mode is disabled.]
Delay	[3012][End of block.]
GetConf	[2029][c_1, c_3, c_5]
GetJoints	[2026][$\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$]
GetStatusRobot	[2007][$as, hs, sm, es, pm, eob, eom$]
GetStatusGripper	[2079][ge, hs, ph, lr, es, fo]
GetPose	[2027][$x, y, z, \alpha, \beta, \gamma$]
GripperClose	[3012][End of block.]
GripperOpen	[3012][End of block.]
Home	[2002][Homing done.] [2003][Homing already done.]
MoveJoints	[3004][End of movement.] [3012][End of block.]
MoveJointsVel	[3004][End of movement.] [3012][End of block.]
MoveLin	[3004][End of movement.] [3012][End of block.]
MoveLinRelTRF	[3004][End of movement.] [3012][End of block.]
MoveLinRelWRF	[3004][End of movement.] [3012][End of block.]
MoveLinVelTRF	[3004][End of movement.] [3012][End of block.]
MoveLinVelWRF	[3004][End of movement.] [3012][End of block.]
MovePose	[3004][End of movement.] [3012][End of block.]
PauseMotion	[2042][Motion paused.] [3004][End of movement.]

Table 2: (continued)

Command	Possible response(s)
ResetError	[2005][The error was reset.] [2006][There was no error to reset.]
ResetPStop	[3032][e]
ResumeMotion	[2043][Motion resumed.]
SetAutoConf	[3012][End of block.]
SetCartAcc	[3004][End of movement.]* [3012][End of block.]
SetCartLinVel	[3012][End of block.]
SetCartAngVel	[3012][End of block.]
SetConf	[3012][End of block.]
SetBlending	[3012][End of block.]
SetCheckpoint	[3030][n]
SetGripperVel	[3012][End of block.]
SetJointAcc	[3004][End of movement.]* [3012][End of block.]
SetJointVel	[3012][End of block.]
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.]
SetOfflineProgramLoop	[1022][Robot was not saving the program.]
SetTRF	[3004][End of movement.]* [3012][End of block.]
SetWRF	[3004][End of movement.]* [3012][End of block.]
SetVelTimeout	[3012][End of block.]
StartProgram	[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: '...']
StopSaving	[2061][n commands saved.] [2064][Offline program looping is enabled.] [2065][Offline program looping is disabled.]

* EOM message sent only if the robot was moving while the command was received.

3.3.3 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.
[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.
[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.

Table 3: (continued)

Message	Explanation
[3026][Robot's maintenance check has discovered a problem. Mecademic cannot guaranty correct movements. Please contact Mecademic.]	A hardware problem was detected. Contact Mecademic.
[3030][n]	Checkpoint n was reached.
[3032][e]	P-Stop 2 enabled ($e = 1$) or cleared ($e = 0$).

3.3.4 Messages over the monitoring port

In addition, the Meca500 is configured to send immediate robot feedback over TCP/IP port 10,001. Four kinds of feedback messages are sent over this port: joint set, TRF pose, and robot status. Joint set feedback returns the angles of the robot joints (i.e., the joint set). TRF pose feedback returns the pose of the TRF with respect to the WRF. By default, these two feedback messages are sent over TCP/IP approximately every 15 ms. You can set the time interval between subsequent feedback messages with the command SetMonitoringInterval.

The format of the responses for the joint set and the TRF pose feedback is, respectively

[2016][$\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6$]
[2027][$x, y, z, \alpha, \beta, \gamma$]

Finally, as soon as a change occurs in the robot status, the message generated by the commands GetStatusRobot is returned, i.e.,

[2007][$as, hs, sm, es, pm, eob, eom$]

Note that in future firmware releases, other status updates will also be sent over the monitoring port.

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 2 ms. Furthermore, you no longer need to parse strings as when using the TCP/IP protocol.

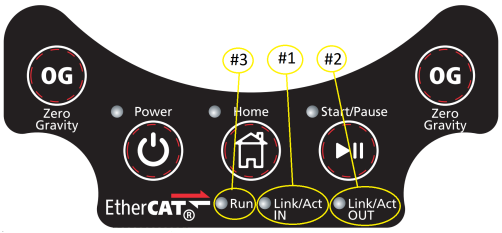


Figure 9: EtherCAT LEDs

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 at https://www.mecademic.com/Downloads/Updates/Robot%20Meca500/R3/.

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. 9) 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.

4.1.4 LEDs

When EtherCAT communication is enabled, the three LEDs on the outer edge of the robot's base (Fig. 9) communicate the state of the EtherCAT connection, as summarized in Table 4.

Table 4: 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. In the tables of this section, SI stands for subindex, and "O. code" for "Object code". There are only two Service Data Objects (SDO), presented in the last two subsections. All other objects are of type Process Data Object (PDO).

4.2.1 Robot control

This object controls the robot's initialization and simulation. It has five subindices:

1. Deactivate: deactivates the robot when set to 1, but you need to first set the subindices Activate and Home to 0
2. Activate: activates the robot when set to 1, but you need to first set the subindex Activate to 0.
3. Home: homes the robot when set to 1 if, of course, the robot is activated.
4. Reset error: resets the error when set to 0.
5. Simulation mode: enables (when set to 1) or disables (when set to 0) the simulation mode. To enable the simulation mode, the robot must be deactivated first.

Table 5 describes the five indices.

4.2.2 Motion control

This object controls the actual robot movement. It has four subindices (Table 6):

1. Move ID: a distinct ID number associated with each motion command.

Table 5: 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

2. SetPoint: has to be set to 1 for information to be sent to the robot.
3. Pause: puts the robot in pause without clearing the commands in the queue.
4. Clear move: erases the motion commands in the queue and puts the robot in pause.
5. ResetPStop: resets the protective stop (Cat. 2).

Table 6: 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

There are two ways to use the SetPoint and the new Move ID indices. In *cyclic mode*, SetPoint has to be set to TRUE and Move ID to 0 for motion commands to be sent to the robot at every cycle. In this mode, remember to reset the Motion command index 7305h to 0, or else the next time you set the SetPoint index to 1, the robot will move again. In *one-off mode*, SetPoint has to be set to TRUE and Move ID to any number different than that of the preceding cycle, in order to send a single motion command.

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, as follows, while the second index has six subindices corresponding to the arguments of the motion command. (Table 7):

0. No movement: all six subindices are ignored.

1. MoveJoints: all six subindices are in degrees.
2. MovePose: subindices 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
3. MoveLin: subindices 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
4. MoveLinRelTRF: subindices 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
5. MoveLinRelWRF: subindices 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
6. Delay: subindex 1 is the pause in seconds.
7. SetBlending: subindex 1 is the percentage of blending, from 0 or 100.
8. SetJointVel: subindex 1 is the percentage of maximum joint velocities, from 0.001 to 100.
9. SetJointAcc: subindex 1 is the percentage of maximum joint accelerations, from 0.001 to 600.
10. SetCartAngVel: subindex 1 is the Cartesian angular velocity limit, from 0.001 to 300, measured in °/s.
11. SetCartLinVel: subindex 1 is the linear velocity limit for the TCP, from 0.001 to 1,000, measured in mm/s.
12. SetCartAcc: subindex 1 is the percentage of maximum Cartesian accelerations, ranging from 0.001 to 100.
13. SetTRF: subindices 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
14. SetWRF: subindices 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
15. SetConf: subindices 1, 2, and 3 are -1 or 1.
16. SetAutoConf: subindex 1 is 0 or 1.
17. SetCheckpoint: subindex 1 in an integer number, ranging from 1 to 8,000.
18. Gripper: subindex 1 is 0 for GripperClose, and 1 for GripperOpen.
19. SetGripperVel: subindex 1 is the percentage of maximum finger velocity (∼100 mm/s), from 10 to 100.
20. SetGripperForce: subindex 1 is the percentage of maximum grip force (∼40 N), from 10 to 100.
21. MoveJointsVel: subindices 1 trhough 6 are in °/s.
22. MoveLinVelWRF: subindices 1, 2, 3 are in mm/s and 4, 5, 6 are in are in °/s.
23. MoveLinVelTRF: subindices 1, 2, 3 are in mm/s and 4, 5, 6 are in are in °/s.
24. SetVelTimeout: subindex 1 is in seconds.

It is strongly recommended to set the 7305h index to zero, when you no longer need to send a motion command.

Table 7: 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 Section 3.1).

4.2.4 Robot status

The robot status object makes it possible to know the actual state of the robot (see Table 8). This object has five subindices:

1. Error: indicates the error number (see Tables 1 and 3), or 0 if no error is present.
2. Busy: indicates whether the robot is being activated, homed or deactivated.
3. Activated: indicates whether the motors are online and ready to home.
4. Homed: indicates whether the robot is homed and ready to receive motion commands.
5. SimActivated: indicates whether the robot simulation mode is activated.
6. Brakes: indicates whether the brakes are engaged (1) or not (0).

Table 8: 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
	6	Variable	BOOL	Brakes	n/a	0	1	RO	1A00h.6

4.2.5 Motion status

The motion status object makes it possible to know the actual state of the robot's motion (see Table 9). This object has six subindices (SI):

1. Checkpoint: indicates the last Checkpoint number that was reached.
2. Move ID: indicates the ID of the last motion command queued for execution.
3. FIFO space: indicates the number of commands that the robot can receive at a particular moment. If 0 (too many commands sent), subsequent commands will be ignored.
4. Paused: Indicates whether the motion is paused.
5. EOB: Indicates whether the robot is not moving AND there is no motion command left in its buffer.
6. EOM: Indicates whether the robot is not moving.
7. Cleared: Indicates whether the robot's command queue is cleared. It the queue is cleared, the robot is not moving.
8. PStop: Indicates whether the protective stop (Cat. 2) is set.

It is best to monitor the Paused and Cleared subindices and ensure they become FALSE when the Pause and Clear move subindices of the Motion control object are set to FALSE, respectively.

Table 9: 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

4.2.6 Gripper status

The Gripper status object contains the current status of the gripper (see Table 10).

Table 10: 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 joint set object contains the current joint angles of the robot, in degrees (see Table 11).

Table 11: 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 End-effector object contains the current pose (position and orientation) of the TRF with respect to the WRF (see Table 12). The first three subindices correspond to the x, y, and z coordinates (in mm), while the next three correspond to the α, β, γ Euler angles (in degrees).

Table 12: 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 Configuration object contains the current inverse-kinematic configuration of the robot (see Section 2.2.3). The three subindices correspond to c_1, c_3, and c_5 and each has a value of -1 or 1 (see Table 13)

Table 13: Configuration object

Index	SI	O. code	Type	Name	Default	Min.	Max.	Access	PDO
6046h		Array		Configuration				RO	none
	1		INT8	c_1	n/a	−1	1	RO	1A08h.1
	2		INT8	c_3	n/a	−1	1	RO	1A08h.2
	3		INT8	c_5	n/a	−1	1	RO	1A08h.3

4.2.10 Joint velocities

Using EtherCAT, you have access to more real-time information about the robot, namely the instantaneous joint velocities, the joint torques, and the values of the accelerometer embedded in the robot's last link. The angular velocity object contains the instantaneous joint velocities of the robot, in $°/s$ (see Table 14).

Table 14: 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 torque ratios object contains the joint torques as percentages of their maximum values (see Table 15).

Table 15: 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 accelerometer object (Table 16) contains the accelerations measured by the accelerometer embedded in the link immediately before the joint 5. Note that the values are in the range of $\pm 32,000$, which corresponds to ± 2g, and are to be used only for verifying for impacts.

Table 16: 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 17). 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 17: 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 18). 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 18: 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 19 and 20, we list all PDOs and recall the objects they are assigned to.

Table 19: RxPDOs

PDO	**Object(s)**	**Name**	**Note**
1600h	7200h	Robot control	see Table 5
1601h	7310h	Motion control	see Table 6
1602h	7305h, 7306h	Movement	see Table 7

Table 20: TxPDOs

PDO	**Object**	**Name**	**Note**
1A00h	6010h	Robot status	see Table 8
1A01h	6015h	Motion status	see Table 9
1A02h	6030h	Angular position	see Table 11
1A03h	6031h	Cartesian position	see Table 12
1A04h	6032h	Angular velocities	see Table 14
1A05h	6033h	Torque ratios	see Table 15
1A06h	6040h	Accelerometer	see Table 16
1A07h	6045h	Gripper status	see Table 10
1A08h	6046h	Configuration	see Table 13

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