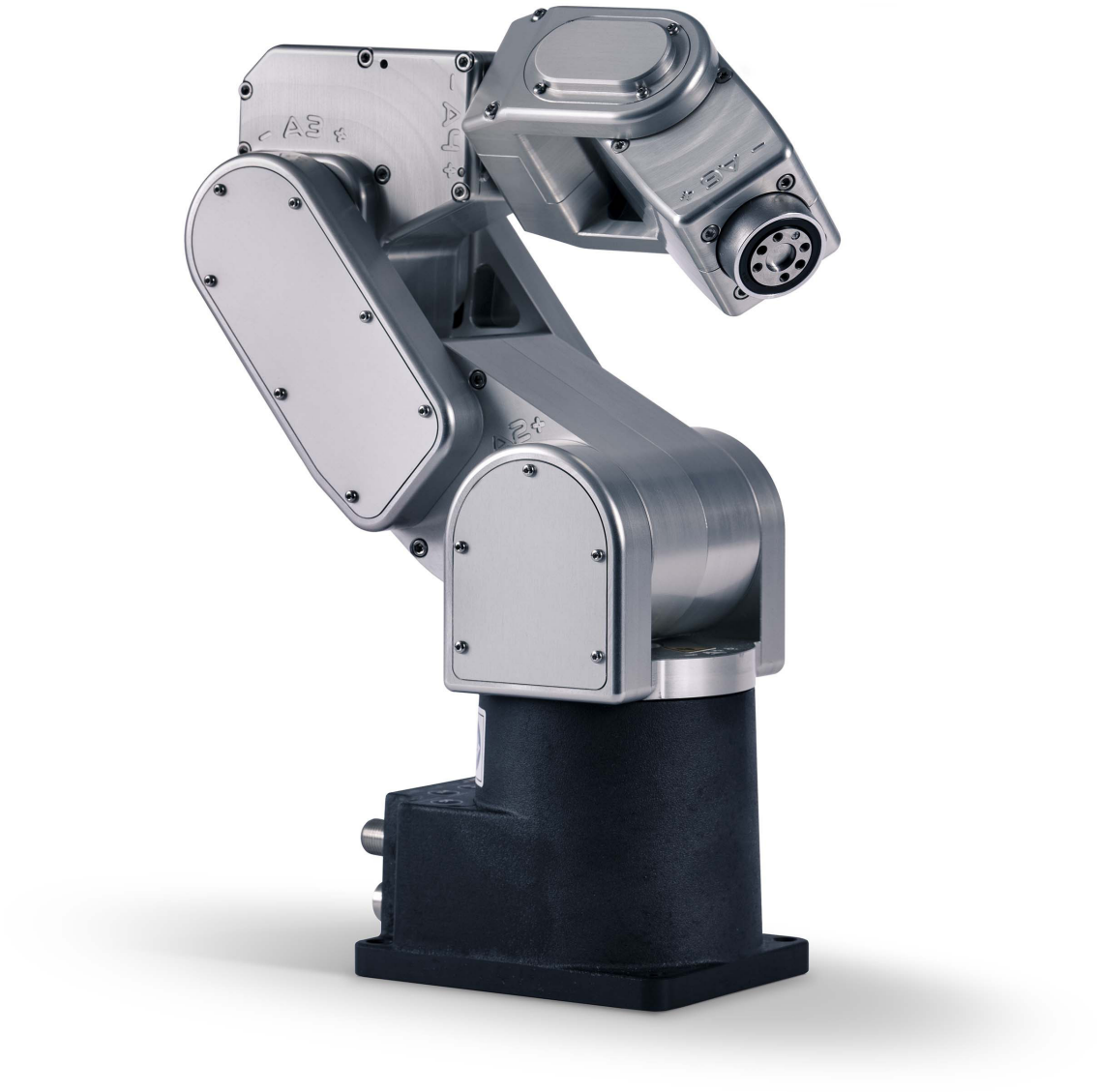
MECADEMIC
INDUSTRIAL ROBOTICS



MECA500

PROGRAMMING MANUAL

For Firmware Version 9.1.x

Document Revision: 1

August 31, 2022

CONTENTS

ABOUT THIS MANUAL

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

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

Reading the Meca500 User Manual and understanding the robot's operating principles is a prerequisite to reading this Programming Manual.

Symbol definitions

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

SYMBOL	DEFINITION
	NOTICE. Identifies information that requires special consideration.
	CAUTION. Provides indications that must be respected in order to avoid equipment or work (data) on the system being damaged or lost.
	WARNING. Provides indications that must be respected in order to avoid a potentially hazardous situation, which could result in injury.

1. BASIC THEORY AND DEFINITIONS

We are dedicated to technical accuracy, detail, and consistency, and use terminology that is not always standard. It is therefore important to read this section very carefully, even if you have prior experience with robot arms.

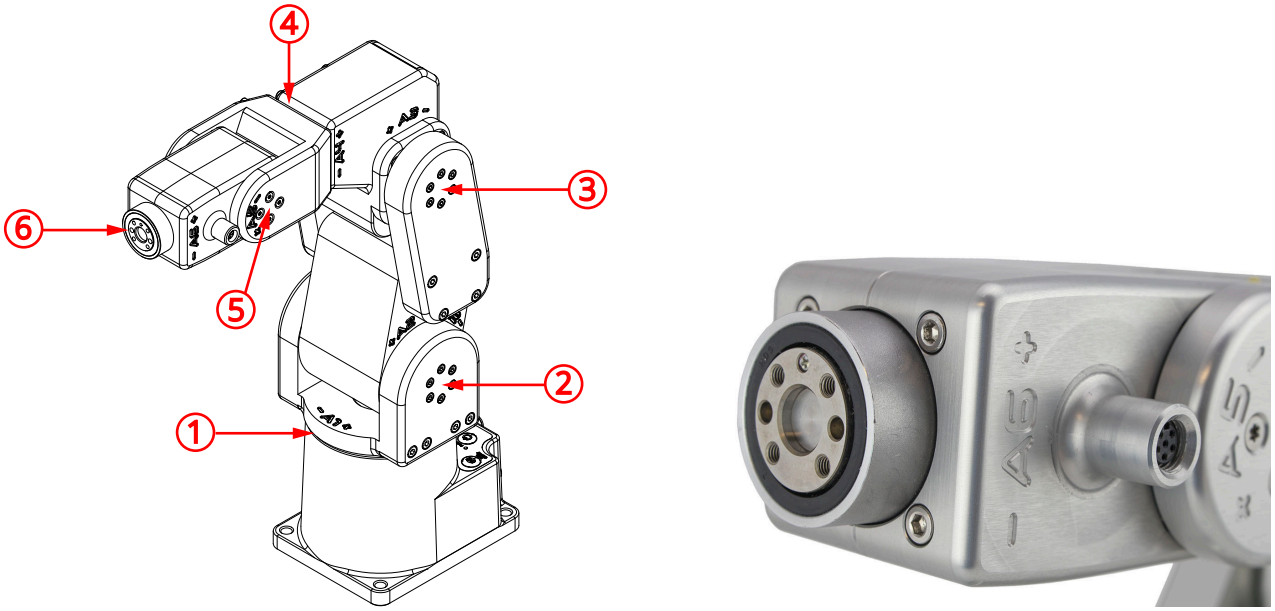
1.1. Definitions and conventions

1.1.1 Units

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

1.1.2 Joint numbering

The joints of the Meca500 are numbered in ascending order, starting from the base, as shown in Figure 1a.



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

Figure 1: Robot's joint numbering and zero-degree joint positions

1.1.3 Reference frames

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

- *BRF: Base reference frame*. Static reference frame fixed to the robot base. Its *z* axis coincides with the axis of joint 1 and points upwards, its origin lies on the bottom of the robot base, and its *x* axis is normal to the base front edge and points forward.
- *WRF: World reference frame*. The main static reference frame coincides with the BRF by default. It can be defined to correspond to the BRF using the *SetWrf* command.
- *FRF: Flange reference frame*. Mobile reference frame fixed to the robot's *flange* (the 20-mm disk with threaded holes at the extremity of the robot, shown in Figure 1b). Its *z* axis coincides with the axis of joint 6, and points outwards. Its origin lies on the surface of the robot's flange. Finally, when all joints are at zero, the *y* axis of the FRF has the same direction as the *y* axis of the BRF.
- *TRF: Tool reference frame*. The mobile reference frame associated with the robot's end-effector. By default, the TRF coincides with the FRF. It can be defined with respect to the FRF with the *SetTrf* command.
- *TCP: tool center point*. Origin of the TRF. (Not to be confused with the Transmission Control Protocol acronym, which is also used in this document.)

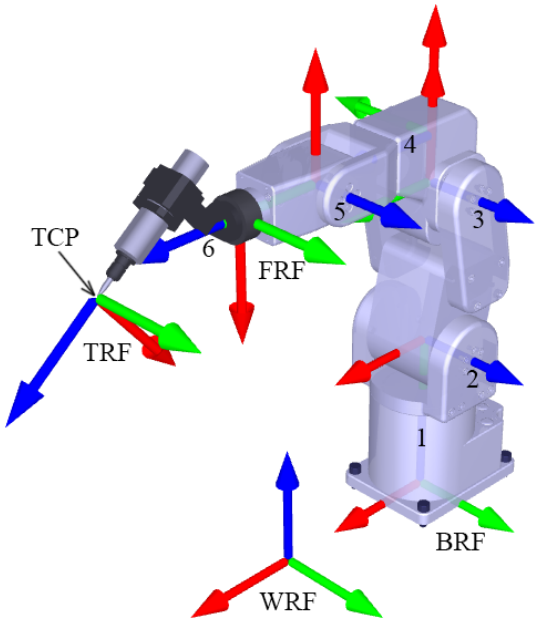


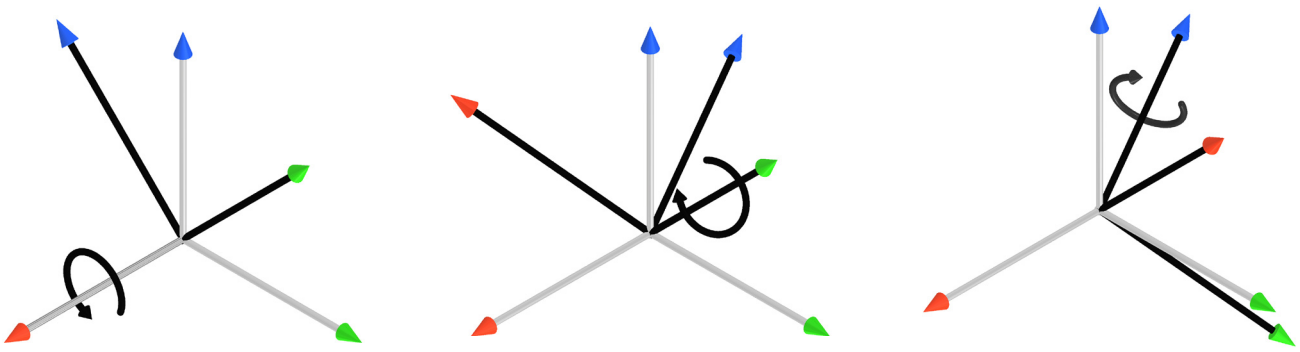
Figure 2: Reference frames for the Meca500

1.1.4 Pose and Euler angles

Some Meca500 commands accept *pose* (position and orientation of one reference frame with respect to another) as an input. In these commands, and in the Meca500 web interface, a pose consists of a Cartesian position, {*x*, *y*, *z*}, and an orientation specified in *Euler angles*, {*α*, *β*, *γ*}, according to the mobile XYZ convention (also referred to as RxRyRz). In this convention, if the orientation of a frame F_1 with respect to a frame F_0 is described by the Euler angles {*α*, *β*, *γ*}, 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.

Figure 3 shows an example of specifying orientation using the mobile XYZ Euler angle convention. The diagram on the right shows the black reference frame orientation with respect to the gray reference frame with the Euler angles {45°, −60°, 90°}.

Because there are infinitely many sets of Euler angles that correspond to a given orientation, the motion commands that accept a pose as arguments, accept any numerical value for the three Euler angles (e.g., the set {378.34°, −567.32°, 745.03°}). However, we output only the equivalent Euler angle set {α, β, γ}, for which $-180° \leq \alpha \leq 180°$, $-90° \leq \beta \leq 90°$ and $-180° \leq \gamma \leq 180°$. If you specify the Euler angles {α, ±90°, γ}, the controller will always return an equivalent Euler angle set in which $\alpha = 0$. Thus, it is perfectly normal that the Euler angles used to specify an orientation are not the same as the Euler angles returned by the controller, once that orientation has been attained (see our tutorial on Euler angles).



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

Figure 3: The three consecutive rotations associated with the Euler angles {45°, −60°, 90°}

The pose of the end-effector alone does not unequivocally define the required joint angles (see Section 1.2.1).

1.1.5 Joint angles and joint 6 turn configuration

The angle associated with joint *i* (*i* = 1, 2, ..., 6), θ_i, will be referred to as *joint angle i*. Since joint 6 can rotate more than one revolution, you should think of the 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 Figure 1. Note that unless you attach an end-effector with cabling to the robot flange, there is no way of knowing the value of θ_6 just by observing the robot.

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

$$-175° \leq \theta_1 \leq 175°,$$

$$-70° \leq \theta_2 \leq 90°,$$

$$-135° \leq \theta_3 \leq 70°,$$

$$-170° \leq \theta_4 \leq 170°,$$

$$-115° \leq \theta_5 \leq 115°.$$

Joint 6 has no mechanical limits, but its software limits are ±100 turns. Finally, define the integer c_t as the axis 6 *turn configuration*, so that $-180° + c_t360° < \theta_6 \leq 180° + c_t360°$.

Joints can be further constrained using the *SetJointLimits* command.

1.1.6 Joint set and robot posture

There are several possible solutions for joint angle values, for a desired location of the robot end-effector with respect to the robot base (i.e., several possible sets $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$). The simplest way to describe how the robot is postured, is by giving its set of joint angles. This set will be referred to as the *joint set*, and occasionally as *joint position*.

For example, in Figure 3, the joint set is {0°, 0°, 0°, 0°, 0°, 0°}, although, it could have been {0°, 0°, 0°, 0°, 0°, 360°}, and you wouldn't be able to tell the difference.

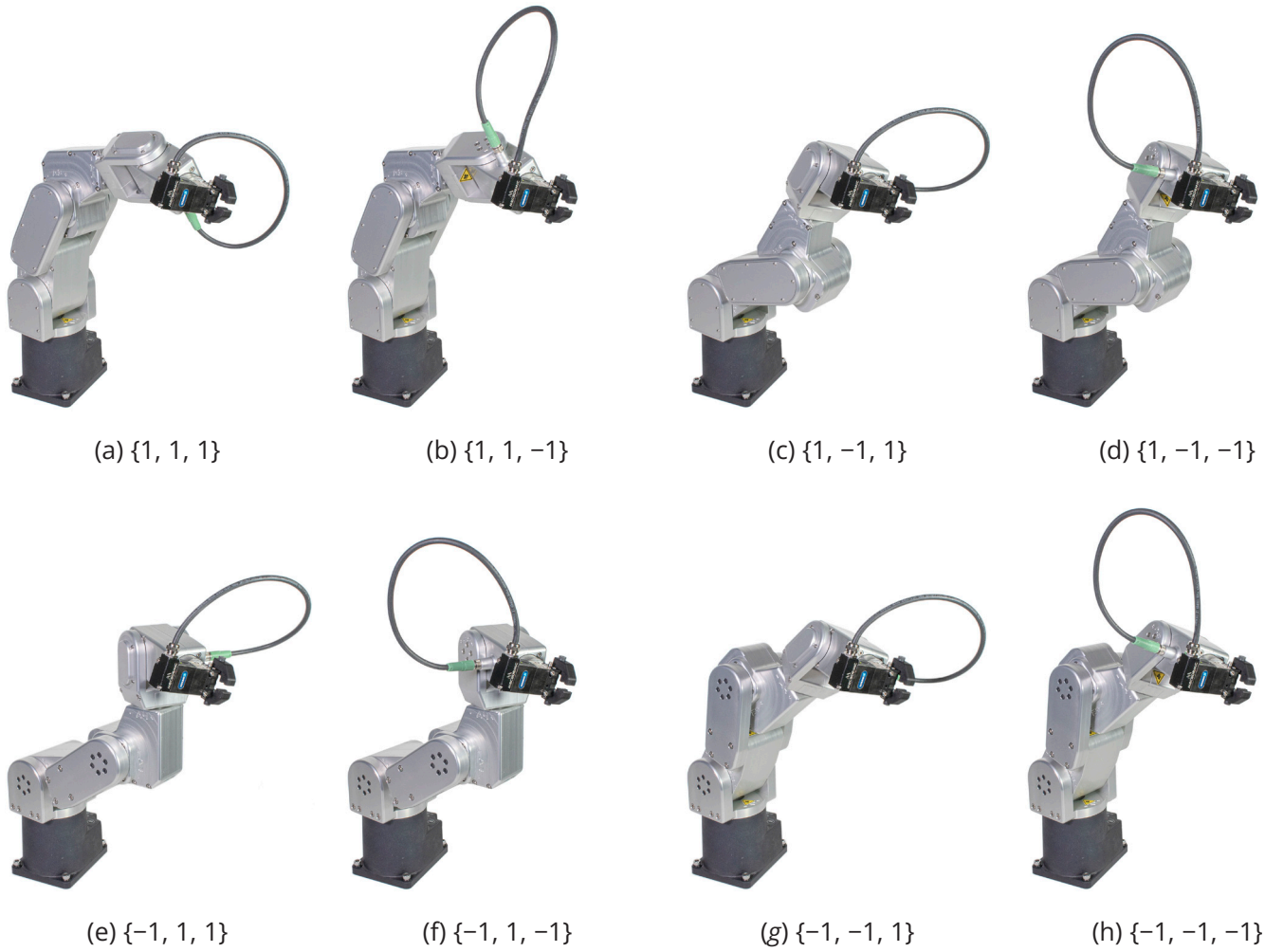
A joint set completely defines the relative poses (i.e., the "arrangement," of the seven robot links, starting with the base and ending with the end-effector). This arrangement is called the *robot posture*. Thus, the joint sets $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$ and $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6 + c_t360°\}$, where $-180° < \theta_6 \leq 180°$ and c_t is the axis 6 turn configuration, correspond to the same robot posture. Therefore, a joint set has the same information as a robot posture AND an axis 6 turn configuration.

1.2. Configurations, singularities and workspace

1.2.1 Inverse kinematic solutions and configuration parameters

Meca500's inverse kinematics generally provide up to eight feasible robot postures for a desired pose of the TRF with respect to the WRF (Figure 4), and many more joint sets (since if θ_6 is a solution, then $\theta_6 \pm n360°$, where n is an integer, is also a solution). Each of these solutions is associated with one of eight *robot posture configurations*, defined by three parameters: c_s, c_e and c_w. Each of these parameters corresponds to a specific geometric condition on the robot posture (see Figure 5):

- c_s (shoulder configuration parameter):
  - $c_s = 1$, if the *wrist center* (where the axes of joints 4, 5 and 6 intersect) is on the "front" side of the plane passing through the axes of joints 1 and 2 (see Figure 5a). The condition $c_s = 1$ is often referred to as "front".
  - $c_s = -1$, if the wrist center is on the "back" side of this plane (see Figure 5c).
- c_e (elbow configuration parameter):
  - $c_e = 1$, if $\theta_3 > -\arctan(60/19) \approx -72.43°$ ("elbow up" condition, see Figure 5d);
  - $c_e = -1$, if $\theta_3 < -\arctan(60/19) \approx -72.43°$ ("elbow down" condition, see Figure 5f).
- c_w (wrist configuration parameter):
  - $c_w = 1$, if $\theta_5 > 0°$ ("no flip" condition, see Figure 5g);
  - $c_w = -1$, if $\theta_5 < 0°$ ("flip" condition, see Figure 5i).



(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}

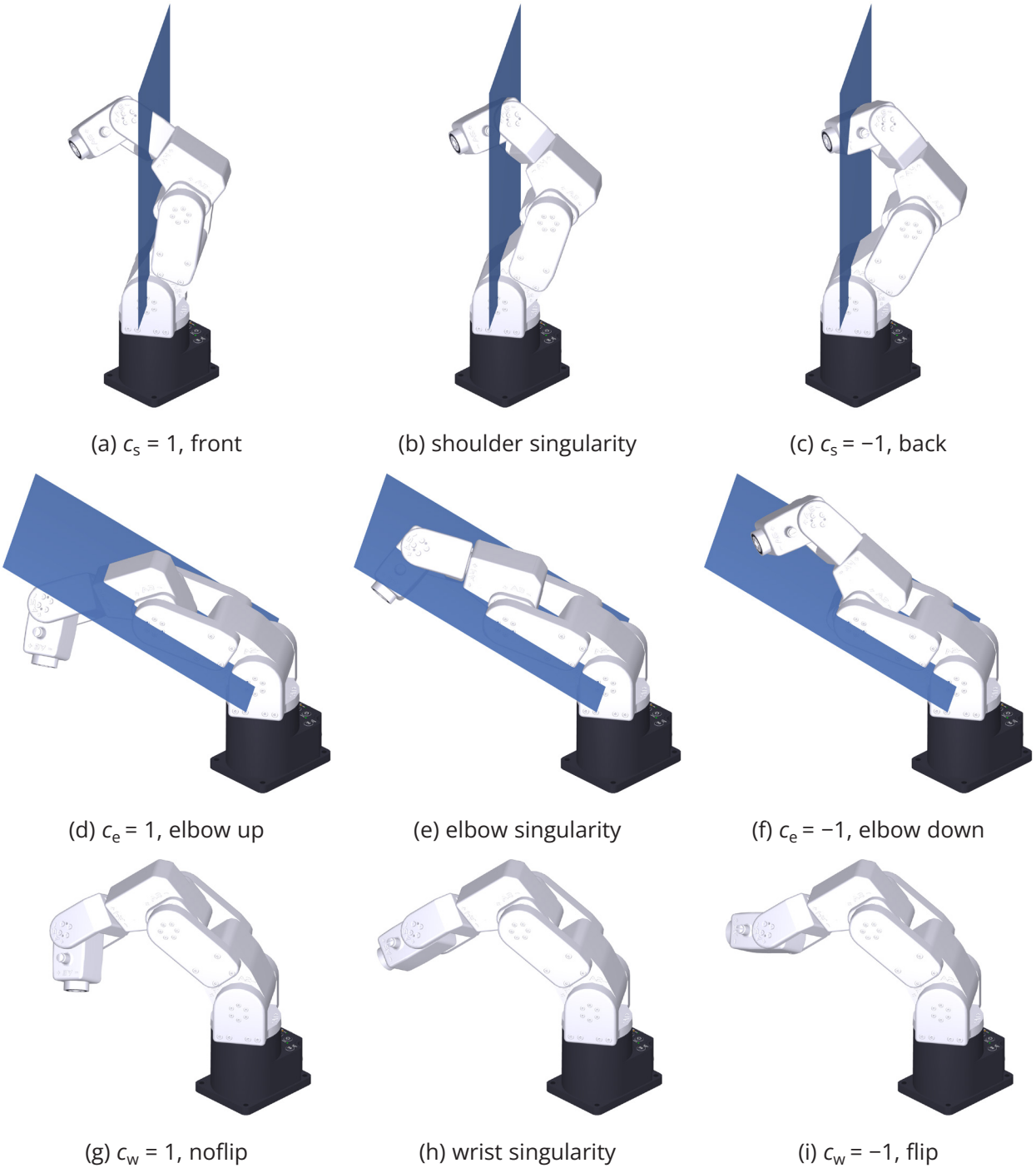
Figure 4: An example showing all eight possible robot postures

Figure 4 shows an example all eight possible robot postures, described by the posture configuration parameters {c_s, c_e, c_w}, for the pose {77 mm, 210 mm, 300 mm, −103°, 36°, 175°} of the FRF with respect to the BRF.

Figure 5 shows an example of each robot posture configuration parameter, and 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.

The robot calculates the solution to the inverse kinematics that corresponds to the desired posture configuration, {c_s, c_e, c_w}, defined by the *SetConf* command. In addition, it solves θ_6 by choosing the angle that corresponds to the desired turn configuration, c_t (an integer in the range ±100), defined by the *SetConfTurn* command. The turn is therefore the last inverse kinematics configuration parameter.

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



(a) c_s = 1, front

(b) shoulder singularity

(c) c_s = −1, back

(d) c_e = 1, elbow up

(e) elbow singularity

(f) c_e = −1, elbow down

(g) c_w = 1, noflip

(h) wrist singularity

(i) c_w = −1, flip

Figure 5: Posture configuration parameters and the three types of singularities

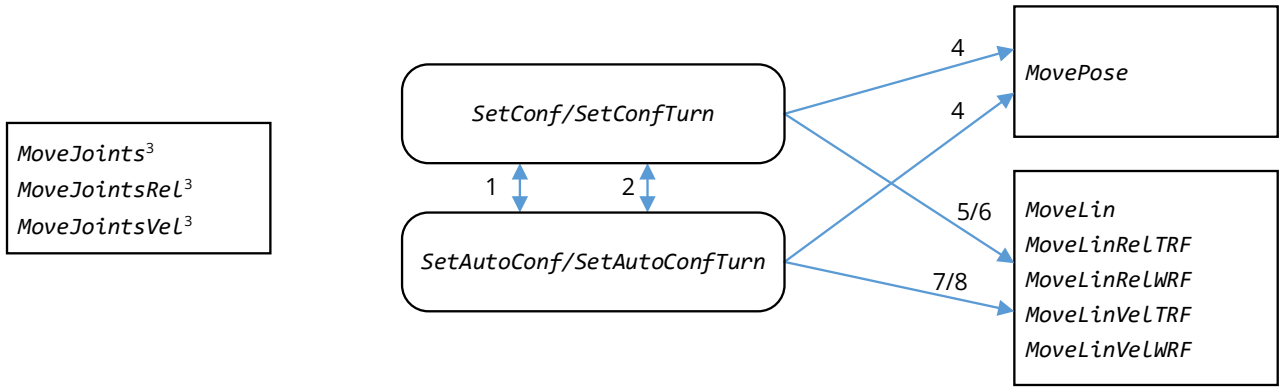
Though it is possible to calculate the optimal (the shortest move from current robot position) inverse kinematic solution (commands *SetAutoConf* and *SetAutoConfTurn*), we highly recommend that you always specify the desired values for the configurations parameters (with the commands *SetConf* and *SetConfTurn*) for every Cartesian-coordinates motion command (i.e., *MovePose* and the various *MoveLin** commands), when programming your robot in *online mode*.

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

1.2.2 Automatic configuration selection

The automatic configuration selection should only be used once you understand how this selection is done, and mainly while programming and testing. For example, when jogging in Cartesian space with the Meca500 web interface, the automatic configuration selection is always enabled. Or, if a target pose is identified in real-time based on input from a sensor (e.g., a camera), enabling the automatic configuration selection will increase your chances of reaching that pose, and in the fastest way.

Figure 6 illustrates how the automatic and manual configuration selections work.



Notes:

1. *SetConf*(c_s,c_e,c_w) disables *AutoConf*, while *SetAutoConf*(1) disables the desired posture setting. When *SetAutoConf*(0) is executed, the new desired posture configuration will be the one corresponding to the current robot position.
2. *SetConfTurn*(c_t) disables *AutoConfTurn*, while *SetAutoConfTurn*(1) disables the desired turn setting. When *SetAutoConfTurn*(0) is executed, the new desired turn will be the one corresponding to the current value of θ_6.
3. *MoveJoints** ignores any desired posture or turn configuration and, inversely, has no effect on the posture and turn configuration settings.
4. *MovePose* will respect any desired posture or turn configuration, as long as the desired robot position is reachable.
5. If a desired posture configuration is set, *MoveLin* or MoveLinRel* will be executed only if the initial and final posture configurations are the same as the desired one, while *MoveLinVel** will start being executed only if the initial posture configuration is the same as the desired one and will stop if the robot arrives at a singularity.
6. If a desired turn configuration is set, *MoveLin* or *MoveLInRel** will be executed only if the initial and final turn configurations are the same as the desired one, while *MoveLinVel** will start being executed only if the initial turn configuration is the same as the desired one and will stop if joint 6 has to change its turn configuration.
7. With *AutoConf* enabled, the robot may change its posture if it passes via certain singularities with a *MoveLin**.
8. With *AutoConfTurn* enabled, the robot may change its turn configuration with a *MoveLin**.

Figure 6: Effect of configuration parameters on robot movement commands

Firstly (Figure 6, notes 1 and 2), setting a desired posture or turn configuration (with *SetConf* or *SetConfTurn*, respectively) disables the automatic posture or turn configuration selection, respectively, which are both set by default. Inversely, enabling the automatic posture or turn configuration selection, with *SetAutoConf*(1) or *SetAutoConfTurn*(1), respectively, removes the desired posture or turn configuration, respectively. At any moment, if *SetAutoConf*(0) or *SetAutoConfTurn*(0) is executed, the robot posture or turn configuration of the current robot position is set as the desired posture or turn configuration, respectively.

Secondly (Figure 6, note 3), the commands *MoveJoints*, *MoveJointsRel*, and *MoveJointsVel* ignore the automatic and manual configuration selections. Thus, the robot may end up in a posture or turn configuration different from the desired ones, if such were set. If you want to update the desired configurations with the current ones, simply execute the commands *SetAutoConf*(0) or *SetAutoConfTurn*(0).

Thirdly, *MovePose* respects any desired posture or turn configuration, as long as the desired robot position is reachable (Figure 6, note 4). In contrast, if automatic posture and/or turn configuration selection is enabled, *MovePose* will choose the joint position, corresponding to the desired end-effector pose, that is fastest to reach, and that satisfies the desired posture or turn configuration, if any.

Fourthly, in the case of *MoveLin** commands, the desired posture and turn configurations will force the linear move to remain within the specified configuration or turn (Figure 6, notes 5 and 6). This means that a *MoveLin* or *MoveLinRel** command will be executed only if the posture and turn configurations of the initial and final robot positions are the same as the desired configurations. In the case of *MoveLinVel**, the robot will start to move only if the posture and turn configurations of the initial and final robot positions are the same as the desired configurations, and will stop if desired configuration parameter has to change. When automatic configuration selection is enabled, a *MoveLin** command may lead to changing the posture (if passing through a wrist or shoulder singularity) or turn configuration along the path.

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

1.2.3 Workspace and singularities

Users often 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 (Figure 4). Thus, the Cartesian workspace of a six-axis robot arm is the combination of eight workspace subsets, one for each the eight robot posture configurations. These eight workspace subsets have common parts, but there are also parts that belong to only one subset (i.e, there are end-effector poses accessible with only one configuration, because of joint limits). Therefore, in order to make optimal use of all possible 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). 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 Figure 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).

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

There are three types of singular robot positions, and these correspond to the conditions under which the configuration parameters c_s, c_e, and c_w are not defined. The most common singular robot posture is called a wrist singularity and occurs when $\theta_5 = 0°$ (Figure 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 frequently. The second type of singularity is called an elbow singularity (Figure 5f). 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 a shoulder singularity (Figure 5h). 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.

1.2.4 Crossing singularities with linear Cartesian-space movements

Although singularities can be a nuisance when controlling the robot in Cartesian space and should usually be avoided in production mode, we have made it possible to cross them to facilitate programming our robot. With the release of firmware 9.1, the Meca500 can start at or pass through wrist and shoulder singularities, while executing any *MoveLin** command, or end at any singularity while executing a *MoveLin** or *MovePose* command. Furthermore, the passage respects the posture configuration selection settings (Figure 6). Figure 7 illustrates how this new feature makes it possible to follow longer linear paths. In that figure, we start from an elbow singularity, pass through a wrist singularity, then through a shoulder singularity, and then end at another elbow singularity, all with a single *MoveLin** command, and in *AutoConf*.

There are two possible situations when crossing a wrist singularity. Consider Figure 8a, where AutoConf is enabled, the robot starts from robot position A, passes without any interruption through the singular configuration Z1 (where all joints are at zero degrees) and goes to robot position B, all with a single *MoveLin** command. In the process, the robot changes the posture parameter c_w from 1 to −1. However, if you execute *SetConf*(1,1,1), then request the robot to move with *MoveLin** to the end-effector pose B, starting from robot position A, the robot will refuse the motion, since that would require joint 4 to rotate 180° or −180° when reaching robot position Z1. This is impossible since the range of joint 4 is ±170°.

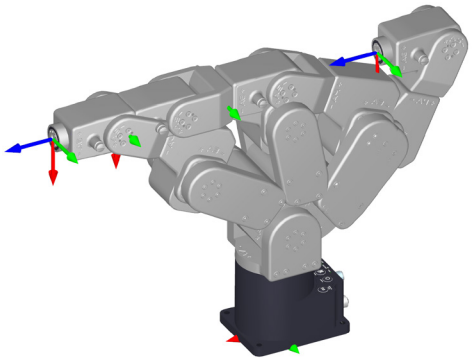
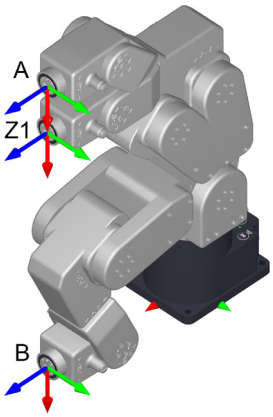
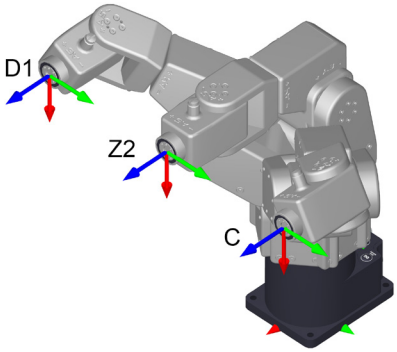


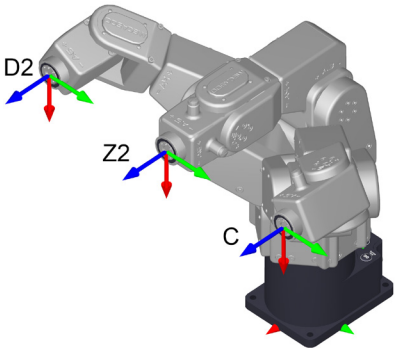
Figure 7: By crossing singularities, the Meca500 can execute longer linear movements



(a) A↔B via Z1, only possible with *AutoConf*(1)



(b) C↔D1 via Z2, with *AutoConf*(1)



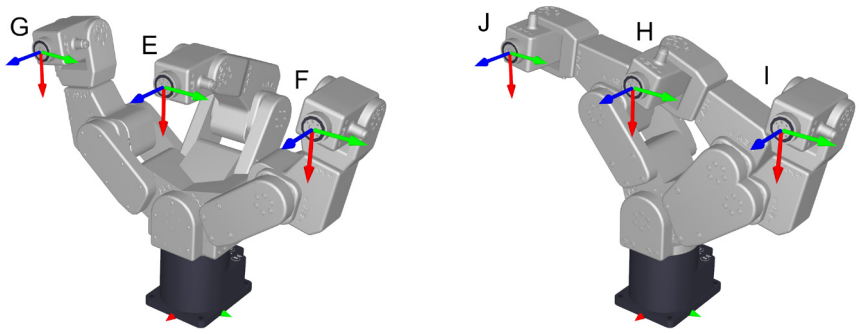
(c) C↔D2 via Z2 and stationary re-orientation, with *SetConf*(1,1, −1)

Figure 8: Crossing a wrist singularity with *AutoConf*(1) or with a desired posture configuration

Similarly, consider Figure 8b, where AutoConf is enabled, the robot starts from position C, passes without any interruption through the singular configuration Z2 (where $\theta_1 = \theta_2 = \theta_3 = \theta_5 = 0°$, $\theta_4 = 90°$, $\theta_6 = -90°$) and goes to robot position D1, all with a single *MoveLin* command. In the process, the robot changes posture parameter c_w from −1 to 1. However, as shown in Figure 8c, if you execute *SetConf*(1,1,−1), then request the robot to move to the end-effector pose D1, starting from robot position C, the robot will execute the *MoveLin* command, but when it reaches configuration Z2, joint 4 will rotate −180° and joint 6 will rotate 180°, at the same time while the end-effector will remain stationary. After that, the robot will continue its linear motion and reach the robot position D2 (which corresponds to the same pose as D1).

In contrast, since shoulder singularities are much less frequent, yet much more complex to handle, the robot can currently cross them only in AutoConf. More precisely, when executing a linear move, the robot will never stop at a shoulder singularity to reorient its joints 1, 4 and 6 while keeping the end-effector stationary. Thus, the motion sequence shown in Figure 9a cannot be executed with a single *MoveLin** command, whatever the state of posture configuration selection. However, in AutoConf, you can cross a shoulder singularity, as shown in Figure 9b.

To experiment with shoulder singularities, simply execute *SetTRF*(0,0,−70,0,0,0), to bring the TCP at the wrist center, then *SetWRF*(0,0,0,0,0,0), and then bring the TCP to a position where its coordinates *x* and *y* are zero.



(a) Impossible motion sequence with a single *MoveLin* command

(b) I↔J via H, with *AutoConf*(1)

Figure 9: Crossing a shoulder singularity can only be done with AutoConf(1) and implies a change of the posture parameter c_w

Passing exactly through singularities could be beneficial for some applications, but you must fully understand the concept. Otherwise, you might end-up with highly suboptimal robot motions. For example, consider the motion shown in Figure 9b. If you try to follow the same linear path, but one micrometer closer to the *z* axis of the WRF, joints 4 and 6, or joints 1, 4 and 6, will rotate very fast while the end-effector's speed will be significantly reduced, in a motion similar to what is shown in Figure 9a. Indeed, passing through or close to singularities usually leads to longer cycle times, and should be avoided in production mode.

1.3. Key concepts for Mecademic robots

1.3.1 Homing

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

During homing, all joints rotate simultaneously. Joints 1, 2 and 3 each rotate 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 these joint movements, or the homing process will fail. Homing will also fail if any of the robot joints are outside their user-defined limits (*SetJointLimits*).

Finally, if your robot is equipped with a Mecademic gripper, the robot controller will automatically detect it, and the homing procedure will end by fully opening, then fully closing the gripper. Make sure there is nothing that restricts the full range of motion of the gripper, except its fingers, while it is being homed.

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 and activate the robot, perform its homing, and zero joint 6. If the screw on the robot's flange is not as in Figure 1b, then rotate joint 6 to +720°, and deactivate the robot. Next, reactivate it, home it and zero joint 6 again. Repeat one more time if the problem is not solved.

1.3.2 Recovery mode

Once activated, if the robot is outside the user-defined joint limits (*SetJointLimits*) or too close to an obstacle, homing the robot is either impossible or presents a risk of collision, it may be necessary to move the robot before homing it, without manual intervention. Mecademic has implemented the recovery mode (see the command *SetRecoveryMode*) for these situations. In this mode, virtually all motion commands are allowed, as long as the robot is activated. However, if the robot was not homed before enabling the recovery mode, it will be less accurate.

Recovery mode is also useful when the robot is already homed, but a collision resulted in some joints rotating outside their user-defined joint limits (*SetJointLimits*). Simply enable the recovery mode, forcing the robot to ignore the user-defined joint limits. If the robot was already homed when enabling the recovery mode, its motions will be as precise as before.

However, whether the robot was homed or not, enabling the recovery mode will limit the joint and Cartesian velocities and accelerations, for safety reasons.

1.3.3 Blending

Industrial robots function similarly when moving in standard manner : either the robot is moved to its end-effector to a certain pose using a *Cartesian-space* command, or its joints moved to a specified joint set using a *joint-space* command. When the 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 let the robot choose the path or require the TCP to follow a linear path. If the robot must follow a complex curve (e.g., a gluing application), the curve must be broken down into multiple linear segments. Then, instead of the robot stopping at the end of each segment and making a sharp change in direction, the segments can be blended. Think of blending as taking a rounded shortcut.

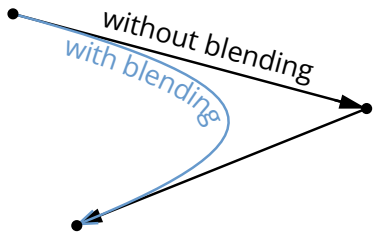


Figure 10: TCP path for two consecutive linear movements, with and without blending

Blending allows the trajectory planner to maintain the end-effector's acceleration to a minimum between two position-mode joint-space movements (*MoveJoints*, *MoveJointsRel*, *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 (Figure 10). The higher the TCP speed, the more rounded the transition will be (the radius of the blending cannot be explicitly controlled, only the blending duration is configurable on the Meca500).

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

1.3.4 Position and velocity modes

As already mentioned in the previous section, the conventional way of moving an industrial robot is by requesting that its end-effector move to a desired pose or that its joints rotate to a desired joint set. This basic control method is called *position mode*. If the robot must also follow a linear path, then you must use the Cartesian-space motion commands *MoveLin*, *MoveLinRelTrf* and *MoveLinRelWrf*. If the goal is to get the robot's end-effector to a certain pose or to rotate the robot's joints to a certain joint set or by a certain amount, then use the joint-space motion commands *MovePose*, *MoveJoints*, or *MoveJointsRel*, respectively.

In position mode, with Cartesian-space motion commands, it is possible to specify the maximum linear and angular velocities, and the maximum accelerations for the end-effector. 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, it is possible to specify the maximum velocity and acceleration of the joints, but it is impossible to limit either the velocity or the acceleration of the robot's end-effector. Figure 11 summarizes the possible settings for the velocity and acceleration in position mode.

There is a second method to control the Meca500, by defining either its end-effector velocity or its joint velocities. This robot control method is called the *velocity mode*. Velocity mode is designed for advanced applications such as force control, dynamic path corrections, or telemanipulation (for example, the jogging feature in Meca500's web interface is implemented using velocity-mode commands).

Controlling the robot in velocity mode requires one of the three velocity-mode motion commands: *MoveJointsVel*, *MoveLinVelTrf* or *MoveLinVelWrf*. Note that the effect from a velocity-mode motion command lasts the time specified in the *SetVelTimeout* command or until a new velocity-mode command is received. This timeout must be very small (the default value is 0.05 s, and the maximum value 1 s). For the robot to continue moving after this timeout, another velocity-mode command can be sent before this timeout. This new command will immediately replace the previous command and restart the timeout. Position-mode and velocity-mode motion commands can be sent to the robot, in any order. However, if the robot is moving in velocity mode, the only commands that will be executed immediately, rather than after the velocity timeout, are other velocity-mode motion commands and *SetCheckpoint*, *GripperOpen* and *GripperClose* commands.

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

In velocity mode, if the robot runs into a singularity or a joint limit, it will simply stop without raising an error. Furthermore, the velocity of the robot's end-effector or of the robot joints is directly controlled. No other command can further limit (or override) these velocities. The *SetJointAcc* command only limits for joints for *MoveJointsVel*. The *SetCartAcc* only limits the end-effector acceleration *MoveLinVelTrf* and *MoveLinVelWrf* (see Figure 11).

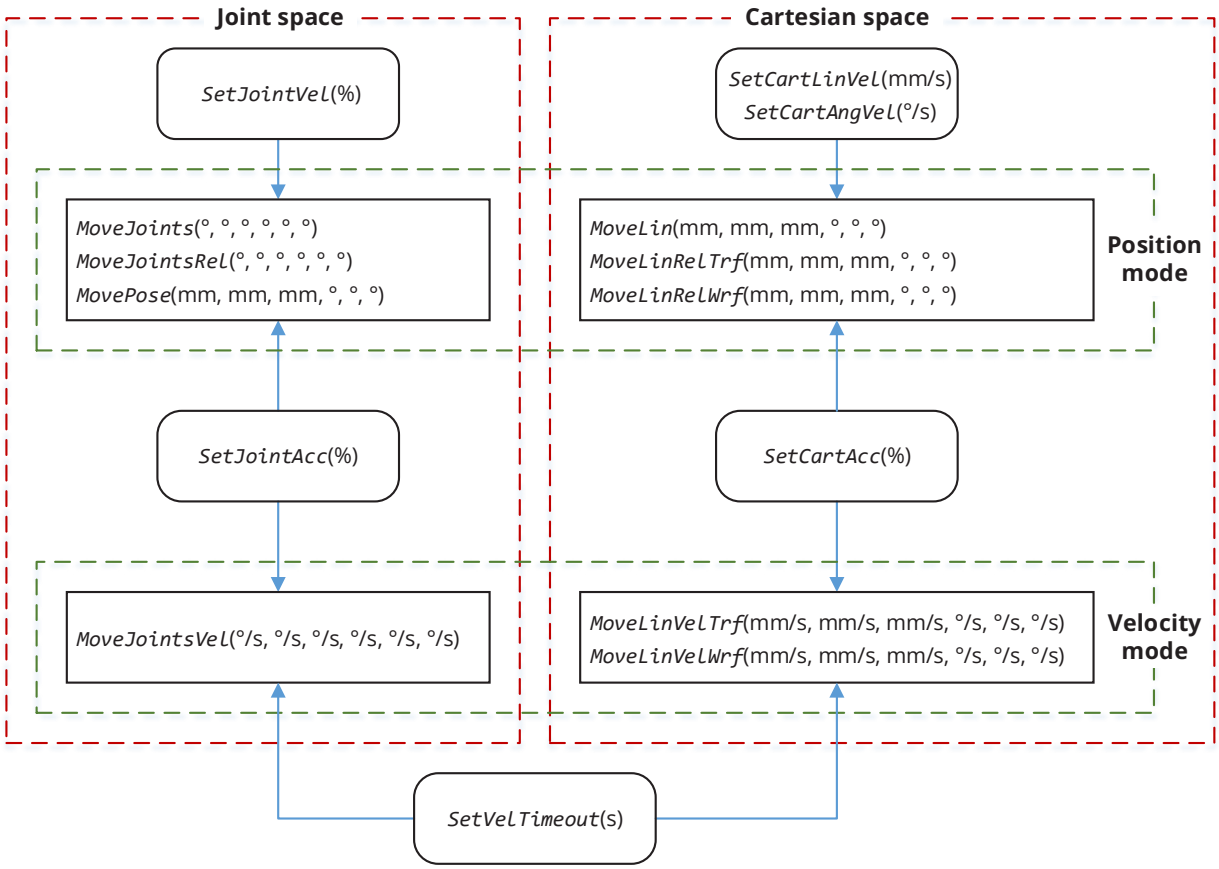


Figure 11: Settings that influence the robot motion in position and velocity modes

2. TCP/IP COMMUNICATION

The Meca500 robot must be connected to a computer or to a PLC over Ethernet. Commands may be sent through Mecademic's web interface or through a custom computer program using either the TCP/IP protocol, which is detailed in the remainder of this Section 2, or any of three cyclic protocols, which will be detailed in the remainder of this manual. When the Meca500 communicates using the TCP/IP protocol, it uses null-terminated ASCII strings. The default robot IP address is 192.168.0.100, and its default TCP port is 10000, referred to as the *control port*. Commands to and messages from the robot are sent over the control port. The robot will periodically send data over TCP port 10001, referred to as the *monitoring port*, at the rate specified by the *SetMonitoringInterval* command. This data includes the joint set and TRF pose (only when it changes), and other optional data enabled with the *SetRealTimeMonitoring* command. To avoid desynchonization between the data received from both parts, it is possible to send a copy of the monitoring port data to the control port data with the *SetCtrlPortMonitoring* command.

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

Some command descriptions refer to *default values*: these are essentially variables that are initialized every time the robot is activated. In contrast, certain parameter values are *persistants*: they have manufacturer's default values, but the changes you make to these are written on an SD drive and persist even if you power off the robot.

2.1. Motion commands

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

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

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

The robot sends a end-of-movement message (*EOM*, code 3004) whenever it has stopped moving, if this option is activated with *SetEom*. Example: if three *MoveJoints* commands are sent with blending enabled, the robot will send an EOM message only after all three *MoveJoints* commands have been executed and the robot has come to a complete stop.

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

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

Finally, motion commands can generate errors, explained in Section 2.5.1.

2.1.1 Delay(*t*)

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

Arguments

- *t*: desired pause duration in seconds.

2.1.2 GripperOpen/GripperClose

These commands are used to open or close MEGP 25E or MEGP 25LS grippers. The gripper will move its fingers apart or together until the grip force reaches 40 N. You can reduce this maximum grip force using the *SetGripperForce* command. You can also control the speed of the gripper with the *SetGripperVel* command.

By default, the *GripperOpen* and *GripperClose* commands open or close the gripper fingers until resistance is met. However, a maximum opening or closing distance can be set using the command *SetGripperRange*.

GripperOpen and *GripperClose* commands behave like a robot motion command, and will be executed only after the preceding motion command has been completed. However, if a robot motion command is sent after either 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 these commands.

2.1.3 MoveGripper(*d*)

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

The *MoveGripper* command makes the gripper fingers move towards the specified fingers opening.

Arguments

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

Unlike other position-mode *Move** commands, *MoveGripper* command does not return any error if the desired finger opening is not reached because of an object limiting the movement of the gripper fingers. The fingers will simply continue to force in the direction of the desired fingers opening with the force set by the *SetGripperForce* command, and the "holding part" gripper status will be true (see *GetRtGripperState*). If, somehow, the object is removed, the fingers will then move to the desired fingers opening. Recall that you can reduce the grip force with the *SetGripperForce* command. In addition, you can control the speed of the gripper with the *SetGripperVel* command.

The *MoveGripper* command behaves like a motion command, and will be executed only after the preceding motion command has been completed. However, if a robot motion command is sent after this 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 the *MoveGripper* command.

2.1.4 MoveJoints(θ_1,θ_2,θ_3,θ_4,θ_5,θ_6)

This command makes the robot simultaneously rotate all Its joints to the specified joint set. The robot takes a linear path in the joint space, but nonlinear in the Cartesian space. Therefore, the TCP trajectory is not easily predictable (Figure 12). Finally, with *MoveJoints*, the robot can cross singularities without any problem.

Arguments

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

  $-175° \leq \theta_1 \leq 175°$,

  $-70° \leq \theta_2 \leq 90°$,

  $-135° \leq \theta_3 \leq 70°$,

  $-170° \leq \theta_4 \leq 170°$,

  $-115° \leq \theta_5 \leq 115°$,

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

These ranges can be further limited with the command *SetJointLimits*.

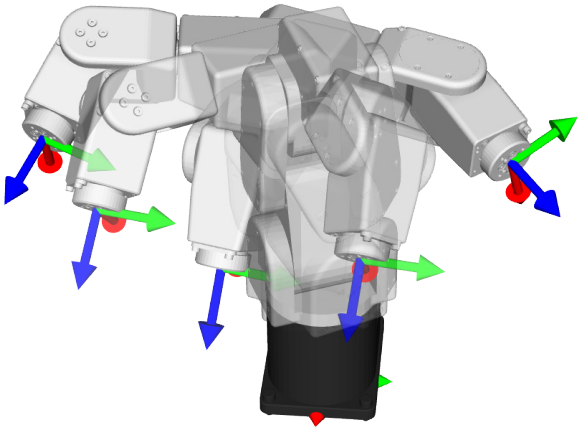


Figure 12: End-effector motion when using the *MoveJoints* or *MovePose* commands

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

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

Arguments

- $\Delta\theta_i$: the desired relative displacement of joint *i* (*i* = 1, 2, ..., 6), in degrees. The value of the argument can be positive, negative or even zero.

2.1.6 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 (Figure 12). With *MoveJointsVel*, the robot can cross singularities without any problem.

Arguments

- $\dot{\theta}_i$ the velocity of joint *i* (*i* = 1, 2, ..., 6), in °/s. The admissible ranges for the joint velocities are as follows:

  $-150°/s \leq \dot{\theta}_1 \leq 150°/s$,

  $-150°/s \leq \dot{\theta}_2 \leq 150°/s$,

  $-180°/s \leq \dot{\theta}_3 \leq 180°/s$,

  $-300°/s \leq \dot{\theta}_4 \leq 300°/s$,

  $-300°/s \leq \dot{\theta}_5 \leq 300°/s$,

  $-500°/s \leq \dot{\theta}_6 \leq 500°/s$.

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

2.1.7 MoveLin(*x*,*y*,*z*,*α*,*β*,*γ*)

This command makes the robot move its end-effector, so that its TRF ends up at a desired pose with respect to the WRF while the TCP moves along a linear path in Cartesian space, as illustrated in Figure 13. If the final (desired) orientation of the TRF is different from the initial orientation, the orientation will be modified along the path using a minimum-torque path. However, the robot will not accept the *MoveLin* command if the required end-effector reorientation is exactly 180°, because there could be two possible paths.

With this command, normally, the initial and final robot postures have to be in the same configuration, {c_s, c_e, c_w}. Only in some very peculiar cases, where the path passes exactly through a shoulder or wrist singularity, and when the automatic posture configuration selection is enabled, a change in c_s or c_w, respectively, is possible (see Section 1.2.4). If the complete motion cannot be performed due to singularities or joint limits, it will not even start, and an error will be generated.

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

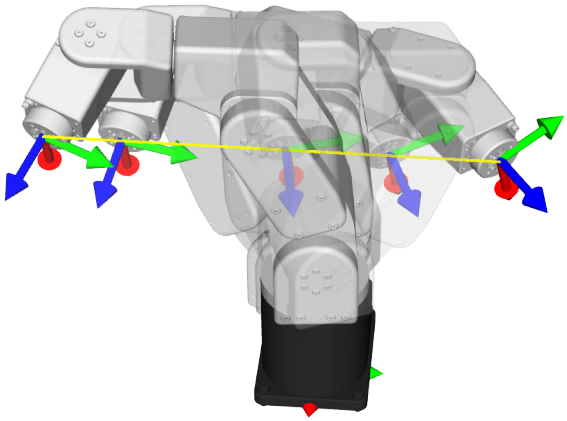


Figure 13: The TCP path when using the *MoveLin* command

Arguments

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

2.1.8 MoveLinRelTrf(*x*,*y*,*z*,*α*,*β*,*γ*)

This command has the same behavior as 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*, and *γ* represent the desired pose of the TRF with respect to the current pose of the TRF (i.e., the pose of the TRF just before executing the *MoveLinRelTrf* command).

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

Arguments

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

2.1.9 MoveLinRelWrf(*x*,*y*,*z*,*α*,*β*,*γ*)

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 and an error will be generated.

Arguments

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

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

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

Arguments

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

Note that the robot will come to a complete stop after a period of time defined by the *SetVelTimeout* command, unless another *MoveLinVelTrf* or a *MoveLinVelWrf* command is sent and, of course, unless a *PauseMotion* command is sent or some motion limit is encountered. Also, bear in mind that this command, unlike position-mode motion commands, generates no motion errors when a joint limit (including the desired turn configuration) or a singularity that cannot be crossed is reached. The robot simply stops before the limit.

2.1.11 MoveLinVelWrf($\dot{x}$,$\dot{y}$,$\dot{z}$,ω_x,ω_y,ω_z)

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

Arguments

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

Note that the robot will come to a complete stop after a period of time defined by the *SetVelTimeout* command, unless another *MoveLinVelWrf* or a *MoveLinVelTrf* command is sent and, of course, unless a *PauseMotion* command is sent or some motion limit is encountered. Also, bear in mind that this command, unlike position-mode motion commands, generates no motion errors when a joint limit (including the desired turn configuration) or a singularity that cannot be crossed is reached. The robot simply stops before the limit.

2.1.12 MovePose(*x*,*y*,*z*,*α*,*β*,*γ*)

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, including those corresponding to a singular robot posture. Then, it either chooses the joint set that corresponds to the desired robot posture and turn configurations, if such were set, or the one that is fastest to reach. Finally, it executes internally a *MoveJoints* command with the chosen joint set.

Thus, all joint rotations start and stop at the same time. The path the robot takes is linear in the joint space, but nonlinear in Cartesian space. Therefore, the path the TCP will follow to its final destination is not easily predictable, as illustrated in Figure 12.

Using this command, the robot can cross any singularity or start from a singular robot posture, or even go a singular robot posture, without any peculiarities. As with the *MoveJoints* command, if the complete motion cannot be performed due to joint limits, it will not even start, and an error will be generated.

Arguments

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

2.1.13 SetAutoConf(*e*)

This command enables or disables the automatic posture configuration selection, to be observed in the *MovePose* and *MoveLin** commands. This automatic selection, in conjunction with the turn configuration selection (Section 1.2.1 and Section 1.2.2), allows the controller to choose the "closest" joint set corresponding to the desired pose. In the case of *MoveLin** commands, enabling the automatic posture configuration selection allows the change of configuration, but only if the path happens to pass exactly through a wrist or shoulder singularity.

Arguments

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

Default values

SetAutoConf is enabled by default. If you disable it, the new desired posture configuration will be the one corresponding to the current robot position, i.e., the one after all preceding motion commands have been completed. Note, however, that if you disable the automatic posture configuration selection in a singular robot posture, the controller will automatically choose one of the 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*.

2.1.14 SetAutoConfTurn(*e*)

This command enables or disables the automatic turn selection for joint 6 (Section 1.2.1 and Section 1.2.2). It affects the *MovePose* command, and all *MoveLin** commands. When the automatic turn selection is enabled, and a *MovePose* command is executed, joint 6 will always take the shortest path, and rotate no more than 180°. In the case of a *MoveLin** command, however, enabling the automatic turn selection simply allows the change of turn configuration along the linear move.

Arguments

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

Default values

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

2.1.15 SetBlending(*p*)

This command enables/disables the robot's blending feature (Section 1.3.3). Note that there is blending only between consecutive movements with the joint-space commands *MoveJoints*, *MoveJointsRel* and *MovePose*, or between consecutive movements with the Cartesian-space commands *MoveLin*, *MoveLinRelTrf* and *MoveLinRelTrf*. For example, there will never be blending between the trajectories of a *MovePose* command followed by a *MoveLin* command.

Arguments

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

Default values

Blending is enabled at 100% by default.

2.1.16 SetCartAcc(*p*)

This command limits the Cartesian acceleration (both the linear and the angular) of the TRF with respect to the WRF during movements resulting from Cartesian-space commands (see Figure 11). 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 TRF, ranging from 0.001% to 600%.

Default values:

The default end-effector acceleration limit is 50%.

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

2.1.17 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.

The actual angular velocity may be slower than requested in some parts of the linear move (near singularities for example) in order to keep joint velocities within their physical limits.

2.1.18 SetCartLinVel(*v*)

This command limits the Cartesian linear velocity of the robot TRF with respect to its WRF. It only affects the movements generated by the *MoveLin*, *MoveLinRelTrf* and *MoveLinRelWrf* commands.

Arguments

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

Default values

The default TCP velocity is 150 mm/s.

The actual TCP velocity may be slower than requested in some parts of the linear move (near singularities for example) in order to keep joint velocities within their physical limits.

2.1.19 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. If a checkpoint is the last queued command, in the absence of blending with another command, the checkpoint response will be sent once the robot has come to a stop (along with an EOB). Finally, note that you can use the same checkpoint number multiple times.

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

Arguments

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

Responses

[3030][*n*]

2.1.20 SetConf(c_s,c_e,c_w)

This command sets the *desired* posture configuration to be observed in the *MovePose* and *MoveLin** commands (see Section 1.2.1 and Section 1.2.2). When a desired posture configuration is set, a *MovePose* command will be executed only if the final robot position can be in the desired posture configuration. In contrast, when a desired posture configuration is set, a *MoveLin** command will be executed only if the final robot position can be and the initial robot position already is in the desired posture configuration. The posture configuration can be automatically selected, when executing a *MovePose* or *MoveLin** command, by using the *SetAutoConf* command. Using *SetConf* automatically disables the automatic posture configuration selection.

Arguments

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

Default values

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

2.1.21 SetConfTurn(c_t)

This command sets the desired turn configuration for joint 6 to be observed in the *MovePose* and *MoveLin** commands (see Section 1.2.1 and Section 1.2.2). When a turn configuration is set, a *MovePose* command is executed only if the final robot position can be in the desired turn configuration. In contrast, when a desired turn configuration is set, a *MoveLin** command will be executed only if the final robot position can be and the initial robot position already is in the desired turn configuration. The turn configuration can be automatically selected, when executing a *MovePose* or *MoveLin** command, by using the *SetAutoConf* command. Using *SetConfTurn* automatically disables the automatic turn configuration selection.

This command is only useful if you have a wired end-effector with long enough cables to allow joint 6 to rotate more than ±180°. For example, if using the MEGP 25E gripper, limit joint 6 to ±180° using the *SetJointLimits* command and then use either *SetAutoConfTurn*(1) or *SetConfTurn*(0). If using a cableless end-effector, then the automatic turn configuration should never be disabled. However, remember to always bring joint 6 within the ±420° range before powering the robot off (recall Section 1.3.1).

Arguments

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

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

Default values

Enabled by default (see *SetAutoConfTurn*), so when you start the robot, there is no default desired turn configuration. The only way to set a desired turn configuration is to specify it with the command *SetConfTurn* or to execute the command *SetAutoConfTurn*(0). The latter sets the desired turn configuration to the one of the current position of joint 6.

2.1.22 SetGripperForce(*p*)

This command limits the grip force of Mecademic grippers.

Arguments

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

Default values

By default, the grip force limit is 50%.

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

This command sets the closed and open states of the gripper and is used mainly to redefine the actions of the *GripperClose* and *GripperOpen* commands, respectively. The *SetGripperRange* command is useful for the MEGP 25LS gripper. If, for example, you are manipulating parts that require fingers opening between 10 mm and 20 mm, but the allowable range of the gripper as detected during the homing is 48 mm, it would be more efficient to redefine the actions of the *GripperClose* and *GripperOpen* commands by calling *SetGripperRange*(8,22), or else the fingers will move more than necessary, and therefore increase your cycle time.

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

Arguments

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

Default values

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

2.1.24 SetGripperVel(*p*)

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

Arguments

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

Default values

By default, the finger velocity limit is 50%.

2.1.25 SetJointAcc(*p*)

This command limits the acceleration of the joints during movements resulting from joint- space commands (see Figure 11). Note that this command makes the robot stop, even if blending is enabled.

Arguments

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

Default values

The default joint acceleration limit is 100%.

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

2.1.26 SetJointVel(*p*)

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

Arguments

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

Default values

By default, the limit is set to 25%.

It is not possible to limit the velocity of only one joint. With this command, the maximum velocities of all joints are limited proportionally. The maximum velocity of each joint will be reduced to a percentage *p* of its top velocity. The top velocity of joints 1 and 2 is 150°/s, of joint 3 is 180°/s, of joints 4 and 5 is 300°/s, and of joint 6 is 500°/s.

2.1.27 SetTorqueLimits(p_1,p_2,p_3,p_4,p_5,p_6)

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

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

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

Arguments

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

Default values

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

2.1.28 SetTorqueLimitsCfg(*s*,*m*)

This command sets the robot behavior when a joint torque exceeds the threshold set by the *SetTorqueLimits* command. It also sets the filtering type used for accurate detection. It also sends a torque limit status every time torque limit status changes (exceeded or not) for events severity greater than 0. Torque limit error is sent when torque exceeds the limit for severity 4.

Arguments

- *s*: integer defining the torque limit event severity as
  - 0, no action;
  - 1, trace warning;
  - 2, pause motion;
  - 3, clear motion;
  - 4, error.
- *m*: integer defining the detection mode as 0, always detect; 1, skip detection during acceleration/deceleration and blending.

Default values

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

2.1.29 SetTrf(*x*,*y*,*z*,*α*,*β*,*γ*)

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 with respect to the FRF, in mm;

- α, β,and γ: the Euler angles representing the orientation of the TRF with respect to the FRF, in degrees.

Default values

By default, the TRF coincides with the FRF.

2.1.30 SetValveState(v_1,v_2)

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

Arguments

- v_1: open (1), close (0) or keep unchanged (−1) valve 1;
- v_2: open (1), close (0) or keep unchanged (−1) valve 2.

Default values

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

Responses

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

2.1.31 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.

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

2.1.32 SetWrf(*x*,*y*,*z*,α,β,γ)

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

Default values

By default, the WRF coincides with the BRF.

2.2. General request commands

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

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

2.2.1 ActivateRobot

This command activates all motors and disables the brakes on joints 1, 2, and 3. It must be sent before homing.

Responses

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

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

2.2.2 ActivateSim/DeactivateSim

The Meca500 supports a simulation mode in which all of the robot's hardware including Mecademic's EOAT (see *SetExtToolSim*) are simulated and nothing moves. This mode allows you to test programs with the robot's hardware (i.e., hardware-in-the-loop simulation), without the risk of damaging the robot or its surroundings. Simulation mode can be activated and deactivated with the *ActivateSim* and *DeactivateSim* commands (this command can only be executed when the robot is deactivated).

Responses

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

2.2.3 ClearMotion

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

Responses

[2044][The motion was cleared.]

2.2.4 DeactivateRobot

This command disables all motors and engages the brakes on joints 1, 2, and 3. This command should be run before powering down the robot.

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

Responses

[2004][Motors deactivated.]

2.2.5 BrakesOn/BrakesOff

These commands engages or disengages the brakes of joints 1, 2 and 3. When the brakes are released, the robot will fall down. This command is only available when the robot is deactivated.

Responses

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

2.2.6 EnableEtherNetIp(*e*)

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

Arguments

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

Default values

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

2.2.7 EnableProfinet(*e*)

This command enables or disables PROFINET slave stack, allowing the robot to be controlled by a PROFINET controller. Please note that enabling PROFINET also enables LLDP packets forwarding between the two Ethernet ports of the robot.

Arguments

- *e*: enable (1) or disable (0) PROFINET.

Default values

PROFINET is disabled when the robot is shipped from Mecademic. Changes in this setting have a persistent effect (remain even after a powering the robot off).

2.2.8 GetFwVersion

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

Responses

[2081][vx.x.x]

2.2.9 GetModelJointLimits(*n*)

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

Arguments

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

Responses

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

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

- $\theta_{n,\text{min}}$: lower joint limit, in degrees;
- $\theta_{n,\text{max}}$: upper joint limit, in degrees.

2.2.10 GetProductType

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

Responses

[2084][Meca500]

2.2.11 GetRobotName

This command returns the robot's name, set with the command *SetRobotName*.

Note that the robot name is used as a host name when the robot's network configuration uses DHCP.

Responses

[2095][*s*]

- *s*: string containing the robot's name.

2.2.12 GetRobotSerial

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

Responses

[2083][robot's serial number]

2.2.13 Home

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

Responses

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

The first response (2002) is sent if homing was completed successfully, while the second one (2003) is sent if the robot is already homed. The third response (1032) is sent if the homing procedure failed because it was started while a robot joint was outside its user-defined limits. The last response (1014) is sent if the homing failed for other reasons.

2.2.14 LogTrace(*s*)

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

Arguments

- *s*: a text string (the comment).

Responses

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

2.2.15 PauseMotion

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

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

The *PauseMotion* command generates the following two responses: the first (2042) is always sent, whereas the second (3004) is sent only if the robot was moving when it received the *PauseMotion* command. If a motion error occurs while the robot is paused (e.g., if another moving body hits the robot), the motion is cleared and can no longer be resumed.

Responses

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

2.2.16 ResetError

This command resets the robot error status. It can generate one of the following two responses: the first response (2005) is generated if the robot was indeed in an error mode, while the second one (2006) is sent if the robot was not in error mode.

Responses

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

2.2.17 ResetPStop

As described in the User Manual of the Meca500 R3, you can connect one 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 immediately put in protective stop, and the message [3032][1] is returned. To exit the protective stop, you must first remove the voltage from the P-Stop 2 terminals. Then, you must send the command *ResetPStop*, which resets the protective stop and generates the message [3032][0].

Responses

[3032][*e*]

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

2.2.18 ResumeMotion

This command resumes the robot movement, if it was previously paused with the command *PauseMotion*. 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.

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. Motion commands sent while the robot is paused will be placed in the queue.

This command must also be sent after the *ClearMotion* command. However, the robot will not move until another motion command is received (or retrieved from the motion queue). This command must also be sent after the *ResetError* command.

Responses

[2043][Motion resumed.]

2.2.19 SetCtrlPortMonitoring(*e*)

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

Arguments

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

Default values

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

Responses

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

2.2.20 SetEob(*e*)

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

Arguments

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

Default values

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

Responses

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

Mecademic does not recommend using the "End of block" message to detect that a program finished executing. Use the command *SetCheckpoint* instead.

2.2.21 SetEom(*e*)

The robot can also send the "[3004][End of movement.]" message 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. 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) the end-of-movement message.

Default values

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

Responses

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

2.2.22 SetExtToolSim(*e*)

This command enables the emulation of one of Mecademic's EOAT. The emulation mode is also automatically enabled or disabled with the *ActivateSim* or *DeactivateSim* commands. You can emulate any of Mecademic's EOAT, even if you have another of these three already installed on the robot.

The robot doesn't need to be deactivated to enable/disable simulation of its physical tool. However, to enable simulation of a tool different from the physical one, you need to deactivate the robot first.

Arguments

- *m*: tool model, where 0 stands for no tool, 1 for current external tool type, 10 for the MEGP 25E gripper, 11 for the MEGP 25LS gripper, and 20 for the MPM500 pneumatic module.

Default values

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

Responses

[2047][*m*]

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

This command redefines the lower and upper limits of a robot joint. It can only be executed while the robot is deactivated. For these user-defined joint limits to be taken into account, you must execute the command *SetJointLimitsCfg*(1). Obviously, the new joint limits must be within the default joint limits (Section 1.1.5) and all the robot joints position must be within the requested limits. Note that these user-defined joint limits remain active even after you power down the robot.

Use *SetJointLimits*(*n*,0,0) to reset the joint limits of a joint to its factory values.

Arguments

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

Responses

[2092][*n*]

2.2.24 SetJointLimitsCfg(*e*)

This command enables or disables the user-defined limits set by the *SetJointLimits* command. It can only be executed while the robot is deactivated. If the user-defined limits are disabled, the default joint limits become active. However, user-defined limits remain in memory, and can be re-enabled, even after a power down.

Example: one of the wrist joints has been inadvertently rotated outside its activated, user-defined limits, preventing the robot from homing. In this situation, you can enable the recovery mode (see *SetRecoveryMode*) which will allow activating the robot even when joints are outside user-defined limits.

Arguments

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

Responses

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

If some robot joints are inadvertently moved outside the defined limits, the robot will refuse to activate. Enable the recovery mode (see Section 1.3.2) to allow moving the robot even when joints are outside the configured limits.

2.2.25 SetMonitoringInterval(*t*)

This command is used to set the time interval at which real-time feedback from the robot is sent from the robot over TCP port 10001 (see the description for *SetRealTimeMonitoring* and Section 2.5.4 for more details).

Arguments

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

Default values

By default, the monitoring time interval is 0.015 s.

2.2.26 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 deactivated. New parameter values will take effect only after a robot reboot.

Arguments

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

Default values

By default, n_1 = 3.

2.2.27 SetOfflineProgramLoop(*e*)

This command is used to define whether the program that is to be saved must later be executed a single time or an infinite number of times, when pressing the Start/Stop button on the robot's base. It has effect only on program number 1 and only when starting a program using the Start/Stop button (not when starting a program using the *StartProgram* command).

Arguments

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

Default values

By default, looping is disabled.

Responses

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

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

2.2.28 SetRealTimeMonitoring(n_1,n_2,...)

TCP port 10001 (i.e., the monitoring port) transmits the robot's joint set and TRF pose, as well as other data (see Section 2.5.4), at the rate specified by the *SetMonitoringInterval* command.

You can enable the transmission of various other real-time data over the monitoring port, with the difference that they are preceded by a monotonic timestamp in microseconds (see *SetRtc*). The arguments of which are a list of numerical codes or alphabetical names.

You can send this command even if the robot is not activated and get the same responses as with the *GetRt** and *GetRtTarget** commands, but on the monitoring port, instead of on the control port, and every monitoring interval, rather than only when requested.

Arguments

- n_1, n_2, ...: a list of number codes or names, as follows:
  - 2200 or TargetJointPos, for the response of the *GetRtTargetJointPos* command;
  - 2201 or TargetCartPos, for the response of the *GetRtTargetCartPos* command;
  - 2202 or TargetJointVel, for the response of the *GetRtTargetJointVel* command;
  - 2204 or TargetCartVel, for the response of the *GetRtTargetCartVel* command;
  - 2210 or JointPos, for the response of the *GetRtJointPos* command;
  - 2211 or CartPos, for the response of the *GetRtCartPos* command;
  - 2212 or JointVel, for the response of the *GetRtJointVel* command;
  - 2213 or JointTorq, for the response of the *GetRtJointTorq* command;
  - 2214 or CartVel, for the response of the *GetRtCartVel* command;
  - 2218 or Conf, for the response of the *GetRtConf* command (sent only when changed);
  - 2219 or ConfTurn, for the response of the *GetRtConfTurn* command (sent only when changed);
  - 2220 or Accel, for the response of the *GetRtAccelerometer* command;
  - 2227 or Checkpoint, for every new checkpoint reached, preceded by a timestamp;

- 2321 or GripperForce, for the response of the *GetRtGripperForce* command;
- 2322 or GripperPos, for the response of the *GetRtGripperPos* command;
- 2323 or GripperVel, for the response of the *GetRtGripperVel* command;
- All, to enable all of the above responses.

Default values

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

Responses

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

- n_1, n_2, ...: a list of response codes.

The *SetRealTimeMonitoring* command does not have a cumulative effect; if you execute the command *SetRealTimeMonitoring*(All) and then the command *SetRealTimeMonitoring*(TargetCartPos) or the command *SetRealTimeMonitoring*(2201), you will only enable message 2201.

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

2.2.29 SetRobotName(*s*)

This command allows you to change the robot's name. The change is persistent and remains even after power-down. The command is useful when multiple Mecademic robots are connected on the same network. The *SetRobotName* command also changes the hostname of the robot in the case of a DHCP connection. The robot's name is displayed in the upper right corner of the web interface, as well as in the browser tab hosting the web interface. You can also retrieve the robot's name with the command *GetRobotName*.

The command can only be executed while the robot is powered but not activated.

Arguments

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

Default values

By default, the robot's name is m500.

Responses

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

2.2.30 SetRecoveryMode(*e*)

As discussed in Section 1.3.1, homing the robot when the robot is too close to an obstacle may lead to a collision. Moving the robot when its joints are outside the user-defined limits is impossible. For these two situations, it is useful to enable the *SetRecoveryMode* command.

When the recovery mode is enabled, and the robot is activated, virtually all motion commands are accepted, but joint and Cartesian velocities and accelerations are significantly limited, for safety reasons. Similarly, in recovery mode, you can still control the Mecademic grippers or the MPM500 pneumatic module connected to the robot, but the gripping force and velocity of the grippers are limited, for safety reasons. Finally, in recovery mode, you can move outside the user-defined joint limits.

If the robot was not homed before enabling the recovery mode, the robot movements will be less accurate. The same applies for the movements of the Mecademic grippers, if such a gripper was installed on the robot. In addition, you would not be able to use the *MoveGripper* command, but can still use the *GripperOpen* and *GripperClose* commands.

If the robot was already homed, when the recovery mode was enabled, the robot and the grippers will be as accurate as before and you can still use the *MoveGripper* command.

Arguments

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

Default values

By default the recovery mode is deactivated.

Responses

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

2.2.31 SetRtc(*t*)

Since our robots do not have batteries, when a Meca500 is powered on, the internal clock starts at the date at which the robot image was built. Each time you connect to the robot via the web interface, the internal clock of the robot is automatically adjusted to UTC. Other than connecting to the robot using the Web Portal, another solution is to send the *SetRtc* command to the robot (from the PLC or any application controlling the robot), if you want all timestamps in the robot's log files to be with respect to UTC.

Arguments

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

2.2.32 StartProgram(*n*)

This command starts a program that has been previously saved in the robot's memory. The robot must be activated and homed before running a program. Executing this command will launch the program *n* only once.

Alternately, pressing the Start/Stop button on the robot base will start program 1 (and only program 1) and execute it the number of times defined by the *SetOfflineProgramLoop* command.

Arguments

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

Responses

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

2.2.33 StartSaving(*n*)

This command is used to save commands in the robot's internal memory. These are referred to as *offline programs* that can later be played using the *StartProgram* command or by pressing the Start/Pause button on the robot's base.

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

The robot records all commands sent between the *StartSaving* and *StopSaving* commands.

The robot will execute but not record *Get** commands (*GetBlending*, *GetRtCartPos*, etc.). If the robot receives a change of state command (*BrakesOn*, Home, *PauseMotion*, *SetEom*, etc.) while recording, it will abort saving the program. FInally, only program 1 can be executed using the Start/Pause button on the robot base.

Arguments

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

Responses

[2060][Start saving program.]

2.2.34 StopSaving

This command will make the controller save the program and stop saving. Two responses will be generated: the first (2061) and the second (2064) or third (2065) of the three responses given below. If you send this command while the robot is not saving a program, the fourth response (1022) will be returned.

Responses

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

2.2.35 SyncCmdQueue(*n*)

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

Arguments

- *n* : a non-negative integer number, ranging from 0 to 4,294,967,295.

Responses

[2097][*n*]

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

2.2.36 SwitchToEtherCat

This command will disable the TCP/IP, EtherNet/IP and PROFINET protocols and enable EtherCAT instead (EtherCAT is an exclusive protocol that cannot be used at the same time as other Ethernet-based protocols, see Section 4).

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

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

1. Use the appropriate EtherCAT command (Section 4.1.3).
2. Perform a network configuration reset (press and hold the power button on the robot base while the robot is rebooting (may require up to 60 seconds).

2.2.37 TcpDump(*n*)

This command starts an Ethernet capture (pcap format) on the robot, for the specified duration. The Ethernet capture will be part of the logs archive, which can be retrieved using the Get logs option in the web portal Options menu.

Arguments

- *n*: duration in seconds.

Responses

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

2.2.38 TcpDumpStop

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

Responses

[3036][TCP dump capture stopped.]

2.3. Data request commands

The request (*Get**) commands in this section generally return (on TCP port 10000) values for parameters that have already been configured (sent and executed) with a *Set** command (or the default values).

Motion commands sent to the robot are executed one after the other, while *Get** commands are executed immediately. Therefore, if you send a *SetTrf* command, then a *MovePose* command, then another *SetTrf* command, and immediately after that a *GetTrf* command, you will get the arguments of the first *SetTrf* command.

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

2.3.1 GetAutoConf

This command returns the state of the automatic posture configuration selection, which can be affected by *SetAutoConf* and *SetConf* commands.

Responses

[2028][*e*]

– *e*: enabled (1) or disabled (0).

2.3.2 GetAutoConfTurn

This command returns the state of the automatic turn configuration selection, which can be affected by *SetAutoConfTurn* and *SetConfTurn* commands.

Responses

[2031][*e*]

– *e* enabled (1) or disabled (0).

2.3.3 GetBlending

This command returns the blending percentage, set with the *SetBlending* command.

Responses

[2150][*p*]

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

2.3.4 GetCartAcc

This command returns the desired limit of the acceleration of the TRF with respect to the WRF, set by the command *SetCartAcc*.

Responses

[2156][*p*]

– *p*: percentage of maximum acceleration of the TRF.

2.3.5 GetCartAngVel

This command returns the desired limit of the angular velocity of the TRF with respect to the WRF, set by the command *SetCartAngVel*.

Responses

[2155][*ω*]

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

2.3.6 GetCartLinVel

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

Responses

[2154][*v*]

– *v*: TCP velocity limit, in mm/s.

2.3.7 GetCheckpoint

This command returns the argument of the last executed *SetCheckpoint*.

Responses

[2157][*n*]

– *n*: checkpoint number.

2.3.8 GetConf

This command returns the desired posture configuration (see Figure 5), or more precisely, the posture configuration that will be applied to the next *MovePose* or *MoveLin** command in the motion queue. This is either the posture configuration explicitly specified with the *SetConf* command, or the one that was automatically assigned when the *SetAutoConf*(0) command was executed.

Responses

[2029][c_s, c_e, c_w]

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

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

2.3.9 GetConfTurn

This command returns the desired turn configuration (see Figure 5), or more precisely, the turn configuration that will be applied to the next *MovePose* or *MoveLin** command in the motion queue. Recall that this is either the turn configuration that you have explicitly specified with the command *SetConfTurn*, or the one that was automatically assigned when the command *SetAutoConfTurn*(0) was executed.

Responses

[2036][c_t]

- c_t: turn configuration parameter, an integer from −100 to 100†.

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

2.3.10 GetGripperForce

This command returns the percentage of maximum grip force for the Mecademic grippers. This percentage is set by the *SetGripperForce* command.

Responses

[2158][p]

- p: percentage of maximum grip force.

2.3.11 GetGripperRange

This command returns the allowable range for the fingers opening of the Mecademic grippers as detected during homing or defined with the *GetGripperRange* command.

Responses

[2162][d_{closed}, d_{open}]

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

2.3.12 GetGripperVel

This command returns the percentage of maximum finger velocity for the Mecademic grippers. This percentage is set by the *SetGripperVel* command.

Responses

[2159][*p*]

– *p*: percentage of maximum velocity of the gripper fingers.

2.3.13 GetJointAcc

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

Responses

[2153][*p*]

– *p*: percentage of maximum joint accelerations.

2.3.14 GetJointLimits(*n*)

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

Arguments

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

Responses

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

– *n*: joint number, an integer ranging from 1 to 6;

– $\theta_{n,\min}$: lower joint limit, in degrees;

– $\theta_{n,\max}$: upper joint limit, in degrees.

2.3.15 GetJointLimitsCfg

This command returns the status of the user-enabled joint limits, defined by *SetJointLimitsCfg*.

Responses

[2094][*e*]

– *e*: status, 1 for enabled, 0 for disabled.

2.3.16 GetJointVel

This command returns the desired joint velocity reduction factor, set with the *SetJointVel* command.

Responses

[2152][*p*]

– *p*: percentage of maximum joint velocities.

2.3.17 GetMonitoringInterval

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

Responses

[2116][*t*]

- *t*: time interval in seconds.

2.3.18 GetNetworkOptions

This command returns the parameters affecting the network connection.

Responses

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

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

2.3.19 GetRealTimeMonitoring

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

Responses

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

2.3.20 GetTorqueLimits

This command returns the desired joint torque thresholds, set with the *SetTorqueLimits* command.

Responses

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

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

2.3.21 GetTorqueLimitsCfg

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

Responses

[2160][*s*, *m*]

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

2.3.22 GetTrf

This command returns the current definition of the TRF with respect to the FRF, set with the *SetTrf* command.

Responses

[2014][*x*, *y*, *z*, *α*, *β*, *γ*]

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

2.3.23 GetVelTimeout

This command returns the timeout for velocity-mode motion commands, set with the *SetVelTimeout* command.

Responses

[2151][*t*]

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

2.3.24 GetWrf

This command returns the current definition of the WRF with respect to the BRF, set with the *SetWrf* command.

Responses

[2013][*x*, *y*, *z*, *α*, *β*, *γ*]

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

2.4. Real-time data request commands

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

There are two types of robot positioning real-time data commands. The first type returns data according to real-time measurements from the robot sensors:

- *GetRtJointTorq*: returns the current joint torques, as measured by the motor currents.
- *GetRtAccelerometer*: returns the current acceleration in link 5, as measured by the embedded accelerometer.
- *GetRtJointPos*: returns the current joint set, as measured by the joint encoders.
- *GetRtCartPos*: returns the current TRF pose as calculated from the real-time joint encoder measurements.
- *GetRtJointVel*: returns the current joint velocities as calculated from the real-time joint encoder measurements.
- *GetRtCartVel*: returns the current Cartesian velocity as calculated from the real-time joint encoder measurements.
- *GetRtConf*: returns the current posture configuration as calculated from the real-time joint encoder measurements.

- *GetRtConfTurn*: returns current turn configuration as calculated from the real-time joint encoder measurements.

The second type returns real-time targets calculated by the trajectory planner:

- *GetRtTargetJointPos*: returns the current target joint pose as calculated by the trajectory planner.
- *GetRtTargetCartPos*: returns the current target TRF pose data as calculated by the trajectory planner.
- *GetRtTargetJointVel*: returns the current target joint velocities as calculated by the trajectory planner.
- *GetRtTargetCartVel*: returns the current target Cartesian velocity as calculated by the trajectory planner.
- *GetRtTargetConf*: returns the current target posture configuration as calculated by the trajectory planner.
- *GetRtTargetConfTurn*: returns the current target turn configuration as calculated by the trajectory planner.

For example, if the robot is active and homed, but not moving, the *GetRtTargetJointPos* command will always return the same joint set, as long as the robot remains stationary. In reality, the robot is never perfectly still since the drives are constantly controlling the motors. This means that the joints oscillate approximately ±0.001° around the desired joint angles. Thus, if you execute the command *GetRtJointPos* twice in a row while the robot is "not moving", you will see that the joint values may differ by a couple of micro-degrees.

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

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

All of the commands in this section return responses on TCP port 10000.

2.4.1 GetCmdPendingCount

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

Responses

[2080][*n*]

Note that the robot will compile several (~25) commands in advance. These compiled commands are not included in this count though they may not yet have started executing.

2.4.2 GetJoints

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

Responses

[2026][θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]

– θ_i: the angle of joint *i*, in degrees (*i* = 1, 2, ..., 6).

2.4.3 GetPose

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

Responses

[2027][x, y, z, α, β, γ]

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

2.4.4 GetRtAccelerometer(*n*)

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

Arguments

- *n*: link number, currently must be 5.

Responses

[2220][t, n, a_x, a_y, a_z]

- t: timestamp in microseconds;
- n: link number, currently 5;
- a_x, a_y, a_z: acceleration in link 5, measured with respect to the WRF, and in units such that 16,000 is equivalent to 9.81 m/s^2 (i.e., 1g).

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

2.4.5 GetRtc

This command returns the current Epoch Time in seconds, set with *SetRtc*, after every reboot of the robot. Note that this is different from the timestamp returned by all *GetRt** commands, which is in microseconds. Furthermore, these two time measurements have different zero references.

Responses

[2140][t]

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

2.4.6 GetRtCartPos

This command returns the pose of the TRF with respect to the WRF, as calculated from the current joint set read by the joint encoders. In also returns a timestamp.

Responses

[2211][t, x, y, z, α, β, γ]

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

2.4.7 GetRtCartVel

This command returns the current Cartesian velocity vector of the TRF with respect to the WRF, as calculated from the real-time data coming from the joint encoders.

Responses

[2214][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z]

- t: timestamp in microseconds;
- $\dot{x}$, $\dot{y}$, $\dot{z}$: components of the linear velocity vector of the TCP with respect to the WRF, in mm/s.
- ω_x, ω_y, ω_z: components of the angular velocity vector of the TRF with respect to the WRF, in °/s.

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

2.4.8 GetRtConf

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

Responses

[2218][t, c_s, c_e, c_w]

- t: timestamp in microseconds;
- c_s: shoulder configuration parameter, either −1 or 1†;
- c_e: elbow configuration parameter, either −1 or 1†;
- c_w: wrist configuration parameter, either −1 or 1†.

† at the corresponding singularity, we return 0, but display the text "n/a" in the web interface.

2.4.9 GetRtConfTurn

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

Response

[2219][t, c_t]

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

2.4.10 GetRtExtToolStatus

This command returns the general status of the external tool connected to the I/O port of the Meca500, preceded with a timestamp. For additional status information, use the commands *GetRtGripperState* or *GetRtValveState*.

Responses

[2300][*t*, *simType*, *phyType*, *hs*, *es*, *oh*]

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

2.4.11 GetRtGripperForce

This command returns the currently applied grip force of the Mecademic gripper, preceded by a timestamp.

Responses

[2321][*t*, *p*]

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

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

2.4.12 GetRtGripperPos

This command returns the current fingers opening of Mecademic grippers (see MoveGripper), preceded with a timestamp.

Responses

[2322][*t*, *p*]

- *t*: timestamp in microseconds;
- *d*: fingers opening.

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

2.4.13 GetRtGripperState

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

Responses

[2320][*t*, *hp*, *dr*, *gc*, *go*]

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

2.4.14 GetRtGripperVel

This command returns the current finger velocity, as percentage of the maximum finger velocity for the Mecademic grippers.

Responses

[2323][*t*, *p*]

- *t*: timestamp in microseconds;
- *p*: current finger velocity, as signed percentage of maximum velocity of the gripper fingers.

2.4.15 GetRtJointPos

This command returns the current joint set read by the joint encoders. It also returns a timestamp.

Responses

[2210][*t*, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]

- *t*: timestamp in microseconds;
- θ_i: the angle of joint *i*, in degrees (*i* = 1, 2, ..., 6).

2.4.16 GetRtJointTorq

This command returns the current joint torques.

Responses

[2213][*t*, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6]

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

2.4.17 GetRtJointVel

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

Responses

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

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

2.4.18 GetRtTargetCartPos

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

Responses

[2201][t, x, y, z, α, β, γ]

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

2.4.19 GetRtTargetCartVel

This command returns the current target Cartesian velocity vector of the TRF with respect to the WRF.

Responses

[2204][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z]

- t: timestamp in microseconds;
- $\dot{x}$, $\dot{y}$, $\dot{z}$: components of the linear velocity vector of the TCP with respect to the WRF, in mm/s.
- ω_x, ω_y, ω_z: components of the angular velocity vector of the TRF with respect to the WRF, in °/s.

2.4.20 GetRtTargetConf

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

Responses

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

- t: timestamp in microseconds;
- c_s: shoulder configuration parameter, either −1 or 1†;
- c_e: elbow configuration parameter, either −1 or 1†;
- c_w: wrist configuration parameter, either −1 or 1†.

† at the corresponding singularity, we return 0, but display the text "n/a" in the web interface.

2.4.21 GetRtTargetConfTurn

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

Responses

[2209][t, c_t]

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

2.4.22 GetRtTargetJointPos

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

Responses

[2200][t, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]

- t: timestamp in microseconds;
- θ_i: the angle of joint i, in degrees (i = 1, 2, ..., 6).

2.4.23 GetRtTargetJointTorq

This command returns the current target joint torques.

Responses

[2203][t, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6]

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

2.4.24 GetRtTargetJointVel

This command returns the current target joint velocities.

Responses

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

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

2.4.25 GetRtTrf

This command returns the current definition of the TRF with respect to the FRF, set by the *SetTrf* command. It returns exactly the same pose as the *GetTrf* command, but the response code is different and a timestamp precedes the pose data.

Responses

[2229][t, x, y, z, α, β, γ]

- t: timestamp in microseconds;
- x, y, z: the coordinates of the origin of the TRF with respect to the FRF, in mm;

- α, β, γ: the Euler angles representing the orientation of the TRF with respect to the FRF, in degrees.

2.4.26 GetRtValveState

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

Responses

[2310][t, v_1, v_1]

- t: timestamp in microseconds;
- v_2: state of valve 1 (0 if closed, 1 if open);
- v_1: state of valve 2 (0 if closed, 1 if open).

2.4.27 GetRtWrf

This command returns the current definition of the WRF with respect to the BRF, set by the *SetWrf* command. It returns exactly the same pose as the *GetWrf* command, but the response code is different and a timestamp precedes the pose data.

Responses

[2228][t, x, y, z, α, β, γ]

- t: timestamp in microseconds;
- x, y, z: the coordinates of the origin of the WRF with respect to the BRF, in mm;
- α, β, γ: he Euler angles representing the orientation of the WRF with respect to the BRF, in degrees.

2.4.28 GetStatusGripper

This deprecated command returns the gripper's status, but it is deprecated as of firmware 9.0. Use *GetRtExtToolStatus* or *GetRtGripperState* instead.

Responses

[2079][*ge*, *hs*, *hp*, *lr*, *es*, *oh*]

- *ge*: gripper enabled, i.e., present (0 for disabled, 1 for enabled);
- *hs*: homing state (0 for homing not performed, 1 for homing performed);
- *hp*: holding part (0 if the gripper does not hold a part, 1 otherwise);
- *lr*: limit reached (0 if the fingers are not fully open or closed, 1 otherwise);
- *es*: error state (0 for absence of error, 1 for presence of error);
- *oh*: overheat (0 if there is no overheat, 1 if the gripper is in overheat).

2.4.29 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 not moving and motion queue is empty, 0 otherwise);
- *eom*: end of movement status (1 if robot is not moving, 0 if robot is moving).

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

2.4.30 GetTorqueLimitsStatus

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

Responses

[3028][*s*]

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

2.5. Responses and messages

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

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

The four-digit code indicates the type of response:

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

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

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

2.5.1 Command error messages

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

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 (... is not in range [...,...] for joint ...). - Command: '...'.]	The robot cannot execute the *MoveJoints* or *MoveJointsRel* command because at least one of its joints is either already or will become outside the user-defined limits.
[1010][Linear move is blocked because a joint would rotate by more than 180deg. - Command: '...']	The linear motion cannot be executed because it requires a reorientation of 180° of the end-effector, and there may be two possible paths.
[1011][The robot is in error.]	A command has been sent but the robot is in error mode and cannot process it until a *ResetError* command is sent.
[1012][Linear move is blocked because it requires a reorientation of 180 degrees of the end- effector - Command: '...'.]	The *MoveLin* or *MoveLinRel** command sent requires that the robot pass through a singularity that cannot be crossed or pass too close to a singularity with excessive joint rotations.
[1013][Activation failed.]	Activation failed. Try again.
[1014][Homing failed.]	Homing procedure failed. Try again.
[1016][Destination pose out of reach for any configuration. - Command: '...'] [1016][Destination pose out of reach for selected conf(...,...,... turn ...). - Command: '...'] [1016][The requested linear move is not possible due to a pose out of reach along the path. - Command: '...']	The pose requested in the *MoveLin*, *MoveLinRel** or *MovePose* command is out of reach, with the desired (or with any) configurations. In the case of the *MoveLin* command, this error code is also produced if a pose along the path is out of reach.
[1022][Robot was not saving the program.]	The *StopSaving* command was sent, but the robot was not saving a program.
[1023][Ignoring command for offline mode. - Command: '...']	The command cannot be executed in the offline program.
[1024][Mastering needed. - Command: '...']	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.
[1029][Offline program full. Maximum program size is 13,000 commands. Saving stopped.]	Memory full.

COMMAND ERROR MESSAGES	
Message	**Explanation**
[1030][Already saving.]	The robot is already saving a program. Wait until finished to save another program.
[1031][Program saving aborted after receiving illegal command. - Command: '...']	The command cannot be executed because the robot is currently saving a program.
[1032][Homing failed because joints are outside limits.] DEPRECATED, USED ONLY FOR RELEASE 8	Homing cannot be done, because the current joint set is outside the user-defined joint limits.
[1033][Start conf mismatch]	Requested move blocked because start robot position is not in the requested configuration.
[1038][No gripper connected.]	No gripper was detected.
[1040][Command failed.]	General error for various commands.
[1041][No Vbox]	No pneumatic model connected.
[1042[Ext tool sim must deactivated]	Switching external tool type is only possible when the robot is deactivated.

Table 1: Command error messages

2.5.2 Command responses

Motion commands do not generate any (non-error) response, other than the optional EOB and EOM messages (see Section for details) and the message eventually generated by the *SetCheckpoint* command. Table 3 presents a summary of all request commands and the possible non-error responses for each of them.

COMMAND RESPONSES	
Response code	**Command**
[2000][Motors activated.] [2001][Motors already activated.]	*ActivateRobot*
[2002][Homing done.] [2003][Homing already done.]	*Home*
[2004][Motors deactivated.]	*DeactivateRobot*
[2005][The error was reset.] [2006][There was no error to reset.]	*ResetError*
[2007][*as*, *hs*, *sm*, *es*, *pm*, *eob*, *eom*]	*GetStatusRobot*
[2008][All brakes released.]	*BrakesOff*
[2010][All brakes set.]	*BrakesOn*
[2013][*x*, *y*, *z*, *α*, *β*, *γ*]	*GetWrf*
[2014][*x*, *y*, *z*, *α*, *β*, *γ*]	*GetTrf*
[2026][θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]	*GetJoints*
[2027][*x*, *y*, *z*, *α*, *β*, *γ*]	*GetPose*
[2028][*e*]	*GetAutoConf*

COMMAND RESPONSES	
Response code	**Command**
[2029][c_s, c_e, c_w]	*GetConf*
[2031][*e*]	*GetAutoConfTurn*
[2036][c_t]	*GetConfTurn*
[2042][Motion paused.]	*PauseMotion*
[2043][Motion resumed.]	*ResumeMotion*
[2044][The motion was cleared.]	*ClearMotion*
[2045][The simulation mode is enabled.]	*ActivateSim*
[2046][The simulation mode is disabled.]	*DeactivateSim*
[2047][External tool simulation mode has changed.]	*SetExtToolSim*
[2051][Requested velocity/acceleration is higher than recovery mode's limits. Requested value will be applied automatically once the recovery mode is disabled.]	*MoveJointsVel*, *MoveLinVelTrf*, *MoveLinVelWrf*, *SetCartAcc*, *SetCartAngVel*, *SetCartAcc*, *SetJointAcc*, *SetJointVel*
[2052][End of movement is enabled.] [2053][End of movement is disabled.]	*SetEom*
[2054][End of block is enabled.] [2055][End of block is disabled.]	*SetEob*
[2060][Start saving program.]	*StartSaving*
[2061][*n* commands saved.]	*StopSaving*
[2063][Offline program *n* started.] [3004][End of movement.]	*StartProgram*
[2064][Offline program looping is enabled.] [2065][Offline program looping is disabled.]	*StopSaving*
[2079][*ge*, *hs*, *hp*, *lr*, *es*, *oh*]	*GetStatusGripper*
[2080][*n*]	*GetCmdPendingCount*
[2081][vx.x.x]	*GetFwVersion*
[2083][robot's serial number]	*GetRobotSerial*
[2084][Meca500]	*GetProductType*
[2090][n, $\theta_{n,min}$, $\theta_{n,max}$]	*GetJointLimits*
[2092][*n*]	*SetJointLimits*
[2093][User-defined joint limits enabled.] [2093][User-defined joint limits disabled.]	*SetJointLimitsCfg*
[2094][*e*]	*GetJointLimitsCfg*
[2095][*s*]	*GetRobotName*
[2096][Monitoring on control port enabled/disabled]	*SetCtrlPortMonitoring*
[2097][*n*]	*SyncCmdQueue*
[2116][*t*]	*GetMonitoringInterval*

COMMAND RESPONSES	
Response code	**Command**
[2117][n_1, n_2, ...]	*GetRealTimeMonitoring*, *SetRealTimeMonitoring*
[2119][n_1, n_2, n_3, n_4, n_5, n_6]	*GetNetworkOptions*
[2140][t]	*GetRtc*
[2150][p]	*GetBlending*
[2151][t]	*GetVelTimeout*
[2152][p]	*GetJointVel*
[2153][p]	*GetJointAcc*
[2154][v]	*GetCartLinVel*
[2155][ω]	*GetCartAngVel*
[2156][n]	*GetCartAcc*
[2157][n]	*GetCheckpoint*
[2159][p]	*GetGripperVel*
[2160][s, m]	*GetTorqueLimitsCfg*
[2161][p_1, p_2, p_3, p_4, p_5, p_6]	*GetTorqueLimits*
[2162][d_{closed}, d_{open}]	*GetGripperForce*
[2200][t, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]	*GetRtTargetJointPos*
[2201][t, x, y, z, α, β, γ]	*GetRtTargetCartPos*
[2202][t, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$]	*GetRtTargetJointVel*
[2203][t, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6]	*GetRtTargetJointTorq*
[2204][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z]	*GetRtTargetCartVel*
[2208][t, c_s, c_e, c_w]	*GetRtTargetConf*
[2209][t, c_t]	*GetRtTargetConfTurn*
[2210][t, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]	*GetRtJointPos*
[2211][t, x, y, z, α, β, γ]	*GetRtCartPos*
[2212][t, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$]	*GetRtJointVel*
[2213][t, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6]	*GetRtJointTorq*
[2214][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z]	*GetRtCartVel*
[2218][t, c_s, c_e, c_w]	*GetRtConf*
[2219][t, c_t]	*GetRtConfTurn*
[2220][t, n, a_x, a_y, a_z]	*GetRtAccelerometer*
[2228][t, x, y, z, α, β, γ]	*GetRtWrf*
[2229][t, x, y, z, α, β, γ]	*GetRtTrf*
[2300][t, *simType*, *phyType*, *hs*, *es*, *oh*]	*GetRtExtToolStatus*

COMMAND RESPONSES	
Response code	**Command**
[2310][*t*, v_1, v_1]	`GetRtValveState`
[2320][*t*, *hp*, *dr*, *gc*, *go*]	`GetRtGripperState`
[2321][*t*, *p*]	`GetRtGripperForce`
[2322][*t*, *p*]	`GetRtGripperPos`
[2323][*t*, *p*]	`GetRtGripperVel`
[3004][End of movement.]	`PauseMotion`
[3012][End of block.]	`StartProgram`
[3032][*e*]	`ResetPStop`
[3035][TCP dump capture started for *n* seconds.]	`TcpDump`
[3036][TCP dump capture stopped.]	`TcpDumpStop`

Table 2: List of request commands and corresponding possible responses

2.5.3 Status messages

Status messages, general or error, occur without any specific action from the network client. Table 3 lists all possible status messages.

STATUS MESSAGES	
Message	**Explanation**
[3000][Connected to Meca500 x_x_x.x.x.]	Confirms connection to robot.
[3001][Another user is already connected, closing connection.]	Another user is already connected to the Meca500. The robot disconnects from the user immediately after sending this message.
[3002][A firmware upgrade is in progress (connection refused).]	The firmware of the robot is being updated.
[3003][Command has reached the maximum length.]	Too many characters before the NULL character. Possibly caused by a missing NULL character
[3004][End of movement.]	The robot has stopped moving.
[3005][Error of motion.]	Motion error. Possibly caused by a collision or overload. Correct the situation and send the `ResetError` command. If the motion error persists, try power-cycling the robot.
[3006][Error of communication with drives]	This error cannot be reset. The robot needs to be rebooted to recover from this error.
[3009][Robot initialization failed due to an internal error. Restart the robot.]	Error in robot startup procedure. Contact 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.

STATUS MESSAGES	
Message	**Explanation**
[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, overheating likely occurred. Let the gripper cool down for a few minutes and send the `ResetError` command. The gripper will stop applying a force; if it was holding a part, the part might fall.
[3026][Robot's maintenance check has discovered a problem. Mecademic cannot guarantee correct movements. Please contact Mecademic.]	A hardware problem was detected. Contact Mecademic.
[3028][*s*]	A torque limit was exceeded.
[3030][*n*]	Checkpoint *n* was reached.
[3031][A previously received text API command was incorrect.]	When using EtherNet/IP, this code (received in the input tag assembly only) indicates that the last command sent by TCP/IP was invalid.
[3032][1/0]	A protective stop was requested (1) or cleared (0).
[3035][TCP dump capture started for *x* seconds]	Sent to indicate that the requested TCP dump capture has started and confirms the maximum duration of *x* seconds.
[3036][TCP dump capture stopped]	Sent after a previously started TCP dump capture has finished.
[3037][Pneumatic module error]	A communication error with the pneumatic module was detected. Contact Mecademic.

Table 3: Status messages and descriptions

2.5.4 Monitoring port messages

The Meca500 is configured to send immediate robot feedback over TCP port 10001. Several kinds of feedback messages are sent over this port, some of which are optional (see `SetRealTimeMonitoring`):

MONITORING PORT MESSAGES	
Message	**Description**
[2007][*as*, *hs*, *sm*, *es*, *pm*, *eob*, *eom*]	Response from the `GetStatusRobot` command (only when the data changes and at connection, but can be sent several times during a monitoring interval)
[2026][θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]	Joint set
[2027][*x*, *y*, *z*, α, β, γ]	Pose of the TRF with respect to the WRF
[2049][Recovery mode enabled]	The recovery mode becomes enabled.
[2050][Recovery mode disabled]	The recovery mode becomes disabled.

MONITORING PORT MESSAGES	
Message	**Description**
[2079][*ge*, *hs*, *hp*, *lr*, *es*, *oh*]	Response from the legacy *GetStatusGripper* command (sent only when a gripper is installed and its status changes and at connection, but can be sent several times during a monitoring interval)
[2200][t, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]	*GetRtTargetJointPos*
[2201][t, x, y, z, α, β, γ]	*GetRtTargetCartPos*
[2202][t, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$]	*GetRtTargetJointVel*
[2203][t, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6]	*GetRtTargetJointTorq*
[2204][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z]	*GetRtTargetCartVel*
[2208][t, c_s, c_e, c_w]	Response from the *GetRtTargetConf* command (only when the data changes and at connection)
[2209][t, c_t]	Response from the *GetRtTargetConfTurn* command (only when the data changes and at connection)
[2210][t, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]	*GetRtJointPos*
[2211][t, x, y, z, α, β, γ]	*GetRtCartPos*
[2212][t, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$]	*GetRtJointVel*
[2213][t, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6]	*GetRtJointTorq*
[2214][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z]	*GetRtCartVel*
[2218][t, c_s, c_e, c_w]	*GetRtConf*
[2219][t, c_t]	*GetRtConfTurn*
[2220][t, n, a_x, a_y, a_z]	*GetRtAccelerometer*
[2228][t, x, y, z, α, β, γ]	Response from the *GetRtWrf* command (only when the data changes and at connection)
[2229][t, x, y, z, α, β, γ]	Response from the *GetRtTrf* command (only when the data changes and at connection).
[2300][t, *simType*, *phyType*, *hs*, *es*, *oh*]	Response from the *GetRtExtToolStatus* command (only when the data changes and at connection).
[2310][t, v_1, v_1]	Response from the command *GetRtValveState* (only when the data changes and at connection).
[2320][t, *hp*, *dr*, *gc*, *go*]	Response from the command *GetRtGripperState* (only when the data changes and at connection).
[2321][t, p]	*GetRtGripperForce*
[2322][t, p]	*GetRtGripperPos*
[2323][t, p]	*GetRtGripperVel*
[2230][t]	End-of-cycle event. Although the messages listed above are not sent in the order shown, this message is always last.

Table 4: Monitoring port messages

By default, these feedback messages are sent every 15 ms. The time interval between subsequent feedback messages can be configured using the *SetMonitoringInterval* command. Note that multiple

ASCII messages are separated by a single null-character and that there are no blank spaces in any of these messages.

Optional messages enabled using *SetRealTimeMonitoring*(2200,2201), are redundant; they provide the same data as messages 2026 and 2027 (legacy messages). Message 2079 provides the same data as messages 2320 and 2300.

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

[2026][-102.6011,-0.0000,-78.9239,-0.0000,15.7848,110.3150]
[2027][-3.7936,-16.9703,457.5125,26.3019,-5.6569,9.0367]
[2208][58675156984,-1,-1,1]
[2209][58675156984,0]
[2230][58675156984]

3. COMMUNICATING OVER CYCLIC PROTOCOLS

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

3.1. Cyclic data

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

3.2. Types of robot commands

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

3.2.1 Status change commands

Some cyclic data fields (bits) directly control robot status:

- PauseMotion
- ClearMotion
- SimMode
- RecoveryMode
- BrakesControl

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

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

3.2.2 Triggered actions

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

- Activate
- Deactivate
- Home
- ResetError
- ResetPStop

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

3.2.3 Motion commands

Most commands related to robot movement are posted to the robot motion queue. The robot will execute these commands sequentially (see Section 3.2.3).

There are two types of motion commands: cyclic (velocity-mode move commands) and non-cyclic (all other move commands):

- velocity-mode commands (e.g., *MoveJointsVel*) are canceled as soon as any subsequent command is received (or after velocity timeout);
- other commands (e.g., *MoveJoints*) are executed completely before the subsequent command starts being executed.

3.3. Sending motion commands

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

3.3.1 Command ID

We have assigned a unique number to each of the available motion commands (see Table 8). By entering this number in the MotionCommandID field, you are specifying the motion command that is to be sent to the robot.

3.3.2 MoveID and SetPoint

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

The SetPoint is a bit that enables or disables the robot's reception of motion commands from the cyclic data. When this bit is cleared, the robot ignores the MotionCommandID and the MoveID fields.

The MoveID field determines if commands are cyclic (MoveID is 0) or non-cyclic (MoveID is not 0, one new command being queued every time the MoveID value is changed).

Always wait for the robot to acknowledge the current MoveID before changing the cyclic data (MoveID, MotionCommandID or the motion command arguments). Otherwise, a motion command may be lost.

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

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

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

When MoveID is changed, the motion command defined in the MotionCommandID field will be added to the motion queue. The robot then acknowledges by updating its own MoveID field to match your MoveID value.

The following sequence must be followed:

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

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

3.3.4 Sending cyclic motion commands (velocity mode)

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

The following sequence must be followed:

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

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

3.4. Cyclic data that can be sent to the robot

The protocols' cyclic data contains the following fields for data that can be sent to the robot, allowing to perform the commands and actions described above.

See Sections 4–6 for detailed protocol-specific information about each field (like bit-offset, or protocol-specific identifier). Below is the detailed description of each field that applies to the cyclic protocols.

3.4.1 Robot control

Table 5 lists the fields that control the status of the robot.

ROBOT STATUS CONTROL FIELDS		
Field	**Type**	**Description**
Deactivate	Bool (action)	Deactivates the robot when set to 1.
Activate	Bool (action)	Activates the robot when set to 1 (only if Deactivate bit is 0).
Home	Bool (action)	Homes the robot when set to 1 (if the robot is activated but not homed).
ResetError	Bool (action)	Resets the error when set to 1.
SimMode	Bool (state)	Enables (when set to 1) or disables (when reset to 0) the simulation mode (only applied when the robot is deactivated).
RecoveryMode	Bool (state)	Enables (when set to 1) or disables (when reset to 0) the recovery mode.

Table 5: Robot control fields

3.4.2 Motion control

Table 6 lists the fields that control the motion of the robot.

MOTION CONTROL FIELDS		
Field	**Type**	**Description**
MoveID	Integer	A user-defined number, the change of which triggers the addition of the command specified in MotionCommandID to the motion queue.
SetPoint	Bool (state)	Has to be set to 1 for motion commands to be sent to the robot.
PauseMotion	Bool (state)	Puts the robot in pause without clearing the commands in the queue. Motion is resumed once both the Pause-Motion and ClearMotion bits are reset to 0.
ClearMotion	Bool (action/ state)	Clears the motion queue and puts the robot in pause. Motion is resumed once both the PauseMotion and the ClearMotion bits are reset to 0.
ResetPStop	Bool (state)	Resets P-Stop 2.

Table 6: Motion control fields

3.4.3 Motion Parameters

The motion parameters include the MotionCommandID and six corresponding arguments. These are illustrated in Table 11. The list of available MotionCommandID values is given in Table 8 along with arguments usage in each case.

MOTION PARAMETERS		
Field	**Type**	**Description**
MotionCommandID	Integer	MotionCommandID (see Table 8).
Motion command argument 1	Real	First argument of the motion command, if applicable, as described in Section 2.
Motion command argument 2	Real	Second argument of the motion command, if applicable, as described in Section 2.

MOTION PARAMETERS		
Field	**Type**	**Description**
Motion command argument 3	Real	Third argument of the motion command, if applicable, as described in Section 2.
Motion command argument 4	Real	Fourth argument of the motion command, if applicable, as described in Section 2.
Motion command argument 5	Real	Fifth argument of the motion command, if applicable, as described in Section 2.
Motion command argument 6	Real	Sixth argument of the motion command, if applicable, as described in Section 2.

Table 7: Motion parameters

MOTION COMMAND ID NUMBERS	
ID	**Description**
0	No movement: all six arguments are ignored.
1	*MoveJoints*, all six arguments are in degrees.
2	*MovePose*, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
3	*MoveLin*, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
4	*MoveLinRelTrf*, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
5	*MoveLinRelWrf*, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
6	*Delay*, argument 1 is the pause in seconds.
7	*SetBlending*, argument 1 is the percentage of blending, from 0 or 100.
8	*SetJointVel*, argument 1 is the percentage of maximum joint velocities, from 0.001 to 100.
9	*SetJointAcc*, argument 1 is the percentage of maximum joint accelerations, from 0.001 to 150.
10	*SetCartAngVel*, argument 1 is the Cartesian angular velocity limit, from 0.001 to 300, measured in °/s.
11	*SetCartLinVel*, argument 1 is the linear velocity limit for the TCP, from 0.001 to 1,000, measured in mm/s.
12	*SetCartAcc*, argument 1 is the percentage of maximum Cartesian accelerations, ranging from 0.001 to 100.
13	*SetTrf*, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
14	*SetWrf*, arguments 1, 2, 3 are in mm and 4, 5, 6 are in degrees.
15	*SetConf*, arguments 1, 2, and 3 are −1 or 1.
16	*SetAutoConf*, argument 1 is 0 or 1.
17	*SetCheckpoint*, argument 1 in an integer number, ranging from 1 to 8,000.
18	Gripper, argument 1 is 0 for *GripperClose*, and 1 for *GripperOpen*.
19	*SetGripperVel*, argument 1 is the percentage of maximum finger velocity (100 mm/s), from 5 to 100.
20	*SetGripperForce*, argument 1 is the percentage of maximum grip force (40 N), from 5 to 100.
21	*MoveJointsVel*, all six arguments are in °/s.
22	MoveLinVelWrf, arguments 1, 2, 3 are in mm/s and 4, 5, 6 are in °/s.

MOTION COMMAND ID NUMBERS	
ID	**Description**
23	MoveLinVelTrf, arguments 1, 2, 3 are in mm/s and 4, 5, 6 are in °/s.
24	*SetVelTimeout*, argument 1 is in seconds.
25	SetConfTurn, argument 1 is an integer number, ranging from −100 to 100.
26	SetAutoConfTurn, argument 1 is 0 or 1.
27	*SetTorqueLimits*, all six arguments are percentage of maximum joint torque, from 0.001 to 100.
28	*SetTorqueLimitsCfg*, argument 1 is severity (0 to 4), argument 2 is detection mode (0 or 1). See Section 2.1.28.
29	*MoveJointsRel*, all six arguments are in degrees.
30	*SetValveState*, both arguments are 0 or 1.
31	*SetGripperRange*, both arguments are in mm.
32	*MoveGripper*, argument 1 is in mm.
100	*StartProgram*, argument 1 is the ID of the offline program to start, from 1 to 500.

Table 8: List of MotionCommandID numbers

3.4.4 Host time

Table 9 lists the fields that allow the host to set robot's date/time.

HOST TIME FIELDS		
Field	**Type**	**Description**
HostTime	Integer	Current time in seconds since epoch (i.e., since 00:00:00 UTC January 1, 1970). If non-zero, the robot will update its own time to this value (same as *SetRtc*). This is useful for robot logs to contain meaningful time (the robot forgets time every time it's rebooted).

Table 9: Host time fields

3.4.5 Brake control

Table 10 lists the fields that allow to control robot brakes (when deactivated) for joints 1, 2 and 3 (all three at the same time).

Brakes behavior:

- Brakes are automatically disengaged when robot is activated (the robot will actively maintain its position when not moving).
- Brakes are automatically engaged when robot is powered-down (or P-Stop 1).
- Brakes are automatically engaged when robot gets deactivated.
- While robot is in 'deactivated' state, brakes state can be controlled using the fields below.

Disable brakes with caution; without brakes, all joints collapse downward.

BRAKES CONTROL FIELDS		
Field	**Type**	**Description**
EnableBrakesControl	Bool	Must be set to 1 to allow brakes control through cyclic data. The purpose of this bit is to ensure that the brakes don't get inadvertently disabled if cyclic data sent to the robot contains all zeroes.
EngageBrakes	Bool	If set to 1, the brakes are engaged, else the brakes are disengaged and the robot might fall down under the effects of gravity. This bit is ignored if EnableBrakesControl bit is cleared. This bit is ignored if the robot is activated.

Table 10: Brakes control fields

3.4.6 Dynamic data configuration

Table 11 lists the fields that allow choosing which dynamic data the robot will return. These values may be set to automatic, to a fixed value, or changed every cycle, as required by the application. See Table 12 for a list of available dynamic data types. Finally, note that there may be a delay of 1 or two cycles before the change takes effect.

DYNAMIC DATA CONFIGURATION ID		
Field	**Type**	**Description**
DynamicDataTypeID 1	Integer	Dynamic data type for index #1 (see Table 12).
DynamicDataTypeID 2	Integer	Dynamic data type for index #2 (see Table 12).
DynamicDataTypeID 3	Integer	Dynamic data type for index #3 (see Table 12).
DynamicDataTypeID 4	Integer	Dynamic data type for index #4 (see Table 12).

Table 11: Dynamic data configuration fields

DYNAMIC DATA TYPE ID	
ID	**Description**
0	Automatic. Robot will automatically choose dynamic data type and change it every cycle to go through them all. This is the easiest way for the host to receive all possible values periodically (round-robin manner).
1	Firmware version. Values: [major version, minor version, patch version, build number]. Same as *GetFwVersion*.
2	Product type. Values: [product type (3=Meca500)]. Same as *GetProductType*.
3	Serial number. Values: [serial number]. Same as *GetRobotSerial*.
4	Joints offset (as calculated by mastering at production). Values: [$\delta\theta_1$, $\delta\theta_2$, $\delta\theta_3$, $\delta\theta_4$, $\delta\theta_5$, $\delta\theta_6$]. In degrees.
5–10	Reserved
11	Joint limits enabled state. Values: [enabled 1/0]. Same as *GetJointLimitsCfg*.
12	Robot model's nominal joint limits for joints 1, 2 and 3. Values: [$\theta_{1,min}$, $\theta_{1,max}$, $\theta_{2,min}$, $\theta_{2,max}$, $\theta_{3,min}$, $\theta_{3,max}$]. Unit is degrees. Same as *GetModelJointLimits*.
13	Robot model's nominal joint limits for joints 4, 5 and 6. Values: [$\theta_{4,min}$, $\theta_{4,max}$, $\theta_{5,min}$, $\theta_{5,max}$, $\theta_{6,min}$, $\theta_{6,max}$]. Unit is degrees. Same as *GetModelJointLimits*.

DYNAMIC DATA TYPE ID	
ID	**Description**
14	Effective joint limits for joints 1, 2 and 3. Values: [$\theta_{1,min}$, $\theta_{1,max}$, $\theta_{2,min}$, $\theta_{2,max}$, $\theta_{3,min}$, $\theta_{3,max}$]. Unit is degrees. Same as *GetJointLimits*.
15	Effective joint limits for joints 4, 5 and 6. Values: [$\theta_{4,min}$, $\theta_{4,max}$, $\theta_{5,min}$, $\theta_{5,max}$, $\theta_{6,min}$, $\theta_{6,max}$]. Unit is degrees. Same as *GetJointLimits*.
20	Motion queue's conf that will be applied to next MovePose. Values: [shoulder −1/1/NaN, elbow −1/1/NaN, wrist −1/1/NaN, last joint turn or NaN]. Value NaN is used to indicate auto-conf or auto-conf-turn. Same as *GetConf* and *GetConfTurn*.
21	Motion queue parameters. Values: [blending ratio percent, velocity timeout in seconds]. Same as *GetBlending* and *GetVelTimeout*.
22	Motion queue velocities and accelerations in percent. Values: [joint velocity, joint acceleration, Cartesian linear velocity, Cartesian angular velocity, Cartesian acceleration]. Unit is percent. Same as *GetJointVel*, *GetJointAcc*, *GetCartLinVel*, *GetCartAngVel*, and *GetCartAcc*.
23	Gripper parameters. Values: [gripper force, gripper velocity, fingers opening corresponding to closed state, fingers opening corresponding to open state]. Arguments 1 and 2 are in percentage, while arguments 3 and 4 are in mm. Same as *GetGripperForce*, *GetGripperVel*, and *GetGripperRange*.
24	Torque limits configuration. Values: [severity, detection mode]. See Section 2.1.28 for corresponding severity/mode values. Same as *GetTorqueLimitsCfg*.
25	Torque limits. Values: [joint 1 limit, joint 2 limit, ..., joint 6 limit]. Unit is percent. Same as *GetTorqueLimits*.
32	Target real-time joint velocity. Values: [$\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$]. Unit is °/s. Same as *GetRtTargetJointVel*.
33	Target real-time joint torque (not implemented yet). Values: [joint 1 torque, joint 2 torque, ..., joint 6 torque]. Unit is percent. Same as *GetRtTargetJointTorq*.
34	Target real-time Cartesian velocity (TRF with respect to. WRF). Values: [$\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z]. Units are mm/s or °/s. Same as *GetRtTargetCartVel*.
40	Actual real-time joint position based on hardware encoders. Values: [θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]. Unit is degrees. Same as *GetRtJointPos*.
41	Actual real-time end-effector pose (TRF with respect to. WRF). Values: [x, y, z, α, β, γ]. Units are mm or degrees. Same as *GetRtCartPos*.
42	Actual real-time joint velocity. Values: [$\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$]. Unit is °/s. Same as *GetRtJointVel*.
43	Actual real-time joint torque. Values: [joint 1 torque, joint 2 torque, ..., joint 6 torque]. Unit is percent. Same as *GetRtJointTorq*.
44	Actual real-time cartesian velocity (TRF with respect to. WRF). Values: [$\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z]. Units are mm/s or °/s. Same as *GetRtCartVel*.
45	Actual conf that corresponds to real-time end-effector pose. Values: [shoulder −1/0/1, elbow −1/0/1, wrist −1/0/1, last joint turn]. Same as *GetRtConf* and *GetRtConfTurn*.
46	Accelerometer reading. Values: [a_x, a_y, a_z]. Unit is 1/16,000 of G. Same as *GetRtAccelerometer*.
52	External tool status. Values: [type, homing done, error state, overheated]. Same as *GetRtExtToolStatus*.
53	EOAT status. Values if type gripper: [holding part, desired fingers opening reached, gripper closed, gripper open, gripper force, fingers opening]. Values if pneumatic module: [valve 1 state, valve 2 state].

Table 12: List of DynamicDataTypeID values with associated values

To avoid data duplication, the dynamic data (above) do not include data that is already provided in the explicit tables mentioned next (i.e., target joint position, target end-effector pose with corresponding configuration, WRF and TRF).

3.5. Cyclic data received from the robot

Every cycle, the robot reports:

- *RobotStatus*, as described in Table 13.
- *MotionStatus*, as described in Table 14.
- *TargetJointSet*. Values: [θ_1, θ_2, θ_3, θ_4, θ_5, θ_6]. Unit is in degrees. Same as *GetRtTargetJointPos*.
- Corresponding target pose (TRF with respect to WRF) and associated information:
  - *TargetEndEffectorPose*. Values: [x, y, z, α, β, γ]. Units are in mm or degrees. Same as *GetRtTargetCartPos*.
  - TargetConfiguration. Values: [shoulder −1/1, elbow −1/1, wrist −1/1, last joint turn]. Same as the combination of *GetRtTargetConf* and *GetRtTargetConfTurn*.
  - WRF (with respect to BRF) Values: [x, y, z, α, β, γ]. Units are in mm or degrees. Same as *GetRtWrf*.
  - TRF (with respect to BRF) Values: [x, y, z, α, β, γ]. Units are in mm or degrees. Same as *GetRtTrf*.
- Robot timestamp, as described in Table 15.
- DynamicData #1, #2, #3, #4. See Table 16.

ROBOT STATUS		
Field	**Type**	**Description**
ErrorCode	Integer	Indicates the error code (see Tables 1 and 3) or 0, if there is no error.
Busy	Bool	True only while the robot is being activated, homed or deactivated.
Activated	Bool	Indicates whether the motors are on (powered).
Home	Bool	Indicates whether the robot is homed and ready to receive motion commands.
SimActivated	Bool	Indicates whether the robot simulation mode is activated.
BrakesEngaged	Bool	Indicates whether the brakes are engaged.
RecoveryMode	Bool	Indicates whether the robot recovery mode is activated.

Table 13: Robot status

MOTION STATUS		
Field	**Type**	**Description**
Checkpoint	Integer	Indicates the last checkpoint number reached (the value stays unchanged until another checkpoint number is reached). See Section 2.1.19 for a detailed description of checkpoints.
MoveID	Integer	Acknowledges the MoveID of the last motion command queued for execution.
FIFOSpace	Integer	The number of commands that can be added to the robot's motion queue at any given time (the maximum is 13,000). If 0 (too many commands sent), subsequent commands will be ignored.
Paused	Bool	Indicates whether the motion is paused. This bit will remain set (and robot will remain paused) as long as motion control bits Pause or ClearMotion remain set. Motion will resume once both Pause and ClearMotion bits become 0.

MOTION STATUS		
Field	**Type**	**Description**
EOB	Bool	The End of Block (EOB) bit is true only when the robot is not moving and there is no motion command left in the motion queue. Note that the EOB bit may be raised before all sent commands have been completed, due to network or processing delays. Therefore, do not rely on this flag to be informed when a sequence of movements has been completed (use a checkpoint instead).
EOM	Bool	The End Of Motion (OOM) bit is true if the robot is not moving. Note that the EOM bit may be raised between two consecutive motion commands. Therefore, do not rely on this flag to be informed when a sequence of movements has been completed (use a checkpoint instead).
Cleared	Bool	Indicates whether the motion queue is cleared. If the queue is cleared, the robot is not moving. This bit will remain true (and robot will remain paused) as long as the motion control bit ClearMotion remains set. Motion will resume once both Pause and ClearMotion bits become 0.
PStop	Bool	Indicates whether P-Stop 2 is set.
ExcessiveTorque	Bool	Indicates whether a joint torque is exceeding the corresponding user-defined torque limit.
OfflineProgramID	Integer	ID of the offline program currently running (0 if none).

Table 14: Motion status

ROBOT TIMESTAMP		
Field	**Type**	**Description**
RobotTimestamp (seconds part)	Integer	Robot's monotonic timestamp (seconds part) based on arbitrary reference.
RobotTimestamp (microseconds part)	Integer	Robot's monotonic timestamp (microseconds within current second).
DynamicDataUpdateCount	Integer	Number of times all available dynamic data has been refreshed (see value 'Automatic' in Table 11).

Table 15: Robot timestamp

ROBOT DYNAMIC DATA		
Field	**Type**	**Description**
DynamicDataTypeID	Integer	Dynamic data type (among values in Table 11).
Value 1	Real	First associated value (see corresponding DynamicDataTypeID in Table 12).
Value 2	Real	Second associated value (see corresponding DynamicDataTypeID in Table 12).
Value 3	Real	Third associated value (see corresponding DynamicDataTypeID in Table 12).
Value 4	Real	Fourth associated value (see corresponding DynamicDataTypeID in Table 12).
Value 5	Real	Fifth associated value (see corresponding DynamicDataTypeID in Table 12).
Value 6	Real	Sixth associated value (see corresponding DynamicDataTypeID in Table 12).

Table 16: Robot dynamic data

4. ETHERCAT COMMUNICATION

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

4.1. Overview

4.1.1 Connection types

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

4.1.2 ESI file

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

4.1.3 Enabling EtherCAT

The default communication protocol of the robot is the Ethernet TCP/IP protocol. The latter is the protocol needed for jogging the robot through its web interface. To switch to the EtherCAT communication protocol, you must send the *SwitchToEtherCAT* command via the TCP/IP protocol from an external client (e.g., from a PC using a Web browser).

As soon as the robot receives this command, the Ethernet TCP/IP connection LED (i.e., #1 or #2 in Figure 14) will go off, then turn back on. This means that the robot is now in EtherCAT mode and can be connected to an EtherCAT master. Note, however, than until EtherCAT is disabled, TCP/IP or EtherNet/IP communication is not possible (e.g., you cannot use the robot's web interface). To disable EtherCAT, use the Communication mode SDO (Section 4.2.16) or perform a network settings reset (by keeping the power button on the robot's base pressed for a few seconds during restart).

4.1.4 LEDs

When EtherCAT communication is enabled, the three LEDs on the outer edge of the robot's base (Figure 14) communicate the state of the EtherCAT connection, as summarized in Table 17.

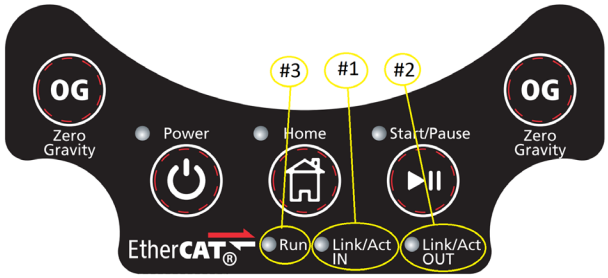


Figure 14: EtherCAT LEDs

ETHERCAT LED DESCRIPTION			
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

Table 17: EtherCAT LED description

4.2. Object dictionary

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

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

4.2.1 Robot control

This object controls the robot's initialization and simulation. Table 18 describes the object's indices. See Table 5 for detailed explanations.

ROBOT CONTROL OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
7200h		Record		RobotControl				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	ResetError	0	0	1	RW	1600h:4
	5	Variable	BOOL	SimMode	0	0	1	RW	1600h:5
	6	Variable	BOOL	RecoveryMode	0	0	1	RW	1600h:6

Table 18: Robot control object

4.2.2 Motion control

This object controls the actual robot movement. Table 19 describes the object's indices. See Table 5 for detailed explanations.

MOTION CONTROL OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
7310h		Record		MotionControl				RO	none
	1	Variable	UINT	MoveID	0	0	65,535	RW	1601h:1
	2	Variable	BOOL	SetPoint	0	0	1	RW	1601h:2
	3	Variable	BOOL	PauseMotion	0	0	1	RW	1601h:3
	4	Variable	BOOL	ClearMotion	0	0	1	RW	1601h:4
	5	Variable	BOOL	ResetPStop	0	0	1	RW	1601h:5

Table 19: Motion control objects

4.2.3 Movement

The movement object is a pair of indices. The first index is the ID number indicating the motion command, while the second index has six subindices corresponding to the arguments of the motion command, as described in Table 20. See Section 3.3 and Section 3.4 for detailed explanations.

MOVEMENT OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
7305h		Variable	UDINT	MotionCommandID	0	0	100	RO	1602h.1
7306h		Array		Arguments				RO	none
	1	Variable	REAL	Motion command argument 1	†	†	†	RW	1602h.2
	2	Variable	REAL	Motion command argument 2	†	†	†	RW	1602h.3
	3	Variable	REAL	Motion command argument 3	†	†	†	RW	1602h.4
	4	Variable	REAL	Motion command argument 4	†	†	†	RW	1602h.5
	5	Variable	REAL	Motion command argument 5	†	†	†	RW	1602h.6
	6	Variable	REAL	Motion command argument 6	†	†	†	RW	1602h.7

† depending on the value of index 7305h (refer to Table 8).

Table 20: Movement objects

4.2.4 Host time

This object controls robot's date/time (real-time-clock). Figure 15 describes the object's indices. See Table 9 for detailed explanations.

HOST TIME OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
7400h		Record		HostTime				RO	none
	1	Variable	UDINT	Time since epoch in seconds	0	0	$2^{32} - 1$	RW	1610h:1

Figure 15: Host time object

4.2.5 Brake control

This object controls robot's brakes (applies only when robot is deactivated). Table 21 describes the object's indices. See Table 10 for detailed explanations about brakes behavior.

Note that robot brakes can also be controlled via the corresponding SDO when bit EnableBrakesControl below is 0 (see Table 39).

BRAKE CONTROL OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
7410h		Record		BrakesControl				RO	none
	1	Variable	BOOL	EnableBrakes-Control	0	0	1	RW	1611h:1
	2	Variable	BOOL	EngageBrakes	0	0	1	RW	1611h:2

Table 21: Brakes control object

4.2.6 Dynamic data configuration

This objects are used to choose which dynamic data type the robot will return. The following four tables describe the object's indices. See Table 12.

DYNAMIC DATA CONFIGURATION OBJECT 1 INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
7420h		Record		DynamicDataConfiguration 1				RO	none
	1	Variable	UDINT	DynamicData-TypeID	0	0	53	RW	1620h:1

Table 22: Dynamic data configuration object 1

DYNAMIC DATA CONFIGURATION OBJECT 2 INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
7421h		Record		DynamicDataConfiguration 2				RO	none
	1	Variable	UDINT	DynamicData-TypeID	0	0	53	RW	1621h:1

Table 23: Dynamic data configuration object 2

DYNAMIC DATA CONFIGURATION OBJECT 3 INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
7422h		Record		DynamicDataConfiguration 3				RO	none
	1	Variable	UDINT	DynamicData-TypeID	0	0	53	RW	1622h:1

Table 24: Dynamic data configuration object 3

DYNAMIC DATA CONFIGURATION OBJECT 4 INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
7423h		Record		DynamicDataConfiguration 4				RO	none
	1	Variable	UDINT	DynamicData-TypeID	0	0	53	RW	1623h:1

Table 25: Dynamic data configuration object 4

4.2.7 Robot status

The structure of the robot status object is described in Table 26. See Table 13 for detailed explanations.

ROBOT STATUS OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6010h		Record		RobotStatus				RO	none
	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	SimMode	n/a	0	1	RO	1A00h.5
	6	Variable	BOOL	BrakesEngaged	n/a	0	1	RO	1A00h.6
	7	Variable	BOOL	RecoveryMode	n/a	0	1	RO	1A00h.7
	1	Variable	UINT	ErrorCode	n/a	0	65,535	RO	1A00h.1

Table 26: Robot status object

4.2.8 Motion status

The structure of the motion status object is described in Table 27. See Table 14 for detailed explanations.

MOTION STATUS OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6015h		Record		MotionStatus				RO	none
	1	Variable	UDINT	Checkpoint	n/a	0	8,000	RO	1A01h.1
	2	Variable	UINT	MoveID	n/a	0	65,535	RO	1A01h.2
	3	Variable	UINT	FIFOSpace	n/a	0	13,000	RO	1A01h.3
	5	Variable	BOOL	Paused	n/a	0	1	RO	1A01h.5
	6	Variable	BOOL	EOB	n/a	0	1	RO	1A01h.6
	7	Variable	BOOL	EOM	n/a	0	1	RO	1A01h.7
	8	Variable	BOOL	Cleared	n/a	0	1	RO	1A01h.8
	9	Variable	BOOL	PStop	n/a	0	1	RO	1A01h.9
	10	Variable	BOOL	ExcessiveTorque	n/a	0	1	RO	1A01h.10
	10	Variable	UINT	(10 unused bits)	n/a	0	0	RO	n/a
	4	Variable	UINT	OfflineProgramID	n/a	0	500	RO	1A01h.4

Table 27: Motion status object

4.2.9 Target joint set

The structure of the real-time target joint set object is described below. The data is the same as that returned by TCP/IP command *`GetRtTargetJointPos`*.

TARGET JOINT SET OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6030h		Array		TargetJointSet				RO	none
	1		REAL	Target position of joint 1	n/a	−175	175	RO	1A02h.1
	2		REAL	Target position of joint 2	n/a	−70	90	RO	1A02h.2
	3		REAL	Target position of joint 3	n/a	−135	70	RO	1A02h.3
	4		REAL	Target position of joint 4	n/a	−170	170	RO	1A02h.4
	5		REAL	Target position of joint 5	n/a	−115	115	RO	1A02h.5
	6		REAL	Target position of joint 6	n/a	−36,000	36,000	RO	1A02h.6

Table 28: Target joint set object

4.2.10 Target end-effector pose

The structure of the real-time target end-effector pose object is described in Table 29. The data is the same as that returned by TCP/IP command *GetRtTargetCartPos*.

END-EFFECTOR POSE OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6031h		Array		TargetEndEffectorPose				RO	none
	1		REAL	Coordinate *x*	n/a	−3.4E38	3.4E38	RO	1A03h.1
	2		REAL	Coordinate *y*	n/a	−3.4E38	3.4E38	RO	1A03h.2
	3		REAL	Coordinate *z*	n/a	−3.4E38	3.4E38	RO	1A03h.3
	4		REAL	Euler angle *α*	n/a	−3.4E38	3.4E38	RO	1A03h.4
	5		REAL	Euler angle *β*	n/a	−3.4E38	3.4E38	RO	1A03h.5
	6		REAL	Euler angle *γ*	n/a	−3.4E38	3.4E38	RO	1A03h.6

Table 29: Target end-effector pose object

4.2.11 Target configuration

The structure of the real-time target configuration object is described in Table 30. The data is the same as that returned by the combination of the TCP/IP commands *GetRtTargetConf* and *GetRtTargetConfTurn*.

TARGET CONFIGURATION OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6046h		Array		TargetConfiguration				RO	none
	1		INT8	c_s (shoulder)	n/a	−1	1	RO	1A08h.1
	2		INT8	c_e (elbow)	n/a	−1	1	RO	1A08h.2
	3		INT8	c_w (wrist)	n/a	−1	1	RO	1A08h.3
	4		INT8	c_t (last joint turn)	n/a	−100	100	RO	1A08h.4

Table 30: Target configuration object

4.2.12 WRF

The structure of the real-time WRF object is described in Table 31. The data is the same as that returned by TCP/IP command *GetRtWrf*.

WRF OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6050h		Array		WRF				RO	none
	1		REAL	Coordinate *x*	n/a	−3.4E38	3.4E38	RO	1A09h.1
	2		REAL	Coordinate *y*	n/a	−3.4E38	3.4E38	RO	1A09h.2
	3		REAL	Coordinate *z*	n/a	−3.4E38	3.4E38	RO	1A09h.3
	4		REAL	Euler angle *α*	n/a	−3.4E38	3.4E38	RO	1A09h.4
	5		REAL	Euler angle *β*	n/a	−3.4E38	3.4E38	RO	1A09h.5
	6		REAL	Euler angle *γ*	n/a	−3.4E38	3.4E38	RO	1A09h.6

Table 31: WRF object

4.2.13 TRF

The structure of the real-time TRF object is described in Table 32. The data is the same as that returned by TCP/IP command *GetRtTrf*.

TRF OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6051h		Array		TRF				RO	none
	1		REAL	Coordinate *x*	n/a	−3.4E38	3.4E38	RO	1A0Ah.1
	2		REAL	Coordinate *y*	n/a	−3.4E38	3.4E38	RO	1A0Ah.2
	3		REAL	Coordinate *z*	n/a	−3.4E38	3.4E38	RO	1A0Ah.3
	4		REAL	Euler angle *α*	n/a	−3.4E38	3.4E38	RO	1A0Ah.4
	5		REAL	Euler angle *β*	n/a	−3.4E38	3.4E38	RO	1A0Ah.5
	6		REAL	Euler angle *γ*	n/a	−3.4E38	3.4E38	RO	1A0Ah.6

Table 32: TRF object

4.2.14 Robot timestamp

The structure of the Robot timestamp object is described in Table 33. See Table 15 for detailed explanations.

TIMESTAMP OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6060h		Array		RobotTimestamp				RO	none
	1		UDINT	RobotTimestamp (seconds part)	n/a	0	$2^{32} - 1$	RO	1A10h.1
	2		UDINT	RobotTimestamp (microseconds part)	n/a	0	232 − 1	RO	1A10h.2
	3		UDINT	DynamicDataUpdate	n/a	0	$2^{32} - 1$	RO	1A10h.3

Table 33: Robot timestamp object

4.2.15 Dynamic data

The structure of the dynamic data objects are described in the following four tables. See Table 11 for detailed explanations.

DYNAMIC DATA 1 OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6070h		Array		DynamicData				RO	none
	1		UDINT	DynamicDataType	n/a	0	53	RO	1A20h.1
	2		REAL	Value 1	n/a	†	†	RO	1A20h.2
	3		REAL	Value 2	n/a	†	†	RO	1A20h.3
	4		REAL	Value 3	n/a	†	†	RO	1A20h.4
	5		REAL	Value 4	n/a	†	†	RO	1A20h.5
	6		REAL	Value 5	n/a	†	†	RO	1A20h.6
	7		REAL	Value 6	n/a	†	†	RO	1A20h.7

† depending on the value of 1A20h.1 (refer to Table 11).

Table 34: Dynamic data object 1

DYNAMIC DATA 2 OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6071h		Array		DynamicData				RO	none
	1		UDINT	DynamicData-TypeID	n/a	0	53	RO	1A21h.1
	2		REAL	Value 1	n/a	†	†	RO	1A21h.2
	3		REAL	Value 2	n/a	†	†	RO	1A21h.3
	4		REAL	Value 3	n/a	†	†	RO	1A21h.4
	5		REAL	Value 4	n/a	†	†	RO	1A21h.5
	6		REAL	Value 5	n/a	†	†	RO	1A21h.6
	7		REAL	Value 6	n/a	†	†	RO	1A21h.7

† depending on the value of 1A21h.1 (refer to Table 11).

Table 35: Dynamic data object 2

DYNAMIC DATA 3 OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6072h		Array		DynamicData				RO	none
	1		UDINT	DynamicData-TypeID	n/a	0	53	RO	1A22h.1
	2		REAL	Value 1	n/a	†	†	RO	1A22h.2
	3		REAL	Value 2	n/a	†	†	RO	1A22h.3
	4		REAL	Value 3	n/a	†	†	RO	1A22h.4
	5		REAL	Value 4	n/a	†	†	RO	1A22h.5
	6		REAL	Value 5	n/a	†	†	RO	1A22h.6
	7		REAL	Value 6	n/a	†	†	RO	1A22h.7

† depending on the value of 1A22h.1 (refer to Table 11).

Table 36: Dynamic data object 3

DYNAMIC DATA 4 OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6073h		Array		DynamicData				RO	none
	1		UDINT	DynamicData-TypeID	n/a	0	53	RO	1A23h.1
	2		REAL	Value 1	n/a	†	†	RO	1A23h.2
	3		REAL	Value 2	n/a	†	†	RO	1A23h.3
	4		REAL	Value 3	n/a	†	†	RO	1A23h.4
	5		REAL	Value 4	n/a	†	†	RO	1A23h.5
	6		REAL	Value 5	n/a	†	†	RO	1A23h.6
	7		REAL	Value 6	n/a	†	†	RO	1A23h.7

† depending on the value of 1A23h.1 (refer to Table 11).

Table 37: Dynamic data object 4

4.2.16 Communication mode (SDO)

When EtherCAT is enabled, subindex 1 of this SDO is equal to 2 (see table below). Currently, you cannot change the communication mode for port ECAT OUT and therefore subindex 2 of this SDO is ignored (will always be the same as that of port ECAT IN). To switch both ports to TCP/IP, change the value of subindex 1 to 1.

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 (ignored)	1	1	2	RW	n/a

Table 38: Communication mode SDO

4.2.17 Brakes (SDO)

The Brakes SDO controls the brakes of joints 1, 2 and 3 (Table 39). See Table 10 for detailed explanations about brakes behavior. Note that robot brakes can also be controlled via the cyclic data (see Table 21), which has priority over this SDO when bit EnableBrakesControl is set (in which case the robot ignores this SDO).

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

Table 39: Brakes SDO

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 object dictionary. PDO assignment is summarized in the next two tables.

RECEIVING PDO			
PDO	**Object(s)**	**Name**	**Note**
1600h	7200h	RobotControl	Mandatory. See Table 18.
1601h	7310h	MotionControl	Mandatory. See Table 19
1602h	7305h, 7306h	Movement	Mandatory. See Table 20.
1610h	7400h	HostTime	Mandatory. See Figure 15.
1611h	7410h	BrakesControl	Mandatory. See Table 21.
1620h	7420h	DynamicDataConfiguration 1	Mandatory. See Table 22.
1621h	7421h	DynamicDataConfiguration 2	Mandatory. See Table 23.
1622h	7422h	DynamicDataConfiguration 3	Mandatory. See Table 24.
1623h	7423h	DynamicDataConfiguration 4	Mandatory. See Table 25.

Table 40: RxPDOs

TRANSMISSION PDO			
PDO	**Object**	**Name**	**Note**
1A00h	6010h	RobotStatus	Mandatory. See Table 26
1A01h	6015h	MotionStatus	Mandatory. See Table 27.
1A02h	6030h	TargetJointSet	Optional. See Table 28.
1A03h	6031h	TargetEndEffectorPose	Optional. See Table 29.
1A08h	6046h	TargetConfiguration	Optional. See Table 30.
1A09h	6050h	WRF	Optional. See Table 31.
1A0Ah	6051h	TRF	Optional. See Table 32.
1A10h	6060h	RobotTimestamp	Optional. See Table 33.
1A20h	6070h	DynamicData #1	Optional. See Table 34.
1A21h	6071h	DynamicData #2	Optional. See Table 35.
1A22h	6072h	DynamicData #3	Optional. See Table 36.
1A23h	6073h	DynamicData #4	Optional. See Table 37.

Table 41: TxPDOs

5. ETHERNET/IP COMMUNICATION

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

Refer to our Support Center for specific PLC examples.

5.1. Connection types

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

5.2. EDS file

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

5.3. Forward open exclusivity

The Meca500 robot will allow only one controlling connection at the time (either a TCP/IP connection or through an EtherNet/IP forward-open request).

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

5.4. Enabling Ethernet/IP

To enable the EtherNet/IP communication protocol, you must connect to the robot via the TCP/IP protocol first from an external client (e.g., from a PC using a Web browser), then send the *EnableEtherNetIP*(1) command. This is a persistent command, so it only needs to be set once. To disable EtherNet/IP, you need to send the *EnableEtherNetIP*(0) command.

Note that EtherNet/IP can be left permanently enabled as it does not prevent using the TCP/IP protocol, unlike EtherCAT and the *SwitchToEtherCAT* command.

5.5. Output tag assembly

The output tag assembly has an Instance of 150 with a size 60-byte array, as detailed below. Refer to Section 3 for a description of the different objects and their fields. The following subsections only define how these objects are mapped to EtherNet/IP output tag assembly (as also described in the EDS file).

OUTPUT TAG ASSEMBLY			
Bytes	**Data Type**	**Name**	**Description**
0–3	DWORD	RobotControl	See Table 43.
4–5	UINT	MoveID	See Table 44.
6–7	WORD	MotionControl	See Table 45.
8–11	UDINT	Movement	See Table 46.
12–15	REAL	Argument 1 for Movement	See Table 47.
16–19	REAL	Argument 2 for Movement	See Table 47.
20–23	REAL	Argument 3 for Movement	See Table 47.
24–27	REAL	Argument 4 for Movement	See Table 47.
28–31	REAL	Argument 5 for Movement	See Table 47.
32–35	REAL	Argument 6 for Movement	See Table 47.
36–39	DINT	HostTime	See Table 48.
40–43	DWORD	BrakesControl	See Table 49.
44–47	UDINT	DynamicDataConfiguration 1	See Table 50.
48–51	UDINT	DynamicDataConfiguration 2	See Table 50.
52–45	UDINT	DynamicDataConfiguration 3	See Table 50.
46–59	UDINT	DynamicDataConfiguration 4	See Table 50.

Table 42: Output tag assembly

5.5.1 Robot control tag

This tag controls the robot's initialization and simulation. Table 43 describes the tag's bits. See Table 5 for detailed explanations.

CONTROL TAGS								
Bytes	**Data Type**	**Bits 6–31**	**Bit 4**	**Bit 5**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
0–3	DWORD	Unused	Recovery-Mode	Sim-Mode	Reset-Error	Home	Activate	Deactivate

Table 43: Robot control tag

5.5.2 MoveID tag

This tag (Table 44) contains the distinct user-defined ID number associated with each motion command sent to the robot. See Table 6 for detailed explanations.

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

Table 44: MoveID tag

5.5.3 Motion control tag

This tag controls the actual robot movement. Table 45 describes the tag's bits. See Table 6 for details.

MOTION CONTROL TAGS						
Bytes	**Data Type**	**Bits 4–15**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
6–7	WORD	Unused	ResetPStop	ClearMotion	PauseMotion	SetPoint

Table 45: Motion control tag

5.5.4 Motion command group of tags

This group of tags will define the type of motion command that is being sent to the robot and the arguments of the respective command. The motion command tag (shown in the table below) contains the ID of the motion command (see Table 8). The motion command argument tags contain the arguments of the motion command (Table 47).

See Section 3.3 and Section 3.4 for detailed explanations.

MOTION COMMANDS			
Bytes	**Data Type**	**Name**	**Possible values**
8–11	UDINT	MotionCommandID	0, 1, 2, ... (see Table 7)

Table 46: Motion command tag

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

Table 47: Motion command arguments tags

5.5.5 Host time tag

This tag controls robot's date/time (real-time-clock). See Table 9 for details.

HOST TIME TAG				
Bytes	**Data Type**	**Name**	**Minimum**	**Maximum**
36–39	DINT	HostTime	0	$2^{32} - 1$

Table 48: Host time tag

5.5.6 Brake control tag

This tag controls robot brakes (applies only when robot is deactivated). This table describes the tag's bits. See Table 10 for detailed explanations about brakes behavior.

BRAKES CONTROL TAG				
Bytes	**Data Type**	**Bits 2–31**	**Bit 1**	**Bit 0**
40–43	DWORD	Unused	EngageBrakes	EnableBrakesControl

Table 49: Brakes control tag

5.5.7 Dynamic data configuration tag

This tag is used to choose which dynamic data type the robot will return (Table 50). See Table 11 for details.

DYNAMIC DATA CONFIGURATION				
Bytes	**Data Type**	**Name**	**Minimum**	**Maximum**
†	DINT	DynamicDataTypeID	0	53

† Index vary on the four available dynamic data configuration tags (see Table 42).

Table 50: Dynamic data configuration tag

5.6. Input tag assembly

The input tag assembly has an Instance of 100 with a size 252-byte array, as detailed in Table 51. Please refer to Section 3.5 for a description of the objects and their fields.

The following subsections define how these objects are mapped to EtherNet/IP input tag assembly (as also described in the EDS file).

INPUT TAG ASSEMBLY			
Bytes	**Data Type**	**Data**	**Description**
0–1	WORD	RobotStatus	See Table 52.
2–3	UINT	ErrorCode	See Table 53.
4–7	UDINT	Checkpoint	See Table 54.
8–9	UINT	MoveID	See Table 55.
10–11	UINT	Motion queue space	See Table 56.
12–13	WORD	MotionStatus	See Table 57.
14–15	UINT	OfflineProgramID	See Table 58.

INPUT TAG ASSEMBLY			
Bytes	**Data Type**	**Data**	**Description**
16–19	REAL	TargetJointSet (joint 1)	See Table 59.
20–23	REAL	TargetJointSet (joint 2)	
24–27	REAL	TargetJointSet (joint 3)	
28–31	REAL	TargetJointSet (joint 4)	
32–35	REAL	TargetJointSet (joint 5)	
36–39	REAL	TargetJointSet (joint 6)	
40–43	REAL	TargetEndEffectorPose *x*	See Table 60.
44–47	REAL	TargetEndEffectorPose *y*	
48–51	REAL	TargetEndEffectorPose *z*	
52–55	REAL	TargetEndEffectorPose *α*	
56–59	REAL	TargetEndEffectorPose *β*	
60–63	REAL	TargetEndEffectorPose *γ*	
64	SINT	TargetConfiguration c_s (shoulder)	See Table 61.
65	SINT	TargetConfiguration c_e (elbow)	
66	SINT	TargetConfiguration c_w (wrist)	
67	SINT	TargetConfiguration c_t (last joint turn)	
68–71	REAL	WRF *x*	See Table 62.
72–75	REAL	WRF *y*	
76–79	REAL	WRF *z*	
80–83	REAL	WRF *α*	
84–87	REAL	WRF *β*	
88–91	REAL	WRF *γ*	
92–95	REAL	TRF *x*	See Table 63.
96–99	REAL	TRF *y*	
100–103	REAL	TRF *z*	
104–107	REAL	TRF *α*	
108–111	REAL	TRF *β*	
112–115	REAL	TRF *γ*	
116–119	UDINT	RobotTimestamp (seconds part)	See Table 64.
120–123	UDINT	RobotTimestamp (microseconds part)	
124–127	UDINT	DynamicDataUpdateCount	
128–131	UDINT	Reserved for future use	
132–135	UDINT	Reserved for future use	
136–139	UDINT	Reserved for future use	

INPUT TAG ASSEMBLY			
Bytes	**Data Type**	**Data**	**Description**
140–143	UDINT	DynamicData #1 type ID	See Table 65.
144–147	REAL	DynamicData #1 value 1	
148–151	REAL	DynamicData #1 value 2	
152–155	REAL	DynamicData #1 value 3	
156–159	REAL	DynamicData #1 value 4	
160–163	REAL	DynamicData #1 value 5	
164–167	REAL	DynamicData #1 value 6	
168–171	UDINT	DynamicData #2 type ID	
172–175	REAL	DynamicData #2 value 1	
176–179	REAL	DynamicData #2 value 2	
180–183	REAL	DynamicData #2 value 3	
184–187	REAL	DynamicData #2 value 4	
188–191	REAL	DynamicData #2 value 5	
192–195	REAL	DynamicData #2 value 6	
196–199	UDINT	DynamicData #3 type ID	
200–203	REAL	DynamicData #3 value 1	
204–207	REAL	DynamicData #3 value 2	
208–211	REAL	DynamicData #3 value 3	
212–215	REAL	DynamicData #3 value 4	
216–219	REAL	DynamicData #3 value 5	
220–223	REAL	DynamicData #3 value 6	
224–227	UDINT	DynamicData #4 type ID	
228–231	REAL	DynamicData #4 value 1	
232–235	REAL	DynamicData #4 value 2	
236–239	REAL	DynamicData #4 value 3	
240–243	REAL	DynamicData #4 value 4	
244–247	REAL	DynamicData #4 value 5	
248–251	REAL	DynamicData #4 value 6	

Table 51: Input tag assembly

5.6.1 Robot status tag

The structure of the robot status tag is described in Table 52. See Table 13 for detailed explanations.

ROBOT STATUS TAG								
Bytes	**Data Type**	**Bits 6–15**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
0–1	WORD	Unused	RecoveryMode	BrakesEngaged	SimMode	Homed	Activated	Busy

Table 52: Robot status tag

5.6.2 Error code tag

The structure of the error code tag is described in Table 53. See Table 13 for detailed explanations.

ERROR CODE TAGS				
Bytes	**Data Type**	**Name**	**Minimum**	**Maximum**
2–3	UINT	ErrorCode	0	65,535

Table 53: Error code tag

5.6.3 Checkpoint tag

The structure of the checkpoint tag is described in Table 54. See Table 14 for detailed explanations.

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

Table 54: Checkpoint tag

5.6.4 MoveID tag

The structure of the MoveID tag is described in Table 55. See Table 14 for detailed explanations.

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

Table 55: MoveID tag

5.6.5 FIFO space tag

The structure of the Motion queue space tag is described in Table 56. See Table 14 for detailed explanations.

FIFO SPACE TAG				
Bytes	**Data Type**	**Name**	**Minimum**	**Maximum**
10–11	UINT	Motion queue space	0	13,000

Table 56: Motion queue space tag

The motion queue space may still be at its maximum value (13000) after several commands, even if they have not yet been executed. In fact, the robot will compile some commands in advance and remove them from the motion queue before they are executed.

5.6.6 Motion status tag

The structure of the motion status tag is described in Table 57. See Table 14 for detailed explanations.

MOTION STATUS TAG								
Bytes	**Data Type**	**Bits 6–15**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
12–13	WORD	Unused	ExcessiveTorque	PStop	Cleared	EOM	EOB	Paused

Table 57: Motion status tag

5.6.7 Offline program ID

This tag indicates the ID of the offline program currently running (Table 58). See Table 14 for details.

OFFLINE PROGRAM ID				
Bytes	**Data Type**	**Name**	**Minimum**	**Maximum**
14–15	UINT	OfflineProgramID	1	500

Table 58: Offline program tag

5.6.8 Target joint set

The structure of the target joint set tag is described in Table 59. The data is the same as that returned by TCP/IP command *GetRtTargetJointPos*.

TARGET JOINT SET TAGS				
Bytes	**Data Type**	**Name**	**Minimum**	**Maximum**
16–19	REAL	Target position of joint 1	−175.000	175.000
20–23	REAL	Target position of joint 2	−70.000	90.000
24–27	REAL	Target position of joint 3	−135.000	70.000
28–31	REAL	Target position of joint 4	−170.000	170.000
32–35	REAL	Target position of joint 5	−115.000	115.000
36–39	REAL	Target position of joint 6	−36,000.000	36,000.000

Table 59: Target joint set tag

5.6.9 Target end-effector pose

The structure of the target end-effector pose tag is described in Table 60. The data is the same as that returned by TCP/IP command *GetRtTargetCartPos*.

TARGET END-EFFECTOR POSE TAGS		
Bytes	**Data Type**	**Name**
40–43	REAL	Coordinate *x*
44–47	REAL	Coordinate *y*
48–51	REAL	Coordinate *z*
52–55	REAL	Euler angle α
56–59	REAL	Euler angle β
60–63	REAL	Euler angle *y*

Table 60: Target end-effector pose tag assembly

5.6.10 Target configuration

The structure of the target configuration tag is described in Table 61. The data is the same as that returned by the combination of the TCP/IP commands *GetRtTargetConf* and *GetRtTargetConfTurn*.

TARGET CONFIGURATION TAGS				
Bytes	**Data Type**	**Name**	**Minimum**	**Maximum**
64	SINT	c_s (shoulder)	−1	1
65	SINT	c_e (elbow)	−1	1
66	SINT	c_w (wrist)	−1	1
67	SINT	c_t (last joint turn)	−100	100

Table 61: Robot target configuration tags

5.6.11 WRF

The structure of WRF tag is described in Table 62. The data is the same as that returned by *GetRtWrf*.

WRF TAG ASSEMBLY		
Bytes	**Data Type**	**Name**
68–71	REAL	Coordinate *x*
72–75	REAL	Coordinate *y*
76–79	REAL	Coordinate *z*
80–83	REAL	Euler angle α
84–87	REAL	Euler angle β
88–91	REAL	Euler angle *y*

Table 62: WRF tag assembly

5.6.12 TRF

The structure of TRF tag is described in Table 63; it is the same data as what is returned by *GetRtTrf*.

TRF TAG ASSEMBLY		
Bytes	**Data Type**	**Name**
92–95	REAL	Coordinate *x*
96–99	REAL	Coordinate *y*
100–103	REAL	Coordinate *z*
104–107	REAL	Euler angle α
108–111	REAL	Euler angle β
112–115	REAL	Euler angle *y*

Table 63: TRF tag assembly

5.6.13 Robot timestamp

The structure of the Robot timestamp tag is described in Table 64. See Table 15 for details.

ROBOT TIMESTAMP TAG ASSEMBLY		
Bytes	**Data Type**	**Name**
116–119	UDINT	RobotTimestamp (seconds part)
120–123	UDINT	RobotTimestamp (microseconds part)
124–127	UDINT	DynamicDataUpdateCount
128–131	UDINT	Reserved for future use
132–135	UDINT	Reserved for future use
136–139	UDINT	Reserved for future use

Table 64: Robot timestamp tag assembly

5.6.14 Dynamic data

The structure of the dynamic data tags are described below. See Table 16 for detailed explanations.

DYNAMIC DATA TAG		
Bytes	**Data Type**	**Name**
†	UDINT	DynamicDataTypeID
†	REAL	Value 1
†	REAL	Value 2
†	REAL	Value 3
†	REAL	Value 4
†	REAL	Value 5
†	REAL	Value 6

† Indices vary with each of the four dynamic data structures (see Table 51).

Table 65: Dynamic data tag assembly

6. PROFINET COMMUNICATION

Certified by PROFIBUS, the Meca500 is compatible with the PROFINET protocol, a common industry standard that can be used with many different PLC brands. Cyclic times up to 1 ms (though not as "hard-real-time" as EtherCAT).

PROFINET—like EtherCAT or EtherNet/IP protocols—controls the robot using cyclic messaging ('CR Input' and 'CR Output' in PROFINET terms).

6.1. PROFINET conformance class

The Electronic Data Sheet (EDS) file for the Meca500 robot is included in the zip file that contains the robot firmware update. These zip files are available in the Downloads section of our web site.

6.1.1 PROFINET limitations on the Meca500 robot

The Meca500 robot does not support the following PROFINET features:

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

6.2. Connection types

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

6.2.1 Limitations when daisy-chaining robots

Please note that the two Ethernet ports on the Meca500 robot act as an un-managed Ethernet switch, not as a "PROFINET-aware" switch. In fact, this Ethernet switch will not respond to LLDP (Local Link Discovery Protocol) packets like a PROFINET-enabled switch would (instead, it forwards LLDP through the daisychain). As a consequence, the LLDP protocol will not properly identify the network topology when the two Ethernet ports of the robots are connected (in a daisy-chain configuration, for example).

Fortunately, this does not prevent the use of PROFINET protocol, since daisy-chained robots will still be detected by the PROFINET controller.

If you need full network topology discovery using LLDP, we recommend connecting the Meca500 robot to a PROFINET-enabled Ethernet switch rather than in a daisy chain.

6.2.2 PROFINET protocol over your Ethernet network

The PROFINET protocol uses non-IP packets to communicate real-time data over the Ethernet network. Please make sure that your Ethernet network and switches are properly forwarding these packets between the PROFINET controller (PLC) and the Meca500 robots.

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

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

Ethernet packets of type PROFINET RT (0x8892) are used for PROFINET cyclic data exchanges between the Meca500 robots and the PROFINET controller (PLC).

6.3. Enabling PROFINET

To enable the PROFINET communication protocol, you must first connect to the robot via the TCP/IP protocol through an external client (e.g., from a PC using a Web browser), then send the *EnableProfinet*(1) command. This is a persistent command; it only needs to be set once. To disable PROFINET, you need to send the *EnablePROFINET*(0) command.

Note that EtherNet/IP can be left permanently enabled since it does not prevent using the TCP/IP protocol, unlike EtherCAT and the *SwitchToEtherCAT* command.

Also note that LLDP forwarding on the Meca500 robot is enabled only when PROFINET is enabled on the robot (so it will not be possible to detect a Meca500 robot using LLDP until PROFINET is enabled on it).

6.4. Exclusivity of AR

On the Meca500, only one AR (Application relationship) can be established with the robot. Only one PROFINET controller (PCL) can control a Meca500 robot.

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

If a PROFINET connection request is refused because the Meca500 robot is already being controlled by another PROFINET controller (PLC), the refused connect request will be returned with standard error codes and the following values:

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

If a PROFINET connection request is refused because the Meca500 robot is already being controlled by another protocol (TCP/IP or EtherNet/IP), the refused connect request will be returned with a vendor-specific error code and the following values:

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

6.5. GSDML file

Each PROFINET slave device is described by a GSDML file (General Station Description XML file). The GSDML file describes the device capabilities, and the PROFINET Modules and SubModules that it supports. The PROFINET controllers (PLC) use this file to properly identify detected PROFINET devices, like the Meca500 robot.

The Meca500 GSDML file is provided along with the Meca500 firmware updates starting with release 9.1. It is also available in the Downloads section of our web site.

Since the GSDML file contains necessary information to identify and list the Meca500 robot capabilities, this manual will only provide a quick summary of the Meca500's GSDML file.

6.5.1 Meca500 modules and sub-modules

The Meca500 robot supports only one module and one sub-module, fixed in a predefined slot.

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

This module provides fixed cyclic data input and output, used to control and monitor the Meca500 robot.

6.6. Cyclic data

Using cyclic data to control and monitor Meca500 robots with PROFINET is explained in Section 3 of this manual.

This cyclic data format is exactly the same with PROFINET, EtherNet/IP and EtherCAT protocols. It is thus very easy to migrate a Meca500-controlling application on a controller/PLC between these different protocols.

Please refer to the Meca500 GSDML file for the list of cyclic input/output fields and refer to Section 3 of this document to learn how to use these cyclic fields.

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

6.7. Alarms

The Meca500 robot will not generate any PROFINET alarms. Any alarm or error condition will be reported by the robot through the corresponding cyclic data fields. This allows the Meca500 to behave the same across various Cyclic protocols (like PROFINET, EtherNet/IP or EtherCAT).

Please refer to Section 3 of this manual for more information about robot status and error states reported in the cyclic input data.

7. GLOSSARY

Table 66 presents of summary of the terms that we use frequently in our manuals and in the robot's web interface.

GLOSSARY OF TERMS

TERM	DESCRIPTION
BRF	Base reference frame (see Figure 2).
Cartesian space	The space where the location of an object, such as the robot's end-effector, is defined by a pose (position and orientation). For example, we can say that a *`MoveLin*`* command forces the TCP to follow a straight line in Cartesian space.
Control port	The TCP port over which commands to the robot and messages from the robot are sent (see Section 2).
Euler angles	Three angles corresponding to three consecutive rotations, used to define the orientation in space of one reference frame with respect to another (see Section 1.1.4).
EOAT	End-of-arm-tooling. Mecademic offers two electric grippers (MEGP 25E and MEGP 25LS) and one pneumatic module (MPM500) that can be controlled directly by the Meca500.
FRF	Flange reference frame (see Figure 2).
Inverse kinematics	The problem of finding all possible joint sets for a desired pose of the TRF with respect to the WRF (see Section 1.2.1).
Joint angle	The angle associated with robot joint i (i = 1, 2, ..., 6), denoted by θ_i and measured in degrees (see Section 1.1.5).
Joint position	The set of all joint angles, i.e., $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$, also referred to as joint set (see Section 1.1.6).
Joint set	The set of all joint angles, i.e., $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$, also referred to as joint position (see Section 1.1.6).
Joint space	The six-dimensional space defined by the positions of the robot joints.
Monitoring port	The TCP port over which data is sent periodically from the robot (see Section 2).
Motion command	A command used to construct a trajectory for the robot. When the meca500 receives a motion command, it places it in a motion queue (see Section 2.1). Examples include the commands delay, *`MoveLin`*, *`SetTrf`*, and *`SetJointVel`*.
Motion queue	A buffer where motion commands that were sent to the robot are stored and executed on a FIFO basis by the robot (see Section 2.1).
Offline program	A sequence of motion commands that can be saved in the robot's storage using the commands *`StartSaving`* and *`StopSaving`*, and later called by the command *`StartProgram`*.
Pose	Position and orientation of one reference frame with respect to another.

GLOSSARY OF TERMS	
TERM	**DESCRIPTION**
Position mode	One of the two control modes, in which the robot's motion is generated by requesting a target robot position or joint position (see Section 1.3.4).
Posture configuration	The set of two-value (−1 or 1) parameters c_s, $c_{e,}$ and c_w that normally define each of the eight possible robot postures for a given pose of the robot's end-effector.
Request command	A command that is executed immediately and returns a specific response (see Section 2.2). Examples includes the commands *`ClearMotion`*, *`Home`*, *`GetRobotName`*, *`GetTrf`*, and *`GetrGripperForce`*.
Robot position	The pose of the robot's end-effector, as well as the four configuration parameters. If the robot is in a singularity, however, we cannot define a robot position, and must define a joint position instead.
Robot posture	The arrangement of the robot links. Thus, the joint sets $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$ and $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6 + n360°\}$, where n is an integer, correspond to the same robot posture (see Section 1.1.6).
Singularity	A robot posture where the robot's end-effector is blocked in some direction even if no joint is at a limit (see Section 1.2.3). There are three types of singularities, corresponding to conditions where each of the three posture parameters are not defined: shoulder singularity, elbow singularity, wrist singularity.
TCP	Tool center point, the origin of the tool reference frame (see Figure 2).
TRF	Tool reference frame (see Figure 2).
Turn configuration	An integer c_t, such that $-180° + c_t360° < \theta_6 \leq 180° + c_t360°$ (see Section 1.1.5).
Velocity mode	One of the two control modes, in which the robot's motion is generated by requesting a target end-effector or joint velocity (see Section 1.3.4).
Workspace	The set of all poses of the TRF with respect to the WRF that are reachable with at least one posture and turn configuration (see Section 1.2.3).
WRF	World reference frame (see Figure 2).
Wrist center	The point where the axes of joints 4, 5 and 6 intersect.

Table 66: Glossary of terms used by Mecademic

Contact Us

Mecademic
1300 St-Patrick Street
Montreal (Quebec) H3K 1A4
Canada

1-514-360-2205
1-833-557-6268 (toll-free in North America)

https://support.mecademic.com

MECADEMIC
INDUSTRIAL ROBOTICS