MECADEMIC
INDUSTRIAL ROBOTICS

MECADEMIC INDUSTRIAL ROBOTS

PROGRAMMING MANUAL

Document ID: MC-PM-EN

For Firmware Version 10.1

Document Revision: B

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, an API, 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.

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

Reading the user manual of your robot 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.

Other conventions

Important terms are typeset in red and in italics, as in *Euler angles*, and are listed at the end of this manual.

Important passages are sometimes underlined, instead of repeating the information in a separate notice.

Occasionally, an asterisk is placed after a word, not formated as a superscript, as a wildcard specifier. For example, MEGP 25* refers to both the MEGP 25E and MEGP 25LS grippers.

TCP/IP commands are typeset in a monospaced typeface font, as in `MoveJoints`.

Revision history

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

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

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

REVISION	DATE	COMMENTS
A	February 14, 2024	Original version.
B	April 8, 2024	Some fixes regarding Dynamic Data Type IDs 12–15, 22, 73.

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

References

The following documents are referred to in this programming manual:

Document title	Document ID
Meca500 Industrial Robot - User Manual	MC-UM-MECA500-EN
MCS500 SCARA Industrial Robot - User Manual	MC-UM-MCS500-EN
MEGP 25E/25LS Electric Parallel Grippers - User Manual	MC-UM-MEGP25-EN
MPM500 Pneumatic Module - User Manual	MC-UM-MPM500-EN
MVK01 Vacuum and I/O Module - User Manual	MC-UM-MVK01-EN

These documents are available as PDF files on Mecademic's web site.

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 industrial robot arms.

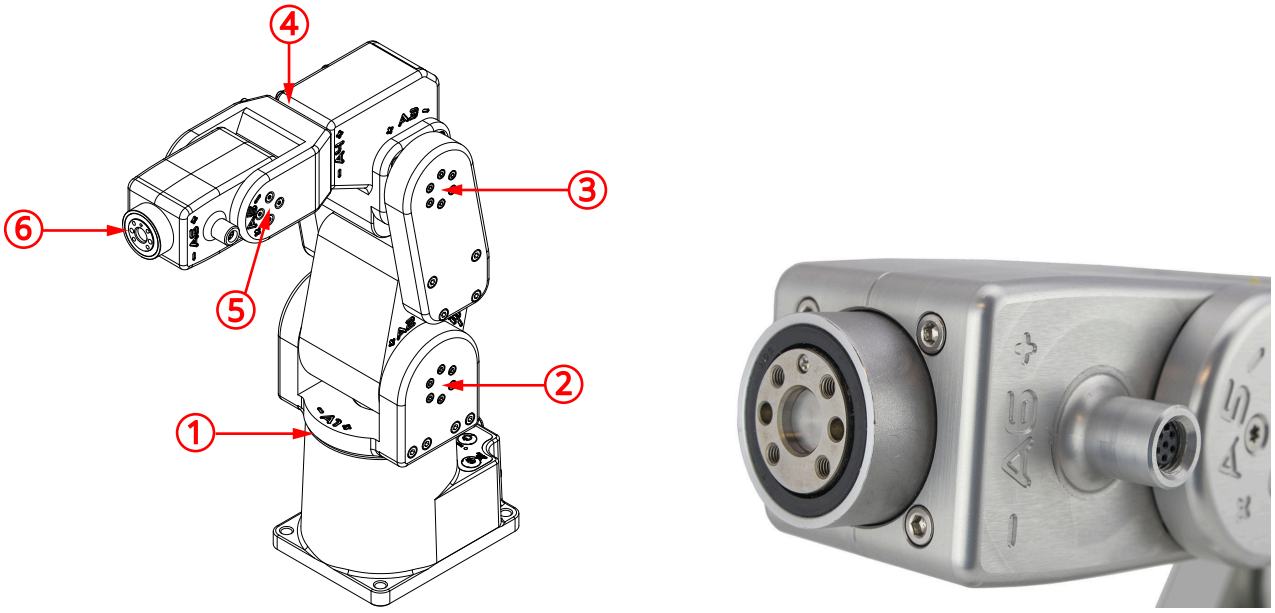
1.1. Definitions and conventions

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. Figure 1 also shows the zero joint positions. In that "zero" robot position, the axis of joint 3 intersects the axis of joint 1, the axes of joints 4 and 6 are aligned and normal to the axis of joint 1 and the axes of joints 2, 3 and 5 are parallel.



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

Figure 1: Meca500's joint numbering and zero-degree joint positions

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

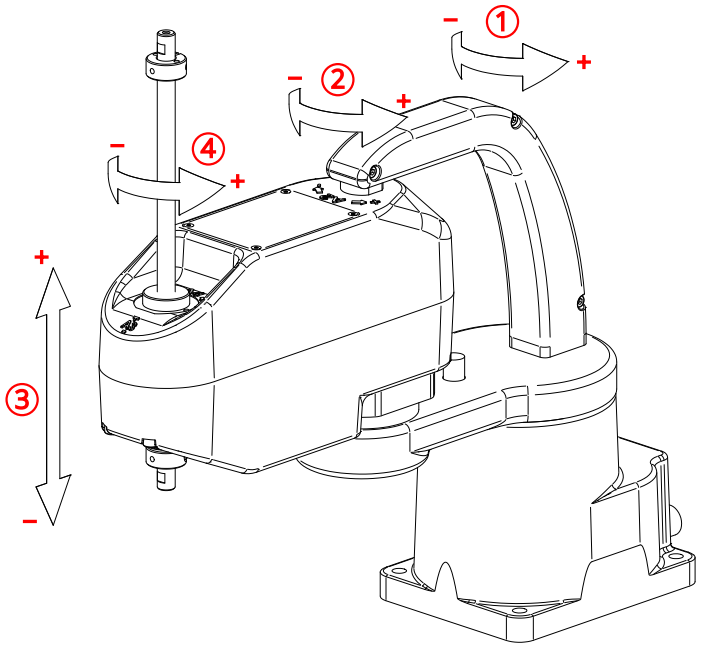


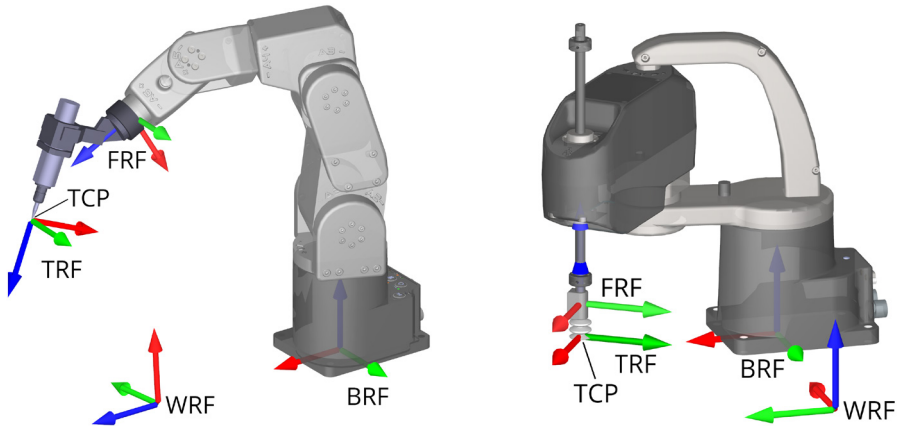
Figure 2: MCS500's joint numbering, joint directions and zero joint positions

1.1.3 Reference frames

We use right-handed Cartesian coordinate systems (*reference frames*). There are only four of them that you need to be familiar with, as shown in Figure 3 (*x* axes are red, *y* axes are green, and *z* axes are blue). These four reference frames and the key term related to them are:

- *BRF: 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 with respect to the BRF using the `SetWrf` command.
- *FRF: Flange reference frame*. Mobile reference frame fixed to the robot's *flange*. In the Meca500, the flange is the 20-mm disk with threaded holes at the extremity of the robot, shown in Figure 1b. The *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.

  In the case of the MCS500, the FRF is fixed to the end of the spline shaft that is closer to the robot's base, so that its *z* axis coincides with the axis of the spline shaft and points away from the robot's base, its origin is at the very end of the spline shaft, and its *x* axis is perpendicular to the plane of the Weldon flat.
- *FCP: flange center point*. Origin of the FRF.
- *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.)



(a) Meca500 (b) MCS500

Figure 3: Reference frames for the Meca500 and the MCS500

1.1.4 Pose and Euler angles

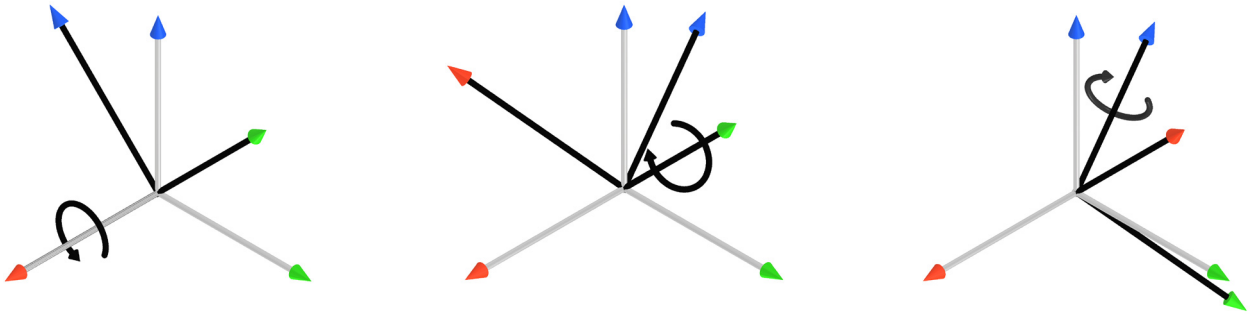
Some Mecademic commands accept *pose* (position and orientation of one reference frame with respect to another) as arguments. In these commands, and in the Mecademic robots' web interface (the MecaPortal), a pose consists of a Cartesian position, {*x*, *y*, *z*}, and an orientation specified in *Euler angles*, {*α*, *β*, *γ*}, according to the mobile XYZ convention (also referred to as RxRyRz, or XY'Z''). In this convention, if the orientation of a frame F_1 with respect to a frame F_0 is described by the Euler angles {*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* (alpha) degrees, then about its *y* axis by *β* (beta)degrees, and finally about its *z* axis by *γ* (gamma) degrees, the final orientation of frame F_m will be the same as that of frame F_1.

Figure 4 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 define a given orientation, the commands that accept a pose as arguments, accept any numerical value for the three Euler angles (e.g., the set {378.34°, −567.32°, 745.03°}). However, we output only the equivalent Euler angle set {*α*, *β*, *γ*}, for which $-180° \le \alpha \le 180°$, $-90° \le \beta \le 90°$ and $-180° \le \gamma \le 180°$. Furthermore, if you specify the Euler angles {*α*, ±90°, *γ*}, the controller will always return an equivalent Euler angle set with $\alpha = 0$. Thus, it is perfectly normal that the Euler angles used to specify an orientation are not the same as the Euler angles returned by the controller, once that orientation has been attained (see our tutorial on Euler angles).

Since the end-effector of the MCS500 SCARA has only one rotational degree of freedom—a rotation about the spline-shaft axis—we have simplified orientations by keeping the alpha and beta angles equal to zero. Thus, in the case of the MCS500, all commands that accept a pose, accept only the spacial position {*x*, *y*, *z*}, and the single orientation angle *γ*. In other words, all reference frames in the MCS500 have their z axes upwards. It is impossible, for example, to specify a TRF with the z-axis pointing downwards.

All commands that take {*x*, *y*, *z*, *α*, *β*, *γ*} as arguments in the case of the Meca500, will only accept {*x*, *y*, *z*, *γ*} or {*x*, *y*, *z*, 0, 0, *γ*} in the case of the MCS500.



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

Figure 4: 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 positions and last joint turn configuration

The angle associated with a rotational joint *i*, θ_i, will be referred to as *joint angle i*, while the position of the translational joint 3 of the MCS500, d_3, will be referred to as *joint position*. However, for simplicity, we will often use the term joint position and the notation q_i, for both the translational and rotational joints.

Since the last joint of the Meca500 (joint 6) and the last joint of the MCS500 (joint 4) can rotate more than one revolution, you should think of the joint angle as a motor angle, rather than as the angle between two consecutive robot links.

Note that the directions of rotation or of translation for each joint are engraved on the robot's body. All joint positions are zero in Figure 1 and Figure 2. Yet, unless you attach an end-effector with cabling to the robot flange, there is no way of knowing the value of the last joint angle just by observing the robot.

In the case of the Meca500, 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, we define the integer c_t as the axis 6 *turn configuration*, so that $-180° + c_t 360° < \theta_6 \leq 180° + c_t 360°$.

In the case of the MCS500, the mechanical limits for the first three robot joints are as follows:

$$-140° \leq \theta_1 \leq 140°,$$

$$-145° \leq \theta_2 \leq 145°,$$

$$-102 \text{ mm} \leq d_3 \leq 0 \text{ mm}.$$

Similarly, in the MCS500, joint 4 has no mechanical limits but its software limits are ±10 turns. Thus, for the MCS500, the integer c_t is the axis 4 *turn configuration*, such that $-180° + c_t360° < \theta_4 \leq 180° + c_t360°$.

Joints can be further constrained using the `SetJointLimits` command (or via the robot web interface).

1.1.6 Joint set and robot posture

There are several possible solutions for joint positions, for a desired pose of the robot end-effector with respect to the robot base, i.e., several possible sets $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$, in the case of the Meca500, and several possible sets $\{\theta_1, \theta_2, d_3, \theta_4\}$, in the case of the MCS500. The simplest way to describe how the robot is postured, is by giving its set of joint positions. This set will be referred to as the *joint set*.

For example, in Figure 1, 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 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 data as a robot posture AND the turn configuration of the last joint.

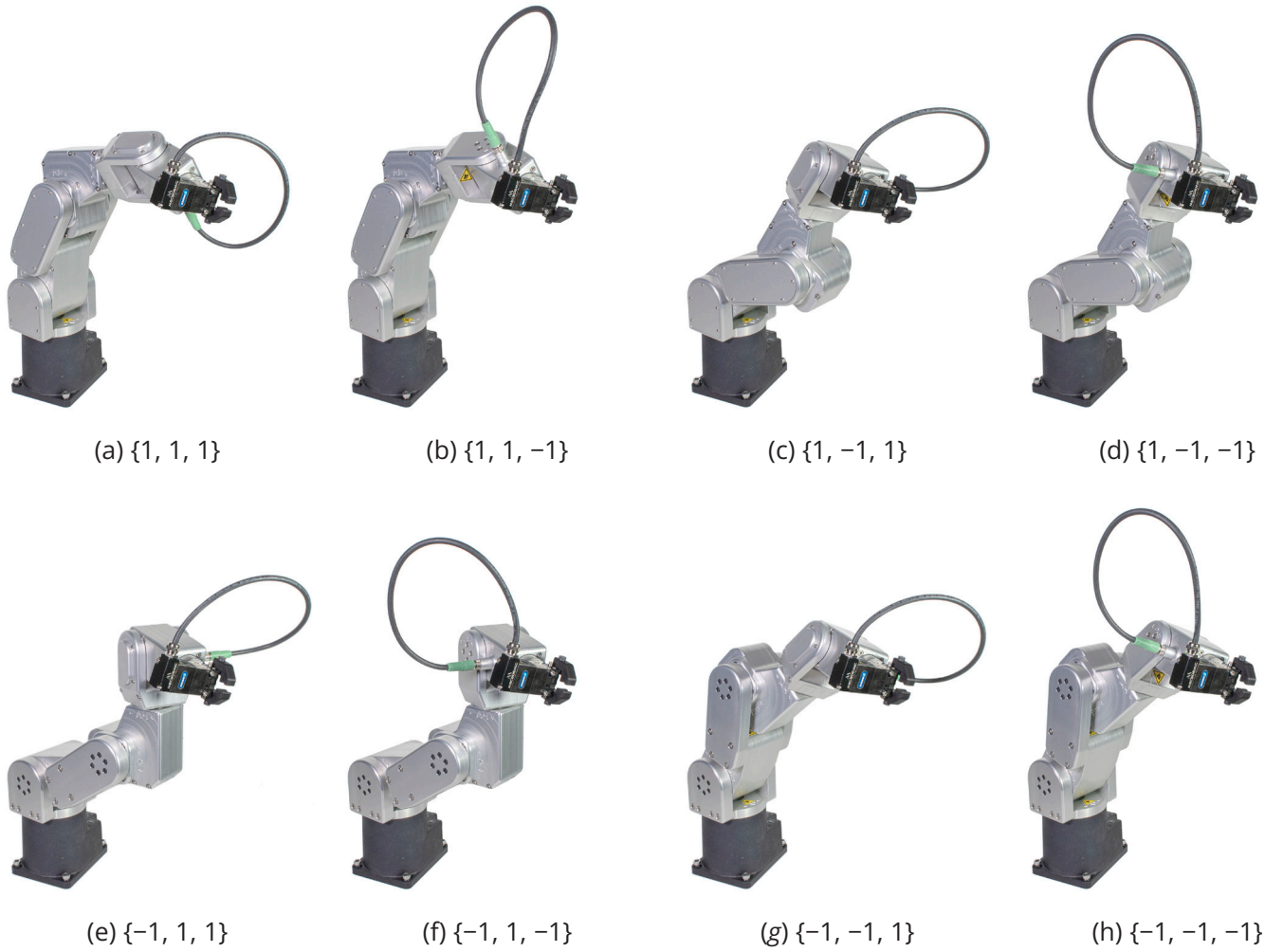
1.2. Configurations, singularities and workspace

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 5), 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 6):

- 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 6a). 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 6c).
- c_e (elbow configuration parameter)
  - $c_e = 1$, if $\theta_3 > -\arctan(60/19) \approx -72.43°$ ("elbow up" condition, see Figure 6d);
  - $c_e = -1$, if $\theta_3 < -\arctan(60/19) \approx -72.43°$ ("elbow down" condition, see Figure 6f).
- c_w (wrist configuration parameter)
  - $c_w = 1$, if $\theta_5 > 0°$ ("no flip" condition, see Figure 6g);
  - $c_w = -1$, if $\theta_5 < 0°$ ("flip" condition, see Figure 6i).

Figure 5 shows an example with all eight possible robot postures, described by the posture configuration parameters $\{c_s, c_e, c_w\}$, for the pose {77 mm, 210 mm, 300 mm, −103°, 36°, 175°} of the FRF with respect to the BRF.



(a) {1, 1, 1} (b) {1, 1, −1} (c) {1, −1, 1} (d) {1, −1, −1}

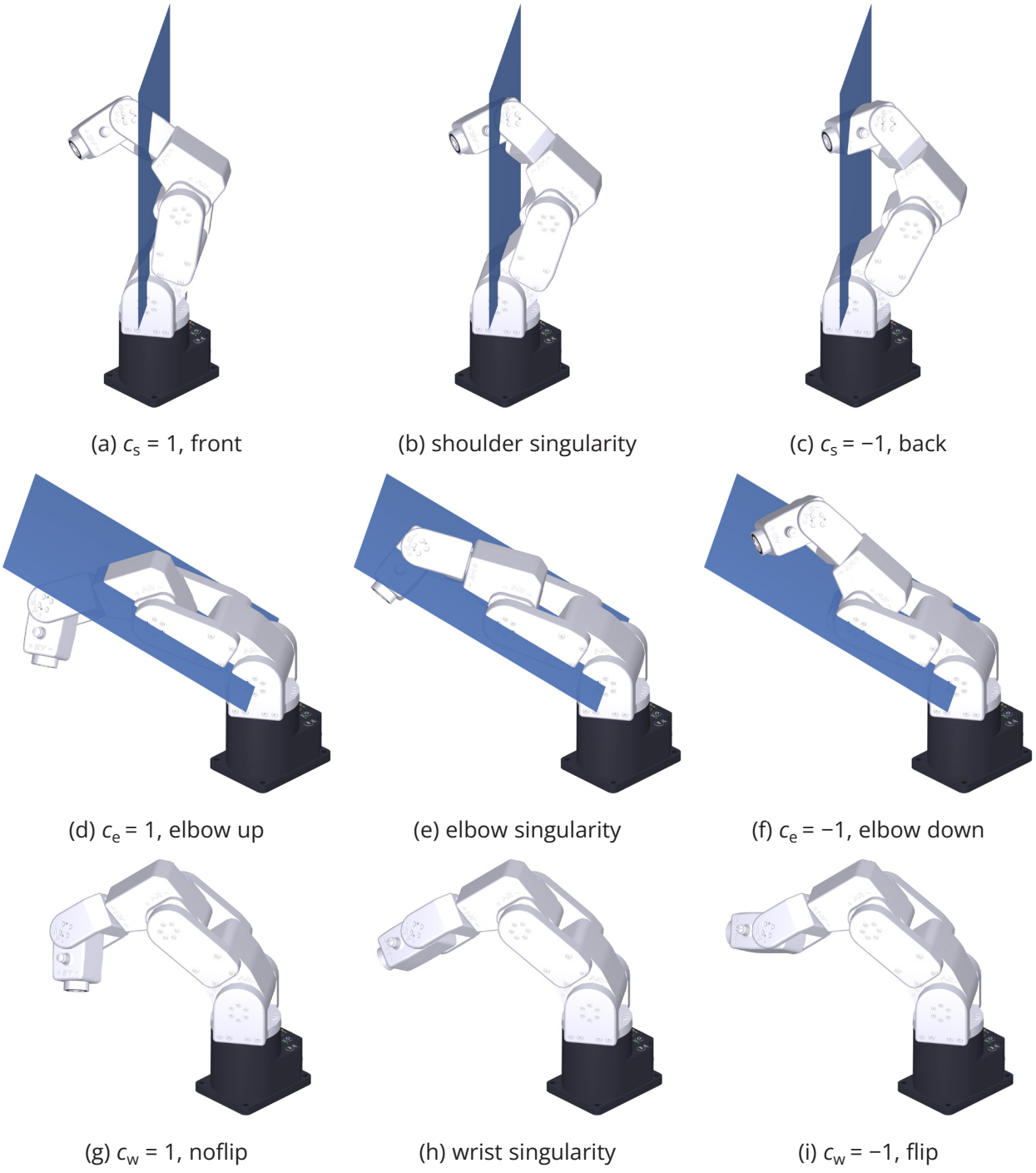
(e) {−1, 1, 1} (f) {−1, 1, −1} (g) {−1, −1, 1} (h) {−1, −1, −1}

Figure 5: An example showing all eight possible robot postures for the Meca500

Figure 6 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 6: Posture configuration parameters and the three types of singularities for the Meca500

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*.

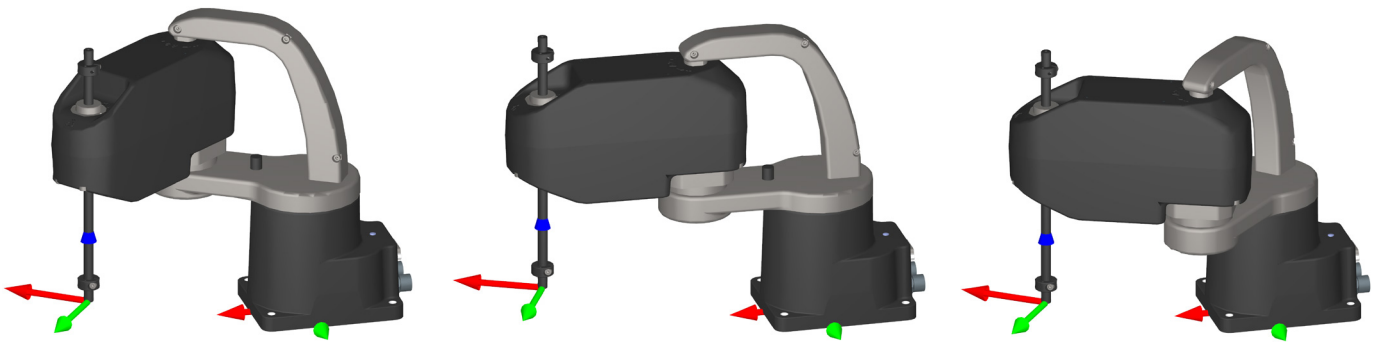
Thus, if you are teaching the *robot position* and want that later its end-effector moves to the current pose along a linear path, you need to record not only the current pose of the TRF with respect to the WRF (by retrieving it with `GetRtCartPos`), but also the definitions of the TRF and the WRF (with `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 four configuration parameters.

The situation is similar in the case of the MCS500, though much simpler. As shown in Figure 7, for the same pose of the end-effector, there are only two possible postures characterized by the only posture parameter c_e:

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

Thus, the `SetConf` command takes only one argument, in the case of the MCS500.

A singularity occurs when $\theta_3 = 0°$ i.e., when the axis of the spline shaft is coplanar with the axes of joints 1 and 2 (the arm is fully stretched).



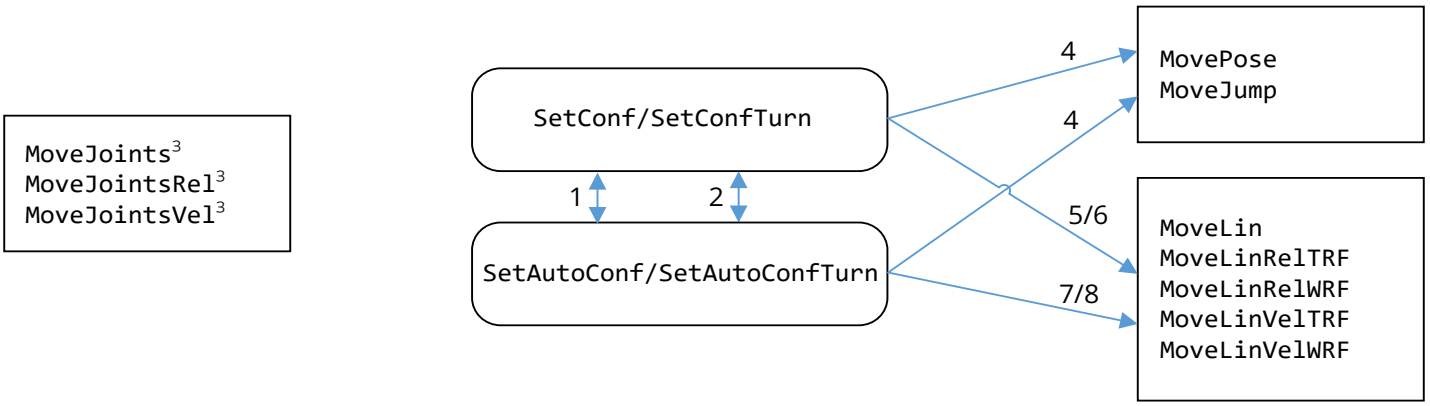
(a) $c_e = 1$, righty (b) elbow singularity (c) $c_e = -1$, lefty

Figure 7: The only posture configuration parameter and singularity in the MCS500

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 MecaPortal, the automatic configuration selection is always enabled. Or, if a target pose is identified in real-time based on input from a sensor (e.g., a camera), enabling the automatic configuration selection will increase your chances of reaching that pose, and in the fastest way.

Figure 8 illustrates how the automatic and manual configuration selections work.



Notes:

1. `SetConf` disables "AutoConf"; `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"; `SetAutoConfTurn(1)` disables the desired turn setting. Executing `SetAutoConfTurn(0)` will make the new desired turn the one corresponding to the current angle of the last joint.
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 the last joint 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 8: Effect of configuration parameters on robot movement commands

Firstly (Figure 8, 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 8, 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 8, note 4). In contrast, if automatic posture and/or turn configuration selection is enabled, `MovePose` will choose the joint set, corresponding to the desired end-effector pose, that is fastest to reach, and that satisfies the desired posture or turn configuration, if any.

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 8, 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 ("AutoConf") is enabled, a `MoveLin*` command may lead to changing the posture (if passing through a wrist or shoulder singularity) or turn configuration along the path.

Finally, note that there is currently no way of specifying only one of the posture configuration parameters and leaving the choice of the others to the robot controller. However, there is an indirect way to specify the elbow and wrist configurations, though this can't be done "on the fly". Indeed, if you prefer to always stick to one of the two possible wrist configurations in the Meca500, you can simply limit the range of joint 5, to either positive or non-negative values, using the command `SetJointLimits`. Similarly, you can fix the elbow configuration parameter by setting the range of joint 3 to be always smaller or larger than $-\arctan(60/19) \approx -72.43°$, in the case of the Meca500, or smaller or larger than 0°, in the case of the MCS500.

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 5). 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 6). 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 6h). 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 6f). 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 6h). 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.

In the case of the MCS500, the situation is similar but much simpler. The workspace of a SCARA robot, or more specifically the set of possible positions for the origin of its FRF, is essentially the set of two zones of height $d_{3,max}$ (102 mm in our case), one for the righty ($c_e > 0$) configuration and one of the lefty ($c_e < 0$) configuration, as shown in Figure 9. In Cartesian mode, you can move only in one of these zones. These overlapping zones are "separated" by the elbow singularity (the thick circular arc in Figure 9).

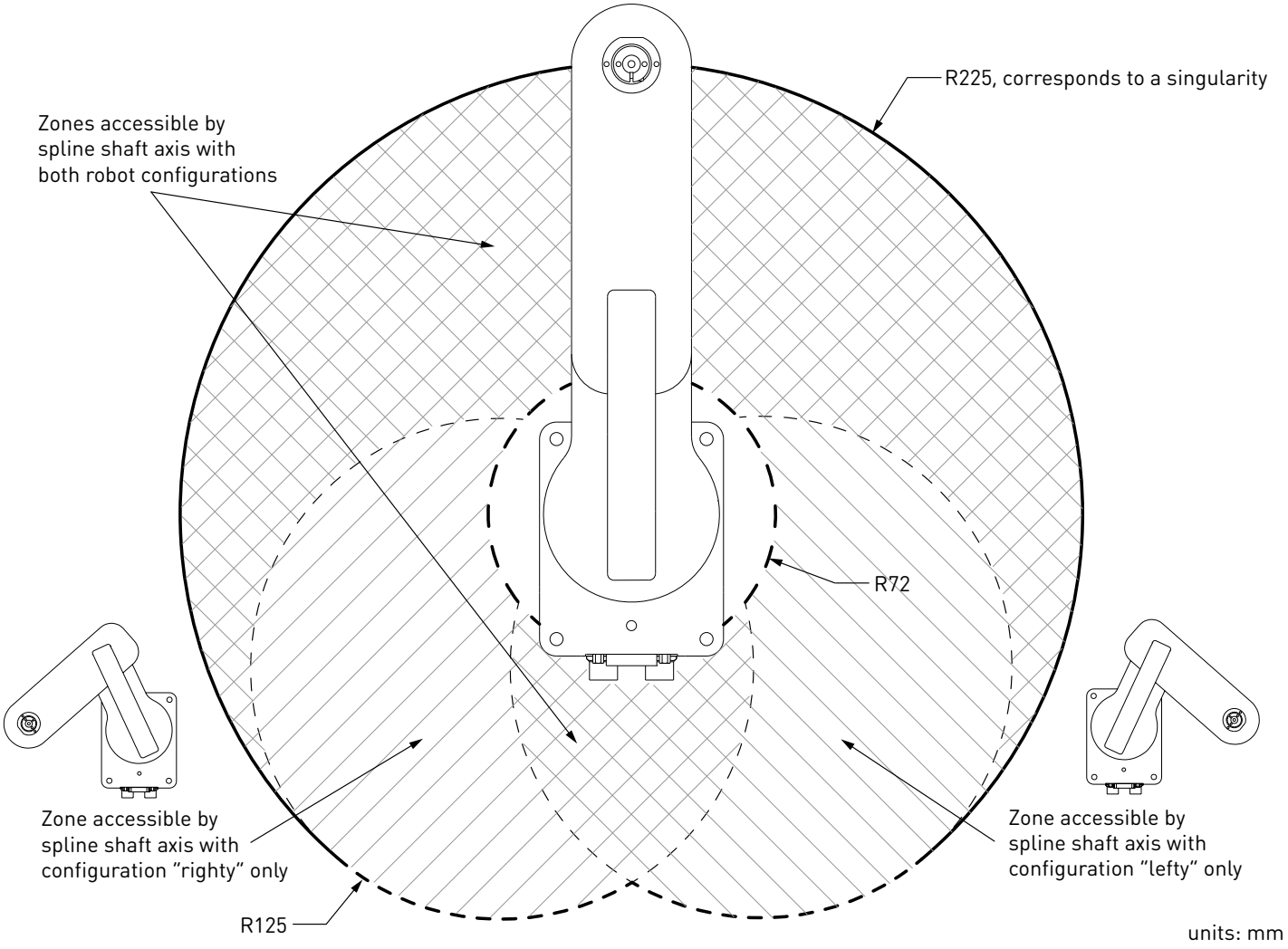


Figure 9: Top view of the workspace of the MCS500, which is a set of two overlapping zones

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 robots. With the release of firmware 9.1, the Meca500 can start at or pass through wrist and shoulder singularities, while executing any `MoveLin*` command, or end at any singularity while executing a `MoveLin*` or `MovePose` command. Furthermore, the passage respects the posture configuration selection settings (Figure 8). Figure 10 illustrates how this feature makes it possible to follow longer linear paths (see also this video). 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 11a, 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 as the range of joint 4 is ±170°.

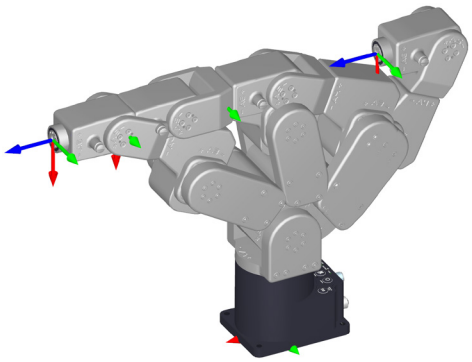
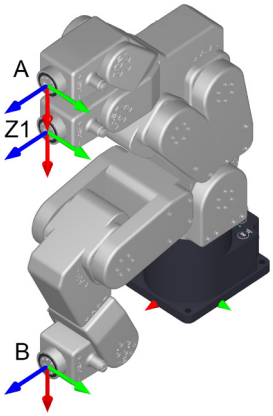
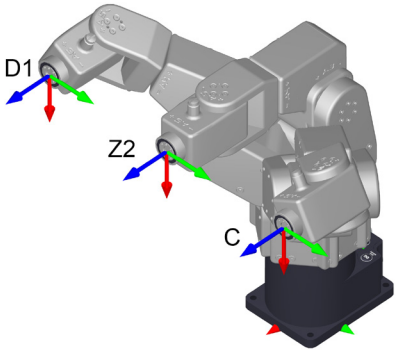


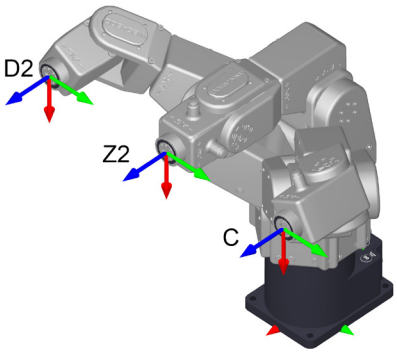
Figure 10: By crossing singularities, the Meca500 can execute longer linear movements



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



(b) C↔D1 via Z2, with `AutoConf(1)`



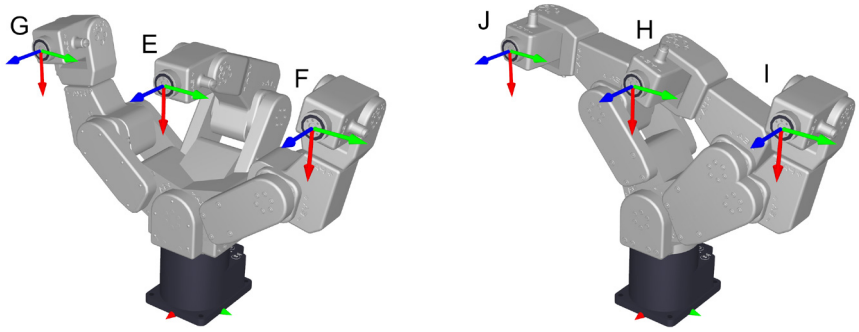
(c) C↔D2 via Z2 and stationary re-orientation, with `SetConf(1,1,−1)`

Figure 11: Crossing a wrist singularity with `AutoConf(1)` or with a desired posture configuration

Similarly, consider Figure 11b, 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 11c, 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 12a 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 12b.

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 12: 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 12b. 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 12a. Indeed, passing through or close to singularities often leads to longer cycle times, and should be avoided in production mode.

In the case of the MCS500, the only advantage of "crossing" a singularity when controlling the robot in Cartesian space is that you can jog the robot in Cartesian mode from a singular configuration or end up in a singular configuration. In other words, in the case of the MCS500 robot, singularity crossing is rather existing from a singularity and is only for convenience purposes.

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`), if the work zone has been breached (`SetWorkZoneLimits`, `SetWorkZoneCfg`) or if a collision has been software detected (`SetCollisionCfg`).

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.

Once the robot is homed, you do not need to home it again, even if you deactivate it, and then reactivate it, unless you use the optional argument 1, i.e., `ActivateRobot(1)`. In Meca500 R4, after an E-Stop has been reset, you do not need to run the homing procedure again, unless the robot is equipped with an MEGP 25* gripper (in that case, only the gripper is homed actually).

If you call the homing process, but homing was not needed, the robot will simply ignore the command (though it will still respond with the [2002][Homing done.] message). If homing was needed only for the MEGP 25* gripper, the gripper fingers will move, but not the robot.

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

The range of the absolute encoder and of the software limits of joint 4 of the MCS500 is only ±3600°. Do not manually rotate joint 4 beyond its software limits (e.g., by disabling the brakes).

1.3.2 Recovery mode

Once activated, if the robot is outside the user-defined joint limits (`SetJointLimits`), if the work zone has been breached (`SetWorkZoneLimits`, `SetWorkZoneCfg`), if a collision has been detected (`SetCollisionCfg`), or if the robot is too close to an obstacle, it may be necessary to move the robot before homing it, without manual intervention. We have 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, in the case of the Meca500, 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 an actual collision resulted in some joints rotating outside their user-defined joint limits (`SetJointLimits`), the robot breaching its work zone (`SetWorkZoneLimits`, `SetWorkZoneCfg`), a collision software detected (`SetCollisionCfg`), or joint torques that are outside their used-defined limits (`SetTorqueLimits`). 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 significantly 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.

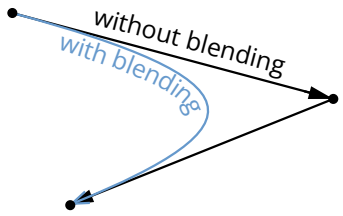


Figure 13: TCP path for two consecutive linear movements, with and without blending

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`, `MoveJump`) 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 13). 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).

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.

Furthermore, if blending is enabled, the gripper motion commands (`MoveGripper`, `GripperOpen`, `GripperClose`) will not cause the robot to stop between two position-mode joint-space commands (blending will occur normally). However, the gripper motion commands will force the robot to stop when used between two position-mode Cartesian-space commands. Once the robot has come to a stop, the gripper's fingers will start moving at the same time as the subsequent Cartesian-space movement.

In contrast, the `SetValveState` command, will not cause the robot to stop. Blending will occur normally, and the `SetValveState` command will be executed at the beginning of the blending path.

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 (as defined by `SetJointVelLimit`) 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 14 summarizes the possible settings for the velocity and acceleration in position mode.

There is a second method to control a Mecademic robot, by defining either its end-effector velocity or its joint velocities. This robot control method is called the *velocity mode*. Velocity mode is designed for advanced applications such as force control, dynamic path corrections, or telemanipulation (for example, the jogging feature in the MecaPortal 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 that cannot be crossed or a joint limit, it will simply stop without raising an error. Furthermore, the velocity of the robot's end-effector or of the robot joints is directly controlled, but is subject to the constraint set by the `SetJointVelLimit` command.

The `SetJointVelLimit` command affects the position-mode commands too. See Figure 14 for a complete description of how velocity and acceleration settings affect the two modes.

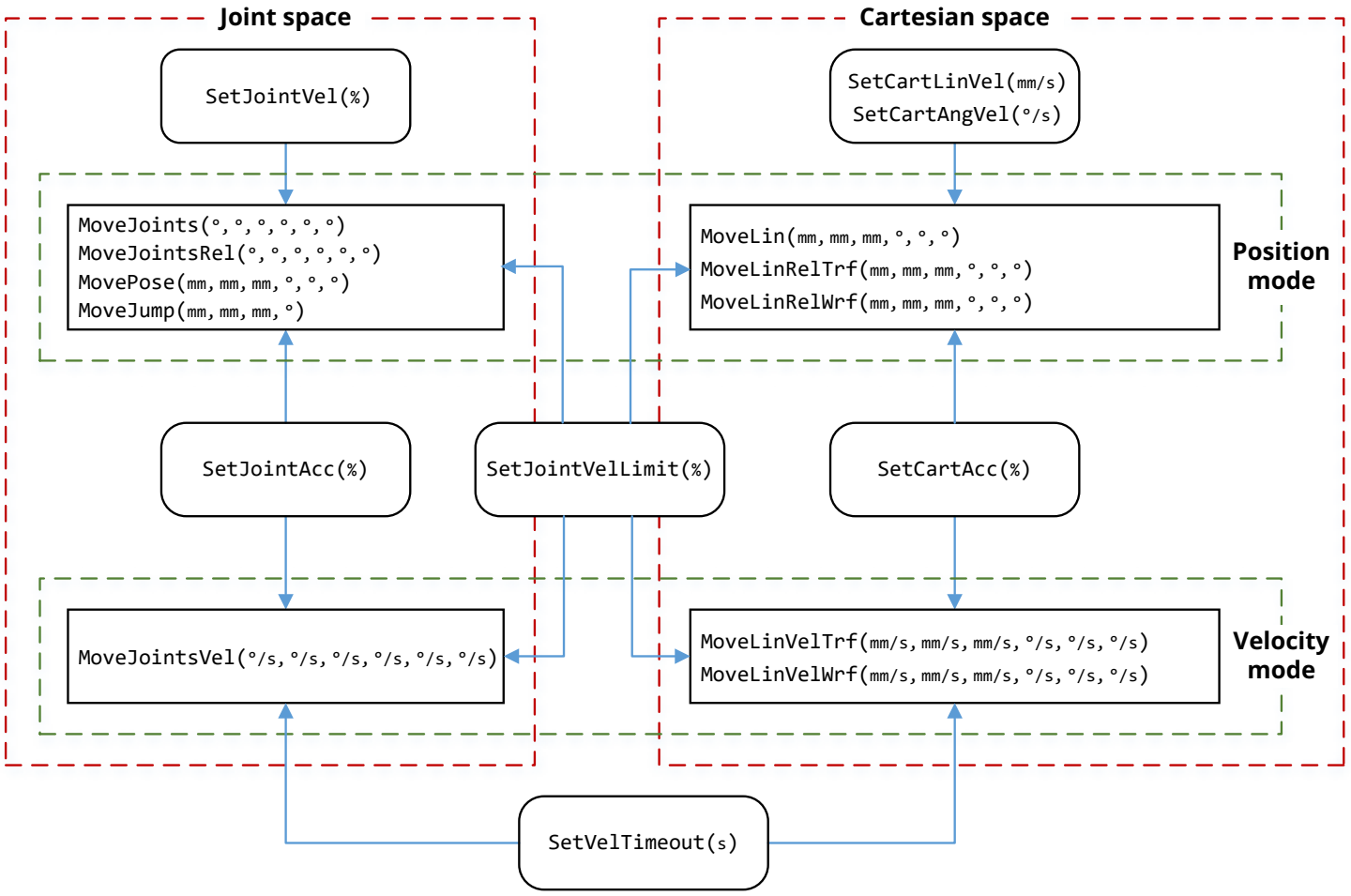


Figure 14: Settings that influence the robot motion in position and velocity modes

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

2. TCP/IP COMMUNICATION

Mecademic robots must be connected to a computer or to a PLC over Ethernet. Commands may be sent through Mecademic's web interface, the MecaPortal, 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, detailed in Sections 3–6. When the robot communicates using the TCP/IP protocol, it uses null-terminated ASCII strings. The default robot IP address is 192.168.0.100, and its default TCP port is 10000, referred to as the *control port*. 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 desynchronization between the data received from both parts, it is possible to send a copy of the monitoring port data to the control port data with the `SetCtrlPortMonitoring` command.

In this section, we will present all commands in the following categories, in terms of functionality:

- *motion commands*, which are the commands used to construct the robot trajectory (e.g., `Delay`, `MoveJoints`, `SetTRF`, `SetBlending`),
- *robot control commands*, which are commands used to control the robot (e.g., `ActivateRobot`, `Home`, `PauseMotion`, `SetNetworkOptions`)
- *robot data request commands*, which are commands used to request some data regarding the robot (e.g., `GetTRF`, `GetBlending`, `GetJointVel`),
- *real-time robot data request commands*, which are commands used to request some real-time data regarding the robot (e.g., `GetRtTrf`, `GetRtCartPos`, `GetStatusRobot`),
- *work zone supervision and collision detection commands*, which are commands used to set a bounding box for the robot and its tooling and define collision conditions, and query these settings and related statuses,
- *external-tool commands*, which are commands used to control or request data from the optional tools and modules for our robots (i.e., the electric grippers and pneumatic module for the Meca500, or the vacuum and I/O module for the MCS500).

However, commands can also be categorized in terms of whether they are executed immediately or not. *Queued commands* are placed in a *motion queue*, once received by the robot, and are executed on a FIFO basis. All motion commands and some external tool commands are queued. *Instantaneous commands* are executed immediately, as soon as received by the robot. All request commands (`Get*`), all robot control commands, all Cartesian limits commands and some external-tool commands (`*_Immediate`) are instantaneous.

Finally, some command descriptions refer to *default values*: these are essentially variables that are initialized every time the robot started. Of these, those that correspond to motion commands are also initialized every time the robot is deactivated (e.g., after an emergency stop). In contrast, certain parameter values are *persistent*: 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 essentially used to generate a trajectory for the robot. When a Mecademic robot receives a motion command, it places it in a motion queue. The command will be run once all preceding motion commands have been executed. In other words, motion commands are synchronous.

Most motion commands have arguments, but not all have default values (e.g., the argument for the command `Delay`). 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 an end-of-movement message (*EOM*, code 3004) whenever it has stopped moving for at least 1ms, if this option is activated with `SetEom`. The EOM message is sent whether or not all queued commands have been executed.

Furthermore, by default, the robot sends an end-of-block message (*EOB*, 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 2.8.1.

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 MoveJoints(q_1,q_2,q_3,q_4,q_5,q_6)

This command makes the robot simultaneously move all Its joints to the specified joint set, as fast as possible but subject to the limits set by the commands `SetJointVel` and `SetJointVelLimit`. All joints start and stop moving at the same time, so there is generally only one joint that moves at the joint velocity indirectly specified in `SetJointVel` and `SetJointVelLimit`. 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 15). Finally, with `MoveJoints`, the robot can cross singularities without any problem.

Arguments

- the (admissible) positions of all joints, in degrees or mm. In the case of the Meca500, the admissi-

ble default ranges for the joint angles are as follows:

$-175° \leq q_1 \leq 175°$,

$-70° \leq q_2 \leq 90°$,

$-135° \leq q_3 \leq 70°$,

$-170° \leq q_4 \leq 170°$,

$-115° \leq q_5 \leq 115°$,

$-36{,}000° \leq q_6 \leq 36{,}000°$.

In the case of the MCS500, there are only four arguments, and their admissible default ranges are:

$-140° \leq q_1 \leq 140°$,

$-145° \leq q_2 \leq 145°$,

$-102 \text{ mm} \leq q_3 \leq 0 \text{ mm}$,

$-3{,}600° \leq q_4 \leq 3{,}600°$.

Note that the above ranges can be further limited with the command `SetJointLimits`.

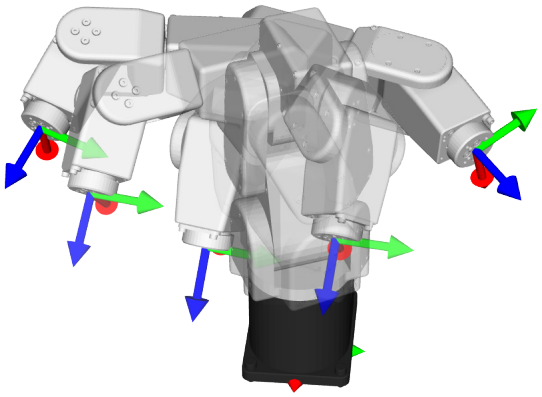


Figure 15: End-effector motion when using the `MoveJoints` or `MovePose` commands

2.1.3 MoveJointsRel(Δq_1,Δq_2,Δq_3,Δq_4,Δq_5,Δq_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 set and wish to avoid having to use the command `GetRtTargetJointPos`.

Arguments

- the desired relative displacement of joint *i* (*i* = 1, 2, ...), in degrees or mm. The value of the argument can be positive, negative or zero. There are only four arguments in the case of the MCS500.

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

This command makes the robot displace simultaneously its joints with the specified joint velocities. All joint displacements 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 15). With `MoveJointsVel`, the robot can cross singularities without any problem.

Arguments

- the velocities of all joints, in °/s or mm/s. In the case of the Meca500 R3, the admissible ranges are:

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

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

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

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

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

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

  In the case of the Meca500 R4, the admissible ranges are:

  $-225°/s \leq \dot{q}_1 \leq 225°/s$,

  $-225°/s \leq \dot{q}_2 \leq 225°/s$,

  $-225°/s \leq \dot{q}_3 \leq 225°/s$,

  $-350°/s \leq \dot{q}_4 \leq 350°/s$,

  $-350°/s \leq \dot{q}_5 \leq 350°/s$,

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

  In the case of the MCS500, there are only four arguments and their admissible ranges are:

  $-300°/s \leq \dot{q}_1 \leq 300°/s$,

  $-500°/s \leq \dot{q}_2 \leq 500°/s$,

  -700 mm/s $\leq \dot{q}_3 \leq 700$ mm/s,

  $-5{,}000°/s \leq \dot{q}_4 \leq 5{,}000°/s$.

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

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

2.1.5 MoveJump(*x*,*y*,*z*,*γ*)

This command can only be used on the MCS500. Essentially, the robot moves up/down its end-effector a certain distance (retract motion along a line and without changing its orientation), then moves it to a pose that is a certain distance over/under the desired pose (lateral motion), and finally moves it down/

up to the desired pose (approach linear motion without changing its orientation), as illustrated in Figure 16. The `MoveJump` command is highly optimized for fast pick and place motion and results in much faster cycle times than when using a simple sequence of `MoveLin` - `MovePose` - `MoveLin` commands.

Arguments

- *x*, *y*, *z*: the coordinates of the origin of the TRF with respect to the WRF, in mm;
- *y*: the orientation of the TRF about its z-axis, with respect to the WRF, in degrees.

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

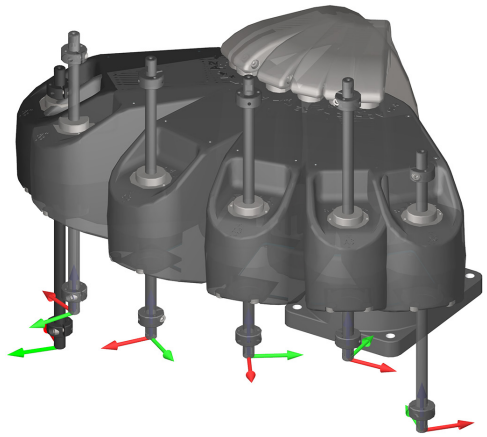


Figure 16: End-effector motion when using the `MoveJump` command

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

The parameters defining the trajectory of the end-effector in a `MoveJump` motion are set by the commands `SetMoveJumpHeight` and `SetMoveJumpApproachVel` (see Figure 18).

2.1.6 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 17. 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.

In the case of the Meca500, 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.

In the case of the MCS500, it is physically impossible to follow a linear path with this command, while changing the configuration (i.e., crossing a singularity).

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

Arguments

- *x*, *y*, *z*: the coordinates of the origin of the TRF with respect to the WRF, in mm;
- *α*, *β*, *γ*: Euler angles representing the orientation of the TRF with respect to the WRF, in degrees. In the case of the MCS500, *α* and *β* must be omitted or zero.

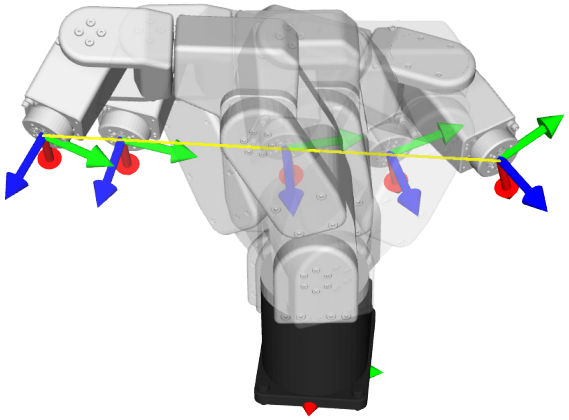


Figure 17: The TCP path when using the `MoveLin` command

The desired Cartesian linear and angular velocity of the TRF with respect to the WRF are specified by the commands `SetCartLinVel` and `SetCartAngVel`, respectively, but the joint velocities are limited by the command `SetJointVelLimit`. There is no guarantee that desired linear and angular velocities will be attained, but they will not be exceeded.

2.1.7 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*: Euler angles, in degrees. In the case of the MCS500, *α* and *β* must be omitted or zero.

2.1.8 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.

Arguments

- *x*, *y*, *z*: the position coordinates, in mm;
- *α*, *β*, *γ*: Euler angles, in degrees. In the case of the MCS500, *α* and *β* must be omitted or zero.

2.1.9 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, under the joint velocity constraint set by the command `SetJointVelLimit`.

Arguments

- $\dot{x}$, $\dot{y}$, $\dot{z}$: the components of the linear velocity of the TCP with respect to the TRF, in mm/s;
- ω_x, ω_y, ω_z: the components of the angular velocity of the TRF with respect to the TRF, in °/s. In the case of the MCS500, ω_x and ω_y must be omitted or zero.

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.

2.1.10 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, under the joint velocity constraint set by the command `SetJointVelLimit`.

Arguments

- $\dot{x}$, $\dot{y}$, $\dot{z}$: the components of the linear velocity of the TCP with respect to the WRF, in mm/s;
- ω_x, ω_y, ω_z: the components of the angular velocity of the TRF with respect to the WRF, in °/s. In the case of the MCS500, ω_x and ω_y must be omitted or zero.

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.

2.1.11 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, and move as fast as possible, but subject to the limits set by the commands `SetJointVel` and `SetJointVelLimit`. 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 15.

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: Euler angles for the orientation of the TRF with respect to the WRF, in degrees. In the case of the MCS500, α and β must be omitted or zero.

2.1.12 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 boundary configurations. For example, if you execute `SetAutoConf(0)` while the Meca500 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`.

2.1.13 SetAutoConfTurn(*e*)

This command enables/disables the automatic turn selection for the last joint of the robot (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, the last joint will always take the shortest path, and rotate no more than 180°. In the case of a `MoveLin*` command, however, enabling the automatic turn selection simply allows the change of turn configuration along the linear move.

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`.

2.1.14 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 position-mode joint-space commands `MoveJoints`, `MoveJointsRel`, `MovePose` and `MoveJump`, or between consecutive movements with the position-mode 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.

A blending of 100% corresponds to a blending that occurs 100% of the duration of the acceleration period, controlled by `SetJointAcc` and `SetCartAcc`.

2.1.15 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 14). 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.

2.1.16 SetCartAngVel(*ω*)

This command sets the desired and maximum angular velocity of the robot TRF with respect to its WRF. It only affects the movements generated by the `MoveLin`, `MoveLinRelTrf` and `MoveLinRelWrf` commands.

Arguments

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

Default values

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

The actual angular velocity may be lower (but never higher) than requested in some parts or the entirety of the linear path, in order to keep joint velocities within the limits set by the command `SetJointVelLimit` and in order to satisfy the limit set by the command `SetCartLinVel`.

2.1.17 SetCartLinVel(*v*)

This command sets the desired and maximum linear velocity of the robot TRF with respect to its WRF. It only affects the movements generated by the `MoveLin`, `MoveLinRelTrf` and `MoveLinRelWrf` commands.

Arguments

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

Default values

The default TCP velocity is 150 mm/s.

The actual TCP velocity may be lower (but never higher) than requested in some parts or the entirety of the linear path, in order to keep joint velocities within the limits set by the command `SetJointVelLimit` and in order to satisfy the limit set by the command `SetCartAngVel`.

2.1.18 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 not at all, if these messages were not enabled).

Arguments

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

Responses

[3030][*n*]

2.1.19 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 (the only argument accepted by the MCS500).
- 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 or the `SetAutoConf(0)` command. The latter sets the desired posture configuration to the one of the current robot posture.

2.1.20 SetConfTurn(c_t)

This command sets the desired turn configuration for the last joint, c_t, to be observed in the `MovePose` and `MoveLin*` commands (see Section 1.2.1 and Section 1.2.2). When c_t is set, a `MovePose` command is executed only if the final robot position can be in the desired turn configuration. In contrast, when a c_t is set, a `MoveLin*` command will 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 the last joint to rotate more than ±180°. For example, if using the MEGP 25E gripper and the Meca500, limit joint 6 to ±180° using the `SetJointLimits` command and then use either `SetAutoConfTurn(1)` or `SetConfTurn(0)`. If using a cable-less end-effector, then the automatic turn configuration should never be disabled. However, remember to always bring joint 6 of the Meca500 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, in the case of the Meca500, and between −10 and 10, in the case of the MCS500.

The turn configuration parameter defines the desired range for joint 6 of the Meca500, according to the inequality $-180° + c_t360° < \theta_6 \leq 180° + c_t360$, or for joint 4 of the MCS500, according to the inequality $-180° + c_t360° < \theta_4 \leq 180° + c_t360°$.

Default values

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

2.1.21 SetJointAcc(p)

This command limits the acceleration of the joints during movements resulting from joint- space commands (see Figure 14). Note that this command makes the robot stop, even if blending is enabled.

Arguments

- p: percentage of maximum acceleration of the joints, from 0.001 to 100 (MCS500) or 150 (Meca500).

Default values

The default joint acceleration limit is 100%.

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

2.1.22 SetJointVel(*p*)

This command specifies the maximal velocities of the robot joints during movements generated by the `MovePose`, `MoveJoints`, and `MoveJointsRel` commands.

Arguments

- *p*: percentage of R3 top rated joint velocities, ranging from 0.001 to 100, for Meca500 R3, and to 150, for R4, in the case of the Meca500, and percentage of the top raged joint velocities, ranging from 0.001 to 100, in the case of the MCS500.

Default values

By default, $p = 25$.

Note that the value of *p* is overridden by the argument of the command `SetJointVelLimit(`p_o`)` if $p_o < p$.

It is not possible to limit the velocity of only one joint. With `SetJointVel` and `SetJointVelLimit,` the maximum velocities of all joints are limited proportionally. In the Meca500 R3 or R4, the maximum velocity of each joint will be reduced to a percentage *p* of its top rated velocity, i.e.,

- 150°/s for joints 1 and 2;
- 180°/s for joint 3;
- 300°/s for joints 4 and 5;
- 500°/s for joint 6.

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

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

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

In the case of the MCS500, each maximum joint velocity is proportionally decreased by the same percentage *p* with respect to the corresponding maximum rated velocity.

2.1.23 SetJointVelLimit(p_o)

In revision 4 of the Meca500, we have introduced the possibility to go beyond the maximum rated joint velocities in any type of movement. For compatibility reasons, we introduced this new command.

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

Arguments

- p_o: percentage of top rated joint velocities, ranging from 0.001 to 100, for Meca500 R3 and MCS500, and to 150, for Meca500 R4.

Default values

By default, p_o = 100.

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

In future firmware releases, joints 3, 4, 5, and 6 in Meca500 R4 may be able to rotate faster, up to 150% of their top rated joint velocities. Thus, in future, for p_o > 100, R4 motions may be faster.

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

This command is intended for reducing the speed during the initial and final moments of the `MoveJump` motion (see Figure 18).

Arguments

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

Default values

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

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

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

This command prescribes the exact distances the end-effector must move up or down, with a pure vertical translational motion, during the vertical portions of the `MoveJump` movement. It also prescribes the minimum and maximum allowed heights for the lateral motion (see Figure 18).

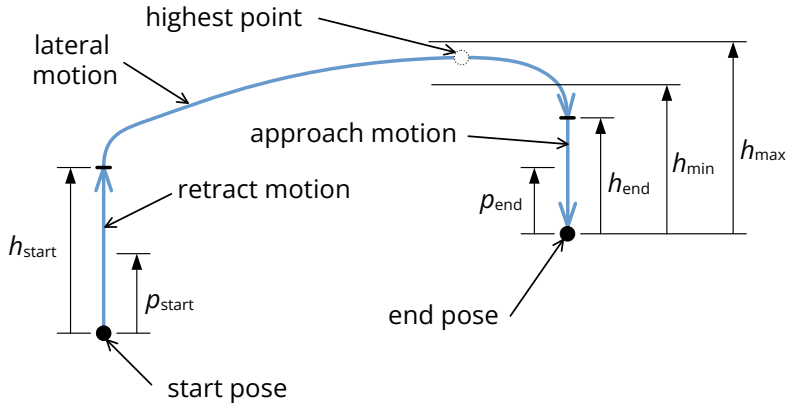


Figure 18: Settings for the `MoveJump` motion (projection on a vertical plane, the actual path is not in one plane)

Arguments

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

The direction the heights (positive or negative) is with respect to the z-axis of the BRF.

Default values

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

2.1.26 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, ..., ranging from 0.001 to 100. In the case of the MCS500, there are only four arguments, and the third argument refers to maximum allowable force.

Default values

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

2.1.27 SetTorqueLimitsCfg(*l*,*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

- *l*: integer defining the torque limit event severity as
  - 0, no action;
  - 1, torque status event only (message [3028]);
  - 2, pause motion and torque status event (message [3028]);
  - 4, torque status event (message [3028]) and torque limit error (message [3029]).
- *m*: integer defining the detection mode 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.

2.1.28 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*, *z*: the coordinates of the origin of the TRF with respect to the FRF, in mm;
- *α*, *β*, *γ*: Euler angles representing the orientation of the TRF with respect to the FRF, in degrees. In the case of the MCS500, *α* and *β* must be omitted or zero.

Default values

By default, the TRF coincides with the FRF.

2.1.29 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 to 1.

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.

2.1.30 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*: Euler angles representing the orientation of the WRF with respect to the BRF, in degrees. In the case of the MCS500, *α* and *β* must be omitted or zero.

Default values

By default, the WRF coincides with the BRF.

2.2. Robot control commands

Contrary to motion commands, robot control and request commands are executed immediately, i.e., are instantaneous, and all return a specific response or the response "[2085][Command successful: '...'.]". The commands described in this section to control the status of the robot (e.g., activate and home the robot) and to configure the robot.

2.2.1 ActivateRobot(e)

This command activates all motors (as well as the EOAT connected to the tool I/O port, in the case of the Meca500) and disables the brakes of the joints.

Arguments

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

Responses

[2000][Motors activated.]

2.2.2 ActivateSim

Our robots support a simulation mode in which all of the robot's hardware including our EOAT are simulated and nothing moves. This mode allows you to test programs with the robot's hardware (i.e., hardware-in-the-loop simulation), without the risk of damaging the robot or its surroundings. Simulation mode can be activated and deactivated with the `ActivateSim` and `DeactivateSim` commands (these commands 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 ConnectionWatchdog(*t*)

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

Arguments

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

Responses

[2177][1] or [2177][0]

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

Note: Even if you do not use the command `ConnectionWatchdog`, the robot will supervise the TCP connection but only when the robot is moving, and as soon as it detects a connection loss, it will stop moving and return the message [3086][1]. However, the delay between the connection loss and the detection may vary from a few milliseconds to several seconds, depending on your network activity.

2.2.4 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.]

2.2.5 DeactivateRobot

This disables all motors (as well as the EOAT connected to the tool I/O port, in the case of the Meca500) and engages the brakes on the robot joints. You must deactivate the robot in order to use certain commands (e.g., `SetJointLimits`, `SetNetworkOptions`). If you deactivate a Meca500 that was already homed, and then reactivate it, you do not need to home it again, unless it has an MEGP 25* gripper installed. In the latter case, however, the homing process is performed only for the gripper, and so the robot does not move. You also need to home the robot again if you reactivated it with `ActivateRobot(1)`. The MCS500 does not need homing.

Even if the MCS500 robot is deactivated, the optional vacuum and I/O module can still function with the appropriate `*_Immediate` command.

Responses

[2004][Motors deactivated.]

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

2.2.6 DeactivateSim

This command deactivate the simulation mode (can only be executed when the robot is deactivated).

Responses

[2046][The simulation mode is disabled.]

2.2.7 EnableEtherNetIp(*e*)

This command enables or disables EtherNet/IP slave stack.

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).

2.2.8 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).

2.2.9 GetFwVersion

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

Responses

[2081][vx.x.x]

2.2.10 GetModelJointLimits(*n*)

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

Arguments

- *n*: joint number, an integer.

Responses

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

- n: joint number, an integer number;
- $q_{n,\min}$: lower joint limit, in degrees or in mm;
- $q_{n,\max}$: upper joint limit, in degrees or in mm.

2.2.11 GetProductType

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

Responses

[2084][Meca500] or [2084][MCS500]

2.2.12 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.

2.2.13 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]

2.2.14 Home

This command is available only on the Meca500. It starts the robot and MEGP 25* gripper homing process (Section 1.3.1). While homing, it is critical to remove any obstacles that could hinder the robot and gripper movements.

Responses

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

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

2.2.15 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: '...'.]

2.2.16 LogUserCommands(e_1,e_2)

This command enables/disables the logging of commands received by the robot, as well as the responses send by the robot.

Arguments

- e_1: enable (1) or disable (0) logging of received commands and sent responses;
- e_2: enable (1) or disable (0) logging of compilation and execution of motion commands.

Responses

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

2.2.17 PauseMotion

This command stops the robot movement. It is executed as soon as received (within 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 path. If you want to know where exactly did the robot stop, you can use the `GetRtCartPos` or `GetRtJointPos` commands.

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.]

2.2.18 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.]

2.2.19 ResumeMotion

This command resumes the robot movement, if it was previously paused (1) with the command `PauseMotion` or (2) with the P-Stop 2 (MCS500) or SWStop (Meca500 R4) external signal, which is no longer present. 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.]

2.2.20 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]

2.2.21 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.

2.2.22 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.]

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

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 joint *n* to its factory values.

Arguments

- *n*: joint number, an integer;
- $q_{n,\min}$: lower joint limit, in degrees or in mm;
- $q_{n,\max}$: upper joint limit, in degrees or in mm.

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.

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 2.8.4 for more details).

Arguments

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

Default values

By default, the monitoring time interval is 0.015 s.

Responses

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

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.

Responses

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

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 Meca500'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 SetPStop2Cfg(*l*)

This command is used to set the behavior of the robot when the P-Stop 2 signal in the MCS500 is activated during automatic mode or when the SWStop signal in the Meca500 R4 is activated. The command can only be executed when the robot is deactivated. This setting is persistent and remains even after power-cycling the robot.

Arguments

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

Default values

By default, the severity level for this setting is 3.

Responses

[2178][PStop2 configuration set successfully]

2.2.29 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 2.8.4), 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;
- 2343 or VacuumPressure, for the response of the `GetRtVacuumPressure` 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. Further details about the monitoring port are presented in Section 2.8.4.

2.2.30 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 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 for the Meca500 and mcs500 for the MCS500.

Responses

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

2.2.31 SetRecoveryMode(*e*)

As discussed in Section 1.3.1, homing the Meca500 when it is too close to an obstacle may lead to a collision. Moving the robot (Meca500 or MCS500) 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 MEGP 25* grippers or the MPM500 pneumatic module connected to the Meca500, 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.

In the case of the Meca500, 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 MEGP 25* 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 Meca500 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]

2.2.32 SetRtc(*t*)

Since our robots do not have batteries, when powered on, their 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).

Responses

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

2.2.33 SetTimeScaling(*p*)

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

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

Arguments

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

Default values

By default, *p* = 100.

Responses

[2015] [*p*]

2.2.34 StartProgram(*s*)

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 *s* only once.

Alternately, pressing the Start/Stop button on the robot's base of the Meca500 will start program named "1", if such a program exists, and execute it the number of times defined by the `SetOfflineProgramLoop` command. (There is no such button on the MCS500.)

Arguments

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

Responses

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

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

2.2.35 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.

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 request commands (`Get*`). If the robot receives a change of state command (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 Meca500's base.

Arguments

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

Responses

[2060][Start saving program.]

2.2.36 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.]

Using the MecaPortal to save, open, and edit programs is much more intuitive than using the commands `StartSaving` and `StopSaving`.

2.2.37 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 $2^{32} - 1$.

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.

2.2.38 SwitchToEtherCat

This command will enable EtherCAT and disable the other three protocols (EtherCAT is an exclusive protocol that cannot be used at the same time as other Ethernet-based protocols, 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 (see the robot's user manual for the procedure).

2.2.39 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 from the MecaPortal.

Arguments

- *n*: duration in seconds.

Responses

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

2.2.40 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 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 added to a motion queue, 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.

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 6), 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] or [2029][c_e]

- 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][*,*,*]. In the case of the MCS500, only c_e is reported.

2.3.9 GetConfTurn

This command returns the desired turn configuration (see Figure 6), i.e., 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 or an asterisk†.

† if automatic turn configuration selection is enabled, the value returned is *.

2.3.10 GetJointAcc

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

Responses

[2153][p]

- p: percentage of maximum joint accelerations.

2.3.11 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.

Responses

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

- n: joint number, an integer;
- $q_{n,\min}$: lower joint limit, in degrees or in mm;
- $q_{n,\max}$: upper joint limit, in degrees or in mm.

2.3.12 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.

2.3.13 GetJointVel

This command returns the desired joint velocities reduction factor, set with the `SetJointVel` command.

Responses

[2152][*p*]

– *p*: percentage of maximum joint velocities.

2.3.14 GetJointVelLimit

This command returns the desired joint velocities override, set with the `SetJointVelLimit` command.

Responses

[2169][*p*]

– *p*: percentage of maximum joint velocities override.

2.3.15 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.

2.3.16 GetMoveJumpApproachVel

This command returns the desired initial and final velocity parameters for the `MoveJump` command, set with the `SetMoveJumpApproachVel` command.

Responses

[2175][v_{start}, p_{start}, v_{end}, p_{end}]

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

2.3.17 GetMoveJumpHeight

This command returns the different height parameters for the `MoveJump` command, set with the `SetMoveJumpHeight` command.

Responses

[2174][h_{start}, h_{end}, h_{min}, h_{max}]

- h_{start}: height from start pose to reach using a pure vertical translation, before the lateral motion begins, in mm;
- h_{end}: height from end pose from where to begin the final pure vertical translation, in mm;
- h_{min}: minimum height to reach while performing the lateral motion, with respect to the highest (if h_{start} and h_{end} are positive) or lowest (if h_{start} and h_{end} are negative) between the start and end poses, in mm;
- h_{max}: maximum height allowed to reach while performing the lateral motion, with respect to the highest (if h_{start} and h_{end} are positive) or lowest (if h_{start} and h_{end} are negative) between the start and end poses, in mm.

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 GetPStop2Cfg(*l*)

This command returns the severity level set with the command `SetPStop2Cfg`.

Responses

[2178][*l*]

- 2, for PauseMotion;
- 3, for ClearMotion.

2.3.20 GetRealTimeMonitoring

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

Responses

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

2.3.21 GetTimeScaling

This command returns the time scaling percentage set by the `SetTimeScaling` command.

Responses

[2015][*p*]

- current time scaling percentage.

2.3.22 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][*l*, *m*]

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

2.3.23 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*] or [2014][*x*, *y*, *z*, *γ*]

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

2.3.24 GetVelTimeout

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

Responses

[2151][*t*]

- *t*: timeout, in seconds.

2.3.25 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*] or [2014][*x*, *y*, *z*, *γ*]

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

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 of the Meca500.
- `GetRtJointPos`: returns the current joint set, as measured by the joint encoders.
- `GetRtCartPos`: returns the current TRF pose as calculated from the joint encoder values.
- `GetRtJointVel`: returns the current joint velocities as calculated from the joint encoder values.
- `GetRtCartVel`: returns the current Cartesian velocity as calculated from the joint encoder values.
- `GetRtConf`: returns the current posture configuration as calculated from the joint encoder values.
- `GetRtConfTurn`: returns the current turn configuration as calculated from the joint encoder values.

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 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 as calculated by the trajectory planner.

For example, if the Meca500 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. The revolute joints oscillate ±0.001° around the desired joint angles. Thus, if you execute `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.

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 GetOperationMode

This command returns the operation mode selected by the key switch on the power supply of MCS500.

Responses

[2076][*m*]

- *m*: 0 for locked mode, 1 for automatic mode, and 2 for manual mode.

2.4.3 GetRtAccelerometer(*n*)

An accelerometer is embedded in link 5 of the Meca500 (i.e., the body with the I/O port). It reports the acceleration of link 5 with respect to the WRF in the range ±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).

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

2.4.4 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).

2.4.5 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] or [2211][t, x, y, z, γ]

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

2.4.6 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] or [2214][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]

- t: timestamp in microseconds;
- $\dot{x}$, $\dot{y}$, $\dot{z}$: components of the linear velocity vector of the TCP with respect to the WRF, in mm/s.
- ω_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 $(\dot{x}^2 + \dot{y}^2 + \dot{z}^2)^{½}$, and the current angular speed of the end-effector with respect to the WRF is $(\omega_x^2 + \omega_y^2 + \omega_z^2)^{½}$. Note that the components of the angular velocity vector are not the time derivatives of the Euler angles.

2.4.7 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] or [2218][t, c_e]

- 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. In the case of the MCS500, only c_e is reported.

2.4.8 GetRtConfTurn

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

Response

[2219][t, c_t]

- t: timestamp in microseconds;
- c_t: turn configuration parameter, an integer number.

2.4.9 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] or [2210][t, θ_1, θ_2, d_3, θ_4]

- t: timestamp in microseconds;
- θ_i: the angle of joint i, in degrees;
- d_3: the position of joint 3, in mm (in the case of the MCS500).

2.4.10 GetRtJointTorq

This command returns the current joint torques, or more specifically, the current motor torques.

Responses

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

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

2.4.11 GetRtJointVel

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

Responses

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

- t: timestamp in microseconds;
- $\dot{\theta}_i$: the rate of change of joint i, in °/s (i = 1, 2, ...);
- $\dot{d}_3$: the rate of change of joint 3, in mm/s (in the case of the MCS500).

2.4.12 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.

Responses

[2201][t, x, y, z, α, β, γ] or [2201][t, x, y, z, γ]

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

N.B. The deprecated command `GetPose`, which is still supported, returns the same date, except for the timestamp. The message ID is also different: 2027.

2.4.13 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] or [2204][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]

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

2.4.14 GetRtTargetConf

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

Responses

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

- 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. In the case of the MCS500, only c_e is reported.

2.4.15 GetRtTargetConfTurn

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

Responses

[2209][t, c_t]

- t: timestamp in microseconds;
- c_t: turn configuration parameter, an integer number.

2.4.16 GetRtTargetJointPos

This command returns the current target joint set.

Responses

[2200][t, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6] or [2200][t, θ_1, θ_2, d_3, θ_4]

- t: timestamp in microseconds;
- θ_i: the angle of joint i, in degrees (i = 1, 2, ...);
- d_3: the position of joint 3, in mm (in the case of the MCS500).

N.B. The deprecated command `GetJoints`, which is still supported, returns the same date, except for the timestamp. The message ID is also different: 2026.

2.4.17 GetRtTargetJointTorq

This command returns the current target joint (actually motor) torques.

Responses

[2203][t, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6] or [2213][t, τ_1, τ_2, τ_3, τ_4]

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

2.4.18 GetRtTargetJointVel

This command returns the current target joint velocities.

Responses

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

- t: timestamp in microseconds;
- $\dot{\theta}_i$: the rate of change of joint i, in °/s (i = 1, 2, ...);
- $\dot{d}_3$: the rate of change of joint 3, in mm/s (in the case of the MCS500).

2.4.19 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] or [2229][t, x, y, z, γ]

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

2.4.20 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] or [2228][t, x, y, z, γ]

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

2.4.21 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.

2.4.22 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. Work zone supervision and collision prevention commands

In addition to being able to further constrain the limits of the robot joints with the commands `SetJointLimits` and `SetJointLimitsCfg`, you can also set a work zone with the command `SetWorkZoneLimits`, which defines a bounding box in the base reference frame (BRF). Similarly, you can also define a "tool sphere" in the flange reference frame (FRF) with the command `SetToolSphere`. See Figure 19 and Figure 20.

You can then specify with the command `SetWorkZoneCfg` that the robot supervises whether the robot links, its tool sphere and optional tooling, or its FCP (flange center point) remain inside the work zone. In addition, you can specify with the command `SetCollisionCfg` that the robot prevents collisions between its links and its tool sphere and optional tooling. In each of the two cases, you can configure that the robot simply generates a warning (i.e., only supervises) or creates a motion error (i.e., prevents a work zone breach or a collision). However, we expect that you would rather prevent collisions, hence the use of the term "collision prevention", whereas you might not necessarily want to prevent a work zone breach, but only detect it, hence the use of the term "work zone supervision".

Note that each robot link is represented by a very accurate STL model. In the near future, it will also be possible to import a CAD model (STL format) for the tool, as well as several CAD models for obstacles in the base reference frame. Therefore, the commands in this section will evolve and it will be much easier to use the MecaPortal to define all these settings.

Figure 19 and Figure 20 illustrate the objects currently supervised. In each robot, the base STL model also includes part of the cables coming from the base (not shown).

The work zone supervision and collision prevention feature is not safety rated. Furthermore, if the robot handles heavy or large objects at large speeds and at high blending, it is possible that the work zone breach or collision detection occurs a few milliseconds too late.

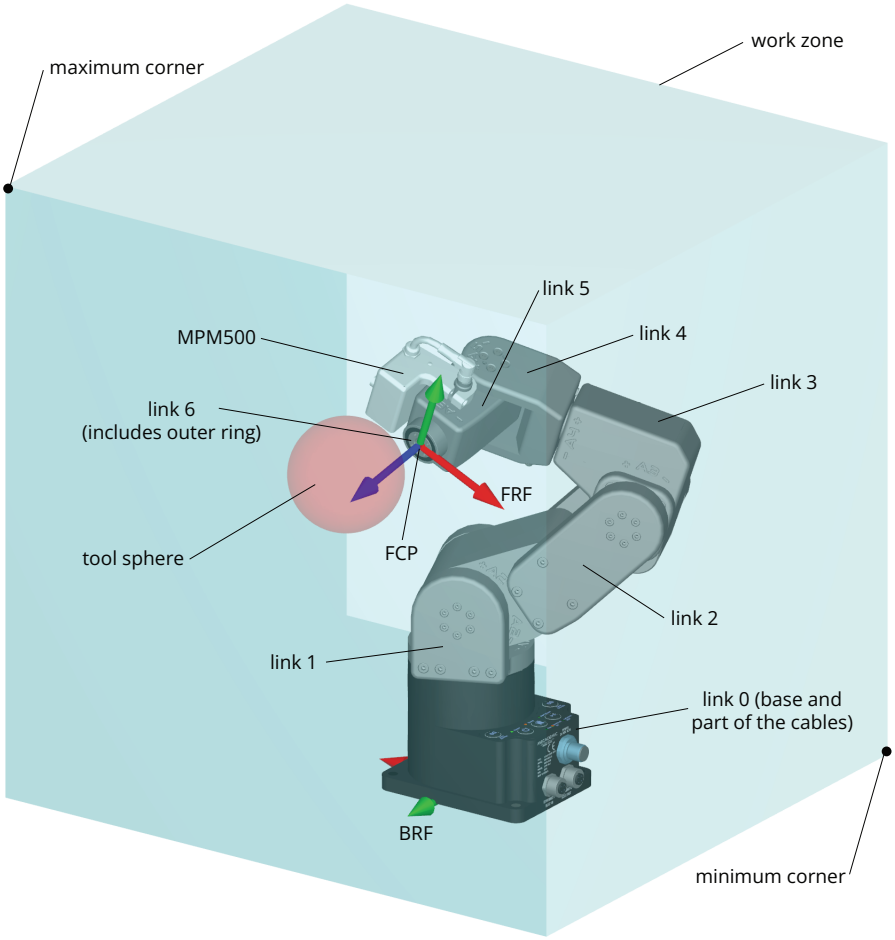


Figure 19: Objects tested in the work zone supervision and collision prevention feature, in the case of the Meca500

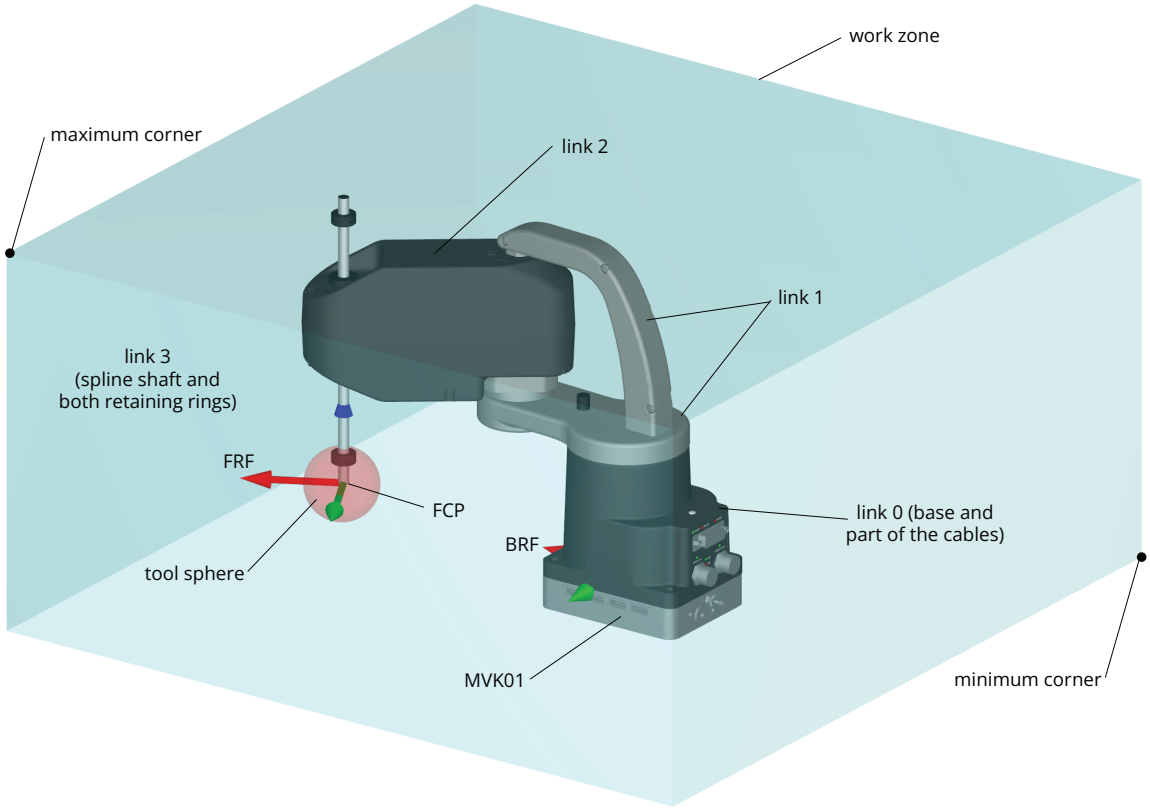


Figure 20: Objects tested in the work zone supervision and collision prevention feature, in the case of the MCS500

2.5.1 GetCollisionCfg

This command returns the severity level set with the `SetCollisionCfg` command.

Responses

[2181][*l*]

– *l*: severity level.

2.5.2 GetCollisionStatus

This command returns the current collision status (refer to Figure 19 and Figure 20).

Responses

[2182][v,g_1,$o_{id,1}$,g_2,$o_{id,2}$]

– v: collision state (1 or 0†),
– g_1, g_2: group identifier of first and second colliding objects, 0 for links, 1 for FCP, and 2 for tool,
– $o_{id,1}$, $o_{id,2}$: object ID of first and second in collision, depending on group identifier, as follows
0 for robot base, 1 for link 1, 2 for link 2, etc., if g = 0,
0 for FCP (flange center point), if g = 1,
0 for tool sphere, 10,000 for MPM500, 20,000 for MVK01, if g = 2.

†If $v = 0$, $g_1 = g_2 = o_{id,1} = o_{id,2} = 0$.

2.5.3 GetToolSphere

This command returns the current definition of the tool sphere, set with the `SetToolSphere` command.

Responses

[2167][*x*, *y*, *z*, *r*]

- *x*, *y*, *z*: the coordinates of the center of the tool sphere with respect to the FRF, in mm;
- *r*: the radius of the tool sphere, in mm.

2.5.4 GetWorkZoneLimits

This command returns the current definition of the bounding box with respect to the BRF, set with the `SetWorkZoneLimits` command.

Responses

[2165][x_{min}, y_{min}, z_{min},x_{max}, y_{max}, z_{max}]

- x_{min}, y_{min}, z_{min}: the coordinates of the minimum corner of the cuboid in the BRF, in mm;
- x_{max}, y_{max}, z_{max}: the coordinates of the maximum corner of the cuboid in the BRF, in mm.

2.5.5 GetWorkZoneLimitsCfg

This command returns the current work zone limits configuration, set with the `SetWorkZoneLimitsCfg` command.

Responses

[2163][*l*,*m*]

- *l*: event severity;
- *m*: supervision mode.

2.5.6 GetWorkZoneStatus

This command returns the current work zone violation status (refer to Figure 19 and Figure 20).

Responses

[2183][*v*,*g*,o_{id}]

- *v*: work zone violation state (1 or 0),
- *g*: group identifier of object in breach, 0 for links, 1 for FCP, and 2 for tool,
- o_{id}: object ID, depending on group identifier number, as follows

  0 for robot base, 1 for link 1, 2 for link 2, etc., if $g = 0$,

  0 for FCP, if $g = 1$,

  0 for tool sphere, 10,000 for MPM500, 20,000 for MVK01, if $g = 2$.

†If $v = 0$, $g = o_{id} = 0$.

2.5.7 SetCollisionCfg(*l*)

The `SetCollisionCfg` command can only be executed while the robot is powered but not activated. The command specifies the event severity for the collision supervision (robot links, tool sphere and MPM500 module).

This setting is persistent and remains even after power-cycling the robot.

Arguments

- *l*: integer defining the collision detection event severity as
  - 0, silent (i.e., collisions are verified but no action is taken, other than to log them internally);
  - 1, generate a warning (message [2182]) every time a new imminent collision is detected;
  - 4, generate a warning (message [2182]) and a motion error (message [3041]) every time a new imminent collision is detected.

Default values

By default, *l* =4 (for both the Meca500 and the MCS500).

Responses

[2180] [Collision configuration set successfully.]

2.5.8 SetToolSphere(*x*,*y*,*z*,*r*)

The `SetToolSphere` command can only be executed while the robot is powered but not activated. It defines a sphere fixed in the flange reference frame (FRF). Interferences between that sphere and the robot links as well as the outside of a bounding box set with the `SetWorkZoneCfg` command can then be supervised, as defined by the `SetWorkZoneCfg` and `SetCollisionCfg` commands.

This setting is persistent and remains even after power-cycling the robot.

Arguments

- *x*, *y*, *z*: the coordinates of the center of the tool sphere in the FRF, in mm;
- *r*: the radius of the tool sphere, in mm.

Default values

By default, $x = y = z = 0$ and $r = 0$. Note that setting all four arguments to zero is equivalent to disabling the tool sphere. However, if $r = 0$, but one of the coordinates is not zero, the tool sphere will be a point.

Responses

[2168] [Tool sphere set successfully.]

N.B. In the case of the Meca500, interferences between the tool sphere and the robot flange, and the tool sphere and link 5 (the one with the I/O port and the "−A6+" engraving) are not tested. Note that, if you set your tool sphere too big, e.g. with `SetToolSphere(0,0,0,60)`, it will always interfere with link 4, i.e., the yoke one with the "−A5+" engraving. In the case of the MCS500, interferences between the tool sphere and the spline shaft are not tested.

2.5.9 SetWorkZoneCfg(*l*,*m*)

The `SetWorkZoneCfg` command can only be executed while the robot is powered but not activated. The command specifies the "event severity" for the work zone limits supervision and the robot parts that need to be verified.

This setting is persistent and remains even after power-cycling the robot.

Arguments

- *l*: integer defining the work zone breach detection event severity as
  - 0, silent (i.e., work zone breach is verified but no action is taken, other than to log them internally);
  - 1, generate a warning (message [2183]) every time a new imminent work zone breach is detected;
  - 4, generate a warning (message [2183]) and a motion error (message [3049]) every time a new imminent work zone breach is detected.
- *m*: integer defining the work zone breach verification mode as
  - 1, verify whether the FCP (flange center point) is inside the work zone;
  - 2, verify whether the tool is completely inside the work zone (tool is the tool sphere defined with the `SetToolSphere` command, and the MPM500 module if detected on the Meca500);
  - 3, verify whether the tool AND all robot links are completely inside the work zone.

Default values

By default, *l* =4 and *m* = 1 (for both the Meca500 and the MCS500).

Responses

[2164] [Work zone configuration set successfully.]

2.5.10 SetWorkZoneLimits(x_{min},y_{min},z_{min},x_{max},y_{max},z_{max})

The `SetWorkZoneLimits` command can only be executed while the robot is powered but not activated. It defines a bounding box (a cuboid), the sides of which are parallel to the axes of the base reference frame (BRF). The arguments of the command are the coordinates of two diagonally opposite corners, referred to as "minimum" and "maximum" corners, such that each coordinate of the minimum corner is smaller that the corresponding coordinate of the maximum corner.

This setting is persistent and remains even after power-cycling the robot.

Arguments

- x_{min}, y_{min}, z_{min}: the coordinates of the minimum corner of the cuboid in the BRF, in mm;
- x_{max}, y_{max}, z_{max}: the coordinates of the maximum corner of the cuboid in the BRF, in mm.

Default values

By default, $x_{min} = y_{min} = z_{min} = –10{,}000$ and $x_{max} = y_{max} = z_{max} = 10{,}000$. To reset the arguments to their default values, deactivate the robot and send the command `SetWorkZoneLimits(0,0,0,0,0,0).`

Responses

[2166] [Workspace limits set successfully.]

2.6. External-tool commands for the Meca500

This section regroups all commands that are used to control or request data from the optional electric grippers (MEGP 25*) and pneumatic module (MPM500) for the Meca500. Refer to their respective user manuals too. Some of the commands in this section are queued, others are instantaneous (`Get*`, `SetExtToolSim`, and `*_Immediate`).

2.6.1 GetExtToolFwVersion

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

Responses

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

Note that in the case of the MCS500, the firmware of the optional vacuum and I/O module is automatically updated to the same version as that of the robot's firmware.

2.6.2 GetRtExtToolStatus

This instantaneous command returns the general status of the external tool connected to the I/O port of the Meca500, preceded with a timestamp. For additional status information, use the commands `GetRtGripperState` 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 Meca500 (0 for none, 10 for MEGP 25E gripper, 11 for MEGP 25LS gripper, 20 for MPM500 pneumatic module);
- *hs*: homing state (0 for homing not performed, 1 for homing performed);
- *es*: error state (0 for absence of error, 1 for presence of error);
- *oh*: overheat (0 if there is no overheat, 1 if the gripper is in overheat).

2.6.3 GetRtGripperForce

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

2.6.4 GetRtGripperPos

This instantaneous command returns the current fingers opening of the MEGP 25* grippers (see `MoveGripper`), preceded with a timestamp.

Responses

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

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

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.

2.6.5 GetRtGripperState

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

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).

2.6.6 GetRtValveState

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

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).

2.6.7 GripperOpen/GripperClose

These queued 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`.

The `GripperOpen` and `GripperClose` commands are queued and behave like a robot motion command, so they 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.

2.6.8 MoveGripper(*d*)

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

The `MoveGripper` 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 is queued and behaves like a motion command, so it 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.

2.6.9 SetExtToolSim(*e*)

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

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*]

2.6.10 SetGripperForce(*p*)

This queued 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%.

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

This queued 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)`.

2.6.12 SetGripperVel(*p*)

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

Arguments

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

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

Default values

By default, the finger velocity limit is 40%.

2.7. External-tool commands for the MCS500

This section regroups all commands that are used to control or request data from the optional vacuum & I/O module (MVK01) for the MCS500. Refer to the MVK01 user manual too. Some of the commands in this section are queued, others are instantaneous (`Get*`, `*_Immediate`, and `SetIoSim`).

2.7.1 GetIoSim(b_{id})

This instantaneous command returns the state of the simulation of the MVK01 vacuum and I/O module set by the command `SetIoSim` (or the default one).

Responses

[2056][b_{id},e]

- b_{id}: I/O bank ID, should be 1 (for MVK01);
- e: status of the simulation mode (1 if enabled, 0 if disabled).

2.7.2 GetVacuumPurgeDuration

This instantaneous command returns the duration of the air purge on the MVK01 vacuum and I/O module set by the command `SetVacuumPurgeDuration` (or the default one).

Responses

[2173][t_p]

- t_p: duration of air purge in seconds.

2.7.3 GetVacuumThreshold

This instantaneous command returns the current pressure thresholds (negative values) for the MVK01 vacuum and I/O module set by the command `SetVacuumThreshold` (or the default ones).

Responses

[2172][p_h, p_r]

- p_h: threshold pressure below which the robot considers that a part is held, in kPa;
- p_r: threshold pressure above which the robot considers that a part is no longer held, in kPa.

2.7.4 GetRtIoStatus(b_{id})

This instantaneous command returns the status of the MVK01 vacuum and I/O module.

Responses

[2330][*t*, b_{id}, *present*, *simMode*, *errorCode*]

- *t*: timestamp in microseconds;
- b_{id}: I/O bank ID, currently 1 (for MVK01);
- *present*: 1 if the MVK01 module has been detected, 0 if otherwise;
- *simMode*: state (1 for enabled, 0 for disabled) of the MVK01 simulation mode (see `SetIoSim`);
- *errorCode*: error code (0 if no error).

2.7.5 GetRtInputState(b_{id})

This instantaneous command returns the state of the eight digital inputs of the MVK01 module.

Responses

[2341][*t*, b_{id}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8]†

- *t*: timestamp in microseconds;
- b_{id}: I/O bank ID, currently 1 (for MVK01);
- p_i: state of input pin *i* (*i* = 1, 2, ..., 8), 1 for high and 0 for low.

†If an MVK01 module is not present, no input values will be included in the response and there will be no error.

2.7.6 GetRtOutputState(b_{id})

This instantaneous command returns the state of the eight digital outputs of the MVK01 module.

Responses

[2340][*t*, b_{id}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8]†

- *t*: timestamp in microseconds;
- b_{id}: I/O bank ID, currently 1 (for MVK01);
- p_i: state of output pin *i* (*i* = 1, 2, ..., 8), 1 for high and 0 for low.

†If an MVK01 module is not present, no output values will be included in the response and there will be no error.

2.7.7 GetRtVacuumPressure

This instantaneous command returns the current pressure in the vacuum chamber of MVK01 vacuum and I/O module.

Responses

[2343][*t*, *p*]

- *t*: timestamp in microseconds;
- *p*: pressure of the vacuum chamber, in kPa (usually non-positive, except during purging).

2.7.8 GetRtVacuumState

This instantaneous command returns the current state of the pneumatic part of the MVK01 vacuum and I/O module.

Responses

[2342][*t*, *v*, *h*, *p*]

- *t*: timestamp in microseconds;
- *v*: state of vacuum generation (1 if vacuum is being generated, 0 if not);
- *p*: state of air purge (1 if air is being purged in order to quickly release a part, 0 if not);
- *h*: state of holding part (1 if vacuum is being generated and a part is being hold, 0 if not).

2.7.9 SetIoSim(b_{id},*e*)

This instantaneous command, availably only for the MCS500, toggles the simulation mode for the MVK01 vacuum and I/O module. With the simulation enabled, you can use all the MVK01 commands (e.g., `VacuumGrip`, `SetOutputState`) and if the MVK01 is physically present, these commands will not have any physical impact (i.e., they will only be simulated).

Arguments

- b_{id}: I/O bank ID, must be 1 (for MVK01);
- *e*: state of simulation mode (1 to enable, 0 to disable).

Default values

By default, the simulation mode of the MVK01 is disabled.

Responses

[2056][b_{id}, *e*]

2.7.10 SetOutputState(b_{id},p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8)

This queued command is used to control the digital outputs of the MVK01 vacuum and I/O module.

Arguments

- b_{id}: I/O bank ID, currently 1 (for MVK01);
- p_i: state of output pin *i* (*i* = 1, 2, ..., 8), 1 to set, 0 to reset and −1 or * to keep unchanged.

2.7.11 SetOutputState_Immediate(b_{id},p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8)

This is the same command as `SetOutputState`, but it is instantaneous, rather than queued and can be executed even when the robot is deactivated.

Responses

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

2.7.12 SetVacuumPurgeDuration(t_p)

This queued command sets the duration of the air purge for ejecting a part when using the commands `VacuumRelease*`.

Arguments

- t_p: duration in seconds.

Default values

By default, t_p = 0.1.

2.7.13 SetVacuumPurgeDuration_Immediate(t_p)

This is the same command as `SetVacuumPurgeDuration`, but it is instantaneous, rather than queued and can be executed even when the robot is deactivated.

2.7.14 SetVacuumThreshold(p_h,p_r)

This queued command sets the thresholds for the vacuum sensor (which measures only negative pressure) in the MVK01 vacuum and I/O module that will be used for reporting whether a part is being hold or not.

Arguments

- p_h: when the negative pressure sensed is smaller than p_h, the robot reports that a part is being held. The value for this argument ranges from −100 kPa to −5 kPa.
- p_r: when the value of the negative pressure sensed is larger than p_r, the robot reports no part is being held. The value for this argument ranges from −95 kPa to 0 kPa. The value of p_r must be larger than the value of p_h by at least 5 KPa.

Default values

By default, p_h = −40 kPa, and p_r = −30 kPa. To reset to the default values, reactivate the robot or send the command `SetVacuumThreshold(0,0)`.

2.7.15 SetVacuumThreshold_Immediate(p_h,p_r)

This is the same command as `SetVacuumThreshold`, but it is instantaneous, rather than queued and can be executed even when the robot is deactivated.

2.7.16 SetValveState(v_1,v_2)

This queued command is used to control independently each of the two valves in the MPM500 pneumatic module (available only for the Meca500).

Arguments

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

Default values

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

Responses

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

Since the MPM500 is often used with pneumatic grippers, you can also use the command `GripperOpen` instead of `SetValveSate(1,0)`, and `GripperClose` instead of `SetValveSate(0,1)`. However, note that these commands do not have the same effect on blending (see Section 1.3.3).

2.7.17 VacuumGrip/VacuumRelease

These queued commands activate/deactivate the suction in the MVK01 vacuum and I/O module for the MCS500 SCARA robot.

2.7.18 VacuumGrip_Immediate/VacuumRelease_Immediate

These instantaneous commands activate/deactivate the suction in the MVK01 vacuum and I/O module for the MCS500 SCARA robot. Unlike their queued equivalents (`VacuumGrip` and `VaccumRelease`), they can be executed even when the robot is deactivated.

Responses

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

2.8. Responses and messages

Every Mecademic robot 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 robot 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.

2.8.1 Command error messages

When the robot encounters an error while executing a command, it goes into error mode. See Section 2.9.1 for details on how to manage these errors. Table 1 lists all command error messages.

Table 1: 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 (for example, because the SWStop is active).
[1014][Homing failed.]	Homing procedure failed. Try again.
[1016][Destination pose out of reach for any configuration. - Command: '...'] [1016][Destination pose out of reach for selected conf(...,...,... turn ...). - Command: '...'] [1016][The requested linear move is not possible due to a pose out of reach along the path. - Command: '...']	The pose requested in the `MoveLin, MoveLinRel*`, `MovePose` or `MoveJump` 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.

Table 1: Command error messages (continued)

COMMAND ERROR MESSAGES	
Message	**Explanation**
[1026][Deactivation needed to execute the command. - Command: '...']	The robot must be deactivated in order to execute this command.
[1027][Simulation mode can only be enabled/ disabled while the robot is deactivated.]	The robot must be deactivated in order to execute this command.
[1029][Offline program full. Maximum program size is 13,000 commands. Saving stopped.]	Memory full.
[1030][Already saving.]	The robot is already saving a program. Wait until finished to save another program.
[1031][Program saving aborted after receiving illegal command. - Command: '...']	The command cannot be executed because the robot is currently saving a program.
[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 module connected.
[1042[Ext tool sim must deactivated]	Switching external tool type is only possible when the robot is deactivated.
[1043][The specified IO bank is not present on this robot]	The argument for the I/O bank ID is different than 1.
[1044][There is no vacuum module present on this robot.]	No MVK01 vacuum and I/O module present or simulated, but a command such as `VacuumGrip` was sent.

2.8.2 Command responses

Motion commands do not generate any (non-error) response, other than the optional EOB and EOM messages (see Section 2.1) and the message eventually generated by the `SetCheckpoint` command. Table 3 presents a summary of all other commands and the possible non-error responses.

Table 2: Request commands and corresponding possible responses

COMMAND RESPONSES	
Response code	**Command**
[2000][Motors activated.]	`ActivateRobot`
[2002][Homing 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`
[2013][*x*, *y*, *z*, *α*, *β*, *γ*] or [2013][*x*, *y*, *z*, *γ*]	`GetWrf`
[2014][*x*, *y*, *z*, *α*, *β*, *γ*] or [2014][*x*, *y*, *z*, *γ*]	`GetTrf`

Table 2: Request commands and corresponding possible responses (continued)

COMMAND RESPONSES	
Response code	**Command**
[2015][*p*]	`SetTimeScaling`, `GetTimeScaling`
[2028][*e*]	`GetAutoConf`
[2029][c_s, c_e, c_w] or [2029][c_e]	`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`
[2049][Robot is in recovery mode] [2050][Robot is not in recovery mode]	`SetRecoveryMode`
[2051][Joint velocity/acceleration ... will be limited to ... due to recovery mode]	`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`
[2056][b_{id},*e*]	`GetIoSim`
[2056][b_{id},*e*]	`SetIoSim`
[2060][Start saving program.]	`StartSaving`
[2061][*n* commands saved.]	`StopSaving`
[2063][Offline program *n* started.]	`StartProgram`
[2064][Offline program looping is enabled.] [2065][Offline program looping is disabled.]	`StopSaving`
[2080][*n*]	`GetCmdPendingCount`
[2081][vx.x.x]	`GetFwVersion`
[2085][Command successful. ...]	Response to various instantaneous commands
[2088][vx.x.x]	`GetExtToolFwVersion`
[2083][robot's serial number]	`GetRobotSerial`
[2084][Meca500] or [2084][MCS500]	`GetProductType`
[2086][vx.x.x]	`GetExtToolFwVersion`

Table 2: Request commands and corresponding possible responses (continued)

COMMAND RESPONSES	
Response code	**Command**
[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`
[2113][q_1, q_2,q_3...]	`GetModelJointLimits`
[2116][t]	`GetMonitoringInterval`
[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] or [2161][p_1, p_2, p_3, p_4]	`GetTorqueLimits`
[2162][d_{closed}, d_{open}]	`GetGripperForce`
[2163][l, m]	`GetWorkZoneCfg`
[2164][Work zone limits configuration set successfully.]	`SetWorkZoneLimitsCfg`
[2165][$x_{\min}$, $y_{\min}$, $z_{\min}$,$x_{\max}$, $y_{\max}$, $z_{\max}$]	`GetWorkZoneLimits`
[2166] [Work zone limits set successfully.]	`SetWorkZoneLimits`
[2167][x, y, z, r]	`GetToolSphere`
[2168][Tool sphere set successfully.]	`SetToolSphere`
[2169][p]	`GetJointVelLimit`

Table 2: Request commands and corresponding possible responses (continued)

COMMAND RESPONSES	
Response code	**Command**
[2172][p_h, p_r]	`GetVacuumThreshold`
[2173][t_p]	`GetVacuumPurgeDuration`
[2174][h_{start}, h_{end}, h_{min}, h_{max}]	`GetMoveJumpHeight`
[2175][v_{start}, p_{start}, v_{end}, p_{end}]	`GetMoveJumpApproachVel`
[2176][*m*]	`GetOperationMode`
[2177][*l*]	`ConnectionWatchdog`
[2178][PStop2 configuration set successfully]	`SetPStop2Cfg`
[2179][*l*]	`GetPStop2Cfg`
[2181][*l*]	`GetCollisionCfg`
[2182][*v*, g_1, $o_{id,1}$, g_2, $o_{id,2}$]	`GetCollisionStatus`
[2183][*v*, *g*, o_{id}]	`GetWorkZoneStatus`
[2200][*t*, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6] or [2200][*t*, θ_1, θ_2, d_3, θ_4]	`GetRtTargetJointPos`
[2201][*t*, *x*, *y*, *z*, α, β, *γ*] or [2201][*t*, *x*, *y*, *z*, *γ*]	`GetRtTargetCartPos`
[2202][*t*, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$] or [2202][*t*, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{d}_3$, $\dot{\theta}_4$]	`GetRtTargetJointVel`
[2203][*t*, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6] or [2203][*t*, τ_1, τ_2, τ_3, τ_4]	`GetRtTargetJointTorq`
[2204][*t*, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z] or [2204][*t*, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]	`GetRtTargetCartVel`
[2208][*t*, c_s, c_e, c_w] or [2208][*t*, c_e]	`GetRtTargetConf`
[2209][*t*, c_t]	`GetRtTargetConfTurn`
[2210][*t*, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6] or [2210][*t*, θ_1, θ_2, d_3, θ_4]	`GetRtJointPos`
[2211][*t*, *x*, *y*, *z*, α, β, *γ*] or [2211][*t*, *x*, *y*, *z*, *γ*]	`GetRtCartPos`
[2212][*t*, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$] or [2212][*t*, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{d}_3$, $\dot{\theta}_4$]	`GetRtJointVel`
[2213][*t*, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6] or [2213][*t*, τ_1, τ_2, τ_3, τ_4]	`GetRtJointTorq`
[2214][*t*, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z] or [2214][*t*, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]	`GetRtCartVel`
[2218][*t*, c_s, c_e, c_w] or [2218][*t*, c_e]	`GetRtConf`
[2219][*t*, c_t]	`GetRtConfTurn`
[2220][*t*, *n*, a_x, a_y, a_z]	`GetRtAccelerometer`
[2227][*t*, n]	Checkpoint reached
[2228][*t*, *x*, *y*, *z*, α, β, *γ*] or [2228][*t*, *x*, *y*, *z*, *γ*]	`GetRtWrf`
[2229][*t*, *x*, *y*, *z*, α, β, *γ*] or [2229][*t*, *x*, *y*, *z*, *γ*]	`GetRtTrf`
[2300][*t*, *simType*, *phyType*, *hs*, *es*, *oh*]	`GetRtExtToolStatus`
[2310][*t*, v_1, v_1]	`GetRtValveState`
[2320][*t*, *hp*, *dr*, *gc*, *go*]	`GetRtGripperState`
[2321][*t*, *p*]	`GetRtGripperForce`
[2322][*t*, *p*]	`GetRtGripperPos`

Table 2: Request commands and corresponding possible responses (continued)

COMMAND RESPONSES	
Response code	**Command**
[2330][*t*, b_{id}, *present*, *simMode*, *errorCode*]	`GetRtIoStatus`
[2340][*t*, b_{id}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8]	`GetRtOutputState`
[2341][*t*, b_{id}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8]	`GetRtInputState`
[2342][*t*, *v*, *h*, *p*]	`GetRtVacuumState`
[2343][*t*, *p*]	`GetRtVacuumPressure`

2.8.3 Status messages

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

Table 3: Status messages and descriptions

STATUS MESSAGES	
Message	**Explanation**
[3000][Connected to ... x_x_x.x.x.]	Confirms connection to robot.
[3001][Another user is already connected, closing connection.]	Another user is already connected to the robot. The robot disconnects from the user immediately after sending this message.
[3002][A firmware upgrade is in progress (connection refused).]	The firmware of the robot is being updated.
[3003][Command has reached the maximum length.]	Too many characters before the NULL character. Possibly caused by a missing NULL character
[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 our technical support team if restarting the robot did not resolve the issue.
[3012][End of block.]	No motion command in queue and robot joints do not move.
[3013][End of offline program.]	The offline program has finished.
[3014][Problem with saved program, save a new program.]	There was a problem saving the program.
[3016][Ignoring command while in offline mode.]	A non-motion command was sent while executing a program and was ignored.
[3017][No offline program saved.]	There is no program in memory.
[3018][Loop ended. Restarting the program.]	The offline program is being restarted.

Table 3: Status messages and descriptions (continued)

STATUS MESSAGES	
Message	**Explanation**
[3020][Offline program ... is invalid]	There was a problem starting a particular program with `StartProgram`.
[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 our technical support team.
[3028][*s*]	A torque limit was exceeded.
[3029][Excessive torque error occurred]	Excessive motor torque was detected.
[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][2/1/0]	A P-Stop 2 is active (1), is no longer active but needs to be reset (2) or is already cleared (0).
[3035][TCP dump capture started for *x* seconds]	Sent to indicate that the requested TCP dump capture has started and confirms the maximum duration of *x* seconds.
[3036][TCP dump capture stopped]	Sent after a previously started TCP dump capture has finished.
[3037][Pneumatic module error]	A communication error with the pneumatic module was detected. Contact our technical support team.
[3039][External tool firmware must be updated.]	Activation has failed, because the robot has detected that the firmware of the EOAT is older than the firmware of the robot.
[3040][0/1]	Indicates when an imminent collision is detected (and would cause robot to stop motion depending on the chosen severity)
[3041][Robot error due to imminent collision.]	Sent when robot is in error due to imminent collision detected while severity is configured to generate an error.
[3042][Detected failure in previous firmware update. Please re-install the firmware again.]	An error was detected during the firmware update. Try to reinstall software.
[3043[Excessive communication errors with external tool.]	Too many communication errors were detected between the I/O port and the EOAT connected to that port. This may mean that the cable is damaged and needs to be replaced or that it is not screwed tightly enough on either side. There may also be a hardware problem with the I/O port.
[3044][Abnormal communication error with external port.]	Detected internal communication errors with the robot's I/O port. Please contact Mecademic support for further diagnostic.
[3046][Power-supply detected a non-resettable power error. Please check power connection then power-cycle the robot]	(MCS500 only) Try to power-cycle the robot.

Table 3: Status messages and descriptions (continued)

STATUS MESSAGES	
Message	**Explanation**
[3047][Robot failed to mount drive. Please try to power-cycle the robot. If the problem persists contact Mecademic support.]	Robot has unexpectedly booted in safe mode. Try to power-cycle the robot.
[3049][Robot error at work zone limit]	Sent when robot is in error due to imminent work zone breach while severity is configured to generate an error.
[3069][0/1/2]	Response to a change in the state of the P-Stop 1 safety stop signal (MCS500 only).
[3070][0/1/2]	Response to a change in the state of the E-Stop safety stop signal.
[3080][0/1/2]	Response to a change in the state of the operation mode safety stop signal state (MCS500 only).
[3081][0/1/2]	Response to a change in the state of the Enabling Device safety stop signal, when the robot is in manual mode (MCS500 only).
[3082][0/1/2]	Occurs when supply voltage fluctuation is detected (MCS500 only).
[3083][0/2]	Occurs after robot is rebooted, and after Reset button is pressed.
[3084][0/1]	Redundancy fault. Occurs if a safety signal mismatch is detected.
[3085][0/2]	Standstill fault. Occurs if robot moves while in pause mode (MCS500 only).
[3086][0/1/2]	Response to a change in the connection drop safety signal change (MCS500 only).

2.8.4 Monitoring port messages

Mecademic robots are 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`), as shown in Table 4. Status messages 3028, 3032, 3040, 3069–3086 are sent on the monitoring port too.

Table 4: Monitoring port messages

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)
[2015][*p*]	`GetTimeScaling`
[2026][θ_1, θ_2, θ_3, θ_4, θ_5, θ_6] or [2026][θ_1, θ_2, d_3, θ_4]	Joint set (same as `GetRtTargetJointPos`, except there is no timestamp and the message ID is different)
[2027][*x*, *y*, *z*, *α*, *β*, *γ*] or [2027][*x*, *y*, *z*, *γ*]	Pose of the TRF with respect to the WRF (same as `GetRtTargetCartPos`, except there is no timestamp and the message ID is different)

[2049][Recovery mode enabled]	The recovery mode becomes enabled.
[2050][Recovery mode disabled]	The recovery mode becomes disabled.
[2079][*ge*, *hs*, *hp*, *lr*, *es*, *oh*]	Response from the legacy `GetStatusGripper` command (ignore this message)
[2176][*m*]	`GetOperationMode`
[2182][*v*, g_1, $o_{id,1}$, g_2, $o_{id,2}$]	`GetCollisionStatus`
[2183][*v*, *g*, o_{id}]	`GetWorkZoneStatus`
[2200][*t*, θ_1, θ_2, θ_3, θ_4, θ_5, θ_6] or [2200][θ_1, θ_2, d_3, θ_4]	`GetRtTargetJointPos`
[2201][*t*, *x*, *y*, *z*, α, β, γ] or [2201][*t*, *x*, *y*, *z*, α, γ]	`GetRtTargetCartPos`
[2202][*t*, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$] or [2202][*t*, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{d}_3$, $\dot{\theta}_4$]	`GetRtTargetJointVel`
[2203][*t*, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6] or [2203][*t*, τ_1, τ_2, τ_3, τ_4]	`GetRtTargetJointTorq`
[2204][*t*, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z] or [2204][*t*, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]	`GetRtTargetCartVel`
[2208][*t*, c_s, c_e, c_w] or [2208][*t*, c_e]	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] or [2210][θ_1, θ_2, d_3, θ_4]	`GetRtJointPos`
[2211][*t*, *x*, *y*, *z*, α, β, γ] or [2211][*t*, *x*, *y*, *z*, γ]	`GetRtCartPos`
[2212][*t*, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$, $\dot{\theta}_4$, $\dot{\theta}_5$, $\dot{\theta}_6$] or [2212][*t*, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{d}_3$, $\dot{\theta}_4$]	`GetRtJointVel`
[2213][*t*, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6] or [2213][*t*, τ_1, τ_2, τ_3, τ_4]	`GetRtJointTorq`
[2214][*t*, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z] or [2214][*t*, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]	`GetRtCartVel`
[2218][*t*, c_s, c_e, c_w] or [2218][*t*, c_e]	`GetRtConf`
[2219][*t*, c_t]	`GetRtConfTurn`
[2220][*t*, *n*, a_x, a_y, a_z]	`GetRtAccelerometer`
[2228][*t*, *x*, *y*, *z*, α, β, γ] or [2228][*t*, *x*, *y*, *z*, γ]	Response from the `GetRtWrf` command (only when the data changes and at connection)

By default, these feedback messages are sent every 15 ms, except for the status-related events which are sent as soon as the state changes. 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]

Finally, when a client (PC, PLC, etc.) connects to a robot, the robot sends a series of messages regarding the initial state of the robot (2007, 2015, 2096, 2310, 2320, 2209, 2228, 2229, etc.).

2.9. Management of errors and safety stops

2.9.1 Errors detected by the robot

The robot goes into *error mode* when it encounters an error while executing a command (see Table 1) or a hardware problem (e.g., a torque limit has been exceeded). It then changes to 1 the value of *es* (error state) in the response [2007][*as*, *hs*, *sm*, *es*, *pm*, *eob*, *eom*] of the `GetStatusRobot`. Recall that you can also receive this message over the monitoring port (see Section 2.8.4). In addition, if you send other commands to the robot, it will respond with the message [1011][The robot is in error.].

When the robot is in error mode, all pending motion commands are canceled (i.e., the motion queue is cleared), the robot stops and ignores subsequent commands (but responds with the [1011] message) until it receives a `ResetError` command. The robot will then execute all request commands and start to accumulate motion commands in its motion queue. However, the commands in the motion queue will be executed only once the `ResumeMotion` command is received by the robot.

2.9.2 P-Stop 2 and SWStop

As soon as the externally wired SWStop on the Meca500 or P-Stop 2 on the MCS500 is activated (see the robot's user manuals), the robot motion is immediately decelerated to a stop and the response [3032][1] is sent by the robot. The motors and the EOAT are still active (i.e. the brakes are not applied) but remain immobilized until the stop is reset.

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

As soon as the stop signal is removed, the message [3032][2] is returned. Then until the stop is reset (with the command `ResumeMotion`), if another motion command is received by the robot, the command will be ignored and the message [3032][2] will be returned.

Finally, to reset the P-Stop 2 or the SWStop, you must remove the stop signal, and then send the command `ResumeMotion`. The robot will respond with the messages [2043][Motion resumed.] and [3032][0]

2.9.3 E-Stop and P-Stop 1

Currently, the Meca500 cannot detect the difference between the E-STOP button on the power supply, an externally wired E-Stop (pins E-Stop on the D-SUB connector) or an externally wired protective stop (pins P-Stop 1), as explained in the Meca500 User Manual.

In revision 3 of the Meca500, the ESTOP completely shuts down the robot. In revision 4 and in the MCS500, when the ESTOP is activated, the robot is decelerated to a full stop, power to the motors and the EOAT connected to the robot is cut, the brakes are applied, and the robot is deactivated. The robot

then sends the message [3070][1], in addition to the messages [2044] [The motion was cleared.] and [2004] [Motors deactivated.]. The only way to reactivate the motors (and the EOAT) is to first clear the E-Stop condition, which will produce the massage [3070][2], and then press the RESET button or activate the external Reset, which will produce the message [3070][0]. Then, you need to re-activate the robot with the `ActivateRobot` command. If the Meca500 R4 was already homed, you do not need to home the robot again, except if an MEGP 25* gripper was connected to the robot's tool I/O port.

In contrast, the MCS500 is able to differentiate between the Emergency Stop function signal and the P-Stop 1. When the P-Stop 1 signal is activated, if the robot is in automatic mode or in manual mode but with the enabling device released, the robot is decelerated to a full stop, power to the motors and the vacuum generator of the MVK01 is cut, the brakes are applied, and the robot is deactivated. The robot then sends the message [3069][1], in addition to the messages [2044] [The motion was cleared.] and [2004] [Motors deactivated.]. The only way to reactivate the motors (and the I/O and vacuum module) is to first clear the P-Stop 1 condition which will produce the message [3069]2], and then press the RESET button or activate the external Reset, which will produce the message [3069][0]. Then, you need to re-activate the robot with the `ActivateRobot` command.

2.9.4 Enabling device

This section applies only for the MCS500, which has input connections for wiring an enabling device that allows you to use the robot in manual mode.

When the operation mode switch of the MCS500 is turned to the "Manual Mode" position, the message [3081][1] is received, no matter whether the enabling device is pressed halfway or not. To start using the MCS500 in manual mode, you need to press the enabling device halfway (or release and press again), which will produce the message [3081][0], and then activate the robot, which will power the robot's motors (when the enabling device is pressed, the P-Stop 1 and P-Stop 2 signals are muted).

If you release the enabling device, while still in manual mode, the message [3081][1] will be received and (1) if the P-Stop 1 signal is present, power from the motors will be removed, otherwise (2) the robot will simply be paused. To start using the MCS500 in manual mode again, you need to press the enabling device halfway again, which will produce the message [3081][2]. Next, you need to reactivate your robot (if it was deactivated because of a P-Stop 1) or simply send the command `ResetMotion`. The robot will then send the message [3081][0]

2.9.5 Operation mode switch

This section applies only for the Meca500, which has a switch for selecting the operation mode.

When the robot is powered, any change in the operating mode switch position will result in decelerating the robot to a full stop and removing power from the robot motors. If the switch position is "Locked", the message [3080][1] will be sent. If the switch position is "Automatic" or "Manual", the message [3080][2] will be sent. You must then press the Reset button on the power supply (or the external reset button), and the robot will send the message [3080][0].

2.9.6 Communication drop

For safety reasons, the robot supervises the TCP connection at all times. If, while the robot is moving, the robot detects that the TCP connection has dropped, it will immediately stop the motion and the message [3081][1] will be sent. If the connection watchdog is enabled, if the TCP connection has

dropped after the timeout specified, the message [3081][1] will be sent regardless of whether the robot is moving or not.

Once a new TCP/IP connection is established, the robot will send the message [3081][2]. Then, you must send the `ResumeMotion` command, to which the robot will respond with the message [3081][0].

2.9.7 Supply voltage fluctuation

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

If the supply voltage fluctuations are more important or longer, the robot will turn itself off and try to reboot a few seconds later.

2.9.8 Robot reboot

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

2.9.9 Redundancy fault

If a redundant safety signal mismatch is detected for more than 1 s, the robot will immediately decelerate to a full stop, remove power from the motors, and send the message [3084][1]. Redundant safety signals are the E-Stop, the P-Stop 1, the P-Stop 2 (MCS500 only), and the three-position enabling device (MCS500 only).

2.9.10 Standstill fault

If the robot is supposed to be in pause mode, but it moves, the message [3085][1] will be sent, power from the motors will be removed, and once the robot has come to a complete stop, the message [3085] [2] will be send. Once the Reset button pressed, the message [3085][0] will be sent.

Recall that if the robot is deactivated (e.g., after an operation mode change, a voltage fluctuation, an E-Stop, a P-Stop 1) all motion command settings, such as the definitions of the TRF and the WRF, and the desired turn of the last joint, will be set to their default values.

3. COMMUNICATING OVER CYCLIC PROTOCOLS

Our robots 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. Limitations

Some of the TCP/IP commands are not available in cyclic protocols. For instance, changing the network configuration (`SetNetworkOptions`), changing the joint limits (`SetJointLimits`), setting up the work zone or the collision configurations, creating, modifying or deleting offline programs, is not possible to do in any of the cyclic protocols.

3.2. Cyclic data

With EtherCAT, EtherNet/IP and PROFINET protocols, our robots are 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.

3.3. Types of robot commands

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

3.3.1 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.

3.3.2 Triggered actions

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

- Activate
- Deactivate
- Home
- ResetError
- ResumeMotion

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.

3.3.3 Motion commands

Most commands related to robot movement are posted to the robot motion queue. The robot will execute these commands sequentially (see Section 3.3.3).

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.

3.4. Sending motion commands

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

3.4.1 Command ID

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

3.4.2 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.

3.4.3 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.

3.4.4 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.

3.5. 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.

3.5.1 Robot control

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

Table 5: Robot control fields

ROBOT CONTROL FIELDS		
Field	**Type**	**Description**
Deactivate	Bool (action)	Deactivates the robot when set to 1.
Activate	Bool (action)	Activates the robot when set to 1 (only if Deactivate bit is 0). The special activation done by `ActivateRobot`(1) is not available in cyclic protocols.
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.
DisableEtherCAT	Bool (state)	Disables (when set to 1) the EtherCAT protocol. This subindex exists only for the MCS500.

3.5.2 Motion control

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

Table 6: Motion control fields

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 PauseMotion and ClearMotion bits are reset to 0, and the ResumeMotion bit is set.

MOTION CONTROL FIELDS		
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, and the ResumeMotion bit is set.
ResumeMotion	Bool (state)	At rising edge, will resume robot motion if it was paused and if resuming motion is allowed (i.e. PauseMotion and ClearMotion bits are cleared and no safety stop conditions are present). Resuming motion will also clear "resettable" safety stop conditions (like P-Stop 2 or enabling device released safety stop signals). Resuming motion will also clear collision or work zone events.

3.5.3 Motion parameters

The motion parameters include the MotionCommandID and six corresponding arguments. These are illustrated in Table 7. The list of available MotionCommandID values is given in Table 8 along with arguments usage in each case.

Table 7: Motion parameters

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 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 8: List of MotionCommandID numbers

MOTION COMMAND ID NUMBERS	
ID	**Description**
0	No movement: all six arguments are ignored.
1	`MoveJoints`†
2	`MovePose`†
3	`MoveLin`†
4	`MoveLinRelTrf`†
5	`MoveLinRelWrf`†

Table 8: List of MotionCommandID numbers (continued)

MOTION COMMAND ID NUMBERS	
ID	**Description**
6	`Delay`†
7	`SetBlending`†
8	`SetJointVel`†
9	`SetJointAcc`†
10	`SetCartAngVel`†
11	`SetCartLinVel`†
12	`SetCartAcc`†.
13	`SetTrf`†
14	`SetWrf`†
15	`SetConf` (in the case of the MCS500, the configuration parameter, c_e, must be entered as the second argument)
16	`SetAutoConf`†
17	`SetCheckpoint`†
18	Gripper, argument 1 is 0 for `GripperClose`, and 1 for `GripperOpen`.
19	`SetGripperVel`†
20	`SetGripperForce`†
21	`MoveJointsVel`†
22	`MoveLinVelWrf`†
23	`MoveLinVelTrf`†
24	`SetVelTimeout`†
25	`SetConfTurn`†
26	`SetAutoConfTurn`†
27	`SetTorqueLimits`†
28	`SetTorqueLimitsCfg`†
29	`MoveJointsRel`†
30	`SetValveState`†
31	`SetGripperRange`†
32	`MoveGripper`†
33	`SetJointVelLimit`†
34	`SetOutputState`, the arguments are BankId (a 32-bit integer), output-values (32 bits, one bit per output, only the first eight are used), and output-mask (32 bits, one bit per output, only the first eight are used). The mask defines which outputs are changed; outputs with corresponding bit equal to 0 in the mask won't be changed.
35	`SetOutputState_Immediate`, the arguments are the same as those of `SetOutputState`

Table 8: List of MotionCommandID numbers (continued)

MOTION COMMAND ID NUMBERS	
ID	**Description**
36	`SetIoSim`†
37	`VacuumGrip`, all six arguments are ignored.
38	`VacuumGrip_Immediate`, all six arguments are ignored.
39	`VacuumRelease`, all six arguments are ignored.
40	`VacuumRelease_Immediate`, all six arguments are ignored.
41	`SetVacuumThreshold`†
42	`SetVacuumThreshold_Immediate`†
43	`SetVacuumPurgeDuration`†
44	`SetVacuumPurgeDuration_Immediate`†
45	`MoveJump`†
46	`SetMoveJumpHeight`†
47	`SetMoveJumpApproachVel`†
48	`SetTimeScaling`†
100	`StartProgram`†

† Arguments are the same as for the corresponding TCP/IP command. For example, if a TCP/IP command has only two arguments, they must correspond to the first and second arguments of the corresponding MotionCommandID, and have the same units, while the other four arguments will be ignored.

In the case of the MCS500 SCARA robot, in the fourth and fifth argument of MotionCommandID of all commands that take a pose (or a Cartesian velocity vector), must be zero and it is the sixth argument that must be the gamma angle (or the rotational velocity about the z axis).

3.5.4 Host time

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

Table 9: Host time fields

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).

3.5.5 Brake control

Table 10 lists the fields that control the robot brakes (when the robot is deactivated) for joints 1, 2 and 3 (simultaneously) on the Meca500 and joints 3 and 4 on the MCS500. The brakes behave as follows:

- Brakes are automatically disengaged when the robot is activated (the robot will actively maintain its position when not moving).
- Brakes are automatically engaged when the robot is powered-down (or P-Stop 1).
- Brakes are automatically engaged when the robot gets deactivated.
- Brakes are automatically engaged when the E-Stop is activated (Meca500 R4 and MCS500).
- While robot is deactivated, the brakes can be controlled using the fields below shown in Table 10.

In the case of the Meca500, disable brakes with caution; without brakes, all links collapse downward. The Brakes control fields will be removed in the upcoming firmware release.

Table 10: Brakes control fields

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 do not get inadvertently disabled if cyclic data sent to the robot contains all zeros.
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.

3.5.6 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.

Table 11: Dynamic data configuration fields

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 12: List of DynamicDataTypeID values with associated values

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 R3, 4-Meca500 R4, 20 = MCS500 R1)]. Same as `GetProductType`.
3	Serial number. Values: [serial number]. Same as `GetRobotSerial`.
4–10	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: [$q_{1,min}$, $q_{2,min}$, $q_{3,min}$, $q_{1,max}$, $q_{2,max}$, $q_{3,max}$]. Unit is degrees. Same as `GetModelJointLimits`.
13	Robot model's nominal joint limits for joints 4, 5 and 6. Values: [$q_{4,min}$, $q_{5,min}$, $q_{6,min}$, $q_{4,max}$, $q_{5,max}$, $q_{6,max}$]. Unit is degrees. Same as `GetModelJointLimits`.
14	Effective joint limits for joints 1, 2 and 3. Values: [$q_{1,min}$, $q_{2,min}$, $q_{3,min}$, $q_{1,max}$, $q_{2,max}$, $q_{3,max}$]. Unit is degrees or mm (in the case of joint 3 of the MCS500). Same as `GetJointLimits`.
15	Effective joint limits for joints 4, 5 and 6. Values: [$q_{4,min}$, $q_{5,min}$, $q_{6,min}$, $q_{4,max}$, $q_{5,max}$, $q_{6,max}$]. Unit is degrees. Same as `GetJointLimits`.
17	Current works zone limits configuration. Values: [work zone limits severity, work zone limits detection mode]. Same as `GetWorkZoneLimitsCfg`.
18	Current work zone limits. Values: [x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]. Units is mm. Same as `GetWorksZoneLimits`.
19	Tool Sphere model. Values: [x, y, z, r]. Units in mm. Same as `GetToolSphere`.
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, joint velocity limit]. Unit is percent. Same as `GetJointVel`, `GetJointAcc`, `GetCartLinVel`, `GetCartAngVel`, `GetCartAcc`, and `GetJointVelLimit`.
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 2.1.27 for corresponding severity/mode values. Same as `GetTorqueLimitsCfg`.
25	Torque limits. Values: [motor 1 limit, motor 2 limit, ...]. Unit is percent. Same as `GetTorqueLimits`.
26	Vacuum configuration. Values: [holdThreshold, releaseThreshold, purgeDuration]. Same as `GetVacuumThreshold` + `GetVacuumPurgeDuration`.
27	Configuration of the `MoveJump` command (MCS500 only). Values: [h_{start},h_{end},h_{min},h_{max}]. Same as `GetMoveJumpHeight`.

Table 12: List of DynamicDataTypeID values with associated values (continued)

DYNAMIC DATA TYPE ID	
ID	**Description**
28	Configuration of the speeds for the approach and retreat of the `MoveJump` command. Values: [v_{start},p_{start},v_{end},p_{end}] Same as `GetMoveJumpApproachVel`.
32	Target real-time joint velocity. Values: [$\dot{q}_1$, $\dot{q}_2$, $\dot{q}_3$, ...]. Unit is mm/s (in the case of joint 3 of the MCS500) or °/s. Same as `GetRtTargetJointVel`.
33	Target real-time joint torque (not implemented yet). Values: [motor 1 torque, motor 2 torque, ..., motor 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`.
36	Current collision configuration. Values: [collision severity level]. Same as `GetCollisionCfg`.
37	Current collision status of the robot. Values: [collision boolean state, group of colliding object 1, ID of colliding object 1, group of colliding object 2, ID of colliding object 2]. Same as `GetCollisionStatus`.
38	Current work zone status of the robot. [work zone breach boolean state, group of object in breach, ID of object in breach]. Same as `GetWorkZoneStatus`.
40	Actual real-time joint set based on hardware encoders. Values: [q_1, q_2, q_3, ...]. Unit is mm (in the case of joint 3 of the MCS500) or 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: [$\dot{q}_1$, $\dot{q}_2$, $\dot{q}_3$, ...]. Unit is mm/s (in the case of joint 3 of the MCS500) or °/s. Same as `GetRtJointVel`.
43	Actual real-time joint torque. Values: [joint 1 torque, joint 2 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 (Meca500 only). 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]. Same as `GetRtGripperState`, `GetRtGripperForce` and `GetRtGripperPos`. Values if pneumatic module: [valve 1 state, valve 2 state]. Same as `GetRtValveState`.
54	Time scaling. Values: [p]. Same as `GetTimeScaling`.
72	Unlike most other dynamic data types that report up to six float values, this one maps the 12 bytes of the first three float values as follows. – Byte index 0: [simMode (1 bit), present (1 bit), reserved (6 bits)]. Least significant byte first. – Byte index 1: errorCode. – Byte index 2: number of outputs. – Byte index 3: number of inputs. – Bytes indices 4–7 : digital outputs values. One bit per output. Least significant bit/byte first. – Bytes indices 8–11 : digital inputs values. One bit per input. Least significant bit/byte first.
73	Vacuum state. Values: [vacuumOn, purgingPart, holdingPart]. Same as `GetRtVacuumState`. The order of these values will be changed in firmware 10.2. Vacuum pressure will be added too.

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 set, target end-effector pose with corresponding configuration, WRF and TRF).

3.6. 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: [q_1, q_2, q_3, ...]. Unit is mm or 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.
- Safety status, as described in Table 16.
- DynamicData #1, #2, #3, #4. See Table 17.

Table 13: Robot status

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.
EStop	Bool	Indicates whether the emergency stop is activated (Meca500 R4 and MCS500 only). Deprecated; use EStop bit from Safety Status instead.
CollisionStatus	Bool	Indicates whether the robot has detected an imminent collision.
WorkZoneStatus	Bool	Indicates whether the robot has detected a work zone breach.

Table 14: Motion 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 2.1.18 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.
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 (EOM) 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.
PStop2	Bool	Indicates whether the SWStop (Meca500) or P-Stop 2 (MCS500) is set. Deprecated; use PStop2 bit from Safety Status instead.
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 15: Robot timestamp

ROBOT TIMESTAMP		
Field	**Type**	**Description**
RobotTimestamp (seconds part)	Integer	Robot's monotonic timestamp (seconds) 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 16: Safety status

SAFETY STATUS		
Field	**Type**	**Description**
EStop	Bool	E-Stop safety stop signal state†
PStop1	Bool	P-Stop 1 safety stop signal state†
PStop2	Bool	P-Stop 2 safety stop signal state†
Operation mode changed	Bool	Operation mode change stop signal state†
Enabling device released	Bool	Enabling device released stop signal state†
Voltage fluctuation	Bool	Voltage fluctuation stop signal state†
Robot rebooted	Bool	Robot rebooted safety signal state†
Redundancy fault	Bool	Redundancy fault safety signal state (1 if present, 0 if not)
Standstill fault	Bool	Standstill fault safety signal state†
Connection dropped	Bool	TCP/IP connection dropped safety signal state†
EStop – reset ready	Bool	E-Stop safety stop signal stop ready to be reset with Reset button
PStop1 – reset ready	Bool	P-Stop 1 safety stop signal stop ready to be reset with Reset button
PStop2 – reset ready	Bool	P-Stop 2 safety stop signal stop ready to be reset with `ResumeMotion` command
Operation mode changed – reset ready	Bool	Operation mode change safety signal stop ready to be reset with Reset button
Enabling device released – reset ready	Bool	Enabling device safety signal stop ready to be reset with `ResumeMotion` command
Voltage fluctuation – reset ready	Bool	Voltage fluctuation safety signal stop ready to be reset with Reset button
Robot rebooted – reset ready	Bool	Robot rebooted safety signal stop ready to be reset with Reset button
Standstill fault – reset ready	Bool	Standstill fault safety signal stop ready to be reset with Reset button
Connection dropped – reset ready	Bool	Connection dropped safety signal stop ready to be reset with `ResumeMotion` command
Operation mode	Integer	0 for locked, 1 for automatic, 2 for manual
Reset ready	Bool	If no more errors causing motor power to be removed are present, and the robot is ready to be reset with the Reset button
Motor voltage on	Bool	Robot motors powered or not

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

Table 17: Robot dynamic data

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).

4. ETHERCAT COMMUNICATION

EtherCAT is an open real-time Ethernet protocol originally developed by Beckhoff Automation. When communicating with a Mecademic robot over EtherCAT, you can 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 Mecademic 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 Mecademic 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 (Link/Act IN) 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 in the case of both the Meca500 and the MCS500, use the RobotConrol PDO (Section 4.2.1) or simply perform a network settings reset. In the case if the Meca500, you can also use the Communication mode SDO (Section 4.2.17).

4.1.4 LEDs

Both the Meca500 and the MCS500 has three green LEDs on their base, labeled Link/Act IN, Link/Act OUT, and Run. When EtherCAT communication is enabled, these three LEDs communicate the state of the EtherCAT connection, as summarized in Table 18.

Table 18: EtherCAT LED description

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

†Only on MCS500

4.2. Object dictionary

This section describes all objects available for interacting with a Mecademic robot. 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 19 describes the object's indices. See Table 5 for detailed explanations.

Table 19: Robot control object

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
	7	Variable	BOOL	DisableEtherCAT	0	0	1	RW	1600h:7

4.2.2 Motion control

This object controls the actual robot movement. Table 20 describes the object's indices. See Table 5 for detailed explanations.

Table 20: Motion control objects

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	ResumeMotion	0	0	1	RW	1601h:5

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 21. See Section 3.4 and Section 3.5 for detailed explanations.

Table 21: Movement objects

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 7).

4.2.4 Host time

This object controls the robot's date/time (real-time-clock). Table 22 describes the object's indices. See Table 9 for detailed explanations.

Table 22: Host time object

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

4.2.5 Brake control

This object controls the robot's brakes (applies only when robot is deactivated). Table 23 describes the object's indices. See Table 10 for detailed explanations about brakes behavior.

Table 23: Brakes control object

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

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.

Table 24: Dynamic data configuration object 1

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 25: Dynamic data configuration object 2

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 26: Dynamic data configuration object 3

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 27: Dynamic data configuration object 4

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

4.2.7 Robot status

The structure of the robot status object is described in Table 28. See Table 13 for detailed explanations.

Table 28: Robot status object

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
	8	Variable	BOOL	EStop	n/a	0	1	RO	1A00h.8
	9	Variable	BOOL	CollisionStatus	n/a	0	1	RO	1A00h.9
	10	Variable	BOOL	WorkZoneStatus	n/a	0	1	RO	1A00h.10
	n/a	Variable	UINT	(7 unused bits)	n/a	0	0	RO	n/a
	1	Variable	UINT	ErrorCode	n/a	0	65,535	RO	1A00h.1

4.2.8 Motion status

The structure of the motion status object is described in Table 29. See Table 14 for detailed explanations.

Table 29: Motion status object

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
	n/a	Variable	UINT	(10 unused bits)	n/a	0	0	RO	n/a
	4	Variable	UINT	OfflineProgramID	n/a	0	500	RO	1A01h.4

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`.

Table 30: Target joint set object

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	†	†	RO	1A02h.1
	2		REAL	Target position of joint 2	n/a	†	†	RO	1A02h.2
	3		REAL	Target position of joint 3	n/a	†	†	RO	1A02h.3
	4		REAL	Target position of joint 4	n/a	†	†	RO	1A02h.4
	5		REAL	Target position of joint 5	n/a	†	†	RO	1A02h.5
	6		REAL	Target position of joint 6	n/a	†	†	RO	1A02h.6

† depending on the robot model. See Section 1.1.5.

4.2.10 Target end-effector pose

The structure of the real-time target end-effector pose object is described in Table 31. The data is the same as that returned by TCP/IP command `GetRtTargetCartPos`.

Table 31: Target end-effector pose object

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 *y*	n/a	−3.4E38	3.4E38	RO	1A03h.6

4.2.11 Target configuration

The structure of the real-time target configuration object is described in Table 32. The data is the same as that returned by the combination of the TCP/IP commands `GetRtTargetConf` and `GetRtTargetConfTurn`.

Table 32: Target configuration object

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	†	†	RO	1A08h.4

† depending on the robot model. See Section 1.1.5

4.2.12 WRF

The structure of the real-time WRF object is described in Table 33. The data is the same as that returned by TCP/IP command `GetRtWrf`.

Table 33: WRF object

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 *y*	n/a	−3.4E38	3.4E38	RO	1A09h.6

4.2.13 TRF

The structure of the real-time TRF object is described in Table 34. The data is the same as that returned by TCP/IP command `GetRtTrf`.

Table 34: TRF object

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

4.2.14 Robot timestamp

The structure of the Robot timestamp object is described in Table 35. See Table 15 for details.

Table 35: Robot timestamp object

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 (microsec. part)	n/a	0	232 − 1	RO	1A10h.2
	3		UDINT	DynamicDataUpdate	n/a	0	$2^{32} - 1$	RO	1A10h.3

4.2.15 Safety status

The structure of the Safety status object is described in Table 36. See Table 16 for details.

Table 36: Safety status object

SAFETY STATUS OBJECT INDICES									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
6065h		Record							
	2	Variable	BOOL	Estop	n/a	0	1	RO	1A11h.2
	3	Variable	BOOL	PStop1	n/a	0	1	RO	1A11h.3
	4	Variable	BOOL	PStop2	n/a	0	1	RO	1A11h.4
	n/a	Variable	BOOL	(1 unused bit)	n/a	0	0	RO	n/a
	5	Variable	BOOL	Operation Mode Change	n/a	0	1	RO	1A11h.5
	6	Variable	BOOL	Enabling device released	n/a	0	1	RO	1A11h.6
	7	Variable	BOOL	Voltage Fluctuation	n/a	0	1	RO	1A11h.7
	8	Variable	BOOL	Reboot	n/a	0	1	RO	1A11h.8
	9	Variable	BOOL	Redundancy Fault	n/a	0	1	RO	1A11h.9
	10	Variable	BOOL	Standstill Fault	n/a	0	1	RO	1A11h.10
	11	Variable	BOOL	Connection Dropped	n/a	0	1	RO	1A11h.11
	n/a	Variable	BOOL	(21 unused bits)	n/a	0	0	RO	n/a
	12	Variable	BOOL	Estop Resettable	n/a	0	1	RO	1A11h.12
	13	Variable	BOOL	PStop1 Resettable	n/a	0	1	RO	1A11h.13
	14	Variable	BOOL	PStop2 Resettable	n/a	0	1	RO	1A11h.14
	n/a	Variable	BOOL	(1 unused bit)	n/a	0	0	RO	n/a
	15	Variable	BOOL	Operation Mode Change Resettable	n/a	0	1	RO	1A11h.15
	16	Variable	BOOL	Enabling device released Resettable	n/a	0	1	RO	1A11h.16
	17	Variable	BOOL	Voltage Fluctuation Resettable	n/a	0	1	RO	1A11h.17
	18	Variable	BOOL	Reboot Resettable	n/a	0	1	RO	1A11h.18
	19	Variable	BOOL	Redundancy Fault Resettable	n/a	0	1	RO	1A11h.19
	20	Variable	BOOL	Standstill Fault Resettable	n/a	0	1	RO	1A11h.20
	21	Variable	BOOL	Connection Dropped Resettable	n/a	0	1	RO	1A11h.21
	n/a	Variable	BOOL	(21 unused bits)	n/a	0	0	RO	n/a
	1	Variable	USINT	Operation Mode	n/a	0	2	RO	1A11h.1
	22	Variable	BOOL	Reset Ready	n/a	0	1	RO	1A11h.22
	23	Variable	BOOL	Vmotor on	n/a	0	1	RO	1A11h.23
	n/a	Variable	BOOL	(22 unused bits)	n/a	0	0	RO	n/a

4.2.16 Dynamic data

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

Table 37: Dynamic data object 1

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 38: Dynamic data object 2

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 39: Dynamic data object 3

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 40: Dynamic data object 4

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).

4.2.17 Communication mode (SDO) - Meca500 only

When EtherCAT is enabled, subindex 1 of this SDO (present only in the Meca500) is equal to 2 (see the 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.

Table 42: Communication mode SDO

COMMUNICATION MODE SDO									
Index	**SI**	**O. code**	**Type**	**Name**	**Default**	**Min.**	**Max.**	**Access**	**PDO**
8000h		Record		Communication 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

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 Mecademic robot) to the EtherCAT master.

In the previous section, we listed the PDOs object dictionary. PDO assignment is summarized in the next two tables.

Table 43: RxPDOs

RECEIVING PDO			
PDO	**Object(s)**	**Name**	**Note**
1600h	7200h	RobotControl	Mandatory. See Table 19.
1601h	7310h	MotionControl	Mandatory. See Table 20
1602h	7305h, 7306h	Movement	Mandatory. See Table 21.
1610h	7400h	HostTime	Mandatory. See Table 22.
1611h	7410h	BrakesControl	Mandatory. See Table 23.
1620h	7420h	DynamicDataConfiguration 1	Mandatory. See Table 24.
1621h	7421h	DynamicDataConfiguration 2	Mandatory. See Table 25.
1622h	7422h	DynamicDataConfiguration 3	Mandatory. See Table 26.
1623h	7423h	DynamicDataConfiguration 4	Mandatory. See Table 27.

Table 44: TxPDOs

TRANSMISSION PDO			
PDO	**Object**	**Name**	**Note**
1A00h	6010h	RobotStatus	Mandatory. See Table 28
1A01h	6015h	MotionStatus	Mandatory. See Table 29.
1A02h	6030h	TargetJointSet	Mandatory. See Table 30.
1A03h	6031h	TargetEndEffectorPose	Mandatory. See Table 31.
1A08h	6046h	TargetConfiguration	Mandatory. See Table 32.
1A09h	6050h	WRF	Mandatory. See Table 33.
1A0Ah	6051h	TRF	Mandatory. See Table 34.
1A10h	6060h	RobotTimestamp	Mandatory. See Table 35.
1A11h	6065h	SafetyStatus	Mandatory. See Table 36
1A20h	6070h	DynamicData #1	Mandatory. See Table 37.
1A21h	6071h	DynamicData #2	Mandatory. See Table 38.
1A22h	6072h	DynamicData #3	Mandatory. See Table 39.
1A23h	6073h	DynamicData #4	Mandatory. See Table 40.

5. ETHERNET/IP COMMUNICATION

Mecademic robots are compatible with the EtherNet/IP protocol. (The Meca500 is Certified by ODVA, and the MCS500 is under certification) A common industry standard, it can be used with many different PLC brands. Tested to work at 10 ms, faster times are also possible. Our robots typically use implicit (cyclic) messaging.

Refer to our Support Center for specific PLC examples.

5.1. Connection types

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

5.2. EDS file

The Electronic Data Sheet (EDS) file for the 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

A Mecademic 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).

Table 45: Output tag assembly

OUTPUT TAG ASSEMBLY			
Bytes	**Data Type**	**Name**	**Description**
0–3	DWORD	RobotControl	See Table 46.
4–5	UINT	MoveID	See Table 47.
6–7	WORD	MotionControl	See Table 48.
8–11	UDINT	Movement	See Table 49.
12–15	REAL	Argument 1 for Movement	See Table 50.
16–19	REAL	Argument 2 for Movement	See Table 50.
20–23	REAL	Argument 3 for Movement	See Table 50.
24–27	REAL	Argument 4 for Movement	See Table 50.
28–31	REAL	Argument 5 for Movement	See Table 50.
32–35	REAL	Argument 6 for Movement	See Table 50.
36–39	DINT	HostTime	See Table 51.
40–43	DWORD	BrakesControl	See Table 52.
44–47	UDINT	DynamicDataConfiguration 1	See Table 53.
48–51	UDINT	DynamicDataConfiguration 2	See Table 53.
52–45	UDINT	DynamicDataConfiguration 3	See Table 53.
46–59	UDINT	DynamicDataConfiguration 4	See Table 53.

5.5.1 Robot control tag

This tag controls the robot's initialization and simulation. Table 46 describes the tag's bits. See Table 5 for detailed explanations.

Table 46: Robot control tag

CONTROL TAGS									
Bytes	**Data Type**	**Bits 7–31**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
0–3	DWORD	Unused	Disable-EtherCAT	Recovery-Mode	Sim-Mode	Reset-Error	Home	Activate	Deactivate

5.5.2 MoveID tag

This tag (Table 47) contains the distinct user-defined ID number associated with each motion command sent to the robot. See Table 6 for detailed explanations.

Table 47: MoveID tag

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

5.5.3 Motion control tag

This tag controls the actual robot movement. Table 48 describes the tag's bits. See Table 6 for details.

Table 48: Motion control tag

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

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 7). The motion command argument tags contain the arguments of the motion command (Table 50).

See Section 3.4 and Section 3.5 for detailed explanations.

Table 49: Motion command tag

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

Table 50: Motion command arguments tags

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

5.5.5 Host time tag

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

Table 51: Host time tag

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

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.

Table 52: Brakes control tag

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

5.5.7 Dynamic data configuration tag

This tag is used to choose which dynamic data type the robot will return (Table 53). See Table 11.

Table 53: Dynamic data configuration tag

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 45).

5.6. Input tag assembly

The input tag assembly has an Instance of 100 with a size 252-byte array, as detailed in Table 54. Please refer to Section 3.6 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).

Table 54: Input tag assembly

INPUT TAG ASSEMBLY			
Bytes	**Data Type**	**Data**	**Description**
0–1	WORD	RobotStatus	See Table 55.
2–3	UINT	ErrorCode	See Table 56.
4–7	UDINT	Checkpoint	See Table 57.
8–9	UINT	MoveID	See Table 58.
10–11	UINT	Motion queue space	See Table 59.
12–13	WORD	MotionStatus	See Table 60.
14–15	UINT	OfflineProgramID	See Table 61.

Table 54: Input tag assembly (continued)

INPUT TAG ASSEMBLY			
Bytes	**Data Type**	**Data**	**Description**
16–19	REAL	TargetJointSet (joint 1)	See Table 62.
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 63.
44–47	REAL	TargetEndEffectorPose *y*	
48–51	REAL	TargetEndEffectorPose *z*	
52–55	REAL	TargetEndEffectorPose α	
56–59	REAL	TargetEndEffectorPose β	
60–63	REAL	TargetEndEffectorPose *y*	
64	SINT	TargetConfiguration c_s (shoulder)	See Table 64.
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 65.
72–75	REAL	WRF *y*	
76–79	REAL	WRF *z*	
80–83	REAL	WRF α	
84–87	REAL	WRF β	
88–91	REAL	WRF *y*	
92–95	REAL	TRF *x*	See Table 66.
96–99	REAL	TRF *y*	
100–103	REAL	TRF *z*	
104–107	REAL	TRF α	
108–111	REAL	TRF β	
112–115	REAL	TRF *y*	
116–119	UDINT	RobotTimestamp (seconds part)	See Table 67.
120–123	UDINT	RobotTimestamp (microseconds part)	
124–127	UDINT	DynamicDataUpdateCount	
128–131	UDINT	Safety stop mask	See Table 68.
132–135	UDINT	Safety stop resettable mask	
136	USINT	Operation mode	
137	USINT	Various safety stop status bits	
138–129	UNIT	Researved for future use	

Table 54: Input tag assembly (continued)

INPUT TAG ASSEMBLY			
Bytes	**Data Type**	**Data**	**Description**
140–143	UDINT	DynamicData #1 type ID	See Table 69.
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	

5.6.1 Robot status tag

The structure of the robot status tag is described in Table 55. See Table 13 for detailed explanations.

Table 55: Robot status tag

ROBOT STATUS TAG											
Bytes	**Data Type**	**Bits 9–15**	**Bit 8**	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
0–1	WORD	Unused	WorkZone-Status	Collision-Status	EStop	Recovery-Mode	Brakes-Engaged	SimMode	Homed	Activated	Busy

5.6.2 Error code tag

The structure of the error code tag is described in Table 56. See Table 13 for detailed explanations.

Table 56: Error code tag

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

5.6.3 Checkpoint tag

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

Table 57: Checkpoint tag

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

5.6.4 MoveID tag

The structure of the MoveID tag is described in Table 58. See Table 14 for detailed explanations.

Table 58: MoveID tag

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

5.6.5 FIFO space tag

The structure of the Motion queue space tag is described in Table 59. See Table 14 for details.

Table 59: Motion queue space tag

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

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 60. See Table 14 for detailed explanations.

Table 60: Motion status tag

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	PStop2	Cleared	EOM	EOB	Paused

5.6.7 Offline program ID

This tag indicates the ID of the offline program currently running (Table 61). See Table 14 for details.

Table 61: Offline program tag

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

5.6.8 Target joint set

The structure of the target joint set tag is described in Table 62. The data is the same as that returned by TCP/IP command `GetRtTargetJointPos`.

Table 62: Target joint set tag

TARGET JOINT SET TAGS				
Bytes	**Data Type**	**Name**	**Minimum**	**Maximum**
16–19	REAL	Target position of joint 1	†	†
20–23	REAL	Target position of joint 2	†	†
24–27	REAL	Target position of joint 3	†	†
28–31	REAL	Target position of joint 4	†	†
32–35	REAL	Target position of joint 5	†	†
36–39	REAL	Target position of joint 6	†	†

† depending on the robot model. See Section 2.1.2.

5.6.9 Target end-effector pose

The structure of the target end-effector pose tag is described in Table 63. The data is the same as that returned by TCP/IP command `GetRtTargetCartPos`.

Table 63: Target end-effector pose tag assembly

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 γ

5.6.10 Target configuration

The structure of the target configuration tag is described in Table 64. The data is the same as that returned by the combination of the TCP/IP commands `GetRtTargetConf` and `GetRtTargetConfTurn`.

Table 64: Robot target configuration tags

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)	†	†

† depending on the robot model. See Section 1.1.5

5.6.11 WRF

The structure of WRF tag is described in Table 65. The data is the same as that returned by `GetRtWrf`.

Table 65: WRF tag assembly

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 γ

5.6.12 TRF

The structure of TRF tag is described in Table 66. The data is the same as that returned by `GetRtTrf`.

Table 66: TRF tag assembly

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 γ

5.6.13 Robot timestamp

The structure of the Robot timestamp tag is described in Table 67. See Table 15 for details.

Table 67: Robot timestamp tag assembly

ROBOT TIMESTAMP TAG ASSEMBLY		
Bytes	**Data Type**	**Name**
116–119	UDINT	RobotTimestamp (seconds part)
120–123	UDINT	RobotTimestamp (microseconds part)
124–127	UDINT	DynamicDataUpdateCount

5.6.14 Safety status

The structure of the Safety status tag is described in Table 68. See Table 68 for details.

Table 68: Safety status tag assembly

SAFETY STATUS TAG ASSEMBLY		
Bits	**Data Type**	**Name**
0	BOOL	Estop
1	BOOL	PStop1
2	BOOL	PStop2
3	n/a	(unused)
4	BOOL	Operation Mode Change
5	BOOL	Enabling device released
6	BOOL	Voltage Fluctuation
7	BOOL	Reboot
8	BOOL	Redundancy Fault
9	BOOL	Standstill Fault
10	BOOL	Connection Dropped
11–31	n/a	(unused)
32	BOOL	Estop Resettable
33	BOOL	PStop1 Resettable
34	BOOL	PStop2 Resettable
35	BOOL	(unused)
36	BOOL	Operation Mode Change Resettable
37	BOOL	Enabling device released Resettable
38	BOOL	Voltage Fluctuation Resettable
39	BOOL	Reboot Resettable
40	BOOL	Redundancy Fault Resettable

Table 68: Safety status tag assembly (continued)

SAFETY STATUS TAG ASSEMBLY		
Bits	**Data Type**	**Name**
41	BOOL	Standstill Fault Resettable
42	BOOL	Connection Dropped Resettable
43–63	n/a	(unused)
64–71	UINT	Operation Mode
72	BOOL	Reset Ready
73	BOOL	Vmotor on
74–96	BOOL	(22 unused bits)

5.6.15 Dynamic data

The structure of the dynamic data tags are described in Table 54. See Table 17 for detailed explanations.

Table 69: Dynamic data tag assembly

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 54).

6. PROFINET COMMUNICATION

Mecademic robots are compatible with the PROFINET protocol, a common industry standard that can be used with many different PLC brands. (The Meca500 is Certified by PROFIBUS, and the MCS500 is under certification). Cyclic times up to 1 ms (though not as "hard-real-time" as EtherCAT) are possible.

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

6.1. PROFINET conformance class

The Electronic Data Sheet (EDS) file for the specific robot model 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 Mecademic robots

Mecademic robots do 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 Mecademic robots, the same as with TCP/IP. Either Ethernet port on the base of the robot can be used. The 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 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 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 Mecademic robots.

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

Ethernet packets of type PN-DCP (0x8892) are used for the DCP protocol (Discovery and Configuration Protocol). This protocol is used to discover PROFINET devices on the network. It'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 Mecademic 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 PROFINET 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 robot is enabled only when PROFINET is enabled on the robot (so it will not be possible to detect a robot using LLDP until PROFINET is enabled on it).

6.4. Exclusivity of AR

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

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

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

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

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

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

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 a Mecademic robot.

The robot GSDML file is provided along with the robot's 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 robot capabilities, this manual will only provide a quick summary of the GSDML file.

6.5.1 Robot modules and sub-modules

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

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

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

6.6. Cyclic data

Using cyclic data to control and monitor Mecademic 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 robot-controlling application on a controller/PLC between these different protocols.

Please refer to the robot 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

Mecademic robots will not generate any PROFINET alarms. Any alarm or error condition will be reported by the robot through the corresponding cyclic data fields. This allows the robots to behave the same across various Cyclic protocols (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 70 presents of summary of the terms that we use frequently in our manuals and in the robot's web interface.

Table 70: Glossary of terms used by Mecademic

GLOSSARY OF TERMS	
TERM	**DESCRIPTION**
BRF	Base reference frame (see Figure 3).
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).
Default value	There are different settings in the robot controller that can be configured using `Set*` commands (e.g., `SetCartAcc`.). Many of these settings have default values. Every time the robot is powered up, these settings are initialized to their default values. In the case of motion commands settings, their values are also initialized to their default values every time the robot is deactivated. In contrast, some settings are persistent and their values are stored on an SD drive.
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, as well as a vacuum and I/O module kit with suction cups, for the MCS500.
Error mode	The robot goes into error mode when it encounters an error while executing a command or a hardware problem (see Section 2.8.1).
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).
FRF	Flange reference frame (see Figure 3).
Instantaneous command	A command that is executed immediately, rather than placed in the motion queue. Examples includes the commands `ClearMotion`, `Home`, `GetRobotName`, `GetTrf`, and `GetrRtGripperForce`.
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 rotary joint *i* (*i* = 1, 2, ..., 6), denoted by θ_i and measured in degrees (see Section 1.1.5).
Joint position	Since our MCS500 SCARA robot has a translational joint, we will use this generic term to designate both the angle of a rotary joint (in degrees) and the position of the translational joint of the MCS500 (in mm). A more scientific alternative would be *joint coordinate* or *joint variable*. We will denote the joint position with q_i.

Table 70: Glossary of terms used by Mecademic (continued)

GLOSSARY OF TERMS	
TERM	**DESCRIPTION**
Joint set	The set of all joint positions, i.e., $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$ or $\{\theta_1, \theta_2, d_3, \theta_4\}$ (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 and later called by the command `StartProgram`.
Persistents	Some settings in the robot controller have default values (e.g., the robot name set by the command `SetRobotName`), but when changed, their new values are written on an SD drive and persist even if the robot is powered off.
Pose	Position and orientation of one reference frame with respect to another.
Position mode	One of the two control modes, in which the robot's motion is generated by requesting a target 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 commands	Instantaneous commands with the `Get` prefix that return immediately some data.
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 set 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 of the Meca500 (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.
Queued command	A command that is placed in the motion queue, rather than executed immediately. All motion commands are queued commands, as well as some external-tool commands.
TCP	Tool center point, the origin of the tool reference frame (see Figure 3).
TRF	Tool reference frame (see Figure 3).

Table 70: Glossary of terms used by Mecademic (continued)

GLOSSARY OF TERMS	
TERM	**DESCRIPTION**
Turn configuration	An integer c_t, such that $-180° + c_t360° < \theta_{last} \leq 180° + c_t360°$, where θ_{last} is the joint angle of the last robot joint (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 3).
Wrist center	The point where the axes of joints 4, 5 and 6 of the Meca500 intersect.

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