



Dorna Robotics  
1306 MONTE VISTA AVE, STE 8  
UPLAND, CA 91786  
USA  
+1 (800) 733-2187  
sales@dorna.ai  
<https://dorna.ai>

---

# Dorna Robot User Manual

Dorna TA

Last update on Oct 1, 2024



Firmware	305
Dorna Lab	2.1.0
Python API	2.1.3

The information contained herein is the property of Dorna Robotics and shall not be reproduced in whole or in part without the prior written approval of Dorna Robotics. The information herein is subject to change without notice and should not be construed as a commitment by Dorna Robotics. This document is periodically reviewed and revised. Dorna Robotics assumes no responsibility for any errors or omissions in this document.

# Table of contents

<b>Table of contents.....</b>	<b>4</b>
<b>Introduction.....</b>	<b>11</b>
How to read this manual.....	11
Robot arm and controller box.....	11
Robot parts.....	12
What do the boxes contain?.....	12
Where to find more information.....	12
<b>Safety.....</b>	<b>12</b>
The brakes.....	13
Power supply.....	13
Robot cables.....	14
Motors.....	14
IO wiring.....	14
Operation environment.....	14
Emergency stop.....	15
Safety function.....	15
Risk assessment.....	16
<b>Product specification.....</b>	<b>17</b>
Specs.....	17
Drawings.....	20
<b>Transportation.....</b>	<b>21</b>
<b>Quick Start.....</b>	<b>21</b>
<b>Mechanical interface.....</b>	<b>23</b>
Robot arm mounting.....	23
Tool mounting.....	24
Controller box installation.....	26
<b>Electrical Interface.....</b>	<b>26</b>
Introduction.....	26
Electrical warnings and cautions.....	27
Controller box I/O.....	27
IO wiring.....	29
Output wiring.....	29
Input wiring.....	31
Sample wiring.....	32
Digital inputs from a button or Emergency Button.....	32



Digital Inputs from a sensor, PLC, or another device's digital output.....	32
Load controlled by a digital output.....	33
Digital output pin to a relay.....	34
Digital output pin to an input pin of another machine or PLC.....	35
<b>TCP network.....</b>	<b>35</b>
Credentials.....	35
Robot connection methods.....	36
Connection through a router.....	36
Method 1.....	36
Method 2.....	37
Direct connection to a computer.....	37
Static IP.....	39
WiFi.....	42
SSH.....	43
SSH error.....	44
Dorna Lab address.....	45
Jupyter Notebook address.....	45
WebSocket server address.....	45
Internet access.....	46
Upgrade process.....	46
Check for new updates.....	46
Software upgrade.....	47
OS upgrade.....	49
Fresh OS image.....	50
Upgrade current OS.....	51
Expand the file system.....	51
Step by step installation of bullseye.....	52
Update the time zone.....	55
Update the configuration.....	56
<b>Motion concepts.....</b>	<b>57</b>
Coordinate system.....	57
Units.....	57
Joint assignment.....	57
Joint limit.....	59
Reference frames.....	60
BRF: Base reference frame.....	60
WRF: World reference frame.....	60
FRF: Flange reference frame.....	60

TRF: Tool reference frame.....	60
TCP: Tool center point.....	60
Joint space.....	60
Assigning values to the joints.....	61
Auto assigning values to the auxiliary axes.....	61
Cartesian coordinates.....	62
Tool Matrix.....	62
Types of motion.....	64
Joint move (jmove).....	64
Line move (lmove).....	64
Circle move (cmove).....	65
The dynamic of a motion.....	66
How to pick the right value.....	67
Continuous motion.....	67
Absolute and relative motion.....	69
<b>Auxiliary axes.....</b>	<b>69</b>
General notes.....	69
Setup.....	69
Scenarios.....	69
Definition.....	70
Example.....	71
Configuration.....	71
Wiring.....	73
Operation.....	75
<b>Command server.....</b>	<b>75</b>
Introduction.....	75
Server address.....	75
Data format.....	76
Messages.....	76
Motion.....	77
Inputs.....	77
Status.....	78
Command response.....	78
Alarm.....	78
Commands.....	79
Format.....	79
Order of commands.....	79
Normal priority queue.....	79

High-priority queue.....	79
Status of a command and its life cycle.....	80
List of commands.....	81
jmove.....	81
lmove.....	84
cmove.....	86
halt.....	88
alarm.....	89
pid.....	91
sleep.....	94
input.....	95
probe.....	97
iprobe.....	99
output.....	101
pwm.....	103
adc.....	106
joint.....	107
axis.....	109
motor.....	111
tool.....	113
version.....	115
<b>Dorna Lab.....</b>	<b>117</b>
URL address.....	117
Halt button.....	118
Alarm information.....	118
Real-time orientation.....	118
Jogging.....	118
Motors.....	119
Hand training.....	120
Setup.....	120
3D view.....	120
I/O.....	120
Script.....	121
Log.....	121
Blockly editor.....	121
Shell viewer.....	121
Python editor.....	121
Processes.....	121

End a process.....	121
Duplicate and run a process.....	122
Info.....	122
Auxiliary axes.....	122
Emergency stop.....	122
Startup programs.....	122
Keyboard and Joystick.....	123
Jupyter Notebook.....	123
Create a kernel.....	123
Shutdown a kernel.....	123
Relaunch the server.....	123
File management.....	124
Exploring the files.....	124
Programs location.....	124
Transferring files.....	124
<b>Python API.....</b>	<b>125</b>
Useful links.....	125
Robot OS Python environment.....	125
Add a python library.....	125
Install the API.....	126
Dorna class.....	127
Getting started.....	127
Connection.....	128
connect(host="localhost", port=443, timeout=5).....	128
close().....	129
Command status.....	129
track_cmd().....	129
Sending command.....	130
play(timeout=-1, **kwargs).....	130
play_dict(cmd={}, timeout=-1).....	133
play_json(cmd='{}', timeout=-1).....	133
play_script(file="", timeout=-1).....	133
Messages.....	135
last_cmd().....	135
last_msg().....	135
union().....	135
val(key="cmd").....	135
Move.....	136

jmove(timeout=-1, **kwargs).....	136
lmove(timeout=-1, **kwargs).....	136
cmove(timeout=-1, **kwargs).....	137
Stop.....	137
halt(accel=None).....	137
get_alarm().....	138
set_alarm(enable=None).....	138
Joint and TCP.....	138
get_all_joint().....	138
get_joint(index=None).....	138
set_joint(index=None, val=None).....	138
get_all_pose().....	139
get_pose(index=None).....	139
get_tool().....	139
set_tool(r00=None, r01=None, r02=None, r10=None, r11=None, r12=None, r20=None, r21=None, r22=None, lx=None, ly=None, lz=None ).....	139
I/O.....	140
get_all_output().....	140
get_output(index=None).....	140
set_output(index=None, val=None, queue=None).....	140
get_pwm(index=None).....	140
set_pwm(index=None, enable=None, queue=None).....	140
get_freq(index=None).....	141
set_freq(index=None, freq=None, queue=None).....	141
get_duty(index=None).....	141
set_duty(index=None, duty=None, queue=None).....	141
get_all_input().....	141
get_input(index=None).....	141
get_all_adc().....	142
get_adc(index=None).....	142
Wait and delay.....	142
probe(index=None, val=None).....	142
iprobe(index=None, val=None).....	143
sleep(val=None).....	143
Setting.....	143
get_motor().....	143
set_motor(enable=None).....	144
get_axis(index=None).....	144
set_axis(index=None, usem=None, usee=None, pprm=None, tprm=None,	

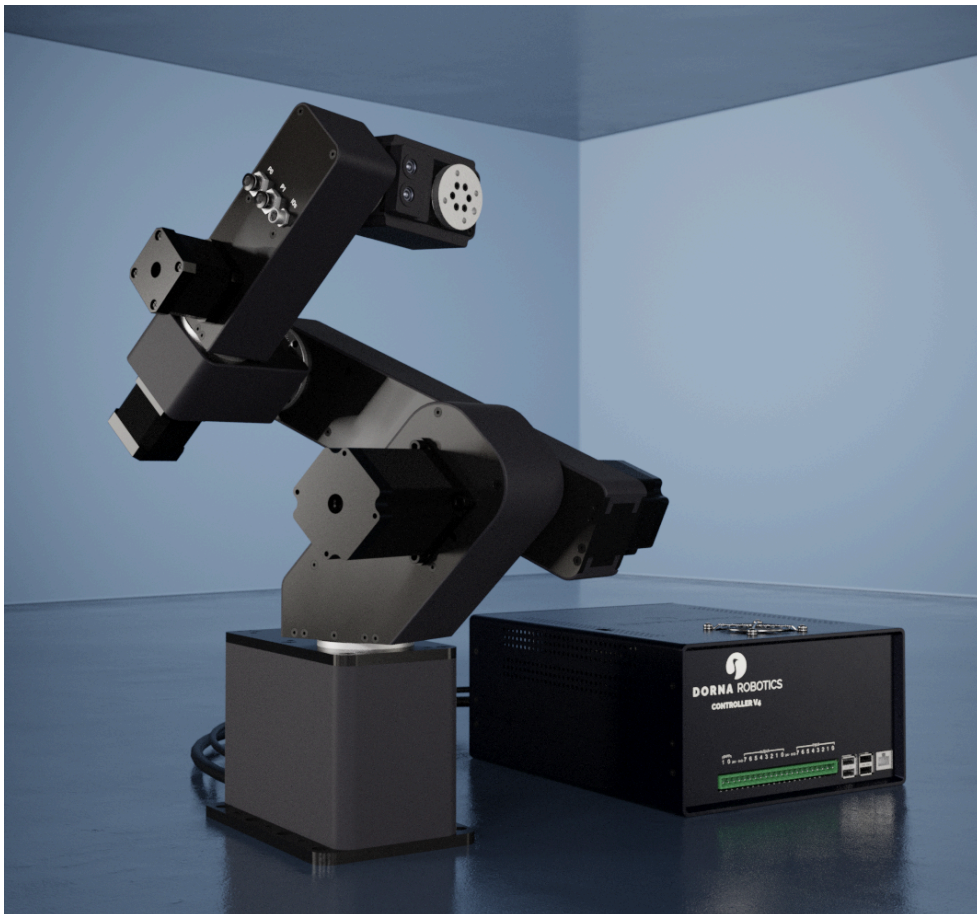
ppre=None, tpre=None).....	144
get_pid(index=None).....	144
set_pid(index=None, p=None, i=None, d=None, thr=None, dur=None).....	145
Info.....	145
version().....	145
uid().....	145
Log.....	145
log(msg).....	145
logger_setup(file="dorna.log").....	145
Event.....	146
get_all_event().....	146
add_event(target=None, kwargs={}).....	146
Format of the target.....	146
target(msg, union, **kwargs).....	146
.clear_event(target=None).....	146
.clear_all_event().....	147
Example.....	147
<b>Troubleshooting.....</b>	<b>148</b>
No LEDs are on upon power up.....	148
No connection to the robot's web interface.....	148
The robot fails to boot.....	149
Joint and position lost.....	149
High-temperature motors.....	149
<b>Maintenance.....</b>	<b>149</b>
Check for upgrade.....	149
Cables and wires.....	149
Belts.....	150
Connectors.....	150

## Introduction

### How to read this manual

This manual contains instructions for installing and programming the robot. The manual is intended for the robot integrator, who must have basic mechanical and electrical training and be familiar with elementary programming concepts.

### Robot arm and controller box



Robot arm and controller box.

## Robot parts

- Axes and joints: The Dorna TA robot has six rotating axes (joints) labeled as `axis0` to `axis5`. We sometimes use the term joint instead of axis and label them as `joint0` to `joint5` or just `j0` to `j5`.
- Base: The robot's lower fixed and stationary part is called the base and is used for mounting the robot to a surface.
- Flange: The robot flange refers to the interface or mounting point at the robot arm's end. It is designed to provide a secure connection between the robot and various end-effectors, tools, or fixtures.
- Motors: The robot has six motors labeled as `motor0` to `motor5`, or just `m0` to `m5`
- Encoders: The robot has six encoders labeled as `encoder0` to `encoder5`, or just `enc0` to `enc5`.

## What do the boxes contain?

When you order the robot, you receive two packages.

The first package contains

- Robot arm.
- Robot cable(s) (cables connecting the robot arm to its controller box).

The other package contains

- Controller box.
- Ethernet cable.
- Power cable compatible with your region.

## Where to find more information

- For technical support, submit a query at: <https://dorna.ai/contact/>
- Tutorials: <https://dorna.ai/tutorial/>
- Programming examples: <https://github.com/dorna-robotics/example>
- Online version of this document is available [here](#).

## Safety

The robot body weighs less than **10 kg** however, it can move fast and cause injuries, especially when certain end-effectors are attached to its flange (e.g., a sharp tool or a laser). The robot also has pinch points where robot joints can squeeze a finger.

It is imperative that you follow the guidelines of ISO 12100:2010 and ISO 10218-2:2011 and conduct a risk assessment of your complete robot cell, including the Dorna robot, its end-effector, and all adjacent equipment.





### Note

- Handle the robot with care.
- The Dorna TA series is equipped with [passive brakes](#).
- Inspect the robot arm and its controller box for damages. If either appears damaged, do not use them and contact us immediately.
- Do not modify or disassemble the robot arm or the controller.
- Do not use or store the robot and its controller in a humid environment.
- Use the proper [AC power source](#), according to your robot setting, to operate the robot.
- If the robot tips and falls from a height, it may cause an injury and certainly get damaged.

## The brakes

[Dorna TA](#) model has a braking mechanism that activates when the power is abruptly disconnected from the robot, preventing crashing and damaging the robot or workpiece.



### Note

- The brakes are designed to slow down the free fall of the robot.
- Brakes are not designed for total position lock.

## Power supply

The robot power supply is compatible with **115/230 Vac**. However, before turning on the robot controller, you have to make sure that the right voltage is selected on the controller. Otherwise, it will damage the robot.

The robot comes with a preselected voltage based on the customer region and is printed on a label attached to the controller. Double-check the operating voltage printed on the controller box, and if you need to change it, follow the instruction below.

1. An opening is available on top of the controller box to access the power supply switch. Access the power supply red switch on top of the controller and change it according to your AC input.

## Robot cables

Always connect the robot cables before turning on the robot.

1. Attach one side of the robot DB-25 and DB-9 cable to the back of the robot base and the other side to the back of the controller. Ensure the connections are secure using the screws available on the DB shells.



### Note

- Always use the cables provided with the robot.
- Contact us if you need to change or extend the robot cables.
- Make sure that the robot cables are straightened and not twisted.
- Make sure the connections are secure using the screws on the robot cables.

## Motors

The robot does not have mechanical brakes. When a motor loses its power, it can cause the robot to collapse, fall and potentially damage itself and other surrounding objects. Here are some cases where the motors can lose power:

- Motions with high speed, acceleration, or jerk.
- Applying too much torque on the motors.
- Turning the controller off or losing power.
- Failure in the motors, cables, or drivers.
- When the robot hits an object.

## IO wiring

- Always run the wiring before turning on the robot.
- Always use shielded and flexible cables for IO wiring to minimize EMI noise and withstands continuous motion without degrading data or signal transmission.

## Operation environment

- Prevent water and dust from entering the robot arm or controller box.
- In a dusty environment, keep the controller box inside a cabinet with proper cooling and airflow.
- Do not operate the robot above 50° C (122° F).

- In the presence of high electromagnetic interference (EMI) noise, keep the robot controller inside a metallic cabinet and properly shield all the cables to reduce the EMI effect.

## Emergency stop

You can configure and set up an emergency stop push-button for the robot to stop the robot immediately and prevent it from executing any further commands.

The emergency button connects to the robot's digital inputs and sends the robot into an alarm mode when the associated input pin gets triggered.

- Visit the wiring section to learn how to connect the emergency stop button to the robot.
- Visit the emergency stop section in [Dorna Lab](#) to activate and configure this feature.

## Safety function

Dorna robots are equipped with special safety functions purposely designed to enable collaborative operation, where the robot system operates without fences and/or with a human. Collaborative operation is only intended for non-hazardous applications, where the complete application, including tool/end effector, workpiece, obstacles, and other machines, is without any significant hazards according to the risk assessment of the specific application.

When the robot motors are enabled, the robot's motion planner and closed-loop feedback system continuously adjust the robot's orientation. The safety function compares the actual and planned positions of the robot, activating an alarm and stopping the robot immediately if a certain amount of error (error threshold) persists for a specified duration (error duration). This helps the robot to detect collisions with an external object.

The sensitivity of the safety function can be configured based on your application and by adjusting the error threshold and the error duration parameter. The two parameters can be set via the PID commands or Dorna Lab.



### Note

- When increasing the error threshold or duration, it is good to consider that:
  - The robot becomes less sensitive to external forces.
  - More risk assessments are required as the robot is less sensitive to detect collisions.
  - The robot can operate at a higher speed or larger payload before entering an alarm.

- The robot can collide with an object without entering an alarm mode.
- When decreasing the error threshold or duration, it is good to consider that:
  - The robot becomes more sensitive to external forces and can go into an alarm mode easier.
  - This mode is recommended when operating the robot around a human or sensitive object.
  - Now, you must operate the robot at a lower speed or payload.

## Risk assessment

One of the most essential things an integrator needs to do is perform a risk assessment. In many countries, this is required by law. The robot itself has partly completed machinery, as the safety of the robot installation depends on how the robot is integrated (e.g., tool/end effector, obstacles, and other machines). It is recommended that the integrator uses ISO 12100 and ISO 10218-2 to conduct the risk assessment. Additionally, the integrator can use the technical specification ISO/TS 15066 as additional guidance. The risk assessment that the integrator conducts shall consider all work tasks throughout the lifetime of the robot application, including but not limited to

- Teaching the robot during set-up and development of the robot installation.
- Troubleshooting and maintenance.
- Normal operation of the robot installation.

A risk assessment must be conducted before the robot arm is powered on for the first time. A part of the risk assessment conducted by the integrator is to identify the proper safety configuration settings and the need for additional emergency stop buttons and/or other protective measures required for the specific robot application.

If the robot is installed in a non-collaborative robot application where hazards cannot be reasonably eliminated, or risks cannot be sufficiently reduced by using the built-in safety-related functions (e.g. when using a hazardous tool/end effector), then the risk assessment conducted by the integrator must conclude the need for additional protective measures (e.g., an enabling device to protect the operator during set-up and programming).

Dorna Robotics identifies the potential significant hazards listed below as hazards that the integrator must consider.

Note: Other significant hazards can be present in a specific robot installation.

1. Penetration of skin by sharp edges and points on the robot belts, joints, or tool/end effector.

2. Penetration of skin by sharp edges and sharp points on obstacles near the robot track..
3. Bruising due to contact with the robot.
4. Sprain or bone fracture due to strokes between a heavy payload and a hard surface.
5. Consequences due to loose bolts that hold the robot arm or tool/end effector.
6. Items falling out of tool/end effector, e.g., due to a poor grip or power interruption.
7. Mistakes due to different emergency stop buttons for different machines.
8. Mistakes due to unauthorized changes to the safety configuration parameters.

## Product specification

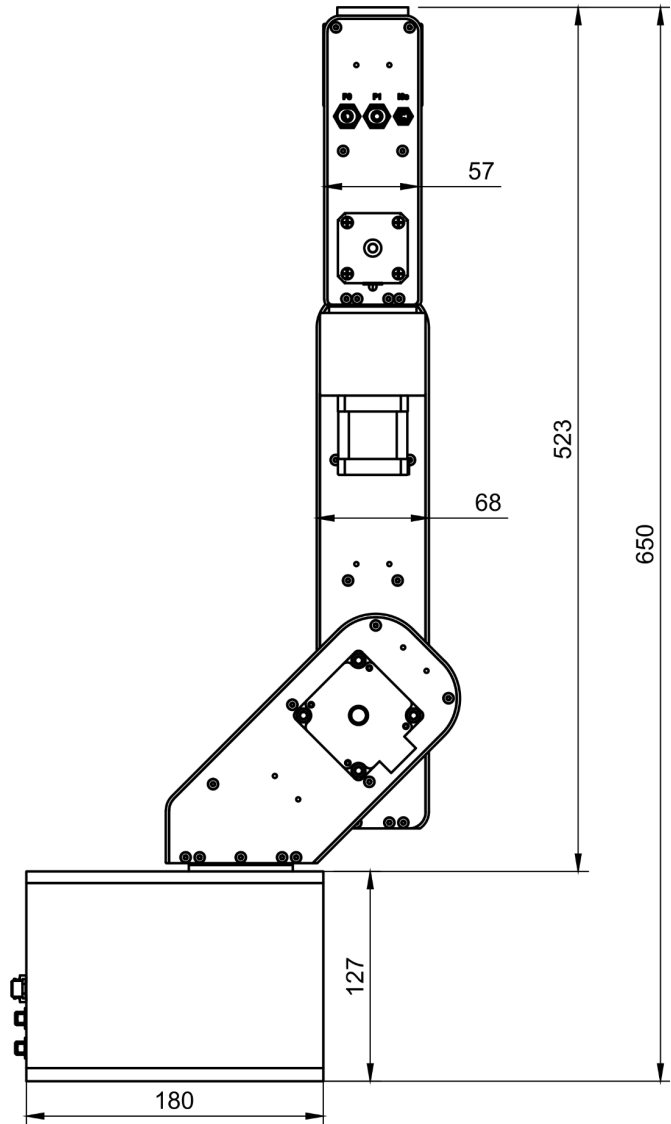
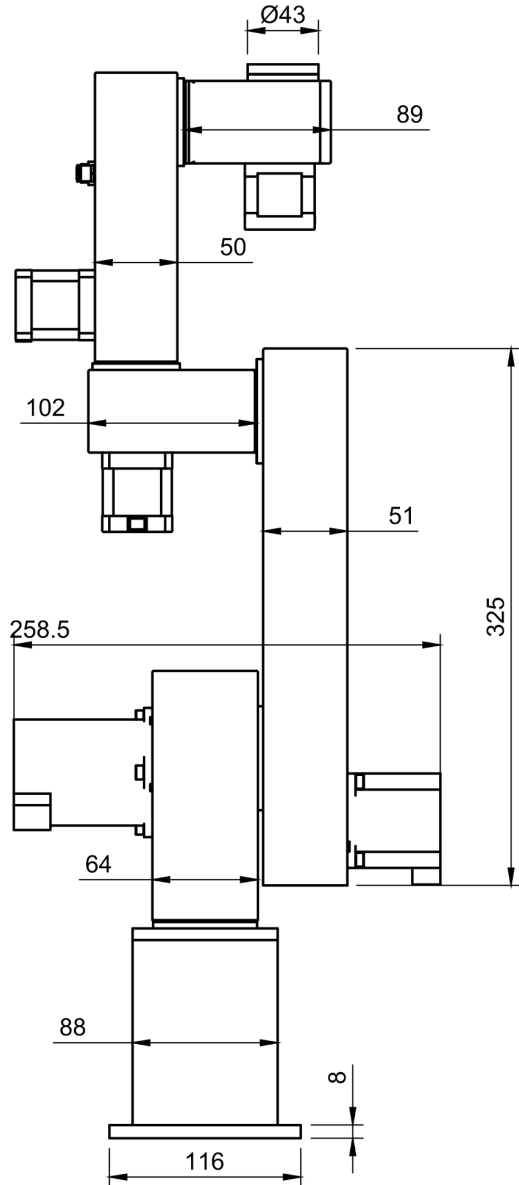
### Specs

<b>Model</b>	Dorna TA	
<b>Mechanical</b>		
<b>Payload</b>	2 kg / 3.3 lb (max 2 kg / 4.4 lb)	
<b>Reach</b>	500 mm / 19.7 in	
<b>Degrees of freedom</b>	6 rotating joints	
<b>Repeatability</b>	0.1 mm / 0.0039 in	
<b>Axis movement arm</b>	<b>Range of motion</b>	<b>Max speed</b>
<b>Axis 0</b>	[-160, 180]	180 deg/s
<b>Axis 1</b>	[-90, 190]	180 deg/s
<b>Axis 2</b>	[-150, 150]	180 deg/s
<b>Axis 3</b>	[-160, 170]	180 deg/s
<b>Axis 4</b>	[-170, 160]	180 deg/s
<b>Axis 5</b>	[-179, 179]	180 deg/s
<b>Tool speed</b>	1,000 mm/s / 39.37 in/s	
<b>Ecnoders</b>	Absolute (no homing required)	
<b>Brakes</b>	Passive braking circuitry	

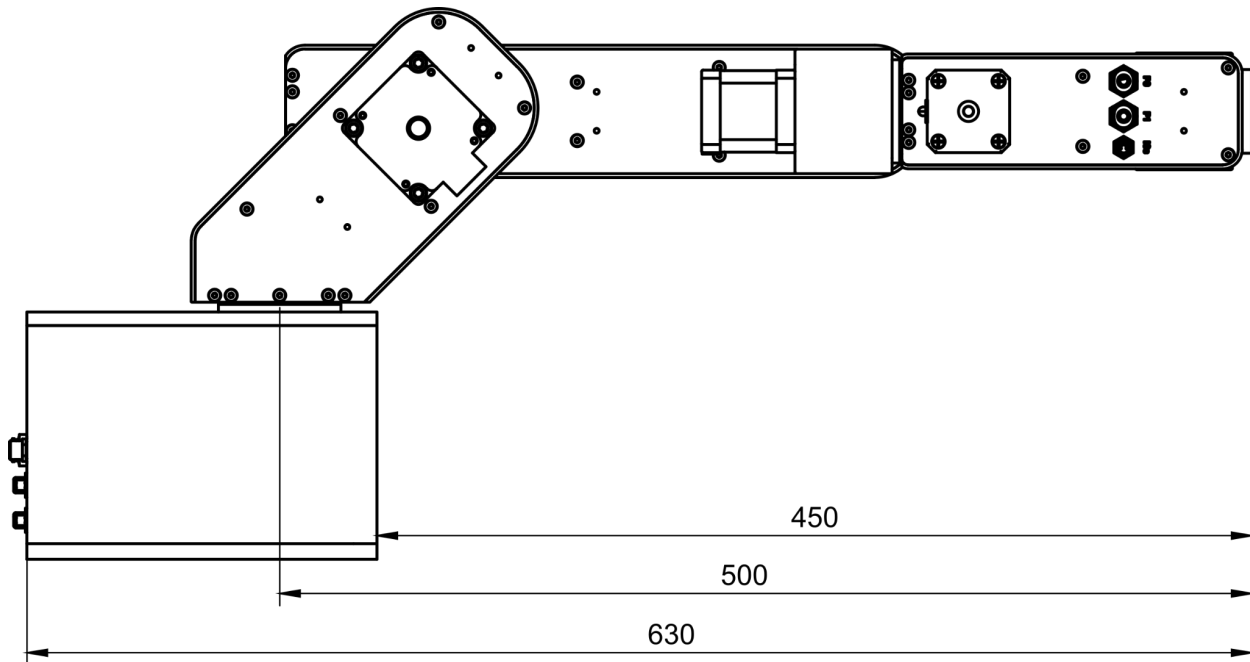
<b>Physical</b>	
<b>Robot arm body weight</b>	9.5 kg / 21 lb
<b>Robot arm footprint</b>	116mm X 180mm
<b>Robot arm material</b>	Aluminum
<b>Controller box weight</b>	3.8 kg / 8.4 lb
<b>Controller box dimension (W × H × D)</b>	280 x 120 x 220 mm / 11.02 x 4.72 x 8.66 in
<b>Controller box material</b>	Steel and plastic
<b>Robot cable length</b>	3 m / 118.11 in
<b>Attachment methods</b>	Floor / ceiling
<b>Robot arm IP classification</b>	IP54
<b>Working temperature</b>	0 - 50° C
<b>Inputs and outputs</b>	
<b>Digital inputs</b>	8 pins, 24 V
<b>Digital outputs</b>	8 pins, 24 V
<b>PWM</b>	2 channels, 3.3 V
<b>Total I/O power supply</b>	2 W
<b>Auxiliary axes</b>	2 axes (step/dir interface)
<b>Encoder inputs</b>	2 encoders (quadrature A, B, and X index, 3.3 V)
<b>Electrical</b>	
<b>Power supply</b>	100-240 VAC (selectable), 47-440 Hz
<b>Power consumption</b>	100 W using a typical program (350 W max)
<b>Interface</b>	
<b>Programming</b>	Dorna Lab (web interface) Python API and Jupyter notebook

	Blockly (drag and drop)
	Socket API
<b>Communication</b>	Gigabit Ethernet (TCP/IP)
	2.4 GHz and 5.0 GHz IEEE 802.11ac wireless
	2 x USB 3; 2 x USB 2.0

## Drawings







## Transportation

Only transport the robot in its original packaging. Save the packaging material in a dry place if you want to move the robot later.

When moving the robot from its packaging to the installation space, hold all the joints and ensure they are not falling. Hold the robot in place until all mounting bolts are securely tightened at the robot's base.

## Quick Start

This is a quick instruction on how to set up the robot and run a simple program. You still need to review the manuals for more information:

- **Mount the robot:** Mount the robot arm securely on a flat, sturdy surface with 4 x M5 screws.
- **Connect the robot cables:** Make sure that the controller box is close enough to the robot and you can attach the robot cables to it.
  - **Dorna TA:** Attach one side of the robot DB-25 and DB-9 cable to the back of the robot base and the other side to the back of the controller. Ensure the connections are secure using the screws available on the DB shells.
- **Connect the robot to a router:** Attach the robot controller to a router via an Ethernet port available on the robot controller. Make sure that your computer is also connected to the same router.

- **Turning on the robot:** The robot is compatible with **115/230Vac**. However, before turning on the controller, you have to make sure that the right voltage is selected on the controller. Otherwise, it will damage the robot. The robot comes with a preselected voltage based on the customer region and is printed on a label attached to the controller box. Please verify the operating voltage and make any necessary adjustments before proceeding. Please follow the instructions below if you need to change the operating voltage. Once you have selected the appropriate AC voltage, connect one end of the AC power cable to the back of the controller box, and connect the other to the AC plug.
  - **Dorna TA:** Access the power supply red AC input switch on top of the controller, and change it according to your AC input.
- **Find the robot IP address:** Use IP scanner software ([example](#)) to find the IP address of the robot attached to the router. For simplicity of the notation, we assume that the IP address of the robot is **10.0.0.14** in this tutorial.
- **Open Dorna Lab:** Dorna Lab is more or less the equivalent of the teach pendant's interface of a traditional industrial robot. Open a [Google Chrome](#) on your computer and type in the Dorna Lab URL at [http://robot\\_ip\\_address](http://robot_ip_address), where the **robot\_ip\_address** is the IP address of the robot we found in the previous section (for example, if the IP address of the robot is **10.0.0.14** then the Dorna Lab address is <http://10.0.0.14>).
- **Run a simple program:** In Dorna Lab, navigate to the Script section under the Main tab. Here is where you can write commands and submit them to the robot.
  - Copy and paste the following commands in the script section.
  - Enable the **Track Line** option to see which lines run during the operation.
  - Make sure that no tool is attached to the robot and that the robot is free to move without hitting any object.
  - Click the **Play** button to submit and run the code to the robot.

Unset

```
{"cmd": "motor", "motor": 1}
{"cmd": "jmove", "rel": 0, "j0": 0, "j1": 10, "j2": -10, "j3": 0, "j4": 0, "j5": 0, "vel": 50, "accel": 500, "jerk": 2000}
{"cmd": "lmove", "rel": 1, "x": -150, "vel": 100, "accel": 500, "jerk": 2000, "cont": 1, "corner": 20}
{"cmd": "lmove", "rel": 1, "y": 150}
{"cmd": "lmove", "rel": 1, "z": 150}
{"cmd": "lmove", "rel": 1, "y": -300}
{"cmd": "lmove", "rel": 1, "z": -150}
```

```
{"cmd": "lmove", "rel": 1, "y": 150}
```

- **Resting the robot:** If the robot is not operational (between the shifts), you can put the robot in rest, by putting the robot in a safe orientation and turning the motors off (via Dorna Lab). This way, the robot controller is still running, and you just de-energize the motors.
- **Turning off the robot:** When turning off the robot, all the motors lose their power. So, it is important to put the robot in a safe orientation before turning off the robot. Dorna TA is equipped with passive brakes, which slow down the robot when the motors are off. When ready, just switch off the power button on the controller to turn off the robot.

## Mechanical interface

### Robot arm mounting

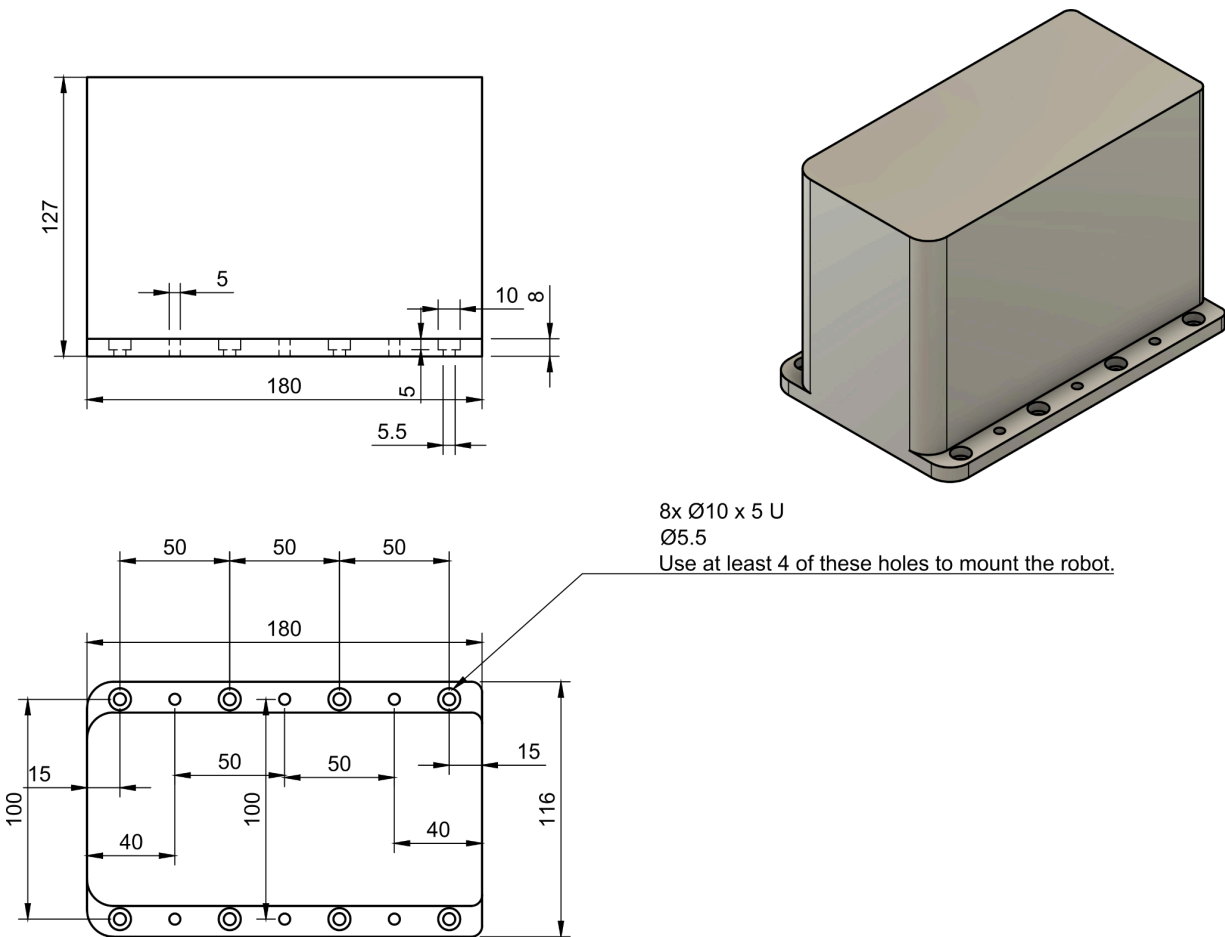
Install and operate the robot either on a floor or to a ceiling. The robot's base has 6 counterbore clearance holes which can be used with **M5** screws. The screws must be tightened with **20 Nm** torque.

Mount the robot on a sturdy, vibration-less surface that can withstand at least ten times the full torque of the base joint and at least five times the weight of the Robot Arm. If the robot is mounted on a linear axis or a moving platform, then the acceleration of the moving mounting base is very low. A high acceleration might cause the robot to make an [alarm](#) stop.



#### Note

Make sure the robot arm is properly and securely screwed in place. Unstable mounting can lead to accidents.



Dorna TA mounting holes. Use at least 4 x M5 screws. All measurements are in mm.

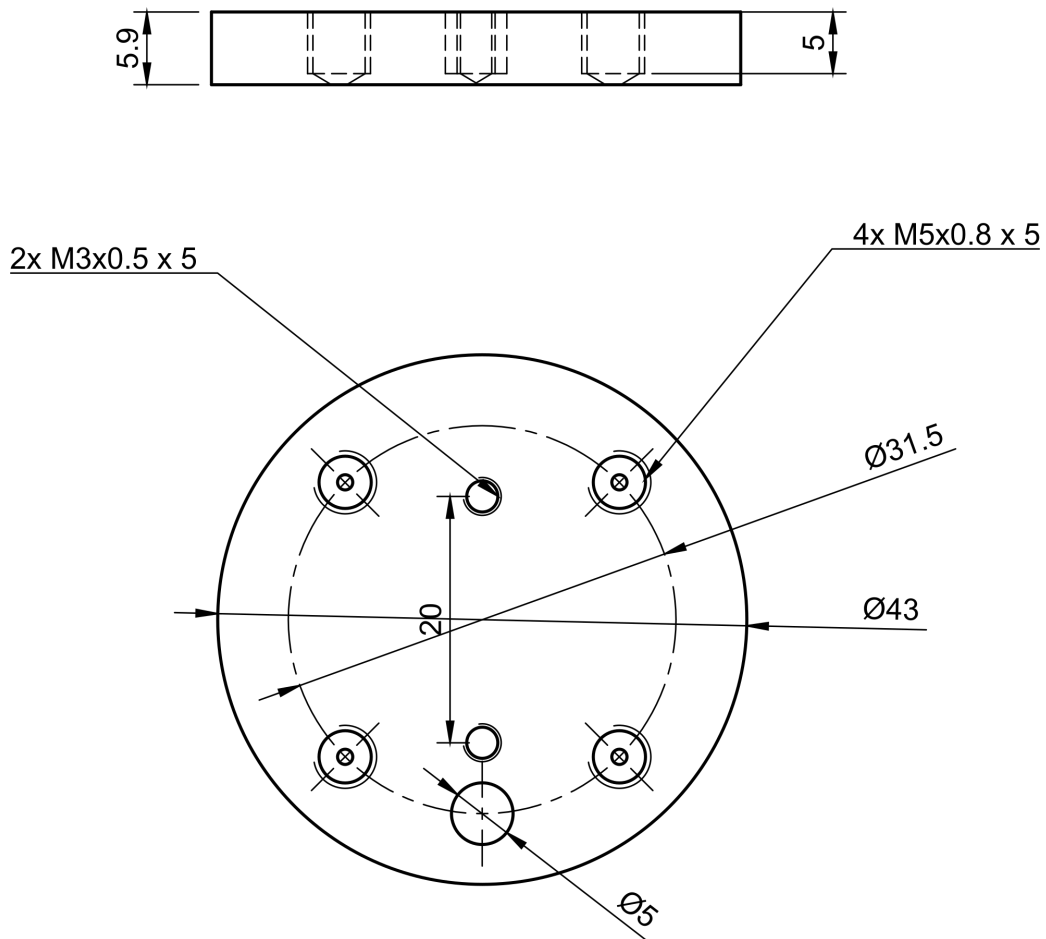
## Tool mounting

You can mount different tools (end effectors) to the robot flange four M5 tapped holes on the robot flange. For compatibility with existing tools that we offer with Dorna 2 series robots, there are two M3 tapped holes on the robot flange that could be used for tool mounting as well. Do not use bolts that extend beyond 5 mm to mount the tool. Very long screws can press against the bottom of the tool flange and damage the robot.



### Note

- Ensure that the tool is properly and securely screwed in place.
- Use at least two screws to attach the tool securely.
- Ensure that the tool is constructed so that it cannot create a hazardous situation by dropping a part unexpectedly.
- Mounting a tool on the robot with screws extending beyond 6 mm can push into the flange and cause irreparable damage, leading to end joint replacement.



Dorna TA tool output flange is where the tool is mounted at the robot's tip. Dorna TA follows the All measurements are in **mm**.

## Controller box installation

The controller box can be placed on the ground. A clearance of **50mm** on each side of the controller box is needed for sufficient airflow.



### Note

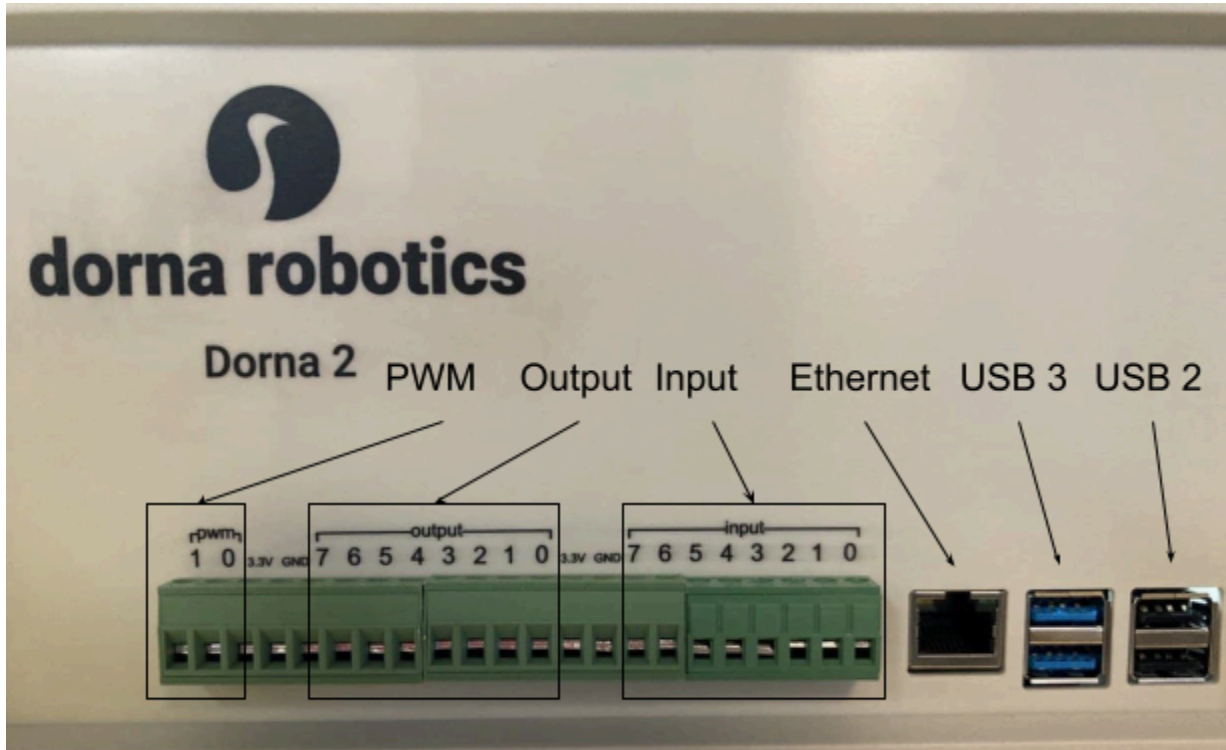
- Make sure the controller box and cables do not come into contact with liquids.
- A wet controller box could cause fatal injury.
- Place the controller box (IP44) in an environment suited for the IP rating.

## Electrical Interface

### Introduction

This chapter describes electrical interface groups for the robot. The main electrical interface groups are listed below:

- Controller I/O
- Ethernet port
- USB ports



## Electrical warnings and cautions

- Voltage beyond the specified range of the I/O channel can potentially harm the controller board.
- Ensure the controller is powered off when connecting an I/O device to avoid potential damage. Connecting an I/O device while the controller is operational may harm the controller.
- To minimize electromagnetic interference (EMI) noise on the I/Os, it is crucial to use shielded cable and ensure proper grounding of the shield to the controller box ground pin.
- To avoid potential malfunctions, utilize relays and an external power supply instead of relying solely on the controller I/Os to power and control external devices. It is necessary to opt for [MOSFET](#) or solid-state [relays](#) instead of magnetic relays, as the latter can draw excessive current from the controller, leading to controller crashes.

## Controller box I/O

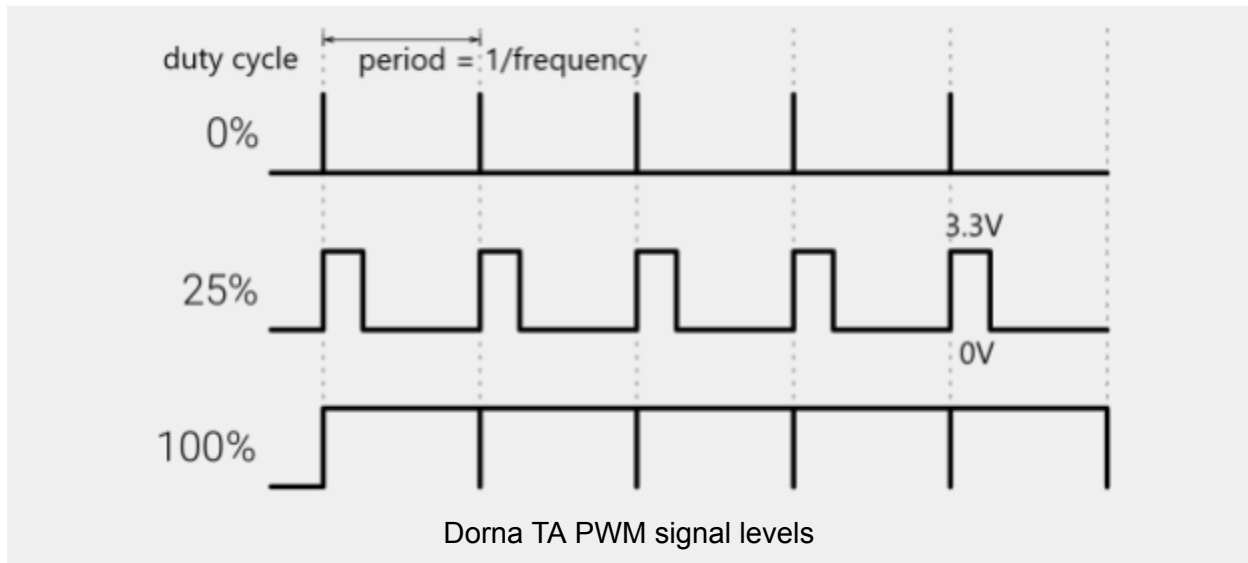
On the front panel of the controller box, the I/O pins are accessible through a 22-position terminal block header connector with male pins. The pitch of the connector is **5.08 mm**. Use a compatible terminal block plug to connect your wires to the I/O pins ([sample](#)).

There are multiple I/O connections available on the controller box and the robot arm.

- I/O power supply shared between the digital inputs, outputs, and PWM channels is **2W** in total.
- The total current on all the I/O pins should not exceed **83 mA**.

Type	Description
Digital inputs	8 pins, 24 VDC
	<ul style="list-style-type: none"> <li>• Labeled as <b>in0, ..., in7</b>.</li> <li>• With approximately <b>100 KHz</b> update rate, the controller sends a message upon any change in their values, reporting the new value.</li> <li>• Use the <a href="#">input command</a> to read the value of input pins.</li> <li>• Voltage level <b>0</b> at an input pin corresponds to digital value <b>0</b>, and voltage level <b>24 VDC</b> at an input pin corresponds to digital value <b>1</b>.</li> </ul>
Digital outputs	8 pins, 24 VDC
	<ul style="list-style-type: none"> <li>• Labeled as <b>out0, ..., out7</b>.</li> <li>• Use the <a href="#">output command</a> to read or set the output pins.</li> <li>• Digital value <b>0</b> appears as <b>0 VDC</b>, and digital value <b>1</b> appears as <b>24 V DC</b> at output pins.</li> <li>• At the startup, all output pins are initialized to <b>0</b>.</li> </ul>
PWM channels	2 pins, 5 / 3.3 VDC
	<ul style="list-style-type: none"> <li>• Labeled as <b>pwm0, pwm1</b>.</li> <li>• PWM pins generate a pulse width modulated signal with a specific frequency (<b>freq</b>) and a specific duty cycle (<b>duty</b>).</li> <li>• Use the <a href="#">PWM command</a> to read or set the PWM channels.</li> <li>• Voltage level of PWM signals for Dorna TA is <b>3.3 V</b></li> </ul>

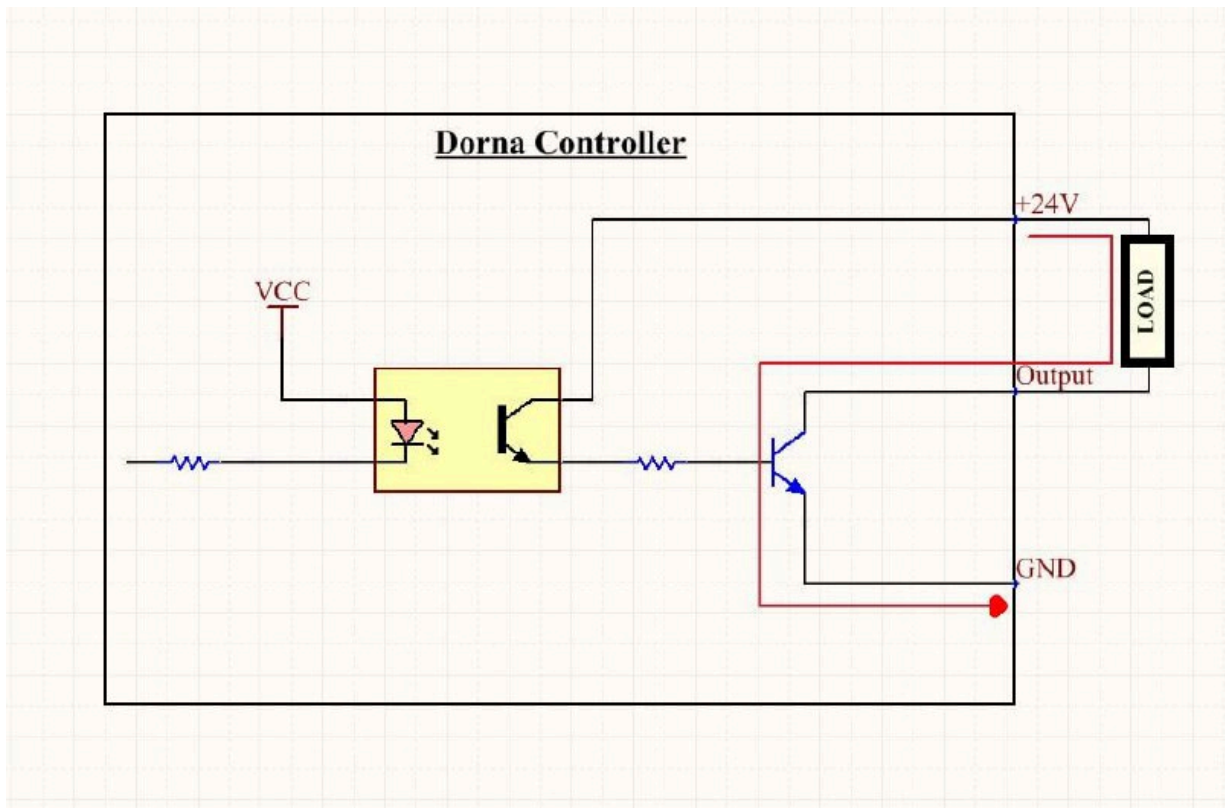




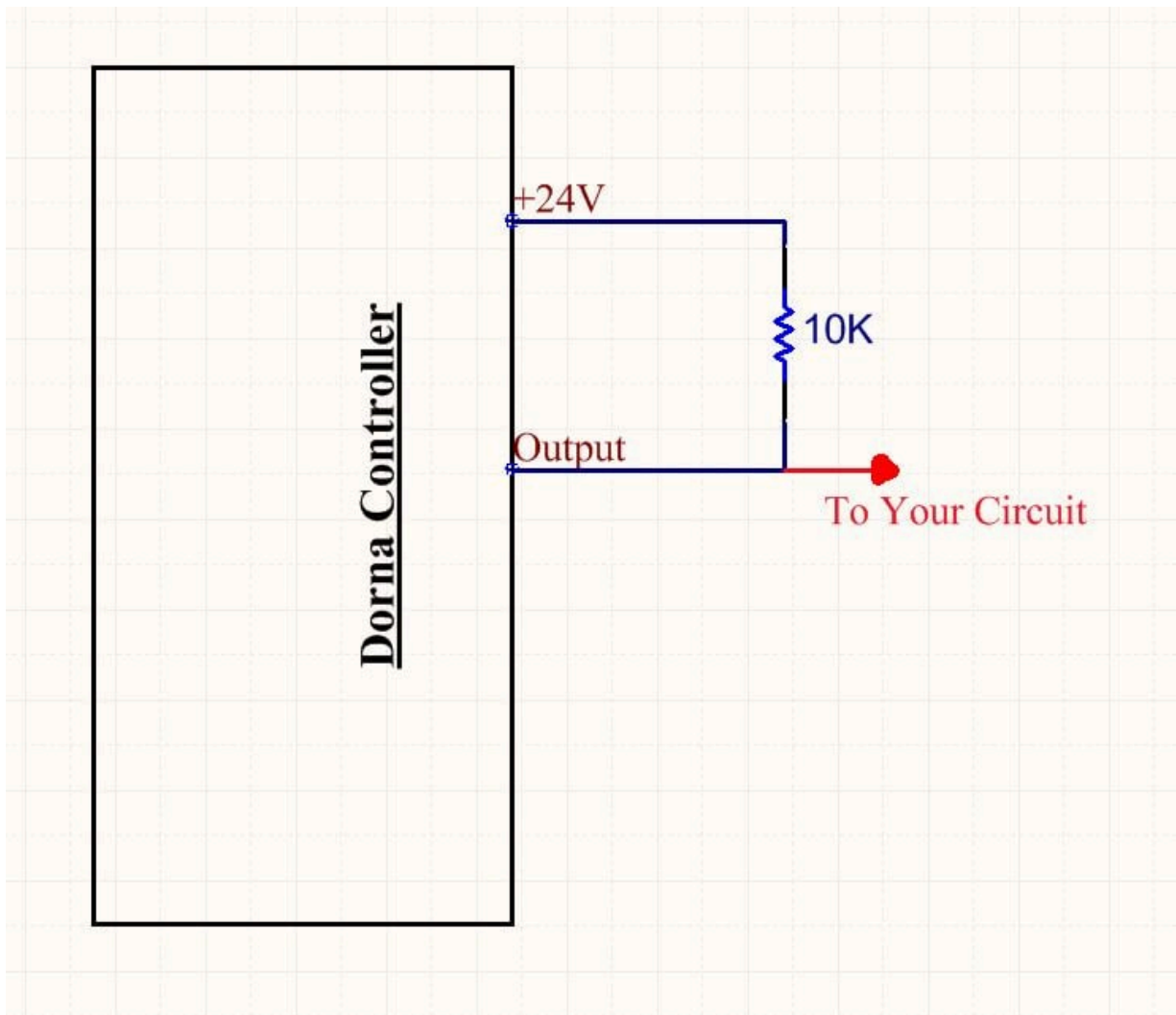
## IO wiring

### Output wiring

The output pins of the front panel of the Dorna TA are optocoupler-isolated open-collector type. Here we explain how the output pins work and how you must connect your load. The I/O pins of the main processor inside the controller box are isolated via optocouplers from the I/O pins available to the user to protect the main processor against external electrical shocks or noise. Transmitting a zero on the output pin turns on the LED inside the optocoupler, enabling the optocoupler output. When the optocoupler output is enabled, the current flows from the DC power source (+24V) through the Load and a transistor to the DC power GND. This current will turn the load on. This way, the output pins of the controller can be used to drive and open/close DC loads. Each output uses a transistor to drive the load with sufficient current. The following diagram depicts the output circuit.



The internal 2W power supply is responsible for providing the current on the output pins of the front panel; the total current on all the output pins of the front panel should not exceed **83 mA**. The red arrow demonstrates the current path when the output is zero. When the output is equal to the logic one, there would be no current, and consequently, the load would be off. This kind of output is common in industrial sensors and is known as NPN output. Therefore to turn on the load, you should set the output pin to 0 and to turn it off you should set the output to 1. If you intend to use the output as a signal (Not turning a load on/off), you should pull up the output pin to the (+24V) with a pull-up resistor (a 10K ohm resistor is sufficient for most applications). In this case, the voltage measured on the output pin will be 24V when the output is set to 1 and 0V when the output is set to 0. The following image shows the connection of the pull-up resistor to the output pin.



## Input wiring

Input signal is connected through an optocoupler to the main processor of the Dorna controller. Any voltage between 0 V and 12 V is interpreted as logic signal 0, and a voltage from 12 V and 24 V is interpreted as logic signal 1.

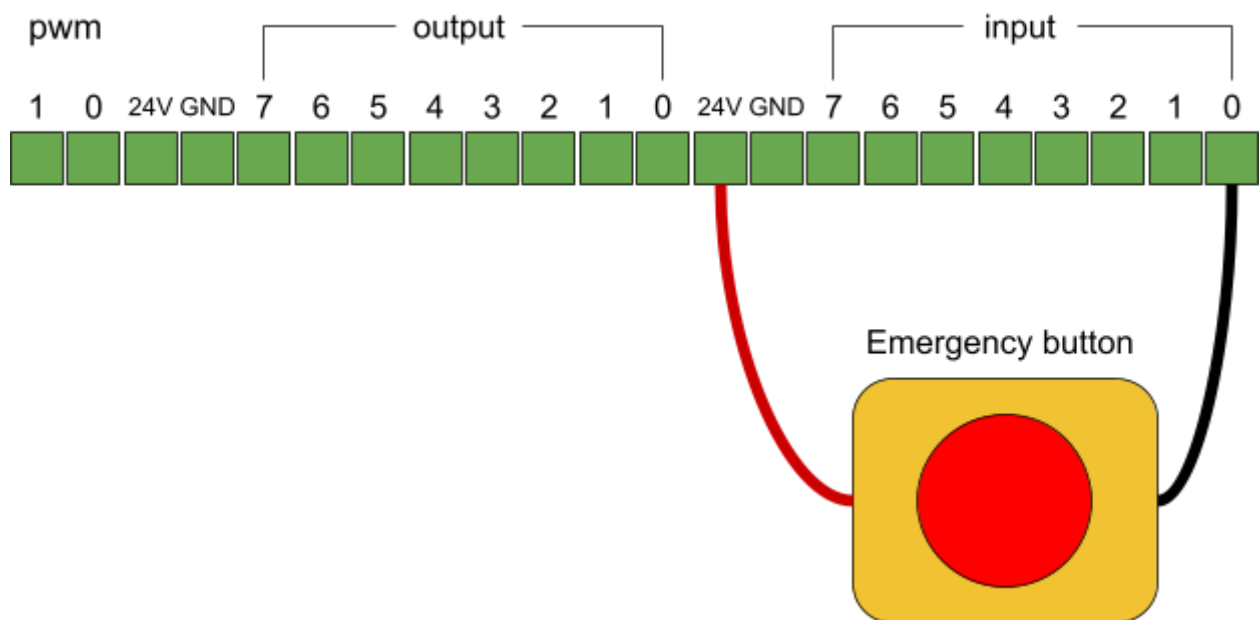
We highly recommend using shielded wires to connect the input pins and grounding the cable shield to the controller box ground.

## Sample wiring

The I/Os can be used to drive equipment like pneumatic relays directly or for communication with other PLC systems. Examples below depict some frequent scenarios that IO pins are used in automation applications:

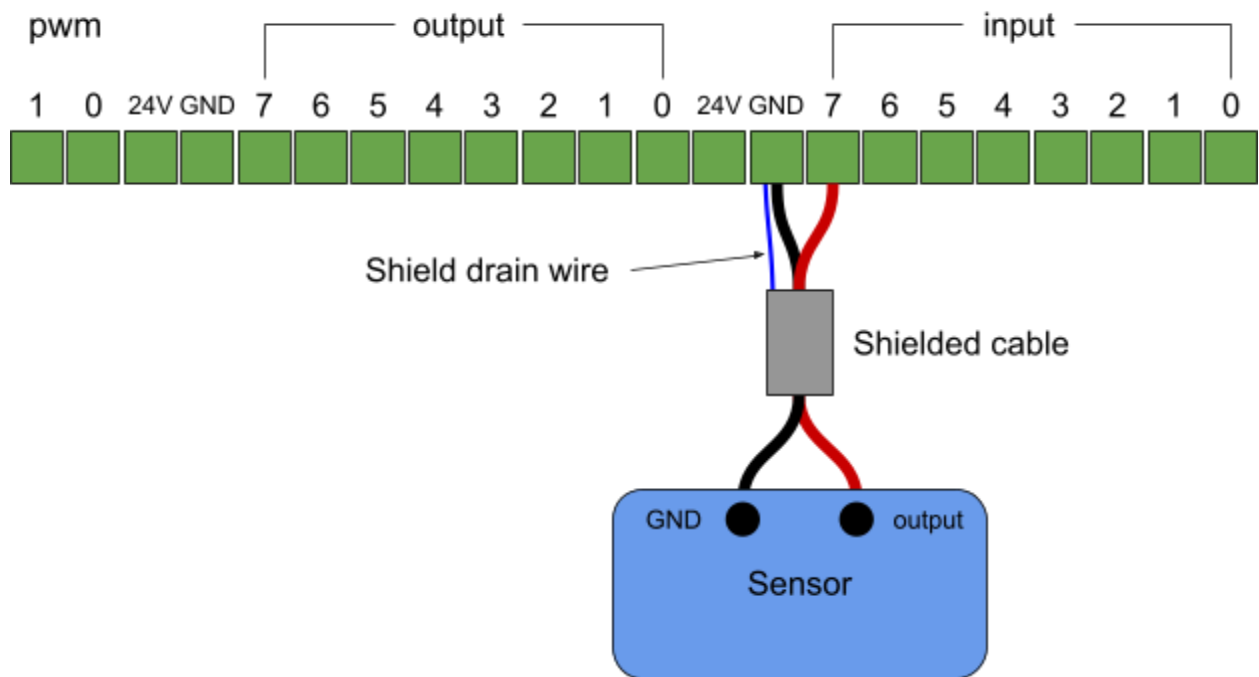
### Digital inputs from a button or Emergency Button

This example illustrates connecting a simple button or an emergency button to a Digital input pin.



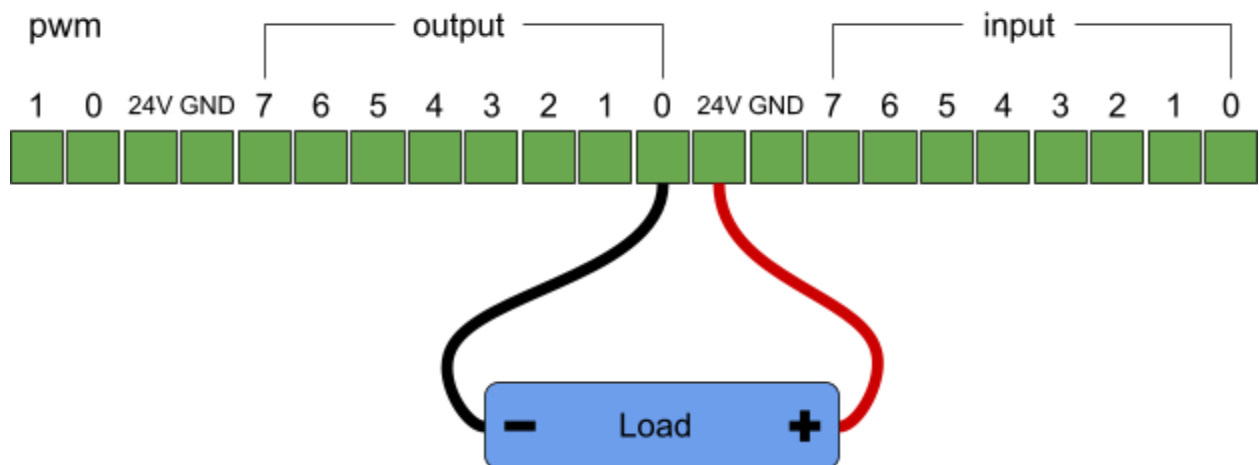
### Digital Inputs from a sensor, PLC, or another device's digital output

We use a shielded cable to connect an output pin of a sensor, or a PLC or another controller device to an input pin of Dorna controller. We will need to establish a common ground between the two devices and connect the shield of the cable to the controller ground.



### Load controlled by a digital output

This example shows how a load is controlled by Digital outputs when connected. The load could be a small DC motor or an LED light or other load that does not consume power less than the maximum power of the controller board (2W).



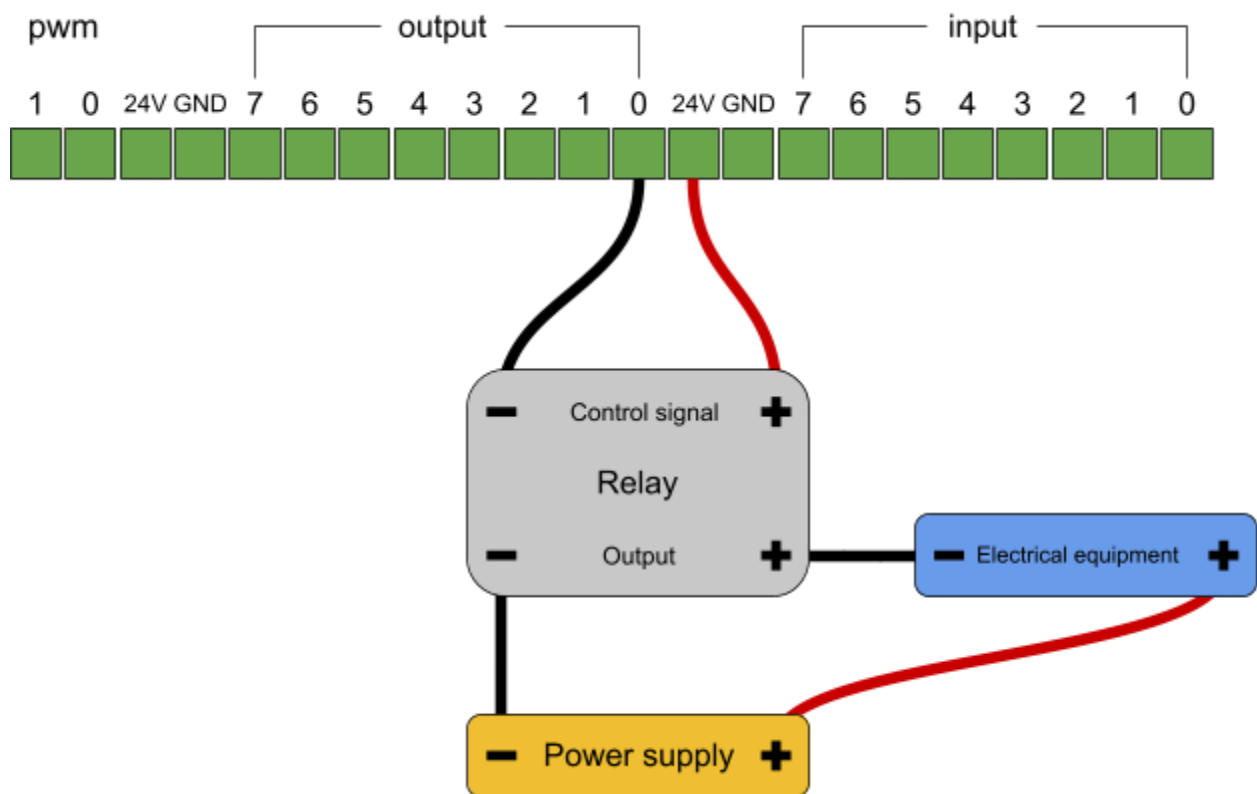
## Digital output pin to a relay

In many scenarios, the user would like to control a high-power load, such as an actuator, or electric magnet, with a digital output pin using a relay circuit. We recommend using Mosfet relays as they consume very little power and have a fast response time. The connection will be as follows.



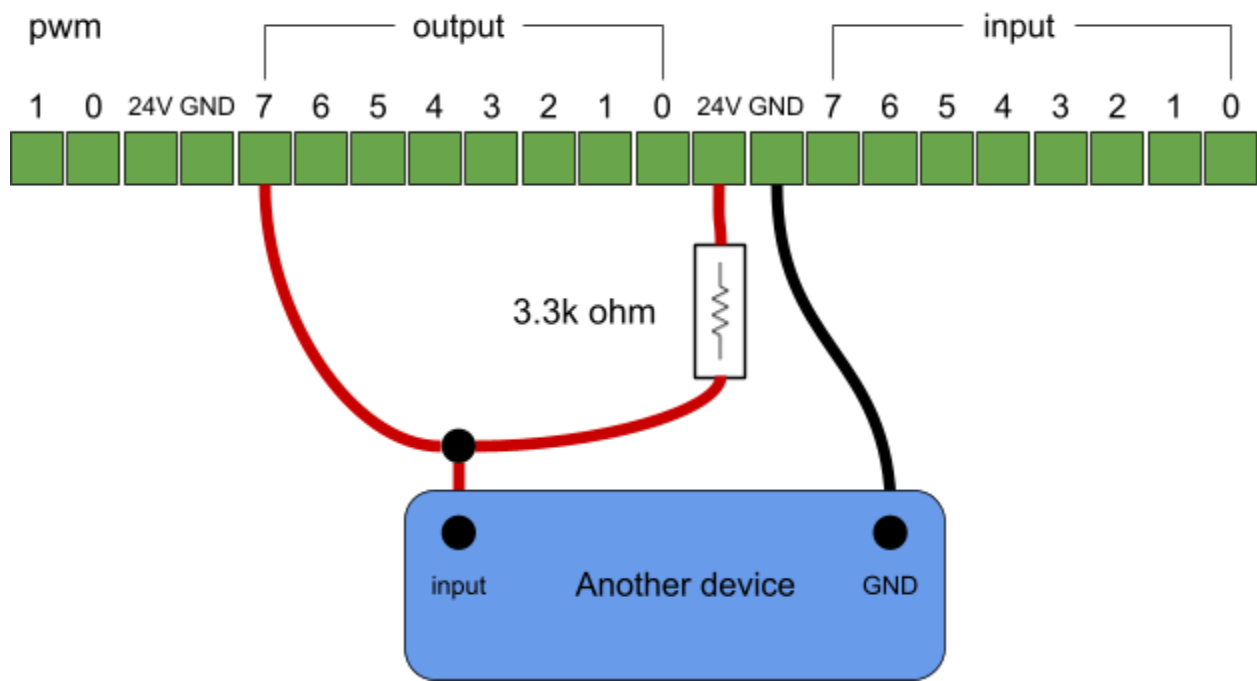
### Note

It is necessary to opt for [MOSFET](#) or [solid-state](#) relays instead of magnetic relays, as the latter can draw excessive current from the controller, leading to controller crashes.



## Digital output pin to an input pin of another machine or PLC

You can use the digital I/O to communicate with other equipment if a common GND (0V) is established and if the machine uses PNP technology, see below. Here we show how the output of the controller box can be connected to the input pin of another controller using a pullup resistor. You can use the same wiring to connect an output pin of the Dorna controller to an input pin of the Dorna controller (you will not need to establish a common ground for this configuration as both pins share their ground internally).



## TCP network

In order to monitor and control the robot, you need to establish a TCP connection with the robot.

## Credentials

The default hostname, username, and password for the robot controller is **dorna**

```
Unset
# hostname
dorna
```

```
#username  
dorna  
  
# password  
dorna
```

## Robot connection methods

Any device connected to a Local Area Network is assigned an IP address. In order to connect to the robot controller ([Dorna Lab](#), [WebSocket](#), [SSH](#), [API](#), remote access, etc.), we need the IP address of the robot controller. Many methods and online resources discuss this issue ([example](#)). Here we discuss two main common ways of setting up the robot TCP network and finding the robot IP address:

- [Connect the robot to a router](#)
- [Directly connect the robot to your computer via Ethernet cable](#)

### Connection through a router

Connect the robot controller to a router and then the robot becomes visible to all the devices connected to that router. We recommend using an Ethernet cable for the connection unless you have set up the robot [WiFi](#) to connect it to your wireless router. Follow the steps below to find the local IP address of the robot.

#### Method 1

1. Open the **Command Prompt (CMD)** on your Windows computer or **Terminal window** on your UNIX device and try all the commands below (one of them usually works) to find the IP address assigned to the robot:

```
Unset  
ping dorna.local -4  
ping dorna.home -4  
ping dorna -4
```



2. This will ping the robot using its hostname (`dorna`). If the ping is successful, you should see the IP address of the robot in the output.

Method 1 works in most cases. If method 1 fails (for example when multiple Dorna controllers are connected to the same router), then use the next method to find the IP address of the robot.

### Method 2

1. Connect your computer to the same router as the robot is connected, and find your computer's local IP address [using the instructions listed here](#).
2. For simplicity and better understanding, we assume that your computer's local IP address is `10.0.0.7` (this is just an example, and your computer IP address can be totally different). Then the robot controller IP address is in the form of `10.0.0.x`. Where `x` is any number from `0` to `255` (your computer already takes `7`).
3. Use any of the following methods to find the list of all the devices connected to the router:
  - a. IP scanner software: Download and install an IP scanner software ([example](#)) on your computer and put `10.0.0.0-255` to search for all the devices connected to the same router as your computer is connected.
  - b. Router devices list: In a web browser, navigate to your router's IP address (usually the ends with digit `1`, so in this case: <http://10.0.0.1>), which is usually printed on a label on your router; this will take you to a control panel. Then log in using your credentials, usually printed on the router or sent to you in the accompanying paperwork. Browse the list of connected devices or similar (all routers are different)
4. Once you have the list of all the devices connected to your router, search for a device with `dorna` in their name (it can be `dorna`, `dorna.local`, etc. In some cases, routers show the robot name under `Raspberry Pi`) and find its IP.

### Direct connection to a computer

In this section, we will provide a step-by-step guide to establish connection between the robot controller and your Windows computer using an Ethernet cable. One end of the cable should be connected to an Ethernet port on your Windows computer, while the other end should be plugged into the Ethernet port of the robot controller box.



## Note

The method below is for a Windows computer, but the procedure is very similar on other operating systems.



The robot controller is connected to a computer via an Ethernet cable.

1. Go to the **Network Connections** page on your Windows computer. You can access this page by going to your **Control Panel > Network and Sharing Center > Change adapter settings**.
2. Next, go to **Wi-Fi > Properties > Sharing**, check both options in the **Internet Connection Sharing** section, and select the proper Ethernet connection the robot connects to under the **Home networking connection**. Click **OK** and close the **Wi-Fi Status** page. If you applied any changes in this section, then it is recommended to restart your computer before going to the next steps.

3. Next, on the **Network Connections** page, go to the **Ethernet** (the one that the robot is connected to) > **Properties** > **Internet Protocol Version 4 (TCP/IPv4)**. Select **Obtain an IP address automatically** and **Obtain DNS server address automatically**.
4. Power on your robot.
5. Wait a few seconds for the robot to obtain an IP address from your computer.
6. Open the **Command Prompt (CMD)** on your Windows computer or **Terminal window** on your mac device and type the following command to check the IP address assigned to the robot:

Unset

```
ping dorna.local -4
```

7. This will ping the robot using its hostname (**dorna**). If the ping is successful, you should see the IP address of the robot in the output.

That's it! Your robot is now connected to your Windows computer via Ethernet and has been assigned a dynamic IP address.

## Static IP

The robot OS utilizes DHCP (Dynamic Host Configuration Protocol) to automatically assign an IP address to the robot whenever it is rebooted. Having a static IP address for your robot ensures that you can always find it at the same fixed IP address rather than a dynamic address that changes every time the device is rebooted. It also helps avoid confusion when multiple robots are connected to your network.

To set a static IP address for your robot, [SSH](#) to your robot controller and follow these steps:

1. Determine your network setup:
  - a. Identify the type of network connection: **wlan0** for wireless or **eth0** for Ethernet. We recommend using **eth0** as it is a more reliable communication with the robot.
  - b. Find the robot's assigned IP address using the **hostname -I** command. It's safest to reuse this for its static IP so that you can be sure the latter hasn't already been to another device on the network. If not, make sure another device isn't already using it.

```
dorna@dorna: ~  
dorna@dorna:~$ hostname -I  
192.168.254.170  
dorna@dorna:~$
```

- c. Locate your router's gateway IP address using the `ip r | grep default` command. The gateway IP address varies depending on the router model but typically starts with `192.168`.

```
dorna@dorna: ~  
dorna@dorna:~$ ip r | grep default  
default via 192.168.254.254 dev eth0 proto dhcp src 192.168.254.170 metric 202  
dorna@dorna:~$
```

- d. Note your router's DNS IP address using the `sudo nano /etc/resolv.conf` command. This is typically the same as its gateway address but may be set to another value to use an alternative DNS – such as `8.8.8.8` for Google or `1.1.1.1` for Cloudflare.

```
dorna@dorna: ~
GNU nano 5.4 /etc/resolv.conf
# Generated by resolvconf
domain home
nameserver 192.168.254.254

^G Help      ^O Write Out  ^W Where Is   ^K Cut        ^T Execute    ^C Location   M-U Undo
^X Exit      ^R Read File  ^\ Replace    ^U Paste      ^J Justify    ^_ Go To Line  M-E Redo
```

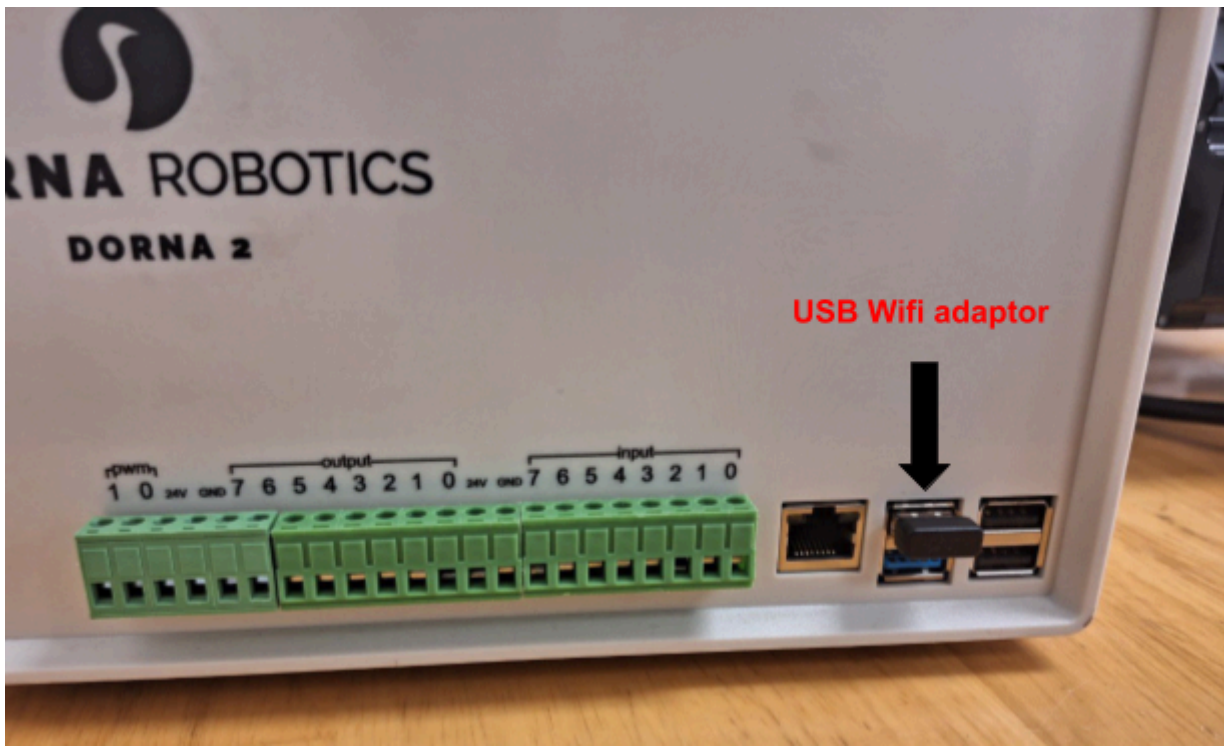
Note the IP address after **nameserver** – the DNS address – and then press **Ctrl + X** to close the file.

2. Edit the **dhcpd.conf** file inside the robot:
  - a. Open the file using the **sudo nano /etc/dhcpd.conf** command.
  - b. Add the following lines at the bottom of the file, replacing the emboldened names with your own network details:

```
Unset
interface NETWORK
static ip_address=STATIC_IP/24
static routers=ROUTER_IP
static domain_name_servers=DNS_IP
```

- c. **NETWORK**: The type of network connection is **wlan0** for wireless and **eth0** for Ethernet.
- d. **STATIC\_IP**: The desired static IP address for the robot.
- e. **ROUTER\_IP**: The gateway IP address for your router on the local network.
- f. **DNS\_IP**: The DNS IP address (typically the same as your router's gateway address).





There are multiple ways to configure the WiFi, and we cover one of them here:

1. [SSH](#) to your robot controller.
2. In the command line, type in `sudo raspi-config`, and this will bring you to your controller configuration tool.
3. Navigate to **Network Options** (usually listed as number 2) and then **Wireless LAN** (usually listed as N2).
4. Follow the steps and set the **SSID** and **Password** of your wireless network.

## SSH

SSH has been enabled by default on the robot controller. Once you have [the IP address](#) of the robot controller, open a terminal (cmd) and use the `SSH` command and the robot IP address to SSH to the robot.

Unset

```
ssh dorna@robot_ip_address
```

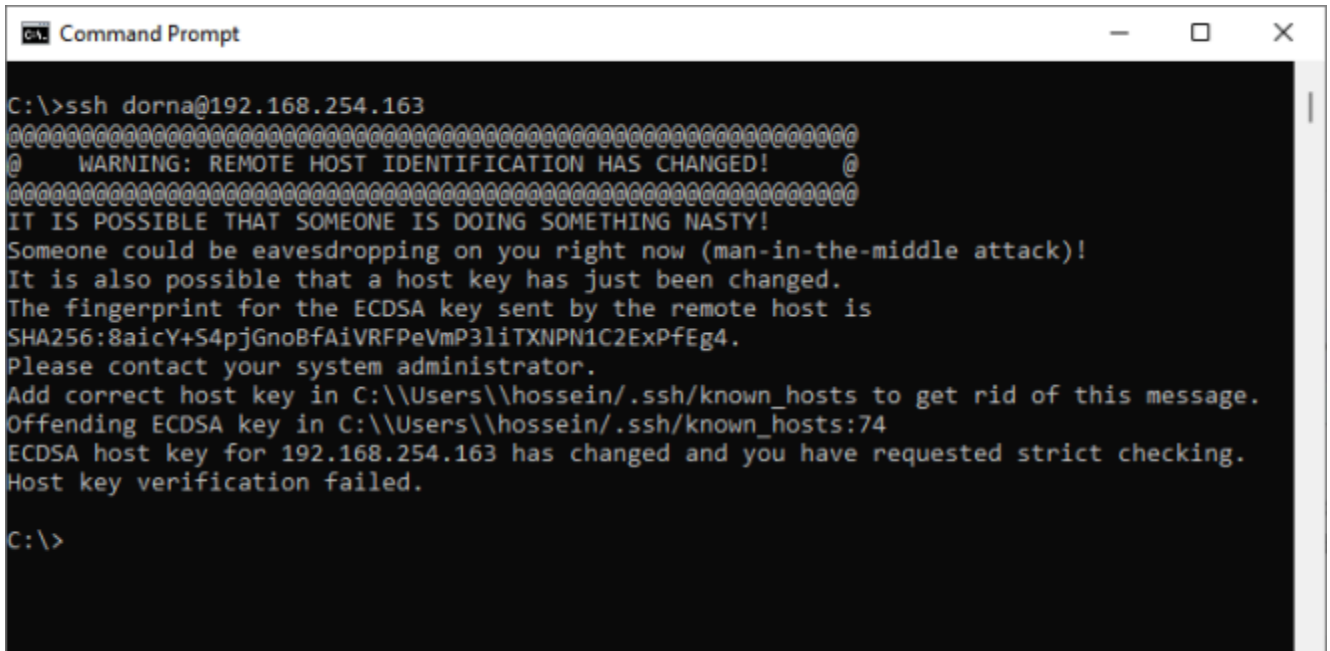


## Note

Notice that the default hostname, username, and password for the robot controller is [dorna](#).

## SSH error

Sometimes, when we run the SSH command, we get an error like this:



```
Command Prompt
C:\>ssh dorna@192.168.254.163
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@  WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!  @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:8aicY+S4pjGnoBfAiVRFPeVmP3liTXNPN1C2ExpFEg4.
Please contact your system administrator.
Add correct host key in C:\\Users\\hossein/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in C:\\Users\\hossein/.ssh/known_hosts:74
ECDSA host key for 192.168.254.163 has changed and you have requested strict checking.
Host key verification failed.

C:\>
```

To solve this issue, run the following command and try to **SSH** again.

```
Unset
ssh-keygen -R robot_ip_address
```

For example, in this case, we run the following command (assume that the robot ip is **10.0.0.14**):



Unset

```
ssh-keygen -R 10.0.0.14
```

You can now **SSH** to the robot again.

## Dorna Lab address

Once you have the robot IP address, you can access [Dorna Lab](#) by typing the robot IP address in a browser `http://robot_ip_address` (Chrome browser is recommended). For example

Unset

```
# Robot IP address is 10.0.0.14  
http://10.0.0.14
```

## Jupyter Notebook address

Once you have the robot IP address, you can access the Jupyter Notebook server in a browser at `http://robot_ip_address:8888` (Chrome browser is recommended). For example

Unset

```
# Robot IP address is 10.0.0.14  
http://10.0.0.14:8888
```

## WebSocket server address

The controller [WebSocket server](#) runs automatically when the controller is turned on. Connect to the WS server via port **443** of the robot and its IP address (hostname also works sometimes).

The WS URL is `ws://robot_ip_address:443`. For example,

Unset

```
# Robot IP address is 10.0.0.14  
ws://10.0.0.14:443
```

## Internet access

The robot does not require internet access during its normal operation. The only time the internet access is required is during the upgrade process of the robot. If the robot is [connected to a router](#), then it is connected to the internet if the router also has access to the internet. If the robot is [connected directly to your computer](#), then it has access to the internet if the computer itself is connected to the internet.

## Upgrade process



### Note

Before running the upgrade process, make sure that:

- The robot controller has access to the [internet](#).
- All your sessions and files on the controller are saved since you need to turn off the controller at some point.
- The robot is in a safe and stable physical position, and no program is running since the motors lose power during the upgrade.

## Check for new updates

To check for the new software updates, make sure that the robot controller has [access to the internet](#), then:

1. Open [Dorna Lab](#) via browser (If the Dorna Lab page didn't appear, then you probably need an update)
2. Go to the **Setting > Info**.
3. Under the Device section, click on **Check for Updates** and wait for the robot to check the latest updates by connecting to the Dorna website.
4. If new updates are available, then follow the [upgrade process](#) and run the updates.

## Software upgrade

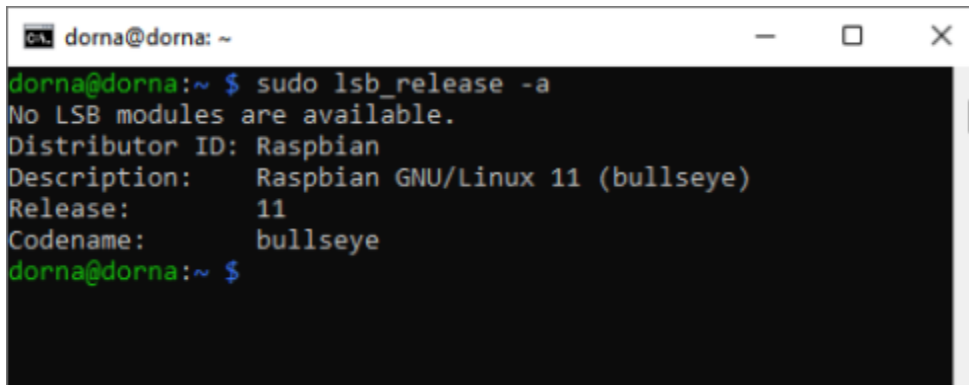
Follow the steps below to get the latest software (firmware, API and Dorna Lab) for the robot:

1. [SSH](#) to the robot controller and run the following line to find the OS version

Unset

```
sudo lsb_release -a
```

If the **Release** item is **10** or less (old OS), then you need to first [upgrade the OS](#).  
Otherwise, continue



```
dorna@dorna: ~  
dorna@dorna:~ $ sudo lsb_release -a  
No LSB modules are available.  
Distributor ID: Raspbian  
Description:    Raspbian GNU/Linux 11 (bullseye)  
Release:       11  
Codename:      bullseye  
dorna@dorna:~ $
```

2. Run the following lines (copy/paste all) in the terminal and press **Enter** to run. Depending on your internet speed, the process can take a few minutes.

Unset

```
sudo rm -rf /home/dorna/Downloads/upgrade && sudo mkdir  
/home/dorna/Downloads/upgrade && sudo git clone -b dorna_ta  
https://github.com/dorna-robotics/upgrade.git  
/home/dorna/Downloads/upgrade && cd /home/dorna/Downloads/upgrade  
&& sudo sh setup.sh dorna_ta
```

3. Once the upgrade process is completed, you should see the following messages on the terminal window. Notice that the progress percentage should reach 100% at the end.



## OS upgrade



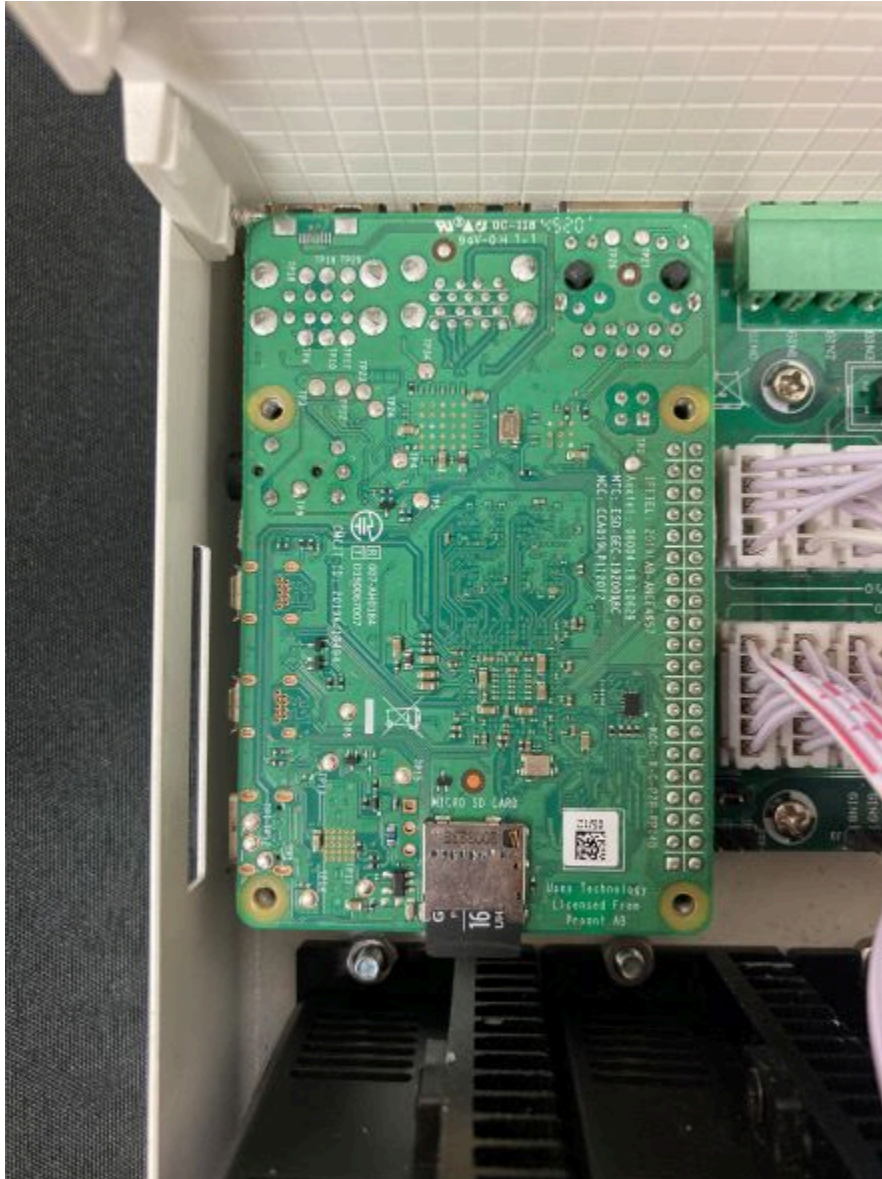
### Note

You will lose all your data on the controller after the OS upgrade. So, upgrade the robot's OS only if the OS is corrupted or outdated (version 10 or less),

We cover two ways to upgrade your OS:

1. [Fresh OS on the SD card](#): Take out the SD card from the controller box and install a new OS on it. For some users, this might be difficult as they need to open the controller box.
2. [Upgrade the current OS](#) (recommended): This method usually takes longer, but it does not require opening the controller box and taking the SD card out.

## Fresh OS image



Follow the steps below to access the controller SD card and install a fresh OS on it:

1. Disconnect the power cable from the controller box.
2. Unscrew the 8 screws on the top lid of the controller box (4 screws are located on each side of the top lid) and remove the top lid. You should now see the Raspberry Pi and the SD card inside it.

3. Unscrew the 4 screws that connect the Raspberry Pi to the controller main board. Notice that the Raspberry Pi pins are connected to the main board. So, gently take the Raspberry Pi out and disconnect it from the main board.
4. Now you have access to the SD card and take it out.
5. Connect the SD card to your own computer, [navigate to the Raspberry Pi software page](#), and download the Raspberry Pi Imager.
6. After downloading and installing the imager. Open the software, on the **CHOOSE OS** option, select the **Raspberry Pi OS (32-bit)** (first option). On the **CHOOSE STORAGE** option select the SD card connected to your computer. Before clicking on the **WRITE**, hold **CTRL + SHIFT + X** to open the **Advanced Options** page.
7. In the **Advanced Options** page:
  - a. Check the Set hostname option and put `dorna` as the hostname.
  - b. Check **Enable SSH** option and select **Use password authentication**.
  - c. Check the **Set username and password** option and put `dorna` for both username and password.
  - d. You can leave the rest unchanged, and click on the **SAVE** button.
8. Click on **WRITE**. This usually takes a few minutes.
9. Once the writing process is over, put back the SD card to the Raspberry Pi. Connect the Raspberry Pi to the board and make sure that its pins seat perfectly on the controller board. Tighten the Raspberry Pi and top lid screws. Now Jump to the [software upgrade](#) section to install the Dorna software on your robot.

## Upgrade current OS



### Note

In this section we will update the version of the OS from 10 (buster) to 11 (bullseye). This process can take up to 1 hour.

## Expand the file system

[SSH](#) to the robot controller and run `df -h /` to check the disk space:

Unset

```
dorna@dorna:~ $ df -h /  
Filesystem      Size  Used Avail Use% Mounted on
```

```
/dev/root      3.8G  3.0G  636M  83% /
```

If the **Size** item is below **4G** then follow this part, otherwise jump to the [installation section](#).

Expand the file system by going over the following steps

1. Run `sudo raspi-config`
2. Go to **Advanced Options**
3. Select **Expand Filesystem**
4. Select **OK**
5. Select **Finish**
6. Select **Yes** to reboot (or run `sudo reboot` manually)

Step by step installation of bullseye

1. [SSH](#) to the robot controller and update the repository lists:

```
Unset
$ sudo apt update
```

2. Install all of the latest packages (and their dependencies):

```
Unset
$ sudo apt full-upgrade
```

3. Reboot your robot controller to activate any packages that require a reboot:

```
Unset
$ sudo reboot
```

4. Update to the latest version of the OS firmware:



Unset

```
$ sudo rpi-update
```

Another reboot may be necessary after updating your OS firmware.

Unset

```
$ sudo reboot
```

- Next, edit your `sources.list` file to switch your repository list from `Buster` to `Bullseye`:

Unset

```
$ sudo nano /etc/apt/sources.list
```

Locate the following line and change `buster` to `bullseye`:

Unset

```
deb http://raspbian.raspberrypi.org/raspbian/ buster main contrib  
non-free rpi
```

Save your changes (`Ctrl+X`) and proceed to the next step.

- Update your repository lists again (this time it will be using the Bullseye-specific repositories):

Unset

```
$ sudo apt update
```

- Install the latest version of `Node.js` (this helps avoid an error you may receive when running `apt full-upgrade` in step 9). This step may take several minutes.

Unset

```
$ sudo apt install nodejs
```

Note that you may receive a prompt to **Restart services during package upgrades without asking**. If so, choose **Yes**.

8. Update to the latest version of GCC 8 (this also helps avoid an error in the next step).

Unset

```
$ sudo apt install gcc-8-base
```

9. Run another full upgrade to install any additional Bullseye requirements and downstream dependencies:

Unset

```
$ sudo apt full-upgrade
```

10. Clean up your packages to remove any that are obsolete or no longer used:

Unset

```
$ sudo apt autoclean  
$ sudo apt autoremove
```

11. Next, you need to alter your `/boot/config.txt` file to enable KMS (the new standard video driver).

Unset

```
$ sudo sed -i  
's/dtoverlay=vc4-fkms-v3d/#dtoverlay=vc4-fkms-v3d/g'  
/boot/config.txt
```

```
$ sudo sed -i 's/\[all\]/\[all\]\ndtoverlay=vc4-kms-v3d/'  
/boot/config.txt
```

12. When you're all done, reboot your robot controller:

```
Unset  
$ sudo reboot
```

13. After a reboot, check to see if everything was updated properly:

```
Unset  
$ cat /etc/os-release
```

This should return the following, with **bullseye** as the new OS:

```
Unset  
PRETTY_NAME="Raspbian GNU/Linux 11 (bullseye)"  
NAME="Raspbian GNU/Linux"  
VERSION_ID="11"  
VERSION="11 (bullseye)"  
VERSION_CODENAME=bullseye  
ID=raspbian  
ID_LIKE=debian  
HOME_URL="http://www.raspbian.org/"  
SUPPORT_URL="http://www.raspbian.org/RaspbianForums"  
BUG_REPORT_URL="http://www.raspbian.org/RaspbianBugs"
```

## Update the time zone

1. [SSH](#) to the robot controller and run

Unset

```
sudo dpkg-reconfigure tzdata
```

2. Select your region to configure your time zone.

## Update the configuration

1. [SSH](#) to the robot controller and run

Unset

```
sudo nano /boot/config.txt
```

Append the following lines to the end of the file,

Unset

```
dtoverlay=pi3-miniuart-bt  
enable_uart=1
```

Save the file and exit the editor.

2. Modify the serial port by running

Unset

```
sudo raspi-config
```

Then:

- a. Select **Interfacing Options**
- b. select **Serial Port**
- c. select **No**
- d. Select **Yes**
- e. Select **OK**
- f. Finish and Reboot

## Motion concepts

In this section we will go over different types of motion that can be achieved by the robot. We will explain the basic concepts that you need to understand before choosing the right motion types and parameters for your application.

### Coordinate system

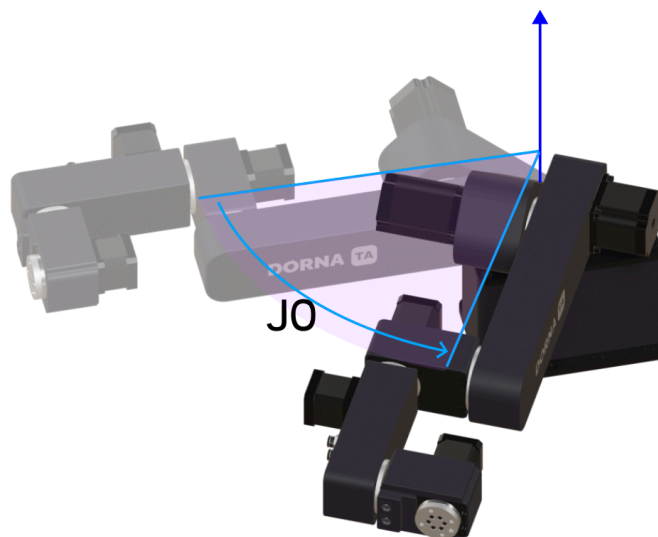
The first concept in the design of a motion for a robotic arm is the coordinate system of the robot. Dorna (and most other robotic arms) work in two main different coordinate systems. The first one is the **joint** coordinate system (joint space) and the second one is the **Cartesian** coordinate system (Cartesian space). Motion commands can be sent in either space and it is a user's choice to decide which one to work with.

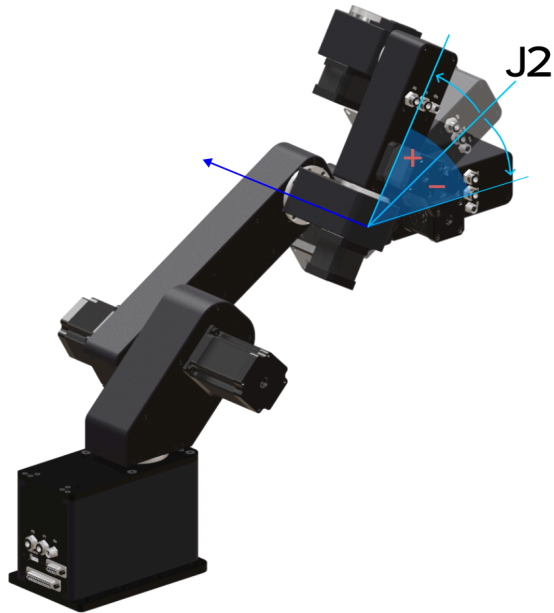
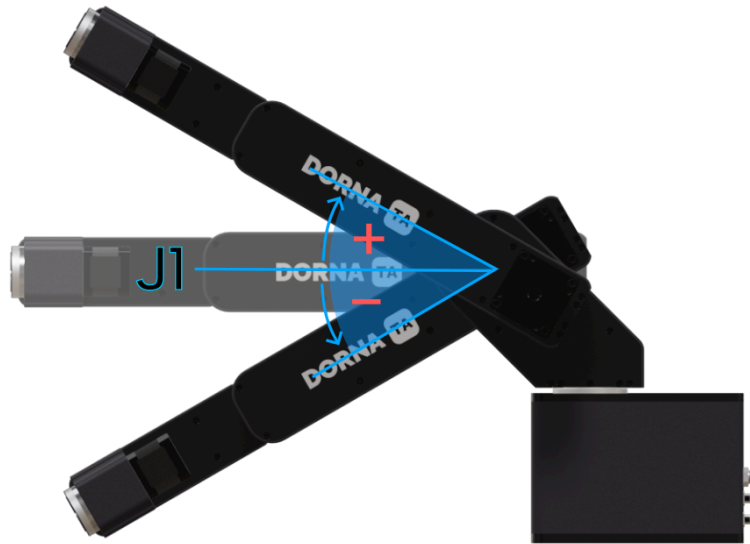
### Units

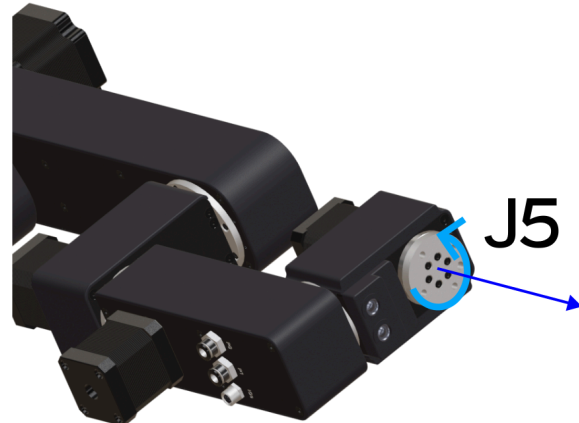
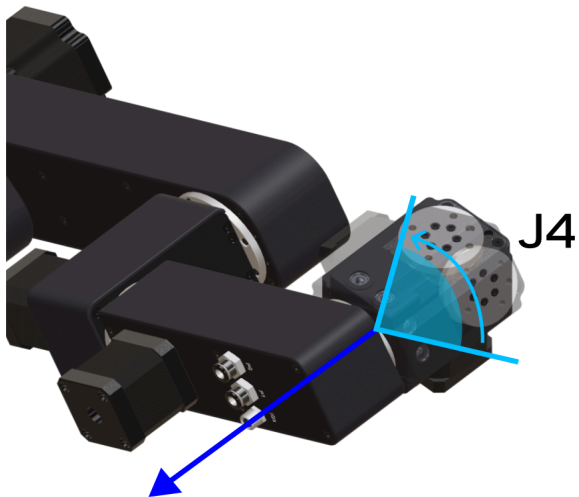
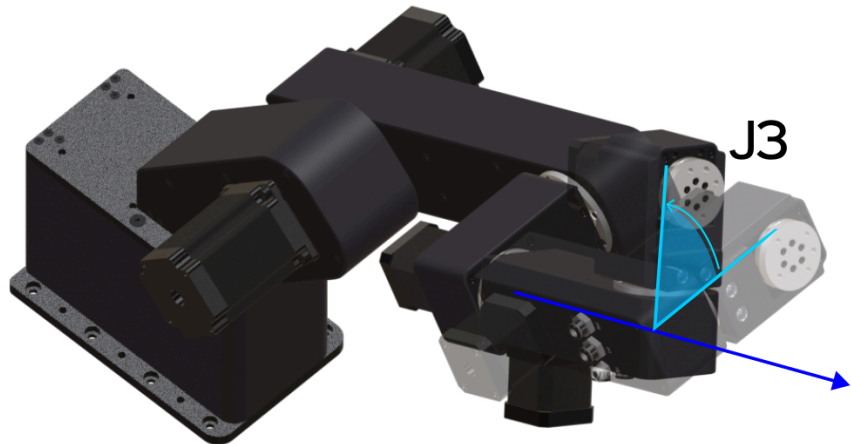
Type	Value	symbol
Time	Seconds	s
Angles	Degrees	deg
Distance	Millimeters	mm

### Joint assignment

The joints of the robot are numbered in ascending order, starting from 0, as shown here.







### Joint limit

Each joint in the robot has a limited range of rotary motion, known as joint limit (upper and lower limits).



## Note

- Depending on the robot's orientation, tools attached to the robot, and the robot workspace some additional limits may apply to each joint.

## Reference frames

We use right-handed Cartesian coordinate systems (reference frames). These four reference frames and the key terms related to them are

### BRF: Base reference frame

Static reference frame fixed to the robot base. Its z-axis coincides with the axis of joint 1 and points upwards, its origin lies on the bottom of the robot base, and its x-axis is normal to the base front edge and points forward.

### WRF: World reference frame

The main static reference frame coincides with the BRF by default.

### FRF: Flange reference frame

Mobile reference frame fixed to the robot's flange.

### TRF: Tool reference frame

The mobile reference frame is associated with the robot's end-effector. By default, the TRF coincides with the FRF.

### TCP: Tool center point

Origin of the TRF.

## Joint space

Joint space or joint coordinate system is the representation of the robot orientation using the joint values of the robot. We use degrees as the unit of each joint. The robot has 5 joints (axes) and 3 additional auxiliary axes that can be controlled by the robot (8 joints in total). The robot joints are denoted by  $j_0$ ,  $j_1$ ,  $j_2$ ,  $j_3$ ,  $j_4$ ,  $j_5$  and  $j_6$ ,  $j_7$  for the auxiliary axes.

The value (angle in degrees) associated with joint  $i$  ( $i = 0, 1, \dots, 5$ ),  $j_i(j_0, j_1, \dots, j_5)$ , will be referred to as  $j_i$

- $j_0$ : Rotation of BRF around the z-axis of C0 to align the x-axis of BRF and C0.



- **j1**: Rotation of C0 around the z-axis of C1 to align the x-axis of C0 and C1.
- **j2**: Rotation of C1 around the z-axis of C2 to align the x-axis of C1 and C2.
- **j3**: Rotation of C2 around the z-axis of C3 to align the x-axis of C2 and C3.
- **j4**: Rotation of C3 around the z-axis of C4 to align the y-axis of C3 and C4.
- **j5**: Rotation of C4 around the z-axis of C5 to align the y-axis of C4 and C5.

Joints 6, and 7 are used for auxiliary axes and can be set for linear rail, z-axis, conveyor belts, or any other motor or encoder connected or controlled by the robot.

## Assigning values to the joints

Setting the robot joints, or the homing process, is the process of identifying the real value of the joints and assigning them to the robot. Multiple hard stops are available on the robot and can be used to identify the true value of a joint. In the homing process, we put the robot in a specific orientation, where the value of each joint is known to us.



### Note

The robots are pre-calibrated and equipped with absolute encoders. So, the homing process is not required. Use the homing process only if you need to re-calibrate the robot.

In the [Dorna lab](#), disable the motors, and navigate to the **set joint** section. All joints 0 to 5 will be set to zero when their value is set in this section. Therefore, you will need to set joint values only when the corresponding joint is at 0 position.

## Auto assigning values to the auxiliary axes

If the auxiliary axis of the robot is equipped with a quadrature encoder and an index channel, the setting of joints can be automated. Here's a summary of the process:

1. Access the **Set Joint** panel in Dorna Lab.
2. Select the desired auxiliary axis for setting its joint.
3. Enter the new value for the joint and click on the motion direction.
4. The associated auxiliary axis starts moving in the specified direction until it reaches its hard stop.
5. The axis then moves backward until the encoder index channel triggers and goes high for the first time.
6. The robot assigns the value in the **New Value** field to the position where the index was triggered.



## Note

- This is a very accurate way of setting the value of an auxiliary axis.
- Make sure the auxiliary axis has a hard stop in the direction of the motion for the setting joint process.
- Make sure that the auxiliary axis is not touching its hard stop initially when running the process.
- Make sure the encoder comes with an index channel (I).
- The process uses `iprobe` command to detect the encoder index and assign the new value to the axis.
- During the process you will see an alarm appears and disappears. Do not interrupt the process, until the joint assignment is completed.

## Cartesian coordinates

The Cartesian coordinate system corresponds to the Cartesian coordinates of the robot tool head with respect to the origin point which is at the bottom on the axis of rotation of  $j0$  in the three-dimensional space. The coordinates are presented as  $x, y, z, a, b, c$  for the robot and  $d, e$  for the auxiliary axes. The first three coordinates of the robot are the  $x, y, z$  which are the respective coordinate values in millimeters.  $a, b, c$  represent the rotation vector of the robot head respect to the robot base in degrees. The definition of the rotation vector can be found here:

[https://en.wikipedia.org/wiki/Axis%E2%80%93angle\\_representation](https://en.wikipedia.org/wiki/Axis%E2%80%93angle_representation)

For the auxiliary axes also their value in Cartesian space is the same as their value in joints space. Therefore,  $d = j6, e = j7$ .

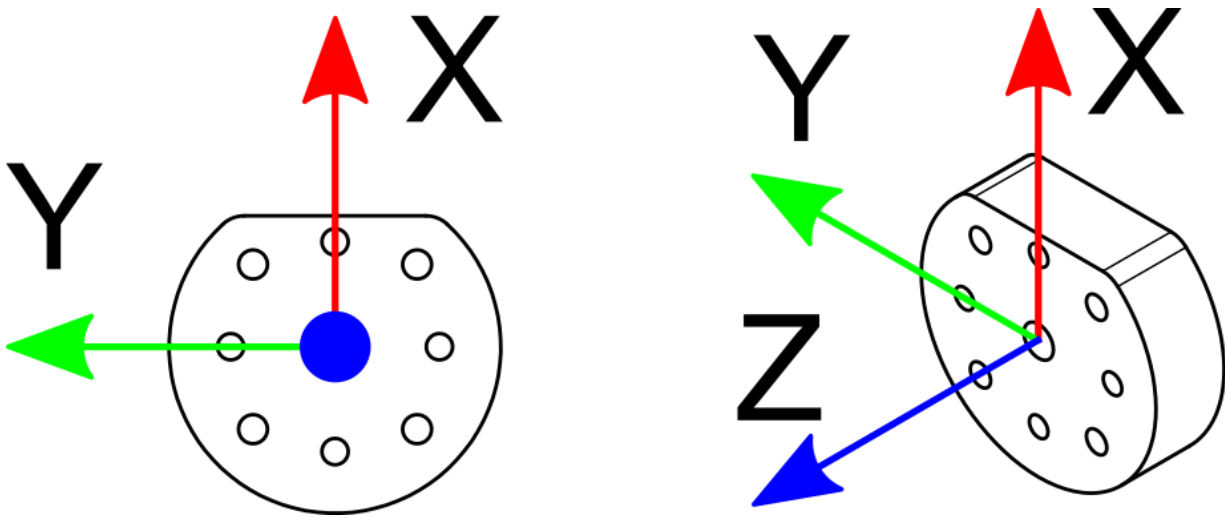
## Tool Matrix

When attaching an end of arm tool to the robot, it is important to set the correct parameters for the tool, so that the coordinates that the robot is reporting, represent the coordinates of the tool instead of the coordinates of the flange of the robot.

In general, to define the parameters of the tool, you will need to define the transformation matrix of the tool. The transformation matrix will define the rotation and translation of tool with respect to the flange of the robot. In general transformation matrices have the following format:

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} & l_x \\ r_{10} & r_{11} & r_{12} & l_y \\ r_{20} & r_{21} & r_{22} & l_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The submatrix  $r_{00}$ ,  $r_{01}$ ,  $r_{02}$ ,  $r_{10}$ ,  $r_{11}$ ,  $r_{12}$ ,  $r_{20}$ ,  $r_{21}$ ,  $r_{22}$  indicates the rotation part of the transformation matrix. The values  $l_x$ ,  $l_y$ ,  $l_z$  also indicate the translation of the toolhead in  $x$ ,  $y$ ,  $z$  direction calculated in  $mm$  with respect to the base of the flange coordinate system as shown in the image below.



Orientation of the robot when all the joints are zero

You can update the tool length based on the tool head that you are using. The correct tool length value is especially important if you work in the Cartesian coordinate system, and you are interested in  $(x, y, z, a, b, c)$  values of the TCP. If you prefer to work in joint coordinates, or you are interested in the  $X, Y,$  and  $Z$  values of the robot flange you might want to leave the TCP at its default value (default value is 0).

## Types of motion

Dorna has three main motion commands [jmove](#), [lmove](#), and [cmove](#). In each motion, the robot starts from its current orientation and moves toward the specified final orientation.



### Note

You can send the coordinates of any motion command in both [joint space](#) or [Cartesian space](#). The robot first looks for coordinates in joint space and if it does not find them in joint space it will look for the TCP coordinates in Cartesian space.

## Joint move (jmove)

[jmove](#) commands the robot to move from its current point to a destination point while the joints are moving uniformly altogether to complete their movements at the same time. This movement results in a curved path of the [TCP](#) in Cartesian space.

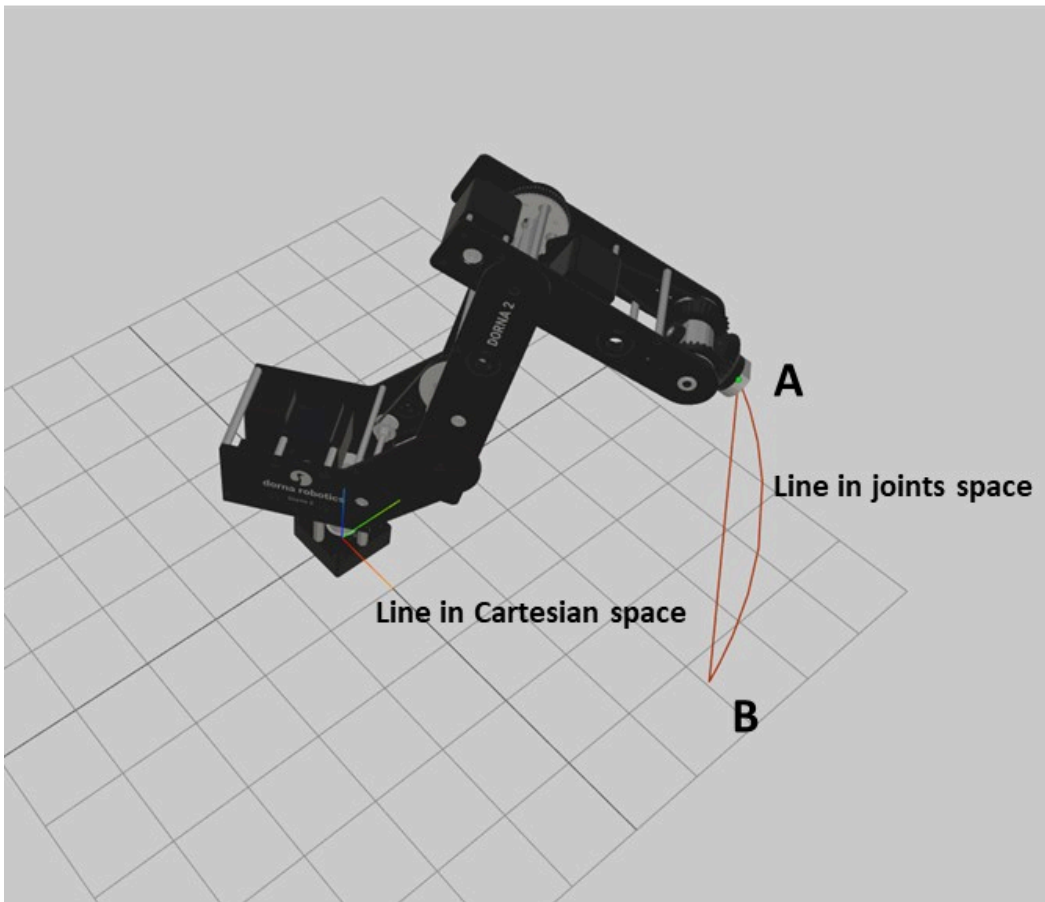


### Note

- [jmove](#) traverses the shortest distance (in terms of motor rotation), between two points. Therefore, they are a better choice for high-speed applications.
- This command is usually used when the trajectory of the motion is not important in the application and it is only important to end the motion at the specified destination point.

## Line move (lmove)

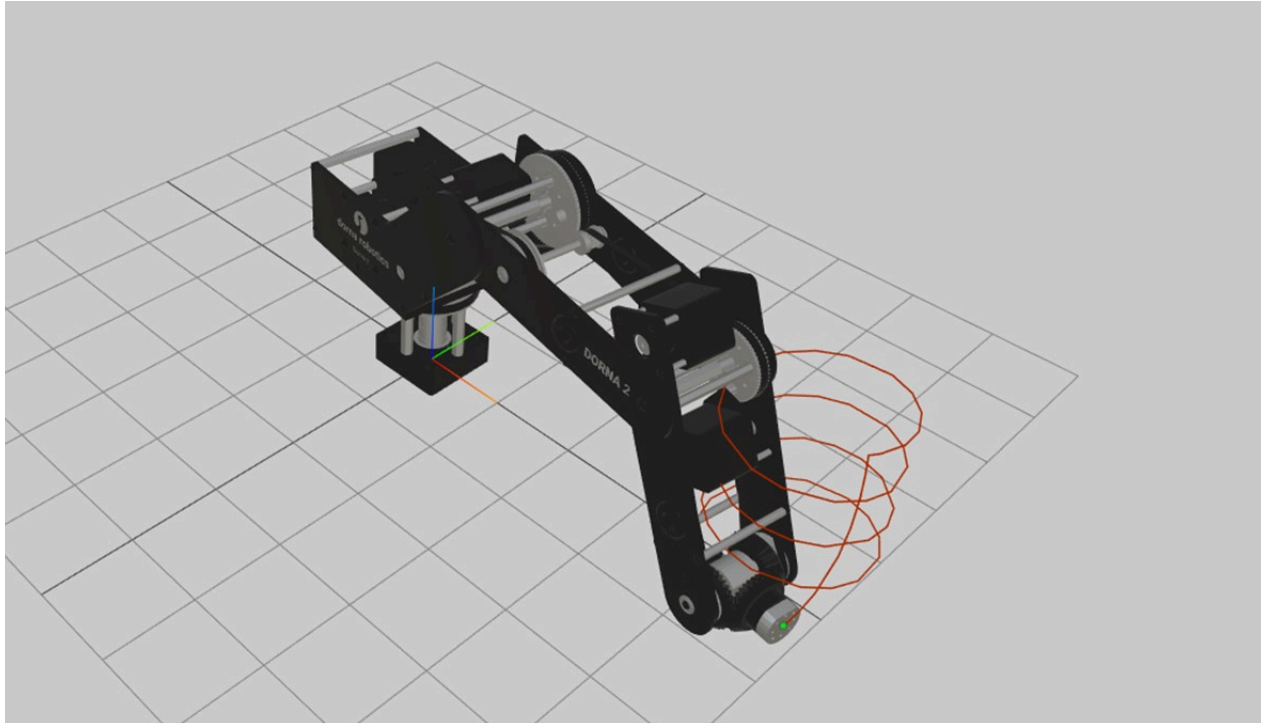
[lmove](#) moves the [TCP](#) linearly from its current pose to a destination pose. This means that each joint performs a more complicated motion to keep the tool on a straight line path.



## Circle move (cmove)

**cmove** moves the robot **TCP** on a circle. Circle is a set of points in space that are at the same distance from a center point and live on a plane. In the robot, a circle is uniquely specified with three points:

- The initial point of the circle, which is the current position of the robot.
- The final point of the circle, which is the final position of the move,
- A midpoint of the circle, which is a point that the circle will pass through before reaching the final point.
- Another parameter of the circle is the **turn** which specifies how many full turns the robot will make in addition to the initial partial circle that connects the initial point to the final point through the midpoint (in a partial circle **turn** is  $\emptyset$ ).



## The dynamic of a motion

Each motion command has associated parameters that define the dynamics of the motion along its trajectory. These parameters are the following:

- **vel**: The maximum velocity along the trajectory of the motion.
- **accel**: The maximum acceleration while speeding up and slowing down at the beginning and end of motion.
- **jerk**: The maximum value of jerk used in motion planning (jerk is the derivative of acceleration).

Depending on the type of the motion space that the motion is defined in, the unit of these parameters will vary

Type	vel	accel	jerk
<b>jmove</b>	$\frac{deg}{s}$	$\frac{deg}{s^2}$	$\frac{deg}{s^3}$
<b>lmove</b>	$\frac{mm}{s}$	$\frac{mm}{s^2}$	$\frac{mm}{s^3}$
<b>cmove</b>	$\frac{mm}{s}$	$\frac{mm}{s^2}$	$\frac{mm}{s^3}$

## How to pick the right value

Usually, by trial and error you will find the optimal value of velocity, acceleration, and jerk in your motion command. The optimal values will result in a motion that is done in the shortest amount of time and has a smooth start and end. Here we will provide a few guidelines to pick the (near) optimal values for your application.

1. Start with moderate values of `vel`, `accel`, and `jerk`. (see table below)
2. Increase or decrease `vel`, depending on your load. For higher payloads, you should limit max `vel`.
3. Increase or decrease `accel` values based on the load. `accel` value defines how fast you will reach the maximum speed and is limited by the torque that motors can generate.
4. Increase or decrease `jerk`, depending on the jerkiness of the motion with the determined `vel` and `accel` parameters. The `jerk` parameter plays a big role in the final quality of the motion, especially at the start and stop of the motion. Higher jerk values translate into more vibration at the start and stop. Lower jerk values, translate into vibration-free and smooth motion. Increase or decrease jerk to find the desired motion smoothness.
5. For more optimal results, you can go over the steps above once more from the beginning.

Parameter	Moderate value		Upper limit for small payloads		Upper limit for high payloads	
	Joint	Cartesian	Joint	Cartesian	Joint	Cartesian
<code>vel</code>	100	200	500	1000	250	500
<code>accel</code>	700	2000	3000	5000	1000	3000
<code>jerk</code>	3000	8000	10000	50000	5000	10000

## Continuous motion

In many applications, you might need to create a continuous transition from one motion to another motion without full stop at the end of each motion. Dorna motion planner makes it possible with an advanced feature called continuous motion.

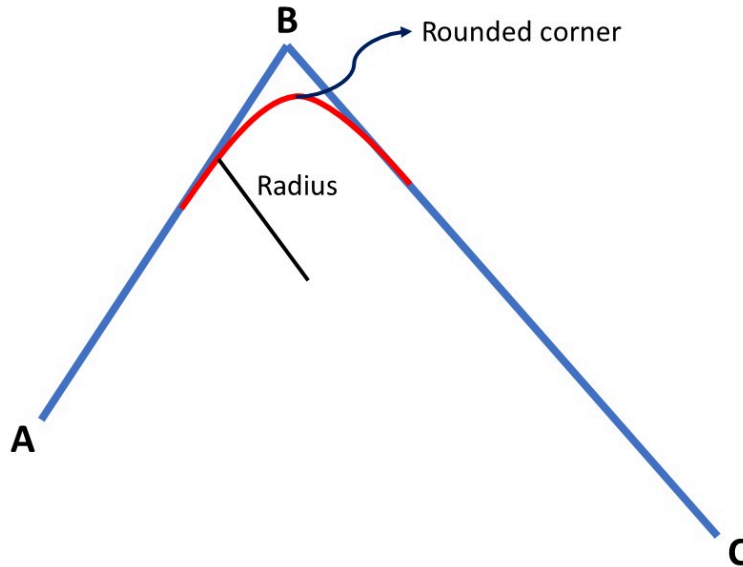
With continuous features activated, the controller will connect each new motion to the previous motion that it has in its queue. As soon as the controller runs out of motion commands in its queue, it will stop at the end of the last motion command.

Continuous motion only applies to consecutive `jmove` or `lmove` motion commands. In other words, it can only connect `jmove` commands or `lmove` commands together. If after a `jmove`

command, there is a `lmove` command, the robot will stop at the end of the `jmove` command and will start from a stop with the `lmove` command, and vice versa.

Since by physics laws, it is impossible to traverse two lines that are not collinear in space, without a full stop at the sharp corner, the continuous motion feature, does two things to traverse connecting lines smoothly:

1. It makes the corners of connecting lines rounded by replacing the corner with a smooth curve that gradually changes direction from one line to the next. The user can control the radius of the rounded corner by changing the parameter `corner` of the move commands.
2. It reduces the speed of the curve.



### Note

To make the corners traverse smoother, reduce the maximum velocity (`vel`) of the motion and increase the `corner` parameter. A larger `corner` radius or smaller `vel` both will make smoother corner traverses.



## Absolute and relative motion

All move commands in Dorna can accept coordinates in both absolute and relative form. The relevant parameter in motion commands is `rel`. When `rel` is set to `1`, all coordinates presented in the command (final point in line and final and midpoints in circle) are relative to the initial point of the motion command. Whereas if the `rel` parameter is set to `0`, all coordinates in the command are interpreted as absolute values.

## Auxiliary axes

In addition to the six axes of the robot, the controller can operate two additional axes, also known as auxiliary axes. In this section, we will go over setting up and using the auxiliary axes with the robot.

## General notes

- The encoder used for an auxiliary axis should be an incremental quadrature encoder with `A` and `B` channels (Index channel `I` is optional) and also be compatible with `3.3 V`.
- The motor driver should have step & direction (PUL/DIR) control and be compatible with `3.3 V`.
- You need a power supply (PSU) to run the motors. There is always the option of using the controller's internal `48 VDC` PSU. In this case, make sure that the motor driver input voltage is within the range of the controller's PSU, and that it also generates enough wattage to run the motors.
- Run all the wiring before turning the controller on, to avoid any damage to the electronics and the controller box.
- Use flex cable for motors and encoders.
- To protect the encoder signal against motor noise, we recommend:
  - Using shielded cable for encoder connection.
  - Connect the `GND` pin of the encoder to the body of the motor.
- Set the proper current setting on the driver based on your motor specification.

## Setup

### Scenarios

There are three main scenarios that the auxiliary axes can be useful in an automation project.

1. In the first scenario, we encounter a motor and a driver that utilize pulse and direction signals to control the motor's rotation, operating without the assistance of an encoder. Stepper motors serve as a prime example of this situation, as they can function accurately without relying on encoder feedback. In other cases, the motor driver itself may possess an internal processor capable of processing encoder data, eliminating the necessity for the Dorna controller to have access to the encoder data during the feedback loop processing. Numerous industrial AC or BLDC servo drivers fall into this category, permitting users to effectively operate these motors by solely sending pulse and direction signals to the driver.
2. The second scenario involves solely reading encoder data without the requirement of motor control. For instance, if a user wishes to track the position of a conveyor belt connected to a motor with a constant RPM, an encoder can be employed for this purpose. The Dorna controller facilitates the reading of encoder values, enabling their utilization for subsequent robot motion planning.
3. In the final setup, the user seeks to operate a stepper motor utilizing encoder feedback data within a closed-loop control system. The Dorna controller lets you control and run the motor and adjust the parameters of the motor and encoder and the controller PID loop.

## Definition

In the context of auxiliary axes, we have a motor (optional) that runs an axis (joint) through a gearing mechanism. The controller uses an encoder (optional) data to report the value of that axis as `j6`, or `j7` (depending on the axis number).

In this configuration, we also need the following motor and encoder parameters in order to properly report the value of an auxiliary axis or run the motor control loop:

- `pprm` (motor pulse per revolution): Also known as the motor driver micro-step setting, refers to the number of electrical pulses generated by a motor driver for each complete revolution of the motor shaft.
- `tprm` (motor travel per revolution): Refers to the amount the axis (joint) travels in one full motor shaft revolution. The unit of the travel is irrelevant and can be set by the user. This parameter can be positive or negative depending on the direction of movement of the motor relative to the axis of rotation.
- `ppre` (encoder pulse per revolution): Represents the number of electrical pulses generated by the encoder for each complete revolution of the encoder shaft (for incremental quadrature encoders, it is commonly referred to as CPR of the encoder).
- `tpre` (encoder travel per revolution): This refers to the amount the axis (joint) travels in one full encoder shaft revolution. This parameter can be positive or negative depending on the direction of movement of the encoder shaft relative to the axis of rotation.



## Note

If the encoder is directly mounted on the motor shaft, then the values of `tprm` and `tpre` will be the same.

## Example

We bring a few examples for better understanding:

1. A motor runs a rotary axis via a `1:10` gearbox. A 10-bit quadrature incremental encoder is mounted directly on the axis, and the driver running the motor is set to a `4000` micro-step setting.

In this case, one full revolution of the axis is `360` degrees, so:

- `pprm` is equal to `4000` due to the driver setting.
  - `tprm` is equal to `36` degrees, as one full rotation of the motor rotates the joint `36` degrees.
  - `ppre` is equal to the encoder resolution (`1024`).
  - `tpre` is equal to `360` degrees, as the encoder and axis shafts are identical.
2. Similar to the previous example but this time the encoder is mounted on the back of the motor and is coupled with the motor shaft. In this case, everything will be similar to the previous case, except `tpre` which will be equal to `36` degrees (similar to the `tprm`)
  3. A `4000` micro-step motor runs a linear rail, where one full rotation of the motor shaft results in `10 mm` travel of the rail. The encoder also has `14` bits resolution (`4096`):
    - `pprm = 4000`
    - `tprm = 10`
    - `ppre = 4096`
    - `tpre = 10`

## Configuration

Depending on the application, there are four distinct configurations available to configure the auxiliary axis of the robot. These configurations can be customized according to your requirements and desired functionality as described [here](#).

1. Use motor and encoder (`usem = 1`, `usee = 1`): In this case, the robot runs the axis closed-loop, using the given encoder and motor parameters, and returns the axis value based on the encoder data (`joint value = tpre x encoder value`). A linear rail with an encoder, or a rotary axis with an encoder are some examples of this

configuration. The closed-loop parameters are also specified by the PID values of that axis.

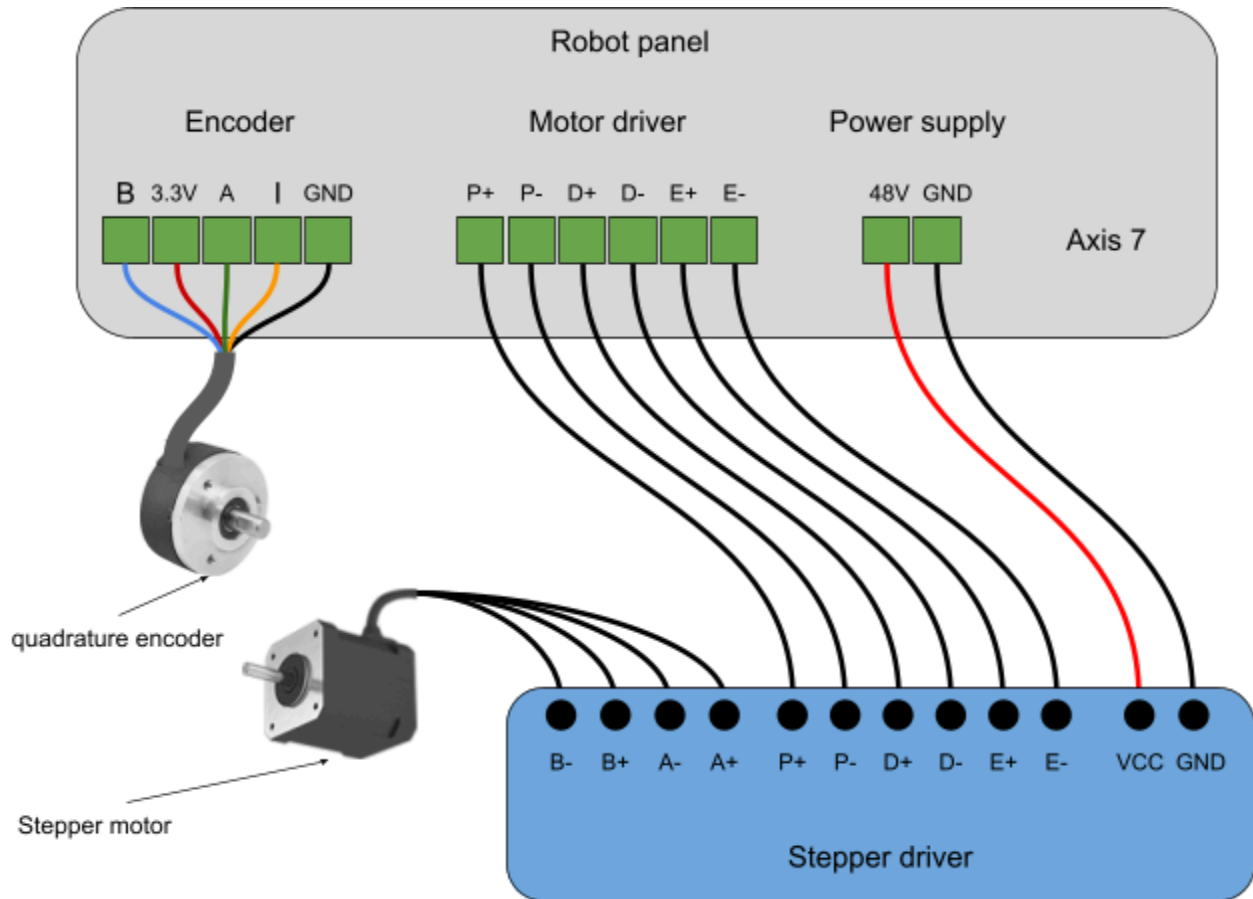
2. Use the motor with no encoder (`usem = 1, usee = 0`): In this case, the robot runs the axis open-loop. This configuration is useful when the encoder is not present, and it is acceptable to run the axis in an open-loop fashion. The robot returns the axis value based on the planned motor position data and not the encoder (`joint value = tprm x motor value`)
3. Use the encoder and not the motor (`usem = 0, usee = 1`): In this case, the robot reports the axis value based on the encoder data (`joint value = tpre x encoder value`). This configuration is useful when you need to track a conveyor position or similar applications.
4. No encoder and no motor (`usem = 0, usee = 0`): Both motor and encoder are missing. In this case, the robot reports the value of the joint as 0.



### Note

In a closed-loop application, it is important to set signs of `tprm` and `tpre` parameters consistent with the direction of the encoder and motor rotation. Otherwise, the feedback loop will not be able to converge to the desired position.

## Wiring



The connectors for the three auxiliary axes are available on the back panel of the controller box. Here is the list of connectors available for the auxiliary axes:

- Quadrature encoder inputs: The pitch of the connector is 3.81 mm and use a 5-position terminal block plug for the connection ([sample](#)).

Pin	Description
B	B channel
3.3 V	Encoder VCC
A	A channel

**I** Index channel

**GND** Encoder GND

- Motor driver with step/dir interface: The pitch of the connector is **3.81 mm** and use a 6-position terminal block plug for the connection ([sample](#)).

Pin	Description
<b>P+</b>	Driver PULSE signal.
<b>P-</b>	Ground for the driver PULSE signal.
<b>D+</b>	Driver DIR signal.
<b>D-</b>	Ground for the driver DIR signal.
<b>E+</b>	Driver ENABLE signal.
<b>E-</b>	Ground for the driver's ENABLE signal.

- Power inputs: The pitch of the connector is **3.81 mm** and use a 2-position terminal block plug for the connection ([sample](#)).

Pin	Description
<b>48V</b>	Power supply positive connection for the driver (48 VDC coming from the controller PSU)
<b>GND</b>	Power supply ground connection for the driver.



### Note

The encoder's index signal can be probed using the [iprobe](#) command. You can use the index signal for homing applications of the corresponding auxiliary axis.

## Operation

You can set up the parameters of the auxiliary axis using the [axis](#) command and operate the auxiliary axis using normal move commands in Dorna lab or Python API, the same way you program the robot's axes. For motion commands in joint space, the auxiliary axes are referred to as `j6`, and `j7`. For motion commands in Cartesian space, the auxiliary axes are referred to as `d`, and `e`, respectively.

## Command server

### Introduction

The robot runs a [WebSocket](#) server to send and receive data. After establishing a WebSocket connection via the IP address of the robot, the user and controller will exchange real-time information. We use the following terminology throughout this document.

- **Command**: Refers to a data message sent from the user (client) to the robot server.
- **Message**: Refers to the data message sent from the robot to the user.

### Server address

Before a user can send commands and receive messages from the robot, they have to first connect to the robot WebSocket server. The WebSocket server runs on port `443` of the robot:

Unset

```
ws://robot_ip_address:443
```

Where `robot_ip_address` is the robot IP address. For example, if the robot IP address is `10.0.0.14`, then the WebSocket address is:

Unset

```
ws://10.0.0.14:443
```



## Note

[Dorna lab](#) and Dorna [Python API](#) are two convenient ways to establish the WebSocket interface with the robot server. However, you can use any other programming language for establishing the WebSocket connection to the robot and sending commands and receiving messages.

## Data format

Each command or message sent over the Websocket channel is a package of data in [JSON](#) format. The general format of each package is as follows:

Unset

```
{"key1": value1, "key2": value2, ... , "keyn": valuen}
```

where string "[key<sub>i</sub>](#)" is the name of the *i*-th parameter and [value<sub>i</sub>](#) is its value.



## Note

- The order in which key and value pairs are presented in each package is not important.
- Keys are always a string in double quotation (" ").
- Values assigned to each key can be number, string, or binary.
- The keys and values (in string format) are case-sensitive. All strings should be in lower letters.

## Messages

Robot sends multiple types of messages to the users. Some are replies to a command that the user sent, some are periodic messages, and some are event-based messages generated due to an event occurring. Here, we will explain each of these messages in detail.



## Motion

These messages are sent periodically to the user and report the position of the robot in real-time at a rate of approximately 100 times a second. Each motion message reports the following information about the current state of the robot:

- Current robot [joint values](#)
- Current robot [TCP](#) Cartesian coordinates
- Current robot [velocity and acceleration](#) in the space of the motion

Here is an example of such a message:

Unset

```
{"cmd": "motion", "j0": 3.4, "j1": 1.2, "j2": 97.3, "j3": 22, "j4": 32.76, "j5": 0, "j6": 0, "j7": 0, "x": 122.3, "y": 674.3, "z": 95.4, "a": 342, "b": 32.6, "c": 0, "d": 0, "e": 0, "vel": 120.43, "accel": 804.11}
```



### Note

Velocity and acceleration are reported in the space of the motion. For instance, if the robot is moving with [line move](#), the space is Cartesian space (mm unit), whereas for [joint move](#) it is [joint space](#) (deg unit).

## Inputs

Upon a change in any of the inputs, a message is sent that includes the input values for all the robot input pins. Here is an example of such a message:

Unset

```
{"in0": 0, "in1": 0, "in2": 1, "in3": 1, "in4": 0, "in5": 0, "in6": 1, "in7": 0, "in8": 1, "in9": 0, "in10": 1, "in11": 0, "in12": 1, "in13": 0, "in14": 0, "in15": 0}
```

## Status

These messages report the [status of each command](#) that was sent to the robot, from the time that the command is submitted, to the time that the execution of the command is completed. These messages will only be sent to the user if the corresponding command has an "id" field with a positive integer value. If the command does not have a positive integer value for its "id" field, or it does not have an "id" field, the commands will be executed as normal, but no status will be reported back to the user.

The returned message will have the same id value as the command itself. An example of such a message is as follows:

```
Unset  
{"id":12, "stat":2}
```

It means that the command with "id":12, is in status 2 (see [here](#) for more information).

## Command response

Certain commands will return a response with information about the current state of the robot. These messages are returned by the robot with the same [id](#) as the original command. They also have a ["cmd"](#) field which is the same as the original command. We will explain the content of these messages for each [command](#) individually.

## Alarm

As soon as the robot goes into the alarm state, an alarm message is sent to the user, which looks like the following message:

```
Unset  
{"cmd":"alarm", "alarm":1, "err0":-1046, "err1":-1, "err2":0, "err3":0  
, "err4":0, "err5":0, "err6":0, "err7":0}
```

In this case the `alarm` message has an `alarm` key set to 1. The message has also `err0` to `err7` fields which are the errors in the readings of the encoders 0 to 7 of the robot. By inspecting the error values, you can identify the joints that have stopped moving and have caused the alarm state. In the example above, motor 0 (the base motor of the robot), has caused the alarm state.

## Commands

### Format

Each command which is sent to the controller has a required "cmd" key that specifies the name of the command (see the full list of commands [here](#)). Each command can also have an optional "id" key with a positive integer value assigned to it. The id is used to follow the [status](#) of the command throughout its lifetime by the [messages](#) that are returned from the robot with the same id value.

Unset

```
{"cmd": command_name, "id": unique_positive_integer_number, ...}
```

### Order of commands

The robot has two different queues for processing commands; normal priority queue and high priority queue.

#### Normal priority queue

This queue is designed for commands that need to run sequentially and the order matters in their execution, such as motion commands.

#### High-priority queue

Sometimes, while executing commands in the [normal priority queue](#), the user needs to issue some higher priority commands that need to be executed immediately upon being received by the controller. For instance, assume that the user needs to [halt](#) the robot immediately. In that case, if the halt command sits in the same queue with other normal priority commands, it will be executed after all other commands are finished, which is useless.

Another use case is reading input values. Sometimes, the user needs the value of an input pin, after a certain command is finished. In that case, the read command should be placed after the desired command in the normal priority queue. There are other instances, where the user needs to read the input value immediately without waiting for the completion of other commands in the normal priority queue. In that case, the user has to submit a read input command to the higher-priority queue.

Some commands are only assigned to a specific queue by the controller. For other commands, the user has the option to submit them to the normal priority queue or high priority queue. For such commands, the "queue" field (key) specifies the queue of the command (normal or high priority). When sending a command to the robot, if

- "queue" is set to 1: The command will be submitted to the [high-priority queue](#) and will be executed instantly.
- "queue" is set to 0: The command will be submitted to the [normal priority queue](#) and will be processed only after all other commands in that queue are processed.

## Status of a command and its life cycle

When sending a valid command to the robot, the controller reports the status of the command from the time that the command is submitted to the time that the execution of that command is completed using a `stat` key and the unique `id` sent initially with the command. An example of these messages are as follows:

Send command to read the alarm status of the system, with `id=12`:

Unset

```
{"cmd": "alarm", "id": 12}
```

Receive the following messages one by one from the controller:

Unset

```
{"id": 12, "stat": 0}
{"id": 12, "stat": 1}
{"cmd": "alarm", "id": 12, "alarm": 0}
{"id": 12, "stat": 2}
```

- `id` key: As you can see in the above example, we sent a command to the robot to get its alarm status. The robot replies back by sending multiple messages. Our initial command had an `id` field equal to 12. So, all the replies from the robot associated with this command have the same `id` value.
- `stat` key: Another important field in the received messages (robot replies) is the `stat` key. The `stat` field can take different values with the following interpretations:
  - `stat = 0`: The command has been received by the controller and has no error.
  - `stat = 1`: The command execution has just begun. Note that after commands are received and processed by the robot, they will be submitted to a [queue](#). Depending on the queue type and the position of the command in the queue, the actual execution of the command will be at a later time.

- `stat = 2`: The execution of the command is now completed with no error. Some commands run almost instantly. For those commands there is no delay between receiving `"stat":1` and `"stat":2` of the command. For example, reading an input value will be executed instantly. However, for some other commands, such as move commands or pause commands, the time that the command starts and the time that the command is completed are different. Users can use the `"stat":2` indicator, to sync different events with the completion of certain commands.
- `stat < 0`: An error happened during the execution of the command and the command will not be executed.



### Note

- We say that a command is completed if the robot sends `stat = 2` or `stat < 0` of that command. That basically means the command is completed, no longer running and its life cycle is over.
- If the command sent by the user does not have an `"id"` field with a positive integer value, the command will still be executed, but no status from that command will be returned by the robot to the user.

## List of commands

### `jmove`

Unset

```
{"cmd": "jmove", ...}
```

Moves the robot joints via a [joint move](#).



## Note

In `jmove` the robot joints travel the shortest possible distance to the destination point and therefore the best option for fast moves.

## Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"jmove"</code>	Yes
Moves the robot joints simultaneously to the target point.		
<code>"id"</code>	Int (>0)	No
ID can be any positive integer. If the ID is not provided, the status of the command will not be returned from the controller.		
Any combination of one of the following sets of keys: <ul style="list-style-type: none"> <li><code>"j0", "j1", "j2", "j3", "j4", "j5", "j6", "j7"</code></li> <li><code>"x", "y", "z", "a", "b", "c", "d", "e"</code></li> </ul>	Double	Yes
The position of the target point. Either you have to use a <a href="#">joint space</a> representation of the target point or the <a href="#">TCP pose</a> representation. In joint representation, at least one of the joint values <code>j0</code> , ..., <code>j7</code> should be present in the command. The joints that are missing will remain the same after the motion is completed. If no joint is present in the command, then the target point should be presented using <a href="#">TCP pose</a> in Cartesian coordinates <code>x</code> , <code>y</code> , <code>z</code> , <code>a</code> , <code>b</code> , <code>c</code> , <code>d</code> , <code>e</code> . In this case, the value of any missing coordinate will not change after the command is completed.		
<code>"rel"</code>	0 / 1	No
This value specifies if the move command is <a href="#">relative or absolute</a> . If it is set to <code>0</code> , the motion will be absolute. Otherwise, the move command will be relative. If this field is not present, the last given value for the same command ( <code>jmove</code> ) will be used.		
<code>"vel"</code>	Double (>0)	No
The maximum velocity of the motion in the joint space ( <i>deg/s</i> ). If this field is not present, the		

last given value for the same command ([jmove](#)) will be used.

**"accel"** Double (>0) No

The maximum acceleration of the motion in the joint space ( $deg/s^2$ ). If this field is not present, the last given value for the same command ([jmove](#)) will be used.

**"jerk"** Double (>0) No

The maximum jerk of the motion in the joints space ( $deg/s^3$ ). If this field is not present, the last given value for the same command ([jmove](#)) will be used.

**"cont"** 0 / 1 No

This parameter is for [continuous motion](#). 1 for continuous motion and 0 for discrete motion. If this field is not present, the last given value for the same command ([jmove](#)) will be used.

**"corner"** Double (>0) No

Set the corner parameter of the [continuous motion](#). If this field is not present, the last given value for the same command ([jmove](#)) will be used.

### Error codes

Stat	Value
-1	General error
-100	Final position is out of range
-102	Midpoint is out of range for circle
-103	Midpoint is not provided for circle
-104	Velocity coefficient is out of range
-105	Acceleration coefficient is out of range
-106	Jerk coefficient is out of range
-107	Velocity should be positive
-108	Acceleration should be positive

-109	Jerk should be positive
-110	Point out of range on the path
-111	Circle cannot be realized
-300	Halt already in process
-400	Alarm activated

Example 1: This command will move **j0**, 10 degrees and **j3**, 20 degrees relative to their current position. Other joints will not change. The maximum velocity will be **234 deg/s** and acceleration and jerk will be at the last given values. Note that if "**rel**" is set to 0, then the robot will move to the coordinates, **j0 = 10** and **j3 = 20**.

Unset

```
{"cmd": "jmove", "id":12, "j0":10, "j3":20, "rel":1, "vel":234}
```

## lmove

Unset

```
{"cmd": "lmove", ...}
```

Moves the robot [TCP](#) on a line using a [line move](#).

### Parameters

Key	Value	Required
"cmd"	"lmove"	Yes
Moves the robot TCP to a new position on a line.		
"id"	Int (>0)	No
Similar to the <b>jmove</b> command		



Any combination of one of the following sets of keys: <ul style="list-style-type: none"> <li>• "j0", "j1", "j2", "j3", "j4", "j5", "j6", "j7"</li> <li>• "x", "y", "y", "z", "a", "b", "c", "d", "e"</li> </ul>	Double	Yes
Similar to the <code>jmove</code> command		
"rel"	0 / 1	No
Similar to the <code>jmove</code> command		
"vel"	Double (>0)	No
The maximum velocity of the motion in the Cartesian space ( $mm/s$ ). If this field is not present, the last given value for the same command ( <code>lmove</code> ) will be used.		
"accel"	Double (>0)	No
The maximum acceleration of the motion in the Cartesian space ( $mm/s^2$ ). If this field is not present, the last given value for the same command ( <code>lmove</code> ) will be used.		
"jerk"	Double (>0)	No
The maximum jerk of the motion in the Cartesian space ( $mm/s^3$ ). If this field is not present, the last given value for the same command ( <code>lmove</code> ) will be used.		
"cont"	0 / 1	No
Similar to the <code>jmove</code> command		
"corner"	Double (>0)	No
Similar to the <code>jmove</code> command		

### Error codes

Similar to the `jmove` command

Example 1: Moves the TCP on a line from its current position to 5 mm in the x and 12 mm in the y direction with a maximum speed of 100 mm/s.

Unset

```
{"cmd": "lmove", "id": 12, "x": 5, "y": 12, "rel": 1, "vel": 100}
```

Example 2: Similar to the previous example, but the final position is given in joint coordinates.

Unset

```
{"cmd": "lmove", "id": 12, "j0": 10, "j1": 20, "rel": 1, "vel": 100}
```

## cmove

Unset

```
{"cmd": "cmove", ...}
```

Moves the robot [TCP](#) on a circle using a [circle move](#). Positions of the target point and midpoint of a circle can be given in joint space or in Cartesian space. Also, the coordinates can be either relative or absolute as defined by parameter `rel`. In relative mode, all positions are relative to the starting point of the circle.

### Parameters

Key	Value	Required
"cmd"	"cmove"	Yes
Moves the robot TCP on a circle.		
"id"	Int (>0)	No
Similar to the <code>jmove</code> command		
Any combination of one of the following sets of keys: <ul style="list-style-type: none"> <li>"j0", "j1", "j2", "j3", "j4", "j5", "j6", "j7"</li> <li>"x", "y", "z", "a", "b", "c", "d", "e"</li> </ul>	Double	Yes
The target point of the circle is expressed either in joint or Cartesian coordinates.		

Any combination of one of the following sets of keys:	Double	Yes
<ul style="list-style-type: none"> <li>“mj0”, “mj1”, “mj2”, “mj3”, “mj4”, “mj5”, “mj6”, “mj7”</li> <li>“mx”, “my”, “mz”, “ma”, “mb”, “mc”, “md”, “me”</li> </ul>		

The coordinates of a midpoint on the circle. The circle will connect the current position to the target position by passing through the midpoint. The midpoint can be either presented in joints space by “mj0”, “mj1”, “mj2”, “mj3”, “mj4”, “mj5”, “mj6”, “mj7” or in Cartesian space by “mx”, “my”, “mz”, “ma”, “mb”, “mc”, “md”, “me”. If a coordinate is not present in the description of the midpoint, it will be set to the initial point of the circle.

“turn”	Int (>=0)	No
--------	-----------	----

Number of turns of the circle. If not present, the default value of 0 is used. For any number of turns larger than 0, the robot will make the additional complete turns before stopping at the target point.

“rel”	0 / 1	No
-------	-------	----

If **rel** is set to 0, both midpoint and target point are interpreted as absolute coordinates. If **rel** is set to 1, both midpoint and target point values are treated relative to the initial point of the circle. If this field is not present, the last given value of the same command (**cmove**) will be used.

“vel”	Double (>0)	No
-------	-------------	----

The maximum velocity of the motion in the Cartesian space ( $mm/s$ ). If this field is not present, the last given value for the same command (**cmove**) will be used.

“accel”	Double (>0)	No
---------	-------------	----

The maximum acceleration of the motion in the Cartesian space ( $mm/s^2$ ). If this field is not present, the last given value for the same command (**cmove**) will be used.

“jerk”	Double (>0)	No
--------	-------------	----

The maximum jerk of the motion in the Cartesian space ( $mm/s^3$ ). If this field is not present, the last given value for the same command (**cmove**) will be used.

## Error codes

Similar to the `jmove` command

Example 1: This command will move the TCP, on a circle that starts from the current position and passes through `mx = 10`, `my = 8`, and `mz = 130`, and stops at `x = 10`, `y = 10`. Final `z`, `a`, `b`, `c`, `d`, `e` values will be the same as the initial point. The robot will make two extra full turns since `turn = 2`.

Unset

```
{"cmd": "cmove", "rel": 0, "id": 12, "x": 10, "y": 10, "mx": 6, "my": 8, "mz": 130, "turn": 2}
```

## halt

Unset

```
{"cmd": "halt", ...}
```

This command will force the robot to stop the motion immediately by decelerating from the current speed to zero. All remaining commands in the [queue](#) of the robot will also be deleted. Until the halt motion is completed, no new command except the [alarm](#) command, will be accepted by the robot.

The halt process by default uses the same deceleration as the original move command acceleration that is being stopped by the halt command. However, using parameter `accel` (default is 1), the user can increase the deceleration rate. If the `accel` parameter is present, the controller will stop the motion by deceleration rate which is the `accel` parameter of the halt command multiplied by the acceleration parameter of the original command. This will help the robot to decelerate faster during emergency halt commands.

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"halt"</code>	Yes

Halts the robot by decelerating to zero speed.

<code>"id"</code>	Int (>0)	No
-------------------	----------	----

Similar to the `jmove` command

<code>"accel"</code>	Double (>=1)	No
----------------------	--------------	----

If present, the robot will decelerate to zero speed by deceleration rate which is the `accel` value times the original motion command `accel` value. Otherwise, the robot will decelerate with the deceleration value equal to the original move command `accel` value. Note that this parameter cannot be less than 1.

### Error codes

Stat	Value
------	-------

-1	General error
----	---------------

-2	<code>accel</code> value provided is not valid
----	--

-300	Halt already in process
------	-------------------------

-400	Alarm activated
------	-----------------

Example 1: Immediately after receiving this command, the robot will decelerate at the rate of  $7.5 \times \text{accel}$  to a full stop, where `accel` is the parameter value from the original move command which the `halt` command is acting on.

Unset

```
{"cmd": "halt", "id": 12, "accel": 7.5}
```

### alarm

Unset

```
{"cmd": "alarm", ...}
```

If an external force (or an obstacle) prevents the robot from maintaining its planned orientation, then the robot goes into the alarm state. In the alarm state, all commands that are already submitted to the robot will be ignored and removed, and the existing motion command will be immediately suspended and the robot will abruptly stop its motion without proper deceleration. No new commands will be accepted by the controller until an alarm-clearing command (`alarm = 0`) is issued to the robot by the user. The user has to clear any obstacle to the motion of the robot before clearing the alarm. After the alarm is cleared, the robot will accept and run new commands as usual.

In addition to accidents that will force the robot into alarm mode, the user can also manually send an alarm command to the robot, which will result in exactly the same behavior from the robot. For instance, if the user can detect an obstacle that will hit the robot shortly, and the time or distance to the obstacle is not large enough for a `halt` command to operate properly, the user can send an alarm command.

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"alarm"</code>	Yes
Forces the robot into an alarm state or clears an already existing alarm state.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command		
<code>"alarm"</code>	0 / 1	No
If the <code>alarm</code> has value 1, the robot will be forced into alarm mode. If the value is 0, the existing alarm will be cleared. If it is not presented, then the current value of the alarm will be returned in response.		

### Response

Key	Value
<code>"cmd"</code>	<code>"alarm"</code>
The <code>cmd</code> key in the response is set to <code>"alarm"</code> .	
<code>"alarm"</code>	0 / 1

The value of the `alarm` indicates the current alarm state in the controller.

`"id"` Int (>0)

Same id as the original command.

### Error codes

Stat	Value
------	-------

None	
------	--

Example 1: This command will clear the alarm state of the robot. It has to be issued after the robot enters the alarm state.

Unset

```
{"cmd": "alarm", "alarm": 0, "id": 12}
```

Example 2: This command will return the current state of the alarm signal.

Unset

```
{"cmd": "alarm", "id": 10}
```

## pid

Unset

```
{"cmd": "pid", ...}
```

This command is used to set and get the closed-loop PID parameters of the robot (P, I and D), and also the threshold and duration parameters of the [safety function](#).

- PID values: Each robot joint ( $j_0, \dots, j_7$ ) uses three sets of parameters  $p_i$ ,  $i_i$ , and  $d_i$  (for  $i$  from 0 to 7) for the position PID loop of the associated joint. Use the PID command to set and get these parameters.
- `threshold` and `duration` parameters: An error  $err_i$  (for  $i$  from 0 to 7) is a function of the robot's actual joint value  $j_i$  and the value that is supposed to be based on the planned motion trajectory. Higher error (in absolute form  $|e_i|$ ) means that the associated joint is facing more external force, and therefore the PID loop cannot converge to the correct position. An alarm happens when for a period of `durationi`, the absolute value of at least one of the  $err_i$  is greater than or equal to the `thresholdi`.



### Note

- The `durationi` parameter is not measured in seconds.
- The `thresholdi` parameter is not measured in degrees.
- Modify the PID parameter with caution. The wrong PID parameters will greatly impact the quality of the motion and will cause unstable and jerky movements by the robot.
- To operate the robot in an open loop with no active safety alarms, set all `p`, `i`, and `d` for all the joints to 0 and `duration` and the `threshold` to 10000000.
- The PID parameters will all be reset to their default values upon resetting the controller.

The default values of PID parameters are set based on extensive tests and work for a wide range of load and speed profiles. We do not recommend changing the default values for the joints 0 to 4 (five axes of the robot). For auxiliary axes, the default values should serve as a good starting point. However, the user might need to change the default values for certain motor or load or speed requirements. The default values for the PID parameters are

Unset

```
# Dorna TA
```

```
p0 = 0, i0 = 0.0003, d0 = 0.02, threshold0 = 200, duration0 = 10000
```

```
p1 = 0, i1 = 0.0003, d1 = 0.02, threshold1 = 200, duration1 = 10000
```

```
p2 = 0, i2 = 0.0003, d2 = 0.02, threshold2 = 200, duration2 = 10000
```



```

p3 = 0, i3 = 0.0003, d3 = 0.02, threshold3 = 200, duration3 = 10000
p4 = 0, i4 = 0.0003, d4 = 0.02, threshold4 = 200, duration4 = 10000
p5 = 0, i5 = 0.001, d5 = 0.01, threshold5 = 200, duration5 = 10000
p6 = 0, i6 = 0.001, d6 = 0.01, threshold6 = 200, duration6 = 10000
p7 = 0, i7 = 0.001, d7 = 0.01, threshold7 = 200, duration7 = 10000

```

## Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"pid"</code>	Yes
Set or get the robot PID parameters.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command		
Any combination of one of the following sets of keys: <ul style="list-style-type: none"> <li><code>"p0"</code>, <code>"p1"</code>, <code>"p2"</code>, <code>"p3"</code>, <code>"p4"</code>, <code>"p5"</code>, <code>"p6"</code>, <code>"p7"</code></li> <li><code>"i0"</code>, <code>"i1"</code>, <code>"i2"</code>, <code>"i3"</code>, <code>"i4"</code>, <code>"i5"</code>, <code>"i6"</code>, <code>"i7"</code></li> <li><code>"d0"</code>, <code>"d1"</code>, <code>"d2"</code>, <code>"d3"</code>, <code>"d4"</code>, <code>"d5"</code>, <code>"d6"</code>, <code>"d7"</code></li> <li><code>"Threshold0"</code>, ..., <code>"Threshold7"</code></li> <li><code>"duration0"</code>, ..., <code>"duration7"</code></li> </ul>	Double(>=0)	No
Use these keys to adjust the PID, threshold and duration parameters. No more than 16 PID parameters can be specified in a single command.		
<code>"pid"</code>	0 / 1	No
Enable or disable the pid functionality (1 enable and 0 disable).		

## Response

Key	Value
<code>"cmd"</code>	<code>"pid"</code>

The `cmd` key in the response is set to `"pid"`.

- `"p0", "p1", "p2", "p3", "p4", "p5", "p6", "p7"` Double( $\geq 0$ )
- `"i0", "i1", "i2", "i3", "i4", "i5", "i6", "i7"`
- `"d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7"`
- `"Threshold0", ..., "Threshold7"`
- `"duration0", ..., "duration7"`

In response the robot sends back the PID, threshold and duration parameters. The response will be sent in two separate messages where the first message returns the first 16 parameters and the second message will return the remaining 10 parameters.

`"pid"` 0 / 1

Indicate whether the pid functionality is enabled (1) or disabled (0)

`"id"` Int ( $> 0$ )

Same id as the original command.

### Error codes

Stat	Value
None	

## sleep

Unset

```
{"cmd": "sleep", ...}
```

The robot sleeps for the number of seconds specified in this command. It can be used to create a delay between the execution of motion commands or other commands in the [normal priority queue](#) as it executes in the normal priority queue in the same order that it was received by the controller.

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"sleep"</code>	Yes
Sleep between commands.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command		
<code>"time"</code>	Double (>=0)	No
The amount of sleep time in seconds (s).		

### Error codes

Stat	Value
-21	<code>time</code> field is missing or invalid
-300	Halt already in process
-400	Alarm activated

Example 1: After finishing all the commands before this command, the controller waits for 5 seconds, before executing the next command.

Unset

```
{"cmd": "sleep", "time": 5}
```

### input

Unset

```
{"cmd": "input", ...}
```

Get the input values. The command by default will report the input values immediately. However, by setting the `queue` value to `0`, the user can send the command to the [normal priority queue](#), where the input values will be reported after all other commands before they are concluded.

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"input"</code>	Yes
Get input values.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command		
<code>"queue"</code>	0 / 1	No
If the value is 0, the command will be submitted to the <a href="#">normal priority queue</a> . Otherwise, it will be submitted to the <a href="#">high-priority queue</a> . The default value is 1.		

### Response

Key	Value
<code>"cmd"</code>	<code>"input"</code>
The response of an <code>input</code> command will be a message with the <code>cmd</code> field set to <code>input</code> .	
<code>"id"</code>	Int (>0)
Same id as the original command.	
<code>"in0", ..., "in15"</code>	0 / 1
The value of each input.	

### Error codes

Stat	Value
-300	Halt already in process
-400	Alarm activated

Example 1: This example will read the input values,

```
Unset
{"cmd": "input", "id": 12}
```

and will respond with a message as follows which specifies the current value of each input pin.

```
Unset
{"cmd": "input", "id": 12, "in0": 0, "in1": 0, "in2": 1, "in3": 1,
 "in4": 0, "in5": 0, "in6": 1, "in7": 0, "in8": 1, "in9": 0,
 "in10": 1, "in11": 0, "in12": 1, "in13": 0, "in14": 0, "in15": 0}
```

## probe

```
Unset
{"cmd": "probe", ...}
```

Match the input values with the pattern that the user specifies. Whenever such a pattern appears at the input pins, the robot will send a response which is the [joint values](#) of the robot at the time that the match happens. This command could be useful for [homing](#) an actuator with a sensor connected to an input pin. Note that the controller will only send the response after the first match happens and consequent matches won't be responded to. This command by default will be submitted to the [high-priority queue](#). By setting the `queue` value to `0`, the command will be submitted to the [normal priority queue](#).

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"probe"</code>	Yes
Match input values with a given pattern.		
<code>"id"</code>	Int (>0)	No

Similar to the `jmove` command from the controller.

`"in0", ..., "in15"` 0 / 1 No

Probe command will wait until the value of each input that has a key in the command matches its value. If an input key is not present in the command, its value won't impact the match.

`"queue"` 0 / 1 No

If the value is `0`, the command will be submitted to the [norma-priority queue](#). Otherwise, it will be submitted to the [high-priority queue](#). The default value is `1`.

### Response

Key	Value
<code>"cmd"</code>	<code>"probe"</code>
Response of a <code>probe</code> command will be a message with the <code>cmd</code> field set to <code>probe</code> which will include the joint values at the moment of input match.	
<code>"id"</code>	Int (>0)
Same id as the original command.	
<code>"j0", "j1", ..., "j7"</code>	Double
The robot joint values	

### Error codes

Stat	Value
-300	Halt already in process
-400	Alarm activated

Example 1: This command will wait until the input 4 (`in4`) value is `0` and input 7 (`in7`) value is `1`

Unset

```
{"cmd": "probe", "id": 12, "in4": 0, "in7": 1}
```

Then it will send a message that includes the joint values of the robot at that moment. While the robot is waiting for the match to happen all other commands including move commands will be executed as normal.

Unset

```
{"cmd": "probe", "id": 12, "j0": 12.3, "j1": 2.11, "j2": 85, "j3": 653.4, "j4": 56.34, "j5": 67.32, "j6": 23.74, "j7": 90.00}
```

## iprobe

Unset

```
{"cmd": "iprobe", ...}
```

This command is similar to the [probe](#) command but here we are waiting for a specific pattern in the encoder indices (only applies to quadrature encoders with index channel), instead of an input pin. Whenever such a pattern appears at the encoder indices, the robot will send a response which is the [joint values](#) of the robot at the time that the match happens. This command could be useful for [homing](#) of an auxiliary axis with quadrature encoders and index channel. Note that the controller will only send the response after the first match happens and consequent matches won't be responded to. This command by default will be submitted to the [high-priority queue](#). By setting the `queue` value to `0`, the command will be submitted to the [normal priority queue](#).



### Note

- The index channel is an additional channel available in some quadrature encoders. It generates a single pulse per revolution, providing a reference point or marker for a complete rotation. This pulse is typically used to reset or synchronize position counts.

- In some encoders, the index generates the pulse 8 times per revolution (quadrature encoders that come with the Dorna rail kit).
- Make sure to read the encoder datasheet, in order to understand the index channel behavior.
- You can use the index channel for [setting the joints automatically](#) auto homing of the auxiliary axis.

### Parameters

Key	Value	Required
"cmd"	"iprobe"	Yes
Match input values with a given pattern.		
"id"	Int (>0)	No
Similar to the <code>jmove</code> command from the controller.		
"in5", "in6", "in7"	0 / 1	No
The iprobe command will wait until the value of the encoder index pins in the command appears. If an encoder index key is not present in the command, its value won't impact the match.		
"queue"	0 / 1	No
If the value is 0, the command will be submitted to the <a href="#">normal priority queue</a> . Otherwise, it will be submitted to the <a href="#">high-priority queue</a> . The default value is 1.		

### Response

Key	Value
"cmd"	"iprobe"
Response of an <code>iprobe</code> command will be a message with the <code>cmd</code> field set to <code>iprobe</code> which will include the joint values at the moment of input match.	



---

`"id"` Int (>0)

Same id as the original command.

`"j0", "j1", ..., "j7"` Double

The robot joint values

---

### Error codes

Stat	Value
-300	Halt already in process
-400	Alarm activated

Example 1: This command will wait until the encoder index channel of the auxiliary `axis5` is 1

```
Unset
{"cmd": "iprobe", "id": 12, "in5": 1}
```

Then it will send a message that includes the joint values of the robot at that moment.

```
Unset
{"cmd": "iprobe", "id": 12, "j0": 12.3, "j1": 2.11, "j2": 85,
 "j3": 653.4, "j4": 56.34, "j5": 67.32, "j6": 23.74, "j7": 90.00}
```

## output

```
Unset
{"cmd": "output", ...}
```

Set or get the values of the output pins. After the robot receives the command, it will set the output values and in response will return the value of all output pins. The default queue for this

command is the [high-priority queue](#). If the `queue` parameter is set to `0`, then the command will be submitted to the [normal priority queue](#).

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"output"</code>	Yes
Set or get the output values.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command from the controller		
<code>"out0"</code> , <code>"out1"</code> , <code>"out2"</code> , ..., <code>"out15"</code>	0 / 1	No
If any <code>out<sub>i</sub></code> is present as a key, the value of the corresponding output will be set to the given value.		
<code>"queue"</code>	0 / 1 (default 1)	No
If the value is <code>0</code> , the command will be submitted to the normal priority queue. Otherwise, it will be submitted to the high-priority queue. The default value is <code>1</code> .		

### Response

Key	Value
<code>"cmd"</code>	<code>"output"</code>
In response to the <code>output</code> command a message is returned with the <code>cmd</code> field set to <code>output</code> which includes the value of all output pins.	
<code>"id"</code>	Int (>0)
Same id as the original command.	
<code>"out0"</code> , <code>"out1"</code> , ..., <code>"out15"</code>	0 / 1
The value of each output pin.	

### Error codes

Stat	Value
-300	Halt already in process
-300	Alarm activated

Example 1: This command will set `out0` to 1 and `out2` to 0,

Unset

```
{"cmd": "output", "out0": 1, "out2": 0, "id": 12}
```

and will return all output values.

Unset

```
{"cmd": "output", "id": 12, "out0": 1, "out1": 0, "out2": 0,
"out3": 1, "out4": 0, "out5": 0, "out6": 1, "out7": 0, "out8": 1,
"out9": 0, "out10": 1, "out11": 0, "out12": 1, "out13": 0,
"out14": 0, "out15": 0}
```

### pwm

Unset

```
{"cmd": "pwm", ...}
```

This command will enable and disable PWM pins and will set their duty cycle and frequency to the desired values. The default queue for this command is the [high-priority queue](#). If the `queue` parameter is set to 0, then the command will be submitted to the [normal priority queue](#). In response to this command, the state of all PWM pins will be returned.

### Parameters

Key	Value	Required
-----	-------	----------

<code>"cmd"</code>	<code>"pwm"</code>	Yes
Enable or disable PWM pins, set their duty cycle and frequency values.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command from the controller.		
<code>"pwm0"</code> , <code>"pwm1"</code> , <code>"pwm2"</code> , <code>"pwm3"</code> , <code>"pwm4"</code>	0 / 1	No
If <code>pwm<sub>i</sub></code> is set to 0, it will disable <i>i</i> th PWM pin. If <code>pwm<sub>i</sub></code> is set to 1, it will enable <i>i</i> th PWM pin. If <code>pwm<sub>i</sub></code> is not present as a key, then the state of <i>i</i> th PWM pin will not change by the command.		
<code>"duty0"</code> , <code>"duty1"</code> , <code>"duty2"</code> , <code>"duty3"</code> , <code>"duty4"</code>	Double (>=0 and <=100)	No
If <code>duty<sub>i</sub></code> is present as a key, the duty cycle of the <i>i</i> th PWM will be set to its value which is the percentage of the period that the PWM pin will be on.		
<code>"freq0"</code> , <code>"freq1"</code> , <code>"freq"</code> , <code>"freq3"</code> , <code>"freq4"</code>	Double (>=0 and <=120,000,000)	No
If <code>freq<sub>i</sub></code> is present as a key, frequency of the <i>i</i> th PWM pin will be set to its value.		
<code>"queue"</code>	0 / 1	No
If the value is 0, the command will be submitted to the normal priority queue. Otherwise, it will be submitted to the high-priority queue. The default value is 1.		

## Response

Key	Value
<code>"cmd"</code>	<code>"pwm"</code>
In response to the <code>pwm</code> command, a message is returned with a <code>cmd</code> field set to <code>pwm</code> where the state, duty cycle, and frequency of all PWM pins are included in it.	
<code>"id"</code>	Int (>0)
Same id as the original command.	

"pwm0", "pwm1", "pwm2", "pwm3", "pwm4"

0 / 1

Indicates if each PWM pin is enabled or disabled. 1 means the PWM pin is enabled and 0 means it is disabled.

"duty0", "duty1", "duty2", "duty3", "duty4"

Double (>=0 and <=100)

Indicates the duty cycle of each PWM pin.

"freq0", "freq1", "freq", "freq3", "freq4"

Double (>=0 and <=120,000,000)

Indicates the frequency of each PWM pin.

### Error codes

Stat	Value
-300	Halt already in process
-300	Alarm activated
-601	Duty cycle parameter is out of range
-602	Freq parameter is out of range

Example 1: This command will enable `pwm0` and set its frequency to `125 Hz`. It will also disable `pwm2`.

Unset

```
{"cmd": "pwm", "id": 12, "pwm0": 1, "pwm2": 0, "freq0": 125}
```

The response will have information about all PWM pins.

Unset

```
{"cmd": "pwm", "id": 12, "pwm0": 1, "pwm1": 0, "pwm2": 0, "pwm3": 1, "pwm4": 0, "duty0": 5, "duty1": 2, "duty3": 32, "duty4": 9, "freq0": 125, "freq1": 320, "freq2": 450, "freq3": 1200, "freq4": 100}
```

## adc

Unset

```
{"cmd": "adc", ...}
```

This command will read the value of all ADC pins and map the input voltage (in the range of 0V to 3.3V) to an integer between 0 to  $2^{16} - 1$ . The default queue for this command is the [high priority queue](#). If the `queue` parameter is set to `0`, then the command will be submitted to the normal priority queue.

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"adc"</code>	Yes
Reads ADC pins.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command from the controller.		
<code>"queue"</code>	0 / 1	No
If the value is <code>0</code> , the command will be submitted to the normal priority queue. Otherwise, it will be submitted to the high-priority queue. The default value is <code>1</code> .		

### Response

Key	Value
<code>"cmd"</code>	<code>"adc"</code>
In response to <code>adc</code> command a message is returned where the <code>cmd</code> field is set to <code>adc</code> and the values of all ADC pins are included in it.	
<code>"id"</code>	Int (>0)
Same id as the original command.	

---

"adc0", "adc1", "adc2", "adc3", "adc4"      Int ( $\geq 0$  and  $\leq 2^{16} - 1$ )

Value of ADC pins as an integer between 0 and  $2^{16} - 1$ .

---

### Error codes

Stat	Value
-300	Halt already in process
-400	Alarm activated

Example 1: Send an `adc` command to the robot

```
Unset  
{ "cmd": "adc", "id": 12 }
```

The response will include the value of all ADC pins.

```
Unset  
{ "cmd": "adc", "id": 12, "adc0": 0, "adc1": 33, "adc2": 10, "adc3": 0,  
  "adc4": 0 }
```

### joint

```
Unset  
{ "cmd": "joint", ... }
```

This command will set the joint values to the values given in the command. This command is always submitted to the [high-priority queue](#) and upon being received by the controller, all other commands in the controller queue will be deleted. This command is particularly useful for the [homing process](#). The return value from the controller will be the value of all the joint values of the robot after the new values are applied. This command can be used to read the current value of joints if it is used with no joint input.



### Note

- Dorna TA uses absolute encoders on its joints that keep track of the joint positions even after the power shut off. Therefore no homing is required for the first 6 axes of the robot upon starting the robot.
- In Dorna TA, this command will only take the value of one joint at a time.

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"joint"</code>	Yes
Sets the joint values to the values provided in the command.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command.		
<code>"j0", "j1", "j2", "j3", "j4", "j5", "j6", "j7"</code>	Double	No
The value of each joint. If a joint is not present, its value will not change by the command. In Dorna TA, only one joint can be set in a joint command.		

### Response

Key	Value
<code>"cmd"</code>	<code>"joint"</code>
The response of a <code>joint</code> command will be a message with a <code>cmd</code> field set to the <code>joint</code> .	
<code>"id"</code>	Int (>0)
Same id as the original command.	



---

"j0", "j1", "j2", "j3", "j4", "j5", "j6", "j7" Double

The value of all joints will be returned.

---

### Error codes

Stat	Value
-300	Halt already in process
-400	Alarm activated

Example 1: This command will set the **j3** value to the given value

```
Unset  
{"cmd":"joint","id":12, "j3":37.5}
```

and return all joint values:

```
Unset  
{"cmd":"joint","id":12, "j0":0, "j1":30.22, "j2":29, "j3":37.5,  
"j4":90.0, "j5":0, "j6":0, "j7":0}
```

### axis

```
Unset  
{"cmd":"axis", ...}
```

Use this command to set or get auxiliary axes parameters.

This command applies parameters to the [auxiliary axes](#) 6, and 7. It also saves the parameters of these three axes whether they are active or not.

The command only takes parameters for one joint at a time, and the joint is determined from the first available parameter (all other parameters for other joints will be ignored).

The response will include the value of all parameters for all joints.

## Parameters

Key	Value	Required
"cmd"	"axis"	Yes
Sets or gets the auxiliary axes parameters.		
"id"	Int (>0)	No
Similar to the <code>jmove</code> command.		
Use one of the following sets of keys <ul style="list-style-type: none"> <li>"usem6", "usee6"</li> <li>"usem7", "usee7"</li> </ul>	0 / 1	No
<p><code>usemi</code> and <code>useei</code> indicate whether we use motor or encoder for the axis <code>i</code> or not, respectively. Depending on the value of <code>usem</code> and <code>usee</code> we can fall into one of the following four categories explained in the <a href="#">auxiliary axes configuration</a> section:</p> <ul style="list-style-type: none"> <li><code>usemi=1</code>, <code>useei=1</code>: Both motor and encoder are present for the auxiliary axis.</li> <li><code>usemi=1</code>, <code>useei=0</code>: Only encoder is present.</li> <li><code>usemi=0</code>, <code>useei=1</code>: Only motor is present.</li> <li><code>usemi=0</code>, <code>useei=0</code>: No motor or encoder is present.</li> </ul>		
Use one of the following sets of keys based on the index picked for <code>usem</code> and <code>usee</code> : <ul style="list-style-type: none"> <li>"pprm6", "tprm6", "ppre6", "tpre6"</li> <li>"pprm7", "tprm7", "ppre7", "tpre7"</li> </ul>	Double (≠0)	No
Set the <a href="#">motor and encoder parameters</a> "pprm", "tprm", "ppre", "tpre" of an auxiliary axis.		

## Response

Key	Value
"cmd"	"axis"
The response of an <code>axis</code> command will be a message with the <code>cmd</code> field set to the <code>axis</code> .	
"id"	Int (>0)

Same id as the original command.

```
usem6, usem7           Double
usee6, usee7
pprm6, pprm7
tprm6, tprm7
ppre6, ppre7
tpre6, tpre7
```

Response will include the value of all parameters for all the auxiliary axes (6 and 7). The response will be chopped in two messages where the first message will include the parameters of axes 6 and the second message will include the parameters of axis 7.

### Error codes

Stat	Value
-902	Error code for <code>pprm</code> being 0
-903	Error code for <code>tprm</code> being 0
-904	Error code for <code>ppre</code> being 0

Example 1: A stepper motor with 4000 micro step setting and 1:10 reduction is running a rotary axis. The axis is labeled as `j7` and a 10-bit (1024) encoder is also mounted on it to measure its value:

Unset

```
{"cmd": "axis", "id":12, "usem7":1, "usee7":1, "pprm7":4000,
"tprm7":36, "ppre7": 1024, "tpre7":360}
```

### motor

Unset

```
{"cmd": "motor", ...}
```

This command is used to turn the motors on or off. The user can turn motors off to either save power while the robot is not operational or place the robot in a predetermined location as a homing process or hand train the robot by moving it to different locations manually and saving the positions of those points. This command is submitted to the [high-priority queue](#). In response to this command the current state of the motors after the command was applied is returned.



### Note

Please note that when the motors are off, the robot still tracks the positions of the motors, and as soon as the motors are on again, the motors can continue receiving motion commands as usual.

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"motor"</code>	Yes
Turn motors on or off. The response will be the current state of the motors.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command.		
<code>"motor"</code>	0 / 1	No
0 will turn all motors off and 1 will turn them on. If this parameter is not present, the motor's state will not change.		

### Response

Key	Value
<code>"cmd"</code>	<code>"motor"</code>
In response to the <code>motor</code> command a message is returned with a <code>cmd</code> field set to the <code>motor</code> which includes the current state of the motors.	

`"id"` Int (>0)

Same id as the original command.

`"motor"` 0 / 1

0 means the motors are off and 1 means the motors are on.

### Error codes

Stat	Value
-300	Halt already in process
-400	Alarm activated

Example 1: This command will turn all motors off.

```
Unset
{"cmd": "motor", "id": 12, "motor": 0}
```

The response will include the current state of the motors after the command was applied:

```
Unset
{"cmd": "motor", "id": 12, "motor": 0}
```

## tool

```
Unset
{"cmd": "tool", ...}
```

With this command you can set or get the [tool matrix](#). For accurate readings of the TCP, the tool matrix should be set to the correct value for use in forward and inverse kinematics calculations inside the robot. The response returned from the robot is the tool head length. This command

always runs in the [high-priority queue](#) and upon being received by the controller, all other commands in the controller will be deleted.

### Parameters

Key	Value	Required
<code>"cmd"</code>	<code>"tool"</code>	Yes
Ser or get the tool head length.		
<code>"id"</code>	Int (>0)	No
Similar to the <code>jmove</code> command.		
<code>"r00"</code> , <code>"r01"</code> , <code>"r02"</code> , <code>"r10"</code> , <code>"r11"</code> , <code>"r12"</code> , <code>"r20"</code> , <code>"r21"</code> , <code>"r22"</code> , <code>"lx"</code> , <code>"ly"</code> , <code>"lz"</code>	Double	No
Parameters of the tool matrix as defined here <a href="#">tool matrix</a> . If no parameter is present, the current value of the tool matrix will be returned and other commands in the controller won't be removed.		

### Response

Key	Value
<code>"cmd"</code>	<code>"tool"</code>
In response to the <code>tool</code> command, a response is returned with the <code>cmd</code> field set to <code>tool</code> which includes the current elements of the tool matrix.	
<code>"id"</code>	Int (>0)
Same id as the original command.	
<code>"r00"</code> , <code>"r01"</code> , <code>"r02"</code> , <code>"r10"</code> , <code>"r11"</code> , <code>"r12"</code> , <code>"r20"</code> , <code>"r21"</code> , <code>"r22"</code> , <code>"lx"</code> , <code>"ly"</code> , <code>"lz"</code>	Double

The elements of the tool matrix.

### Error codes

Stat	Value
-300	Halt already in process
-400	Alarm activated
-701	The rotation matrix of the tool matrix is not a valid rotation matrix.

Example 1: This command will set the offset of the tool in z direction to 22mm and in x direction to 10mm.

Unset

```
{"cmd": "tool", "id": 12, "lz": 22, "lx": 10}
```

It will return a response as follow:

Unset

```
{"cmd": "tool", "id": 12, "r00": 1, "r01": 0, "r02": 0, "r10": 0, "r11": 1, "r12": 0, "r20": 0, "r21": 0, "r22": 1, "lx": 10, "ly": 0, "lz": 22}
```

## version

Unset

```
{"cmd": "version", ...}
```

This command returns the version of the firmware.

### Parameters

Key	Value	Required
"cmd"	"version"	Yes
Returns the version of the firmware.		
"id"	Int (>0)	No
Similar to the <code>jmove</code> command.		

### Response

Key	Value
"cmd"	"version"
In response to the <code>version</code> command response is returned with <code>cmd</code> field set to <code>version</code> which includes the current version of the firmware.	
"id"	Int (>0)
Same id as the original command.	
"version"	Int (>0)
The current version of the firmware.	

### Error codes

Stat	Value
-300	Halt already in process
-400	Alarm activated

### Example 1: Get the firmware version

Unset

```
{"cmd": "version", "id": 12}
```



In reply you will get:

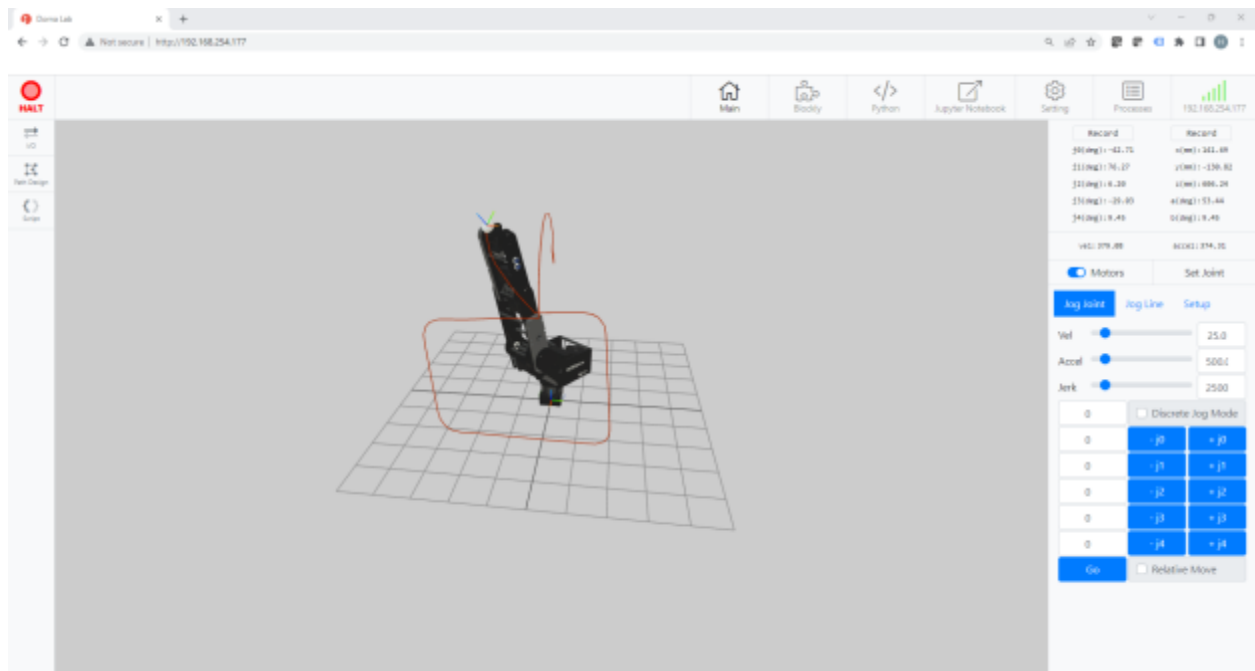
Unset

```
{"cmd": "version", "version" : 109, "id": 12}
```

## Dorna Lab

Dorna web interface, also known as **Dorna Lab**, is more or less the equivalent of the teach pendant's interface of a traditional industrial robot. The interface is essentially a web application that you can use to monitor, control, and program the robot. All of the files and packages reside in the robot's controller, so you do not need to install anything on your computer, but [Google Chrome](#).

The interface basically translates your mouse clicks, joystick movements, and keyboard entries into proprietary commands that are sent to the robot's controller. These are the same commands described in the [commands list](#). In addition, the web interface displays the feedback messages received from the robot and the 3D model of the actual robot.



## URL address

Use the latest [Google Chrome](#) version to connect to Dorna Lab and its URL address is

Unset

`http://robot_ip_address`

Where the `robot_ip_address` is the [IP address of the robot](#). For example, if the IP address of the robot is `10.0.0.14`, then the Dorna Lab URL is: <http://10.0.0.14>

## Halt button

On the top left of the screen, you have access to a [Halt](#) button. Use it to stop the robot immediately.

## Alarm information

When an [alarm](#) appears (`alarm = 1`), you will see a button pop up right beside the halt button, asking for disabling the alarm. You can also see the raw alarm message along with the errors associated with it (`err`) by clicking on the three dots right next to the **Disable Alarm** button.

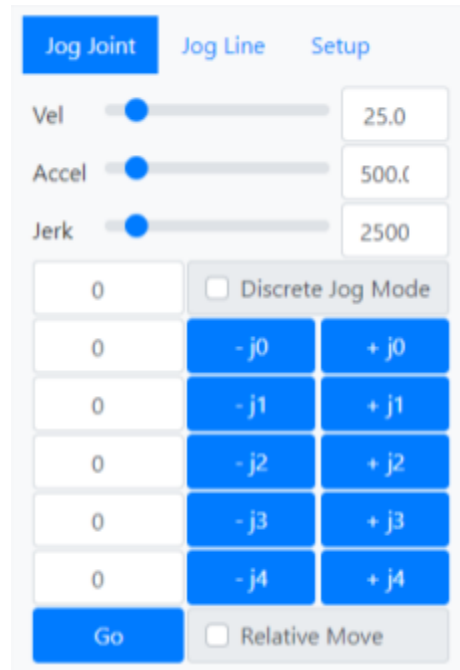
## Real-time orientation

Navigate to the **Main** tab, and you will find the robot's live orientation. This section displays the live values of the robot joints, pose, velocity, and acceleration.

<input type="button" value="Record"/>	<input type="button" value="Record"/>
<code>j0(deg): 180.000</code>	<code>x(mm): 000.000</code>
<code>j1(deg): 180.000</code>	<code>y(mm): 000.000</code>
<code>j2(deg): -142.000</code>	<code>z(mm): 000.000</code>
<code>j3(deg): 135.000</code>	<code>a(deg): 000.000</code>
<code>j4(deg): 000.000</code>	<code>b(deg): 000.000</code>
<code>vel: 000.000</code>	<code>accel: 000.000</code>

## Jogging

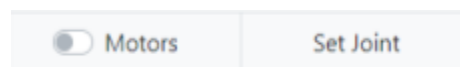
Navigate to the **Main** tab, and you will find the jogging section there. This section is used for jogging the robot with a [joint move](#) or [line move](#).



- **Motion parameters sliders:** Use the slider to set the velocity, acceleration, and jerk of the jog.
- **Jog buttons:** The jog buttons are used for moving the joint (**Jog Joint**) or TCP (**Jog Line**) in the associated direction. By default, when you press a button, the robot starts moving and stops when the button is released.
- **Discrete Jog:** Enable this feature and assign a nonzero value to the discrete jog, to precisely jog. Each time you click on one of the jog buttons, then the robot moves in the proper direction, depending on the button clicked, for the exact amount mentioned in the discrete jog input. Use this option for fine-tuning the final position of the robot.
- **Go button:** Use this button to command the robot to go to the [absolute](#) orientation mentioned in the go input fields.
- **Relative move:** When the relative move is enabled then the go button interrupts the go input fields as the [relative](#) orientation compares to the current orientation and moves the robot relative to the current position based on the values specified in the go input fields.

## Motors

Use the motors switch to enable or disable the robot motors. The robot motors are disabled by default when you turn the robot on. So, make sure to enable the motors before running any program with motion. Otherwise, you will not see any motion.





### Note

- When disabling the motors, the motors immediately lose their power, and the robot can potentially fall.
- Use this feature mainly for [hand training](#) purposes or when you need to put the robot in rest mode.

## Hand training

There are two helper record buttons on the [real-time orientation](#) section to create a motion command in the [script](#) section. The **Record** buttons on the left and right create [joint](#) and [line](#) moves respectively, based on the robot's current orientation. Use these two buttons to quickly record the position and run the result in the **script** panel.

Turn the motors off when you need to hand train the robot and move the robot with your hand. Otherwise, the robot will resist your hand force and try to keep its current orientation. If you put too much force on the robot, and the robot is not able to keep its position, then the robot goes into alarm mode. When the motors are off the robot will no longer resist your hand motion and the alarm never appears.

## Setup

In this section you can

- Disable / enable the 3D view.
- Select a Keyboard or Joystick to control the robot if needed. Visit the [Keyboard and Joystick](#) section in the setting for assigning the keys on the keyboard or joystick to the robot.

## 3D view

In this section you are able to see the live 3D view of the robot.

## I/O



Use this section, to monitor and control the robot I/Os. This section includes the digital outputs, digital inputs, and the PWM channels.

## Script

Use this section to write a script and run it on the robot.

- **Replay:** The replay switch lets you replay your script repeatedly, like a while loop.
- **Track line:** This option marks the last command completed in the script sent to the robot.

## Log

Displays the commands sent to the robot and the messages received from the robot. Time is included in each line. The commands sent from the user to the robot start with the symbol  and the message received from the robot starts with the symbol .

## Blockly editor

Use the blockly section for drag-and-drop programming. There are many blocks available to build your program. You can also see the Python interpretation of your code in the Python viewer available in this section. Once you play your code, the robot translates your Blockly design into Python code and runs it inside the robot controller.

For more examples, visit the example library at: <https://github.com/dorna-robotics/example>

## Shell viewer

This section displays the shell messages (just like a terminal in your computer). When using the [Log](#) method, your log appears here with a PID number indicating which process printed this message.

## Python editor

Use this section to program your robot in Python language.

## Processes

This section lists all the programs that have been executed on the robot since the robot's startup. It also shows their running status. You also have the option of ending a process or rerunning it.

## End a process

When you end a process, the robot OS just kills that program. That does not necessarily mean the robot stops once you end a process. For example, if your program sends multiple commands to the robot, then ending that process just stops the program from sending more

commands to the robot. Still, the commands sent to the robot will be executed according to their orders.

If you need to stop the robot completely, you must ensure that the robot queues are empty and no process sends commands to the robot. For example, you can send a [halt command](#) to clean all the queues.

## Duplicate and run a process

When you re-run a process, the robot OS clone it (takes a separate copy) and runs it. As a result, you can run multiple processes of the same program simultaneously.

## Info

This section displays information about the robot

- Device ID: The unique identifier for your robot and controller
- Model: The model of your robot.
- Firmware, Dorna Lab, and API versions: Information about the current software running on the robot and the latest version available. Click on **Check for Updates...** to see the latest versions available. You need internet access on the robot to fetch this information online. If an update is necessary, follow the [upgrade process](#).

## Auxiliary axes

Use this section to configure the auxiliary axes

Set the auxiliary axes [parameters](#): `usem`, `usee`, `pprm`, `tprm`, `ppre`, `tpre`.

Also, use the switches in the **Display** section to hide or show a specific auxiliary axis in the jog, real-time orientation, and set joint sections.

## Emergency stop

You can trigger the alarm behavior when a specific input triggers (get high or low). The alarm will remain active unless the specified input is toggled.

## Startup programs

Add a list of programs to run automatically after the robot startup. These programs will also appear in the [Processes](#) for more control over their life cycle.

## Keyboard and Joystick

Use this section to connect a keyboard or joystick to the robot. Set your desired keyboard or joystick functionality, or use the default settings.

[Xbox](#) and [PlayStation](#) controllers are all compatible with Windows computers. You can also always check your joystick's button and axis names from the [Gamepad tester](#).

Once the Keyboard or Joystick is configured, use the dropdown menu in the [setup](#) section to select the joystick or keyboard.

## Jupyter Notebook

The Jupyter Notebook tab redirects you to a [Jupyter](#) session running on the robot. The session runs on

Unset

[http://robot\\_ip\\_address:8888](http://robot_ip_address:8888)

Where the `robot_ip_address` is the IP address of the robot.

## Create a kernel

You can navigate between the folders and create a kernel by clicking on the **New** dropdown (top right).

## Shutdown a kernel

You can navigate to the following address and locate the running kernels and shut down the ones that are no longer required, to free up some CPU usage and RAM.

Unset

[http://robot\\_ip\\_address:8888/tree?#running](http://robot_ip_address:8888/tree?#running)

## Relaunch the server

If you quit the Jupyter server by accident and stop the server, then you can relaunch it again by submitting the following command in the [Shell Viewer](#) and clicking the **Submit** button:

Unset

```
nohup jupyter notebook --ip 0.0.0.0 --no-browser --port=8888
--allow-root --notebook-dir="/home/dorna/" --NotebookApp.token=' '
--NotebookApp.password=' ' &
```

## File management

You can use the robot to host your files and run your programs directly inside the robot controller without needing an external computer.

### Exploring the files

There are different ways to access the files inside the robot controller:

- Dorna Lab file browser: This is an easy way to create and manage files and folders.
- [Jupyter Notebook](#) provides a powerful platform for file management.
- Install available file transfer software on your computer to explore and manage files and folders inside the robot controller ([example](#)).

### Programs location

We strongly recommend organizing your programs in the **Projects** folder and try not to store anything outside of this folder. Create folders here to manage your projects and programs.

Unset

```
/home/dorna/Projects/
```

There are also example programs available at:

Unset

```
/home/dorna/Downloads/example/
```

### Transferring files

Use FTP and SFTP software to transfer folders and files to (from) the robot ([example](#)). You can also upload or download files using Dorna Lab and the Jupyter Notebook.



## Python API

The Python API for Dorna is a set of helper functions and modules around the [command server](#). Use the API for quick application development and easy communication with external devices.

### Useful links

- Report your technical issues for the API [here](#).
- Find some sample codes here: <https://github.com/dorna-robotics/example>

## Robot OS Python environment

Inside the robot controller, the Python environment the robot OS uses is called `python3`. If you are accessing the robot via [SSH](#), make sure to call your program via `sudo`

Unset

```
sudo python3 /path/to/the/python_file.py
```

### Add a python library

[SSH](#) to the robot and use one of the following ways to add a new module or library to the robot:

- Via pip: The `pip3` command is available in the robot OS, and if you need to install and add a package from pip server to the robot controller (with a specific `package_name`), then use the following command

Unset

```
sudo pip3 install package_name
```

- Via git: `git` command is available for the robot OS. We strongly recommend keeping all the git repos inside `/home/dorna/Projects/git/`  
To add a new repo, use the `git` command

Unset

```
sudo git clone https://github.com/sample\_repo.git
/home/dorna/Projects/git/sample_repo
```

The above command downloads a repo located at [https://github.com/sample\\_repo.git](https://github.com/sample_repo.git) to `/home/dorna/Projects/git/sample_repo`.

If the package is a module then you can also install it as a module for Python, and make it accessible to all other Python programs (read the package documentation on how to install it)

Unset

```
cd /home/dorna/Projects/git/sample_repo
sudo pip3 install -r requirements.txt
cd ..
sudo pip3 install --upgrade --force-reinstall sample_repo/
```

## Install the API

The [upgrade process](#) installs the latest Dorna Python API and other dependencies on the robot OS. So, you do not need to go over this section and try to install the latest API on the robot controller.

Follow this topic only if you need to directly run the API on your own computer and host your code and program there.



### Note

Note that the program has been tested only on Python 3.9+.

1. Download: First, use `git clone` to download the repository, or simply download the zip file and unzip the file.

Unset

```
git clone https://github.com/dorna-robotics/dorna2-python.git
```

2. Install: Next, go to the downloaded directory, where the `setup.py` file is located, and run:

Unset

```
python setup.py install --force
```



### Note

- On UNIX systems, you might need to use the `sudo` prefix for admin privileges and install the requirements.
- Depending on your Python setup environment, you might also call `python3` instead of `python`:

Unset

```
sudo python3 setup.py install --force
```

## Dorna class

### Getting started

First, import the `Dorna` class from the `dorna2` module, and then create a `Dorna` object.

Python

```
from dorna2 import Dorna
```

```
# create the Dorna object
robot = Dorna()
```

## Connection

The robot server ([WebSocket server](#)) runs on `ws://host:443`, where the `host` parameter is the host address (IP) of the robot controller, and `443` is the default port number. Once the connection has been established between the robot and the client (user), they communicate by sending and receiving data in [JSON](#) format.

```
connect(host="localhost", port=443, timeout=5)
```

Connect to the robot server at `ws://host:port`. Return `True` on a successful connection, otherwise `False`.

### Parameter

Key	Type	Default value	Required
<code>host</code>	String	<code>"localhost"</code>	No
The robot host address (IP).			
<code>port</code>	Int	<code>443</code>	No
The robot port number.			
<code>timeout</code>	Float (>0)	<code>5</code>	No
Wait a maximum of <code>timeout</code> seconds to establish a connection to the robot.			



### Note

The `host` (string) and `port` (integer) arguments are similar to the Python [`socket.connect\(\(host, port\)\)`](#) method.

## close()

Use this method to close an opened connection. This method instantly closes the socket and terminates the communication loop. After this, the `Dorna` object cannot send or receive any message from (to) the robot unless you [connect](#) to the robot again.



### Note

Once your task is completed and the connection is no longer required, it is necessary to close the open connection.

Python

```
from dorna2 import Dorna

# create the Dorna object
robot = Dorna()

# check if the connection is successful and then run you code
if robot.connect("10.0.0.14"):
    #####
    # your code goes here #
    #####

# always close the socket when you are done
robot.close()
```

## Command status

### track\_cmd()

Return the replies of the last commands sent to the robot via the same `Dorna` object. This method returns a nested [Python dictionary](#) with three main keys as follows:

- `"cmd"`: The value assigned to this key is a [Python dictionary](#), representing the initial command sent to the robot.

- "msgs": The value assigned to this key is a [list](#) of all [replies from the robot](#) controller with the same `id` as the initial command. Each element in the list is a [Python dictionary](#). Elements in the list are also sorted in ascending order based on the time they have been received by the API. So, the first element in the list was received earlier than the last element in the list.
- "union": This is a [Python dictionary](#) formed by merging all the elements in the "msgs" list and keeping the most recent value for each key.

Here is an example of showing the result of `.track_cmd()`, based on the replies we got in the [status of a command](#) section.

```
Python
robot.track_cmd()
"""
{
  "cmd": {"cmd": "alarm", "id": 12},
  "all": [{"id": 12, "stat": 0},
          {"id": 12, "stat": 1},
          {"cmd": "alarm", "id": 12, "alarm": 0},
          {"id": 12, "stat": 2}],
  "merge": {"id": 12, "stat": 2, "cmd": "alarm", "alarm": 0}
}
"""
```

## Sending command

In this section we cover methods to send commands to the robot.

### `play(timeout=-1, **kwargs)`

Send a command to the robot, and return a [.track\\_cmd\(\)](#) object associated with the command sent.

There are multiple ways to send a message via `.play()`. For a better understanding, we send a simple [alarm](#) status command in three different ways:

- Case 1: (Recommended) Key and value format: `play(cmd="alarm", id=10)`
- Case 2: [Python dictionary](#) format: `play({"cmd": "alarm", "id": 10})`
- Case 3: [JSON string](#) format: `play('{"cmd": "alarm", "id": 10}')`

#### Parameter

Key	Type	Default value	Required
<code>timeout</code>	Float	-1	No

We can assign different values to the `timeout` parameter depending on your code logic:

- `timeout < 0`: Send command and wait for the command completion (`stat = 2` or `stat < 0`) and then return from the function. At this moment, we are sure that the command is no longer running in the robot.
- `timeout >= 0`: Send a command and wait for a maximum of `timeout` seconds for its completion. Notice that in this case, we might have returned from the `.play()` method, but the command which was sent to the robot is still running or waiting inside for the controller queue for its turn to get executed. If we do not want to wait for the execution of a command at all, then we can always set `timeout = 0`.

<code>msg</code>	Python dictionary or JSON string	None	No
------------------	----------------------------------	------	----

Use this parameter if you want to send your command in a [Python dictionary](#) format (Case 2), or in a [JSON format](#) (Case 3).

<code>kwargs</code>			No
---------------------	--	--	----

Use this parameter to send your [command](#) in a key and value format.



### Note

- Throughout this API, we use and refer to the [timeout](#) key as an argument inside many methods that are sending commands to the robot. These functions use the [timeout](#) argument for tracking the completion or any error during the execution of the command that they are sending.
- The `.play()` method and any other method sending commands to the robot always includes a random `id` field to your command if it is not present.

For a better understanding of the [timeout](#) parameter, we send a [joint move](#) command to the robot in four different ways.

Python

```
import time

# motion 1
start = time.time()
robot.play(timeout=-1, cmd="jmove", rel=1, j0=10, vel=1)
print("Motion 1 is completed, and took ", time.time()-start, "
seconds.")

# motion 2
start = time.time()
robot.play(timeout=100, cmd="jmove", rel=1, j0=10, vel=1)
print("Motion 2 is completed, and took ", time.time()-start, "
seconds.")

# motion 3
robot.play(timeout=2, cmd="jmove", rel=1, j0=10, vel=1)
print("2 seconds has passed and motion 3 is still running.")

# motion 4
robot.play(timeout=0, cmd="jmove", rel=1, j0=10, vel=1)
print("Motion 3 is still running and motion 4 is waiting for its
execution.")

# Output:
#
# Motion 1 is completed and took 10.199519157409668 seconds.
# Motion 2 is completed and took 10.207868814468384 seconds.
# 2 second has passed and motion 3 is still running.
# Motion 3 is still running and motion 4 is waiting for its
execution.
```



```
play_dict(cmd={}, timeout=-1)
```

similar to the [.play\(\)](#) method, but this time sends a command to the robot in a Python dictionary format.

```
play_json(cmd='{}', timeout=-1)
```

Similar to the [.play\(\)](#) method, this time send a command to the robot in a JSON string format. In this example, we send a similar command sent in the [.play\(\)](#) method, in both dictionary and JSON format.

Python

```
# play_dict
robot.play_dict({"cmd":"jmove", "rel":1, "j0":10, "vel":1})

# play_json
robot.play_json('{"cmd":"jmove", "rel":1, "j0":10, "vel":1}')
```

```
play_script(file="", timeout=-1)
```

Send all the messages that are stored in a script file to the robot controller. The method opens the script file located at the `file`, reads the file line by line, and sends each line as a command instantly.

#### Parameter

Key	Type	Default value	Required
<code>timeout</code>	Float	-1	No

The `timeout` parameter acts similarly to the [timeout](#) parameter in [.play\(\)](#) method:

- `timeout < 0`: The method sends all the commands in the script, and returns when all those commands are completed.
- `timeout >= 0`: The method sends all the commands in the script file and waits a maximum of `timeout` seconds for the completion of those commands before returning.

<code>script_path</code>	String	""	Yes
--------------------------	--------	----	-----

Path to the script file.

This method returns the overall status of the commands sent:

- 2: If all the commands in the script file are completed.
- 0 or 1: If the commands in the script file are still running.
- < 0: If there is any error during the execution of the commands in the script file.



## Note

Use this function to send multiple messages at once to the robot. Notice that each message has to occupy exactly one line. Multiple messages in one line or one message in multiple lines is not a valid format. As an example, here we show a valid and invalid script format:

Unset

```
# valid format: Each message occupies exactly one line
{"cmd": "jmove", "rel": 0, "j0": 0}
{"cmd": "jmove", "rel": 0, "j0": 10}
{"cmd": "jmove", "rel": 0, "j0": -10}

# invalid format: Multiple commands in one line or one command in
multiple lines
{"cmd": "jmove", "rel": 0, "j0": 0}{ "cmd": "jmove", "rel": 0, "j0": 10}
{"cmd": "jmove", "rel": 0,
"j0": -10}
```

Here are two examples, of how to run a script file multiple times using a simple loop:

Python

```
# case 1: run a script 10 times
for i in range(10):
    robot.play_script("test.txt")
    robot.log("Script is completed")
```

Another example, that also keeps track of each script file as well (the safe way):

Python

```
# case 2: a safe way of running a script in a for loop, by
checking the return status of the script
for i in range(10):
    stat= robot.play_script("test.txt")
    if stat!= 2:
        robot.log("Error happened")
        break
    robot.log("Script is completed")
```

## Messages

`last_cmd()`

Return the last command sent to the robot, in a [Python dictionary](#) format.

`last_msg()`

Return the last message received from the controller, in a [Python dictionary](#) format.

`union()`

Return a [Python dictionary](#), consisting of all the keys and their most up-to-date values received from the controller since the [connection](#) has been established with the robot.

`val(key="cmd")`

Return the value of an specific key of the [union\(\)](#).

Python

```
print(robot.last_msg())
```

```
# Output:
```

```
#
```

```
{'cmd': 'output', 'id': 81513, 'out0': 1, 'out1': 0, 'out2': 0, 'out3': 0, 'o
```

```

ut4':0,'out5':0,'out6':0,'out7':0,'out8':0,'out9':0,'out10':0,'ou
t11':0,'out12':0,'out13':0,'out14':0,'out15':0}

print(robot.val("out0"))

# print => 1

```

## Move

In this section we cover robot motion functions.

```
jmove(timeout=-1, **kwargs)
```

A helper function to send a [joint move](#) (`jmove`) command to the robot, and return the stat of the motion command sent.

This method is basically similar to the `.play()` method but the `cmd` key is set to "jmove". So, `.jmove(rel=1, j0=10, id=10)` is equivalent to `.play(cmd='jmove', rel=1, j0=10, id=10)`.

### Parameter

Key	Type	Default value	Required
<code>timeout</code>	Float	-1	No
Similar to the <a href="#">timeout</a> in the <code>.play()</code> method			
<code>kwargs</code>	Key and value		Yes
The keys and values associated with the <a href="#">joint move</a> ( <code>jmove</code> ) command.			

```
lmove(timeout=-1, **kwargs)
```

A helper function to send a [line move](#) (`lmove`) command.

### Parameter

Key	Type	Default value	Required
<code>timeout</code>	Float	-1	No

Similar to the [timeout](#) in the [.play\(\)](#) method

<code>kwargs</code>	Key and value	Yes
---------------------	---------------	-----

The keys and values associated with the [line move](#) (`lmove`) command.

```
cmove(timeout=-1, **kwargs)
```

A helper function to send a [circle move](#) (`cmove`) command.

#### Parameter

Key	Type	Default value	Required
<code>timeout</code>	Float	-1	No

Similar to the [timeout](#) in the [.play\(\)](#) method

<code>kwargs</code>	Key and value	Yes
---------------------	---------------	-----

The keys and values associated with the [circle move](#) (`cmove`) command.

## Stop

Series of helper functions to send stop ([halt](#)) commands, read and set the [alarm](#) status of the robot.

```
halt(accel=None)
```

A helper function to send a [halt](#) command to the robot, with a given acceleration ratio (`accel`), and return the final status of the halt command (`stat`).

#### Parameter

Key	Type	Default value	Required
<code>accel</code>	Double (> =1)	None	No

The acceleration ratio parameter associated with the [halt](#) command.

Python

```
robot.halt() # sends a halt command to the controller
robot.halt(5) # send a halt command with an acceleration ratio
equal to 5
```

### `get_alarm()`

Get the robot alarm status (0 for disabled and 1 for enabled).

### `set_alarm(enable=None)`

Enable or disable the alarm status of the robot (set `enable` to 0 for disabling and 1 for enabling the alarm), and return the final status of the command (`stat`).

Python

```
# Disable the alarm, if the alarm exists
if robot.get_alarm():
    robot.set_alarm(0)
```

## Joint and TCP

In this section we cover methods that are related to robot orientation.

### `get_all_joint()`

Get the joint values of the robot, in a list of size 8. Where index `i` in the list is the value of joint `i` (`ji`).

### `get_joint(index=None)`

Get the value of the joint `index` ( $0 \leq \text{int} < 8$ ).

### `set_joint(index=None, val=None)`

Set the value of the joint `index` ( $0 \leq \text{int} < 8$ ) to `val` (float) and return the final status of the joint command (`stat`) sent to the robot.

Python

```
# [180, 180, -142, 135, 0, 0, 0, 0]
robot.get_all_joint()

# return the value of j1: 180
robot.get_joint(1)

# set the value of j1 to 30
robot.set_joint(1, 30)
```

### get\_all\_pose()

Get the value of the robot tool head (TCP) in the Cartesian coordinate system (with respect to the robot base frame). in a list of size 8. Where indices 0 to 7 in this list are associated with the coordinates *x*, *y*, *z*, *a*, *b*, *c*, *d*, *e* respectively.

### get\_pose(index=None)

Get the value of the TCP *index* ( $0 \leq \text{int} < 8$ ).

### get\_tool()

Get the value of the robot tool length in *mm*. The tool length is measured in the *Z* direction of the robot flange frame.

```
set_tool(r00=None, r01=None, r02=None, r10=None, r11=None,
r12=None, r20=None, r21=None, r22=None, lx=None, ly=None,
lz=None )
```

Set the robot tool length (*mm*) to *length* and return the final status of the tool length command (*stat*) sent to the robot.

Python

```
robot.get_tool() # get the robot tool length in mm
robot.set_tool(lz=10) # set the robot tool length in z direction
to 10 mm
```

## I/O

In this section we cover methods that are related to the robot inputs and outputs.

### `get_all_output()`

Get the value of all the 16 output pins in a list of size 16. Where item `i` in the list is the value of `outi`.

### `get_output(index=None)`

Get the value of output pin `index` ( $0 \leq \text{int} < 16$ ).

### `set_output(index=None, val=None, queue=None)`

Set the value of the output pin `index` ( $0 \leq \text{int} < 16$ ) to `val` (0 or 1) and return the final status of the output command (`stat`) sent to the robot.

Python

```
robot.get_all_output() # return the value of all the 16 outputs
                        in a list of size 16
robot.get_output(0) # return the value of the out0
robot.set_output(0, 1) # set the value of the out0 to 1
```

### `get_pwm(index=None)`

Get the value of the `pwm` channel `index` ( $0 \leq \text{int} < 5$ ).

### `set_pwm(index=None, enable=None, queue=None)`

Enable or disable (set `enable` to 0 for disable and 1 for enable) the PWM channel `index` ( $0 \leq \text{int} < 5$ ), and return the final status of the PWM command (`stat`) sent to the robot.

Python

```
robot.get_pwm(0) # return the value of the pwm0
robot.set_pwm(0, 1) # enable pwm channel 0
```



### `get_freq(index=None)`

Get the frequency of a pwm channel `index` ( $0 \leq \text{int} < 5$ ).

### `set_freq(index=None, freq=None, queue=None)`

Set the frequency value of the PWM channel `index` ( $0 \leq \text{int} < 5$ ) to `freq` ( $0 \leq \text{float} \leq 120,000,000$ ) and return the final status of the PWM command (`stat`) sent to the robot.

Python

```
robot.get_freq(0) # return the frequency value of the PWM channel
0 (freq0)
robot.set_freq(0, 1000) # set freq0 to 1000
```

### `get_duty(index=None)`

Get the duty cycle of the PWM channel `index` ( $0 \leq \text{int} < 5$ ).

### `set_duty(index=None, duty=None, queue=None)`

Set the duty cycle of the PWM channel `index` ( $0 \leq \text{int} < 5$ ) to `duty` ( $0 \leq \text{float} \leq 100$ ), and return the final status of the PWM command (`stat`) sent to the robot.

Python

```
robot.get_duty(0) # return the value of duty0
robot.set_duty(0, 10) # set the value of duty0 to 1 and return
its value
```

### `get_all_input()`

Get the value of all the input pins in a list of size 16, where index `i` in the list is the value of `ini`.

### `get_input(index=None)`

Get the value of the input pin `index` ( $0 \leq \text{int} < 16$ )

Python

```
robot.get_all_input() # return the value of all the 16 input pins
in a list of size 16
robot.get_input(0) # return the value of in0
```

### get\_all\_adc()

Get the value of all the adc channels in a list of size 5, where item *i* in the list is the value of *adci*.

### get\_adc(index=None)

Get the value of the adc channel *index* ( $0 \leq \text{int} < 5$ )

Python

```
robot.get_all_adc() # return the value of all the 5 ADC channels
in a list of size 5
robot.get_adc(0) # return the value of adc0
```

### Wait and delay

Wait for an input pin pattern, encoder indices or delay for a certain amount of time in the program.

### probe(index=None, val=None)

Return the joint values of the robot in a list of size 8 ([.get\\_all\\_joint\(\)](#)), the moment that the input pin *index* ( $0 \leq \text{int} < 16$ ), is equal to the *val* (0 or 1).



#### Note

Use this method to wait for a pattern of inputs.

Python

```
robot.probe(1, 0) # return the joint values, the moment in1 gets  
equal to 0
```

### `iprobe(index=None, val=None)`

This method is similar to the `probe` function but here we are waiting for a specific pattern in the encoder index, instead of an input pin. Return the joint values of the robot in a list of size 8 (`.get_all_joint()`), the moment that the encoder `index` ( $0 \leq \text{int} < 8$ ), is equal to the `val` (0 or 1).



#### Note

Notice that the encoder on the motors gets high (1), 8 times during one full rotation of the encoder, and we can locate these points by calling the `.iprobe` function.

Python

```
robot.iprobe(1, 1) # return the value of the joints, the moment  
that index1 (encoder 1 index) gets 1
```

### `sleep(val=None)`

Sleep for `val` (float  $\geq 0$ ) seconds and return the status of the command.

Python

```
robot.sleep(10) # the controller sleeps for 10 seconds
```

## Setting

### `get_motor()`

Get the robot motors status (0 for disabled and 1 for enabled).

### set\_motor(enable=None)

Enable or disable the motors and return the final status of the motor command (`stat`) sent to the robot.

Python

```
robot.get_motor() # get the robot motor status
robot.set_motor(0) # disable the motors
```

### get\_axis(index=None)



#### Note

This method applies only to Dorna TA.

Get the [axis](#) parameters of the auxiliary axis `index` ( $6 \leq \text{int} < 8$ ), in a list of size 6, [`usem`, `usee`, `pprm`, `tprm`, `ppre`, `tpre`].

```
set_axis(index=None, usem=None, usee=None, pprm=None, tprm=None,
ppre=None, tpre=None)
```



#### Note

This method applies only to Dorna TA.

Set the [axis](#) parameters of the auxiliary axis `index` ( $5 \leq \text{int} < 8$ ), and return the final status of the axis command (`stat`) sent to the robot.

### get\_pid(index=None)

Return the PID parameters (`p`, `i`, `d`, `the`, `dur`) of the joint `index` ( $0 \leq \text{int} < 7$ ) in a list of size 3 (`[p, i, d]`).

`set_pid(index=None, p=None, i=None, d=None, thr=None, dur=None)`

Set the PID parameters (`p`, `i`, `d`, `thr`, `dur`) of the joint `index` ( $0 \leq \text{int} < 7$ ), and return the final status of the `pid` command (`stat`) sent to the robot.

## Info

`version()`

Get the firmware version of the robot.

`uid()`

Get the robot controller's Universal Identification number.

Python

```
robot.version() # get the firmware version
robot.uid() # get the controller UID
```

## Log

`log(msg)`

Print a message in a terminal and a file at `dorna.log`.

The printed format includes time in the beginning followed by the `msg`.

Get the firmware version of the robot.

`logger_setup(file="dorna.log")`

The default path for the logs is at `dorna.log`. If you need to change the path, use this method and assign a different log file to store the logs.

Python

```
if robot.connect():
    robot.log("connected")

""" output
2023-06-14 03:27:29,868 connected
```

```
"""
```

## Event

Every time a message is received from the robot, we can call (trigger) a function. This is useful when you want to create an event, based on the messages received from the robot.

`get_all_event()`

Return the list of all the running events as a list.

`add_event(target=None, kwargs={})`

Register a function `target` to be called every time a message is received from the robot controller.

Format of the `target`

The `target` function always starts with two required parameters, `msg`, and `union`. Other necessary parameters can also be passed via `**kwargs`

```
Python
```

```
target(msg, union, **kwargs)
```

- `msg` is the message received from the controller when the target was called ([last\\_msg\(\)](#)).
- `union` is the dictionary defined by the [union\(\)](#), the moment that target was called.
- `**kwargs` are all the remaining parameters we pass to the `target` function.

`.clear_event(target=None)`

This method acts opposite of [add\\_event\(\)](#), and it removes the event (function) `target` from the event list.

It is important to call this method when we no longer need the registered event.

## `.clear_all_event()`

Removes all the events from the event list.

### Example

Assume that you have a program running in a while loop, and you want to stop and exit the program when `out0` is enabled (1). So, we register an event that checks the messages received from the robot and send a halt command if `out0 == 1`.

Python

```
from dorna2 import Dorna

# halt the robot when "out0" gets to 1
def stop_event(msg, union, dorna_robot):
    if "out0" in msg and msg["out0"] == 1:
        # instant stop
        dorna_robot.halt()

        # change the robot state to alarm to ensure that the
        robot ignores all the future commands
        dorna_robot.set_alarm(1)

        # exit the while loop of the main function
        dorna_robot.prm_stop = True

def main(robot):
    # initial stop condition
    robot.prm_stop = False

    # register a stop event
    robot.add_event(target=stop_event, kwargs={"dorna_robot":
robot})

    # motion loop
    while not robot.prm_stop:
```

```
robot.log("forward motion")
robot.jmove(rel=1, j0=10)

robot.log("reverse motion")
robot.jmove(rel=1, j0=-10)

if __name__ == '__main__':
    ip = "localhost"
    robot = Dorna()

    # connect to the robot
    if robot.connect(ip):
        main(robot)

    # close the connection
    robot.close()
```

## Troubleshooting

### No LEDs are on upon power up

There are LED lights on the Ethernet port of the robot controller. If they are not blinking when turning on the controller:

- Make sure all connectors are properly attached.
- Make sure you use the right AC voltage for the robot (If you use the wrong AC outlet, then immediately turn off the robot).

### No connection to the robot's web interface

- If you are using an Ethernet cable, make sure the Ethernet cable is properly connected. The green Ethernet LED should blink. If the green LED is not illuminated, detach and reconnect the Ethernet cable.
- Make sure that the router/switch works by checking the LEDs of the connection socket.
- Make sure you are connected to the same network as the robot.



- If you are using static IP addresses, ensure the robot's default IP address does not conflict with any other device on the network.
- If you use a dynamic IP address (DHCP), verify the robot's IP address via your router's web interface or other IP scanner software ([example](#)).
- Verify that you can [SSH](#) to the robot.
- If you just went over an [upgrade process](#), which was unsuccessful, rerun the upgrade steps again.

## The robot fails to boot

- Disconnect the power cable from the AC outlet, reconnect the power cable, and turn the robot's controller on.

## Joint and position lost

- Properly home the robot.
- Make sure all the pulleys are tight inside their shaft (tighten their screws if it is necessary)

## High-temperature motors

- Make sure that the motors are running under  $70^{\circ}\text{C}$ .
- Make sure that the environment temperature is under  $50^{\circ}\text{C}$ .
- Continuously running the robot at high speed and also high payload can increase the motor's temperature over time.
- Double check motor driver settings to apply the right current to each motor (follow the instruction [here](#)).

## Maintenance

### Check for upgrade

Periodically check for the [upgrades](#) and make sure that your robot runs the latest firmware and software.

### Cables and wires

Periodically check the moving cables inside the robot and check for any signs of wear and damage.

## Belts

Unlike other industrial robots, the Dorna TA requires no lubrication. Occasionally, check the robot belts and look for any sign of wear, defect, or damage. You might still be able to operate the robot with a defective belt, but these belts are more prone to damage and failure in the future. So, ensure you have the spare parts ready if you need to change the belts.



### Note

For more information about spare parts, make sure to [contact](#) us.

## Connectors

Make sure that the connectors for encoders and motors are not hitting and colliding with any object.