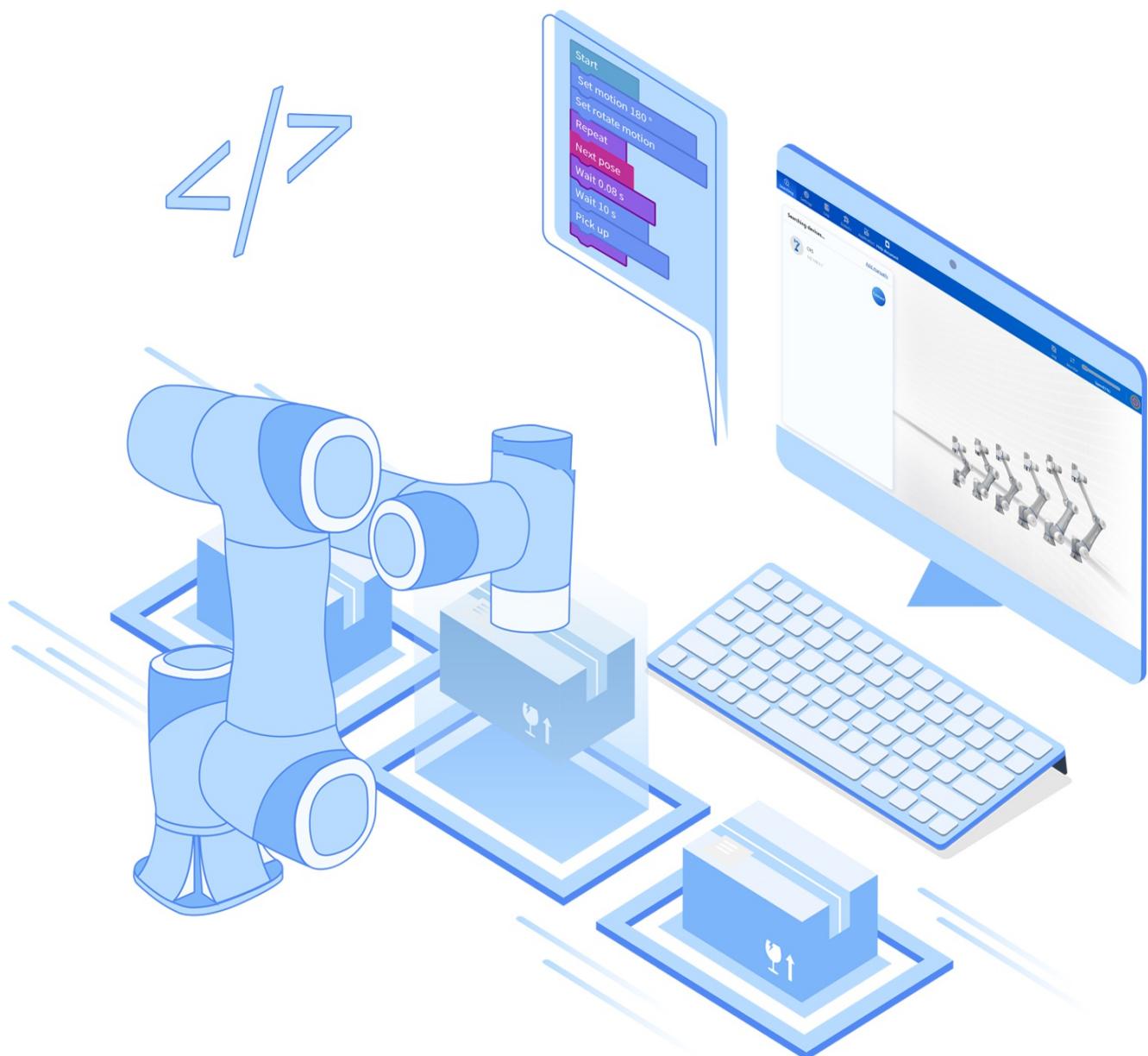




# DobotStudio Pro User Guide

## (Six-axis Robots)



# Table of Contents

---

## Preface

---

### 1 Getting Started

---

1.1 Overview

---

1.2 Configuration requirements

---

1.3 DobotStudio Pro installation

---

1.4 How to use this Guide

---

### 2 Connecting to Robot

---

2.1 Power on

---

2.2 Wireless connection

---

2.3 Wired connection

---

2.4 Manual adding

---

### 3 Interface Overview

---

3.1 Main interface

---

3.2 Settings interface

---

3.3 Log interface

---

3.4 Dobot+ interface

---

3.5 Application interface

---

3.6 Jog panel

---

3.7 Monitor panel

---

### 4 Quick Start

---

### 5 Robot Basic Operations

---

5.1 User login

---

5.2 Enable

---

5.3 Recovery mode

---

5.4 Remote control

---

5.4.1 Device mode

---

5.4.2 IO/Modbus configuration

---

5.5 Manual/Automatic mode

---

5.6 Jog

---

5.7 Drag

---

5.8 Emergency stop and recovery

5.9 Speed

---

5.10 Load settings

---

5.11 Collision detection settings

---

5.12 Alarm

---

## 6 Application

---

6.1 Application main interface

---

6.2 Blockly programming

---

6.2.1 Overview

---

6.2.2 Project management

---

6.2.3 Points page

---

6.2.4 Programming

---

6.3 Script programming

---

6.3.1 Overview

---

6.3.2 Project management

---

6.3.3 Points page

---

6.3.4 Programming

---

6.4 Python programming (Magician E6)

---

6.4.1 Overview

---

6.4.2 Project management

---

6.4.3 Points page

---

6.4.4 Programming

---

6.5 Debugging and running (blockly/script programming)

---

6.6 Trajectory recovery

---

## 7 Dobot+

---

## 8 Monitor

---

8.1 Controller DI/DO

---

8.1.1 DI/DO monitoring

---

8.1.2 I/O settings

---

8.2 Controller AI/AO

---

8.3 Tool I/O

---

8.4 Safety I/O

---

8.5 Modbus

---

8.5.1 Modbus monitoring

---

8.5.2 Modbus settings

---

---

## 8.6 Global Variable

---

## 8.7 Program variables

---

## 9 Log

---

## 10 Settings

---

### 10.1 System settings

---

### 10.2 User management

---

### 10.3 Coordinate system management

---

#### 10.3.1 User coordinate system

---

#### 10.3.2 Tool coordinate system

---

### 10.4 Load parameters

---

### 10.5 Button settings (Magician E6)

---

### 10.6 Motion parameters

---

#### 10.6.1 Motion parameters (CRA)

---

#### 10.6.2 Motion parameters (Magician E6)

---

### 10.7 Posture settings

---

### 10.8 Trajectory playback

---

### 10.9 Communication settings

---

### 10.10 Installation settings

---

### 10.11 Drag settings

---

### 10.12 Power voltage (DC controller/Magician E6)

---

### 10.13 Security settings

---

#### 10.13.1 Collision detection (Magician E6)

---

#### 10.13.2 Safety limits (CRA series)

---

#### 10.13.3 Joint limits (CRA series)

---

#### 10.13.4 Safety wall

---

#### 10.13.5 Safety zone

---

#### 10.13.6 Safe home position

---

#### 10.13.7 Joint brake

---

### 10.14 Operation mode settings

---

### 10.15 Home calibration

---

### 10.16 Advanced functions

---

### 10.17 File migration

---

### 10.18 Firmware upgrade

---

## Appendix A Modbus Register Definition

---

---

A.1 Modbus introduction

---

A.2 Coil register (map1, control robot)

---

A.3 Contact register (map1, robot status)

---

A.4 Input register (map1, robot real-time feedback data)

---

A.5 Holding register (map1, interaction between robot and PLC)

---

## Appendix B Blockly Programming Command

---

B.1 General description

---

Block types

---

Motion mode

---

Coordinate system parameters

---

Speed parameters

---

CP parameters

---

Stop condition

---

B.2 Event

---

Start command

---

Sub-thread start command

---

B.3 Control

---

Wait until...

---

Repeat n times

---

Repeat continuously

---

End repeat

---

if...then...

---

if...then...else...

---

Repeat until...

---

Set label

---

Goto label

---

Fold command

---

Pause

---

Stop program

---

Custom log

---

Set collision detection

---

Set collision backoff distance

---

Modify user or tool coordinate system

---

Calculate and update user coordinate system

---

Calculate and update tool coordinate system

---

Set user coordinate system

---

Set tool coordinate system

---

Set payload parameters

---

Wait

---

Set safety wall switch

---

Set safety zone switch

---

Get system time

---

Start timing

---

Get timing result

---

Set tool mode

---

Set data type for tool RS485

---

Set tool power switch

---

Custom popup

---

Comment

---

#### B.4 Operator

---

Arithmetic command

---

Comparison command

---

AND command

---

OR command

---

Not command

---

Modulo command

---

Rounding command

---

Unary operation

---

Print command

---

#### B.5 String

---

Get the Nth character of a string

---

Check if String A contains String B

---

Concatenate two strings

---

Get length of string or array

---

Compare two strings

---

Convert Array to String

---

Convert String to Array

---

Get Element at a specific Index in an Array

---

Get multiple Elements from an Array

---

Set Element at a specific Index in an Array

---

## B.6 Custom

---

Call global variable

---

Set global variable

---

Create custom variable

---

Custom number variable

---

Set value of custom number variable

---

Add value of number variable

---

Custom string variable

---

Set value of custom string variable

---

Create array

---

Custom array

---

Add variable to array

---

Delete item from array

---

Delete all items from array

---

Insert item into array

---

Replace item in array

---

Get item from array

---

Get total number of items in array

---

Create a function

---

Custom function

---

Create a subroutine

---

Subroutine

---

## B.7 IO

---

Set digital output

---

Check digital output status

---

Set a group of digital outputs

---

Wait for a group of digital outputs

---

Wait for digital input

---

Wait for a group of digital inputs

---

Set analog output

---

Get analog output

---

Check digital input status

---

Get analog input

---

## B.8 Motion

---

Move to target point

---

Relative motion along coordinate system

---

Get point after offsetting along coordinate system

---

Joint offset motion

---

Get point after joint offset

---

Arc motion

---

Circle motion

---

Trajectory playback

---

Set CP ratio

---

Set joint speed ratio

---

Set joint acceleration ratio

---

Set linear speed ratio

---

Set linear acceleration ratio

---

Set global speed ratio

---

Modify coordinates of a specified point

---

Get coordinates of a specified point

---

Check motion feasibility

---

Get coordinates of a specified axis

---

Get joint angle of specified point

---

Forward kinematics (joint angles to posture)

---

Inverse kinematics (posture to joint angles)

---

Get encoder value

---

## B.9 Modbus

---

Create Modbus master

---

Create tool-RS485-based Modbus master

---

Create RS485-based Modbus master

---

Get result of creating Modbus master

---

Wait for input register

---

Wait for holding register

---

Wait for contact register

---

Wait for coil register

---

Get input register

---

Get holding register

---

Get contact register

---

Get coil register

---

Get multiple values of coil register

---

Get multiple values of holding register

---

Set coil register

---

Set multiple coil registers

---

Set holding register

---

Close Modbus master

---

## B.10 Bus

---

Get bus register value

---

Set bus register value

---

## B.11 TCP

---

Connect SOCKET

---

Get result of connecting SOCKET

---

Create SOCKET

---

Get result of creating SOCKET

---

Close SOCKET

---

Get variable

---

Get result of reading variables

---

Send variable

---

Get result of sending variables

---

## B.12 Tray

---

Create tray

---

Get total tray points

---

Get tray point

---

## B.13 Quick start

---

Reading and writing Modbus register data

---

Transmitting data via TCP communication

---

# Appendix C Script Programming Command

---

## C.1 Basic syntax

---

Basic concepts

---

Variable and data type

---

Operator

---

Flow control

---

Functions

---

General mathematical functions

---

General string-processing functions

---

General table (array) functions

---

## C.2 General description

---

Motion mode

---

Point parameters

---

Coordinate system parameters

---

Speed parameters

---

Continuous path parameters

---

Stop condition

---

IO signal

---

## C.3 Motion

---

Command list

---

MovJ

---

MovL

---

Arc

---

Circle

---

MovJIO

---

MovLIO

---

GetPathStartPose

---

StartPath

---

PositiveKin

---

InverseKin

---

## C.4 Relative motion

---

Command list

---

RelPointUser

---

RelPointTool

---

RelMovJTool

---

RelMovLTool

---

RelMovJUser

---

RelMovLUser

---

RelJointMovJ

---

RelJoint

---

### C.5 Motion parameters

---

Command list

---

CP

---

VelJ

---

AccJ

---

VelL

---

AccL

---

SpeedFactor

---

SetPayload

---

User

---

SetUser

---

CalcUser

---

Tool

---

SetTool

---

CalcTool

---

GetPose

---

GetAngle

---

GetABZ

---

CheckMovJ

---

CheckMovL

---

SetSafeWallEnable

---

SetWorkZoneEnable

---

SetCollisionLevel

---

SetBackDistance

---

### C.6 IO

---

Command list

---

DI

---

DIGroup

---

DO

---

DOGGroup

---

GetDO

---

GetDOGGroup

---

AI

---

[AO](#)

---

[GetAO](#)

---

#### [C.7 Tool](#)

---

[Command list](#)

---

[ToolDI](#)

---

[ToolDO](#)

---

[GetToolDO](#)

---

[ToolAI](#)

---

[SetToolMode](#)

---

[GetToolMode](#)

---

[SetToolPower](#)

---

[SetTool485](#)

---

#### [C.8 TCP&UDP](#)

---

[Command list](#)

---

[TCPCreate](#)

---

[TCPStart](#)

---

[TCPRead](#)

---

[TCPWrite](#)

---

[TCPDestroy](#)

---

[UDPCreate](#)

---

[UDPRead](#)

---

[UDPWrite](#)

---

#### [C.9 Modbus](#)

---

[Command list](#)

---

[ModbusCreate](#)

---

[ModbusRTUCreate](#)

---

[ModbusClose](#)

---

[GetInBits](#)

---

[GetInRegs](#)

---

[GetCoils](#)

---

[SetCoils](#)

---

[GetHoldRegs](#)

---

[SetHoldRegs](#)

---

#### [C.10 Bus register](#)

---

Command list

---

GetInputBool

---

GetInputInt

---

GetInputFloat

---

GetOutputBool

---

GetOutputInt

---

GetOutputFloat

---

SetOutputBool

---

SetOutputInt

---

SetOutputFloat

---

### C.11 Program control

---

Command list

---

Print

---

Log

---

Wait

---

Pause

---

Halt

---

ResetElapsedTime

---

ElapsedTime

---

Systime

---

SetGlobalVariable

---

Popup

---

### C.12 Tray

---

Command list

---

CreateTray

---

GetTrayPoint

---

### C.13 SafeSkin

---

Command list

---

EnableSafeSkin

---

SetSafeSkin

---

## Appendix D Remote Control Signal Sequence Diagram

---

## Appendix E Python Programming Command

---

### E.1 Basic syntax

---

### E.2 General description

---

Motion mode

---

Point parameters

---

Coordinate system parameters

---

Speed parameters

---

Continuous path parameters

---

IO signal

---

### E.3 Motion

---

Command list

---

MovJ

---

MovL

---

Arc

---

Circle

---

MovJIO

---

MovLIO

---

StartPath

---

GetPathStartPose

---

PositiveKin

---

InverseKin

---

### E.4 Relative motion

---

Command list

---

RelPointUser

---

RelPointTool

---

RelMovJTool

---

RelMovLTool

---

RelMovJUser

---

RelMovLUser

---

RelJointMovJ

---

RelJoint

---

### E.5 Motion parameters

---

Command list

---

CP

---

VelJ

---

AccJ

---

VelL

---

AccL

---

SpeedFactor

---

SetPayload

---

User

---

SetUser

---

CalcUser

---

Tool

---

SetTool

---

CalcTool

---

GetPose

---

GetAngle

---

GetABZ

---

CheckMovJ

---

CheckMovL

---

SetSafeWallEnable

---

SetWorkZoneEnable

---

SetCollisionLevel

---

SetBackDistance

---

## E.6 IO

---

Command list

---

DI

---

DIGroup

---

DO

---

DOGroup

---

GetDO

---

GetDOGroup

---

AI

---

AO

---

GetAO

---

## E.7 Tool

---

Command list

---

ToolDI

---

ToolDO

---

GetToolDO

---

[SetToolPower](#)

---

## E.8 TCP&UDP

---

[Command list](#)

---

[TCPCreate](#)

---

[TCPStart](#)

---

[TCPRead](#)

---

[TCPWrite](#)

---

[TCPDestroy](#)

---

[UDPCreate](#)

---

[UDPRead](#)

---

[UDPWrite](#)

---

## E.9 Modbus

---

[Command list](#)

---

[ModbusCreate](#)

---

[ModbusRTUCREATE](#)

---

[ModbusClose](#)

---

[GetInBits](#)

---

[GetInRegs](#)

---

[GetCoils](#)

---

[SetCoils](#)

---

[GetHoldRegs](#)

---

[SetHoldRegs](#)

---

## E.10 Program control

---

[Command list](#)

---

[Print](#)

---

[Wait](#)

---

[Pause](#)

---

[ResetElapsedTime](#)

---

[ElapsedTime](#)

---

[Systime](#)

---

# Preface

## Purpose

This document describes the functions and operations of DobotStudio Pro for controlling CR A and Magician E6 robots, making it easier for users to understand and use DobotStudio Pro.

## Intended audience

This document is intended for:

- Customer
- Sales Engineer
- Installation and Commissioning Engineer
- Technical Support Engineer

## Related documents

Document	NOTE	Download link
Dobot CR A Series Hardware Guide	Introduces the functions, technical specifications, and installation instructions of the Dobot CR A series collaborative robots.	
Dobot TCP_IP Remote Control Interface Guide	Describes how to use the secondary development interface based on TCP/IP protocol. You can also find development DEMOs in various languages on <a href="#">GitHub</a> .	Visit Dobot official website, click "Support > Download Center", and search by document name or filter by product series.
Dobot Bus Communication Protocol Guide (EtherNetIP_Profinet)	Introduces the usage of the robot's bus communication functions (EtherNetIP/Profinet).	
Dobot Cobot Teach Pendant User Guide	Explains how to use the Teach Pendant that accompanies the collaborative robot.	
Dobot Magician E6 User Guide	Introduces the functions, technical specifications, and installation instructions of the Dobot Magician E6 collaborative robot.	

# Revision history

Date	Version	Revised content
2024/12/26	V4.6.0	<ol style="list-style-type: none"><li>Added <a href="#">Debugging and running</a> section for easier debugging.</li><li>Added <a href="#">Trajectory recovery</a> section for more flexible operation.</li><li>Added <a href="#">File migration</a> section for one-click import and export of configuration files.</li><li>Added <a href="#">Firmware upgrade</a> section for one-click firmware version update.</li><li>Other optimizations and updated illustrations.</li><li>Updated to DobotStudio Pro 4.6.0.</li></ol>
2024/03/25	V4.5.1	Update to DobotStudio Pro 4.5.1
2023/11/28	V4.5.0	Update to DobotStudio Pro 4.5.0
2023/10/13	V4.4.1	Update the examples of lua command, i.e., GetInRegs, GetHoldRegs, and SetHoldRegs
2023/08/10	V4.4.0	<ol style="list-style-type: none"><li>Reconstruct the content</li><li>Update to DobotStudio Pro 4.4.0</li></ol>
2023/06/02	V4.1.1	<ol style="list-style-type: none"><li>Update to DobotStudio Pro 4.1.1</li><li>Optimize the content and format</li></ol>
2023/05/16	V4.1.0	First release

## NOTE

The English version of this document is a translation from the Chinese version.

# Symbol conventions

The symbols that may be found in this document are defined as follows.

Symbol	NOTE
 <b>DANGER</b>	Indicates a hazard with a high level of risk which, if not avoided, could result in death or serious injury.
 <b>Warning</b>	Indicates a hazard with a medium level or low level of risk which, if not avoided, could result in minor or moderate injury, robot arm damage.
 <b>NOTICE</b>	Indicates a potentially hazardous situation which, if not avoided, could result in robot arm damage, data loss, or unanticipated result.
 <b>NOTE</b>	Provides additional information to emphasize or supplement important points in the main text.

# 1 Getting Started

## 1.1 Overview

Welcome to DobotStudio Pro. DobotStudio Pro is a multi-functional control software developed by Dobot for its robots. With simple interface, easy-to-use functions and strong practicality, it can help you quickly master the operation of Dobot robots.

This document mainly introduces how to use DobotStudio Pro to control CR A series robots.

DobotStudio Pro is supported on PC, Android tablet, and Dobot's self-developed teach pendant, with the teach pendant version featuring unique functions (e.g., three-position switch) that require specific hardware, which are not covered in this document, please refer to *Dobot Cobot Teach Pendant User Guide*.

## 1.2 Configuration requirements

The configuration requirements for DobotStudio Pro are listed below:

### For PC

Configuration item	Minimum	Recommended
Processor	64-bit Intel or AMD processor, SSE 4.2 or above, 2.9GHz or above	
Operating system	<ul style="list-style-type: none"><li>Windows 10 (64-bit) 1809 or above</li><li>Windows 11</li></ul>	
RAM	8 GB	16 GB or above
Graphics card	<ul style="list-style-type: none"><li>Support DirectX12</li><li>2GB VRAM</li></ul>	<ul style="list-style-type: none"><li>Support DirectX12</li><li>4GB VRAM for 4K and higher resolutions</li></ul>
Display resolution	1440 x 900, 100% scaling	1920 x1080 or higher resolution
Hard drive	4 GB of available space	<ul style="list-style-type: none"><li>4 GB of available space</li><li>Built-in SSD</li></ul>

### NOTE

- Processor/RAM/Graphics card below the minimum requirement may lead to software lag or crashes.
- Display resolution below the minimum requirement may result in incomplete display of the interface.

- Additional hard drive space may be required during software installation.
- Operating system below the minimum requirement may result in software incompatibility and require further evaluation.

#### For App

Customers can request a specific tablet configuration from Dobot. If you need to purchase it by yourself, the recommended specifications are as follows:

Configuration item	Recommended
Processor	4-core
Operating system	Android 10 and above
RAM	2 GB
Storage space	32 GB
Display	8-inch

## 1.3 DobotStudio Pro installation

#### For PC

Download the latest DobotStudio Pro installation package form [Dobot website](#) and follow these steps to install:

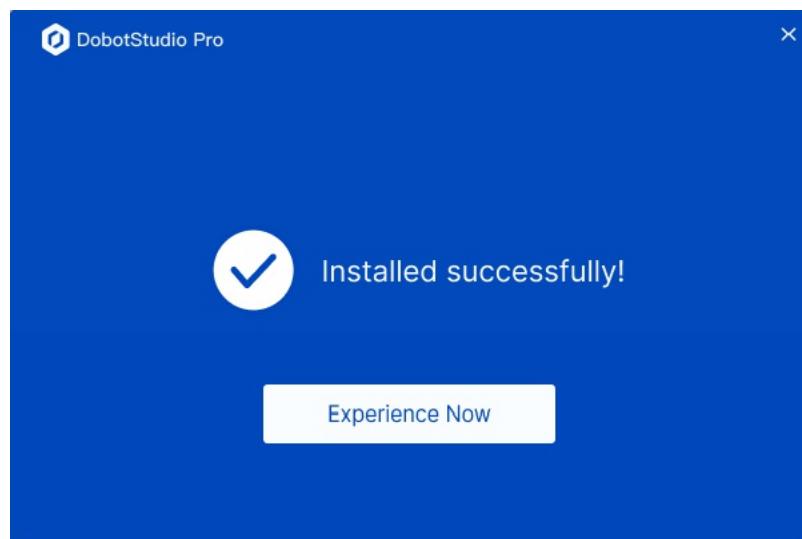
1. Double-click DobotStudio Pro installation package, select a language for installation, and click **Start Installation**.



2. Click **One click installation**, or start installation after setting the installation path in the **Custom** option.



3. After installation, click **Experience Now** to enter DobotStudio Pro.



#### For App

- Android: Download the latest DobotStudio Pro installation package from [Dobot website](#) and install it.

## 1.4 How to use this Guide

Phase	NOTE	Reference section
Connecting to the robot	DobotStudio Pro supports wired and wireless connections to the robot. Most functions require a connection, but you can set the display language or upload App crash logs when not connected.	<a href="#">Connecting to Robot</a>
Understanding the DobotStudio Pro	Get a quick overview of the main interface and functions of DobotStudio Pro.	<a href="#">Interface Overview</a>
Installation/Voltage settings	If the robot is not installed on a level surface, adjust the installation angle first. Additionally, set the voltage range if using a DC power supply.	<ul style="list-style-type: none"> <li>• <a href="#">Installation settings</a></li> <li>• <a href="#">Power voltage</a></li> </ul>
Quickly experience robot functions	Write and run a program for the robot to move in a loop between two points.	<a href="#">Quick Start</a>
Security settings	Before using the robot, set the safety functions based on your risk assessment results.	<ul style="list-style-type: none"> <li>• <a href="#">Safety I/O</a></li> <li>• <a href="#">Safety settings</a></li> </ul>
Understanding basic robot operations	Learn basic operations, including user login, jogging, and alarm handling.	<a href="#">Robot Basic Operations</a>
Programming	You can write programs to control the robot's automatic operations. Select the appropriate programming method first and learn how to use the programming interface.	<a href="#">Application</a>
	Explore specific programming commands, including their functions and usage.	<ul style="list-style-type: none"> <li>• <a href="#">Appendix B Blockly Programming Command</a></li> <li>• <a href="#">Appendix C Script Programming Command</a></li> <li>• <a href="#">Appendix E Python Programming Command</a></li> </ul>
	You can view and adjust I/O status for debugging related functions.	<ul style="list-style-type: none"> <li>• <a href="#">Controller DI/DO</a></li> <li>• <a href="#">Controller AI/AO</a></li> <li>• <a href="#">Tool I/O</a></li> </ul>
Using eco-accessories	The Dobot+ plugin helps you quickly configure and use Dobot eco-accessories, eliminating the need for secondary development. For information on compatible eco-accessories, please visit <a href="#">Dobot website</a> .	<a href="#">Dobot+</a>
	Modify settings related to the software and	

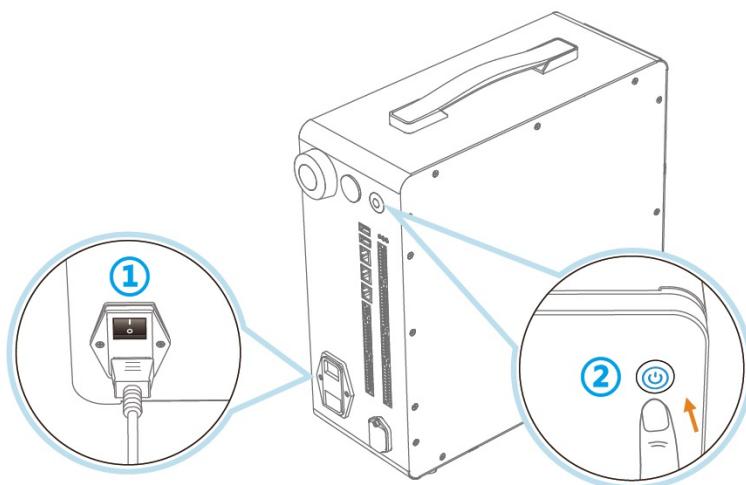
Software and robot settings	the robot. Settings related to the robot are saved in the controller and must be set individually for each robot.	<a href="#">Settings</a>
Remote controlling the robot	You can remotely control the robot through the controller's I/O, including project execution and getting robot status.	<a href="#">Controller DI/DO</a>
	You can also remotely control the robot via Modbus (Modbus-TCP or RTU-over-TCP), including project execution and getting status and real-time feedback.	<ul style="list-style-type: none"> <li>• <a href="#">Modbus</a></li> <li>• <a href="#">Appendix A Modbus Register Definition</a></li> </ul>
Viewing and exporting logs	Logs can help diagnose issues, and you can export them for technical support if needed.	<a href="#">Log</a>
Firmware upgrade	Easily upgrade the robot's firmware to the latest version with one click, or roll back to a previous firmware version.	<a href="#">Firmware upgrade</a>

## 2 Connecting to Robot

### 2.1 Power on

#### CRA series

After completing the installation and wiring of the controller and turning on the external power supply, set the switch above the power interface to "|". Then, short-press the circular button on top of the controller. When the blue indicator is steady on, it means the controller has powered on.

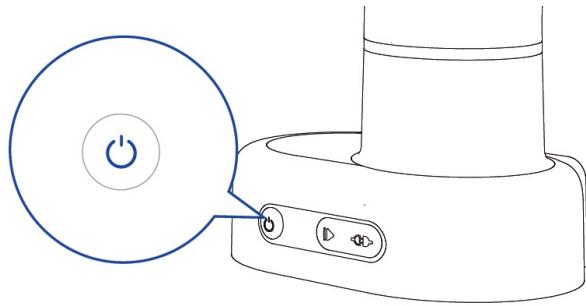


#### *i* NOTE

During the start-up of the controller, the orange light of the LAN2 interface flashes, indicating that the built-in WiFi router is initializing. At this time the LAN, WiFi and Teach Pendant are unable to connect to the robot. As the initialization progresses, the flashing speed of the light will increase until it turns off, indicating that the initialization is completed, and you can try connecting to the robot.

#### Magician E6

After completing the installation and wiring of the robot and turning on the external power supply, short-press the circular button on the base. When the blue indicator is steady on, it means the robot has powered on.

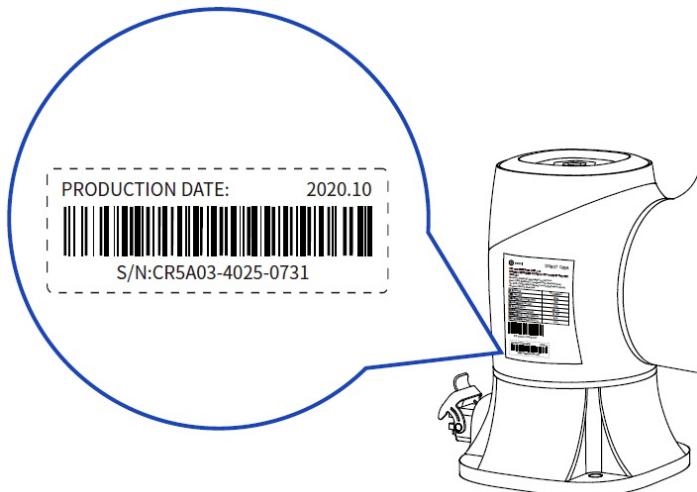


## 2.2 Wireless connection

### NOTE

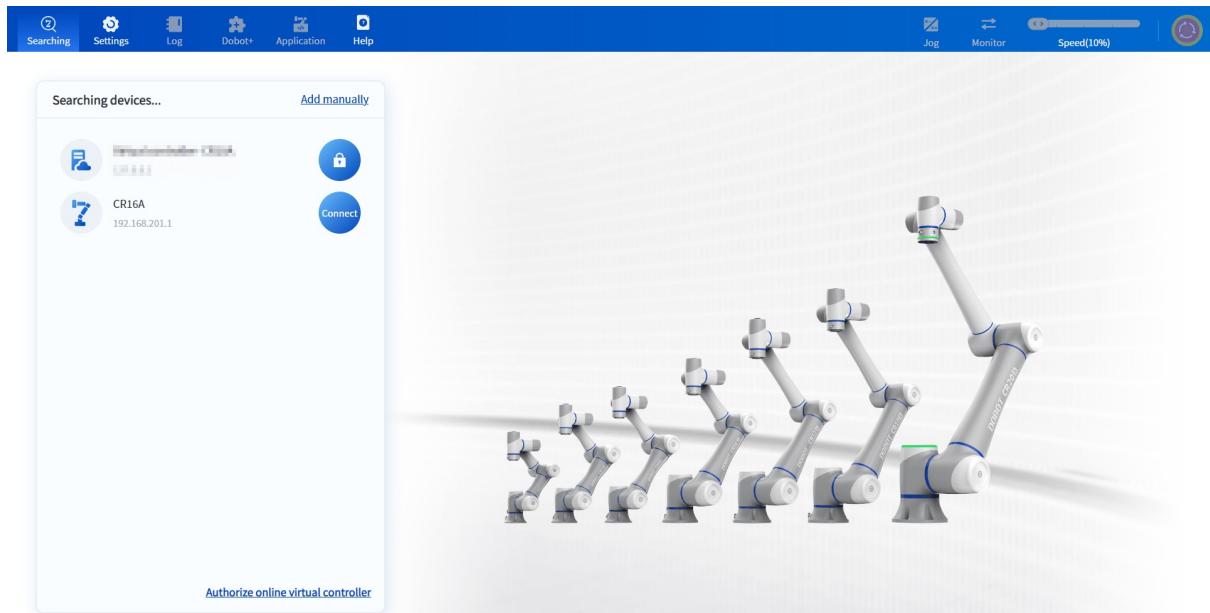
To connect to the Magician E6 robot wirelessly, you need to purchase a wireless module separately and plug it into the robot base's USB interface.

On your PC or tablet, search for and connect to the robot's Wi-Fi. The default SSID format is "DobotProductModel-SerialNumber", where the serial number consists of two sets of four-digit numbers connected by "-" (the S/N code is located on the nameplate at the robot base; for example, the CR5A model would have a default SSID of "DobotCR5A-4025-0731", and similar formats apply to other models). The initial Wi-Fi password is: 1234567890. You can modify the WIFI SSID and password in [Communication settings](#).



For WiFi connection, the robot IP is 192.168.201.1.

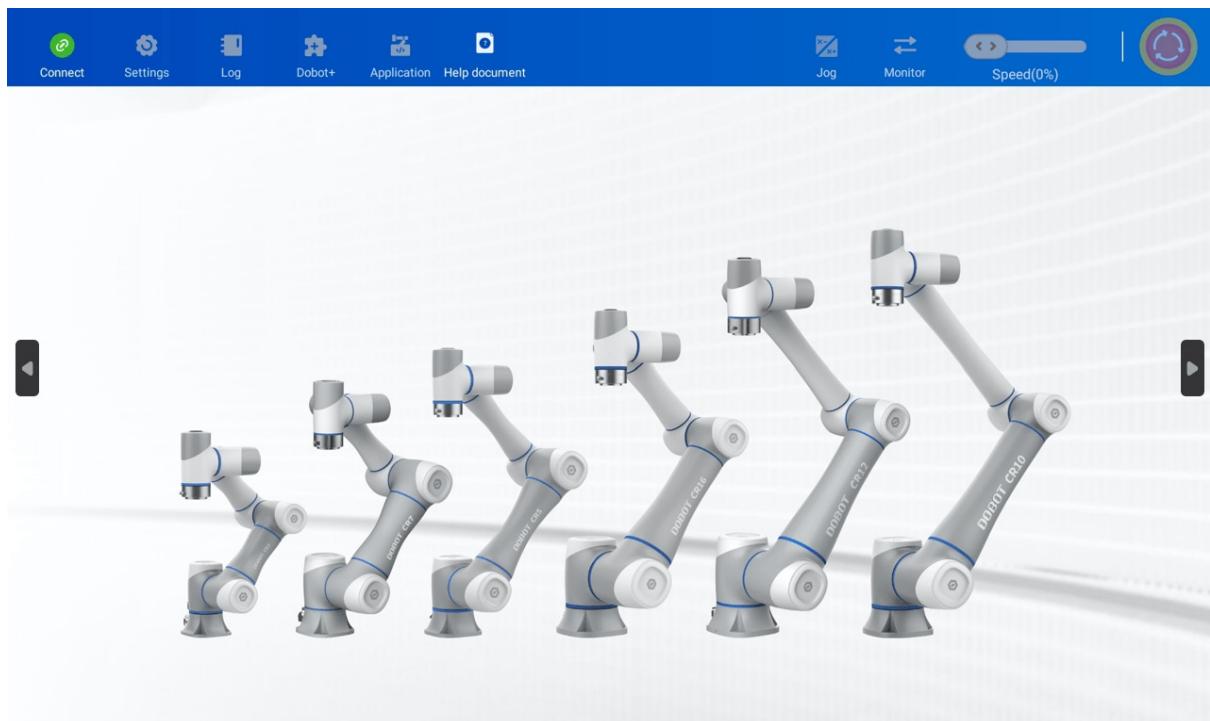
**For PC**



After you enter DobotStudio Pro, the software will automatically search for robots and display the results on the interface. Due to the internal network architecture of the controller, DobotStudio Pro may detect multiple IP addresses for the same robot. You can connect to the robot using any of these IPs, but it is recommended to use 192.168.201.1 for the connection.

Click next to the robot to connect it.

## For App



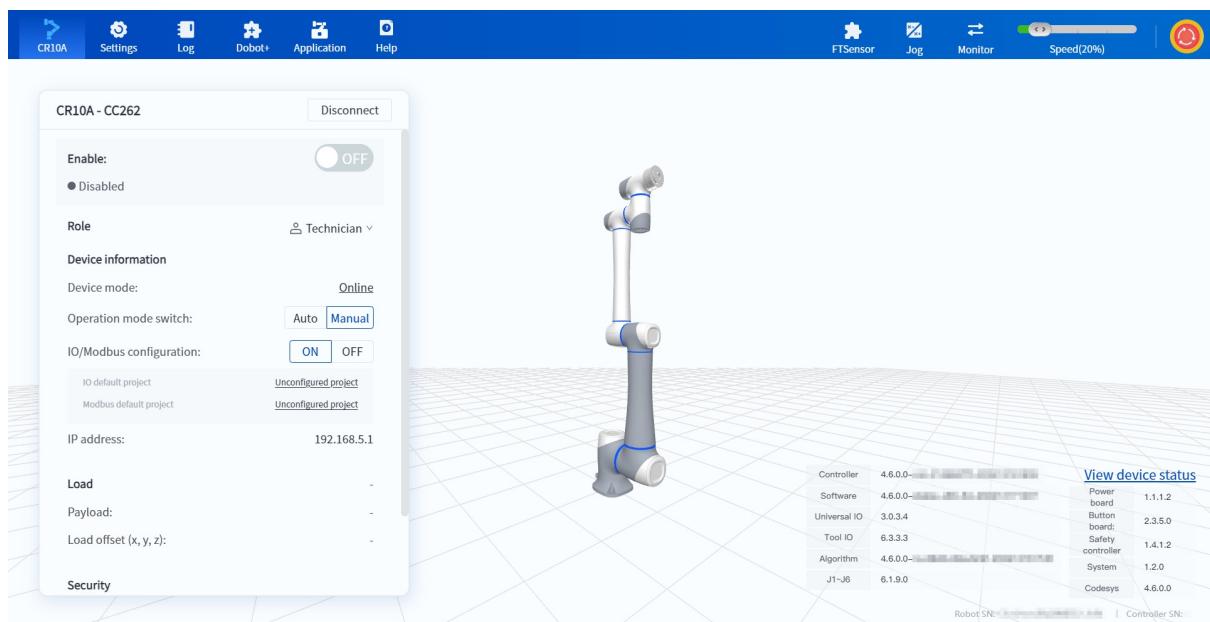
Click at the top left of the interface to connect to the robot.

### **NOTE**

When connecting the software to the robot, it will check whether the software version matches the controller version:

- If the first two digits of the version No. match, the robot will connect successfully.
- If the first two digits of the version No. do not match, the software pops up a window which prompts that the version does not match and automatically disconnects.

Once the robot is successfully connected, the software interface will refresh to display the connected robot's information and 3D model. You can click the **Disconnect** button next to the robot's name to disconnect from the current robot.



## 2.3 Wired connection

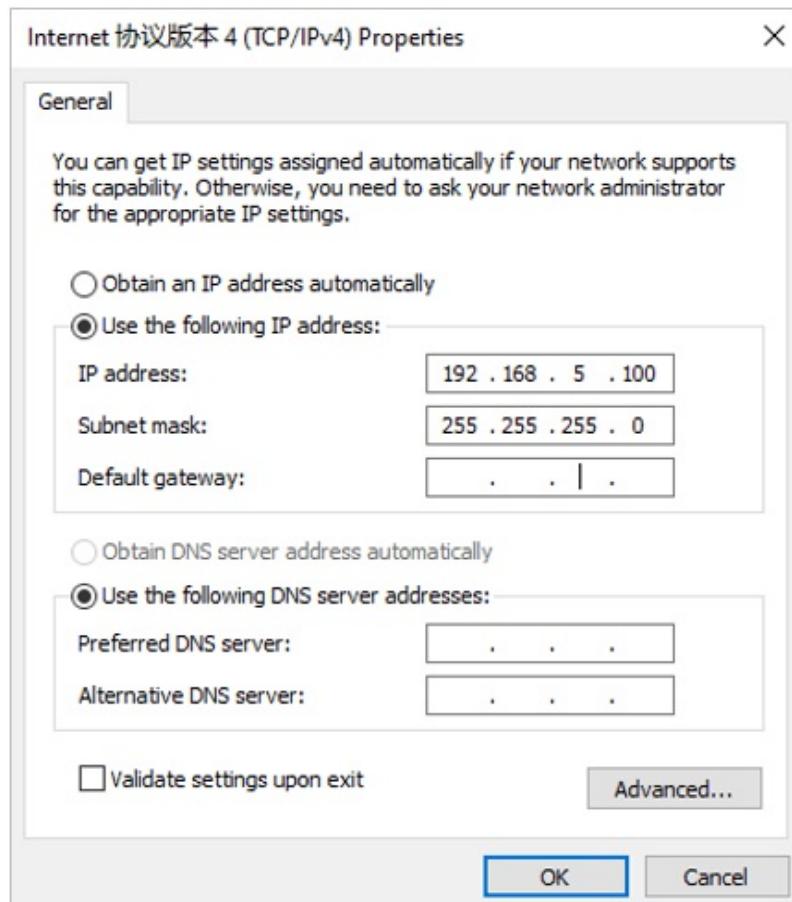
**Wired connections are only supported on PC.**

Connect one end of the network cable to the LAN interface on the controller and the other end to your PC. Then, modify the PC's IP address to ensure it is in the same subnet as the controller. The default IP for the LAN1 interface is 192.168.5.1, while for the LAN2 interface it is 192.168.200.1. The IP address for the LAN1 interface can be changed in the [Communication settings](#), but the LAN2 interface's IP address is fixed.

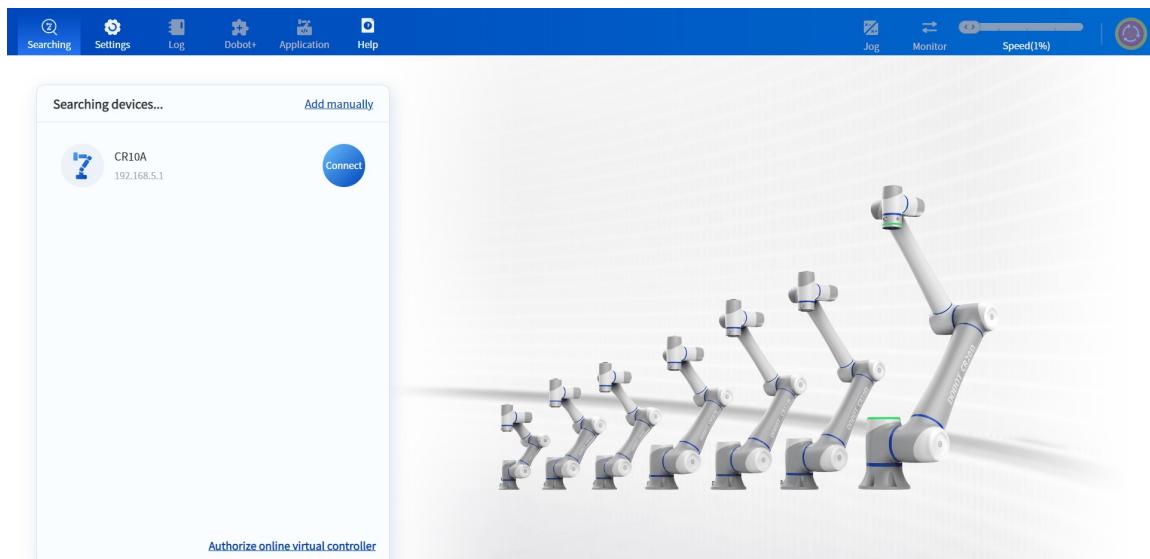
Different Windows versions vary in modifying the IP address. This section takes Windows 10 as an example to introduce specific operations.

1. Search for **View network connections** in the taskbar, and click **Open**.

2. Right-click the icon for your current network connection and select **Properties**. Then, find and double-click **Internet Protocol Version 4 (TCP/IPv4)** in the pop-up window.
3. In the **Internet Protocol Version 4 (TCP/IPv4) Properties** page, select **Use the following IP address**, and modify the PC's IP address, subnet mask, and default gateway. You can set the PC's IP address to any available address in the same subnet as the controller, ensuring the subnet mask and default gateway match those of the controller. For example, set the PC's IP to 192.168.5.100, and the subnet mask to 255.255.255.0.



4. Click  next to the robot to connect it.

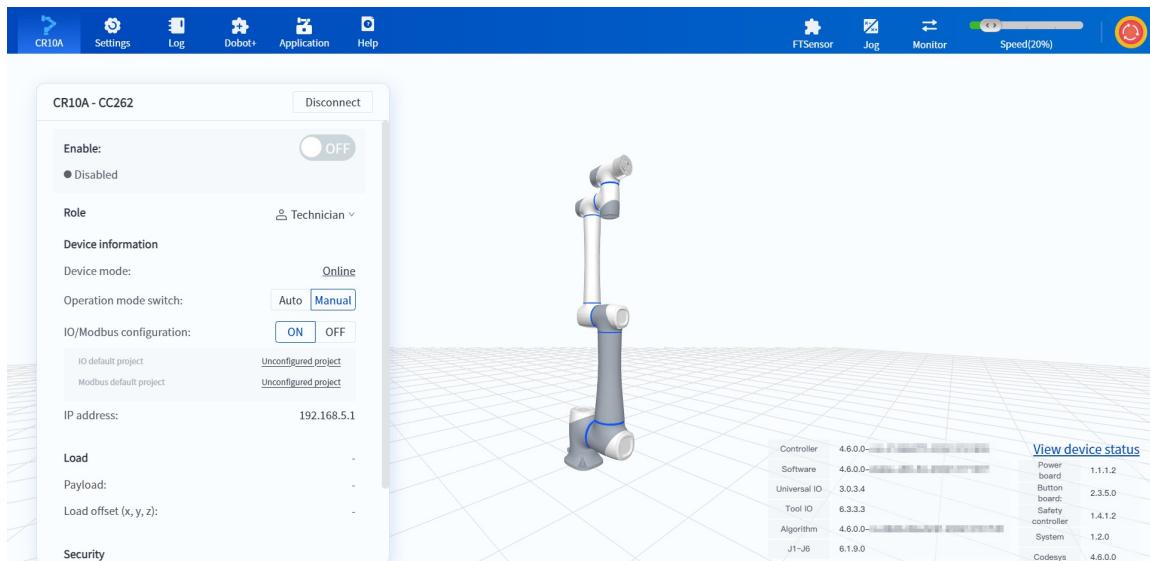


### *NOTE*

When connecting the software to the robot, it will check whether the software version matches the controller version:

- If the first two digits of the version No. match, the robot will connect successfully.
- If the first two digits of the version No. do not match, the software pops up a window which prompts that the version does not match and automatically disconnects.

- Once the robot is successfully connected, the software interface will refresh to display the connected robot's information and 3D model. You can click the **Disconnect** button next to the robot's name to disconnect from the current robot.



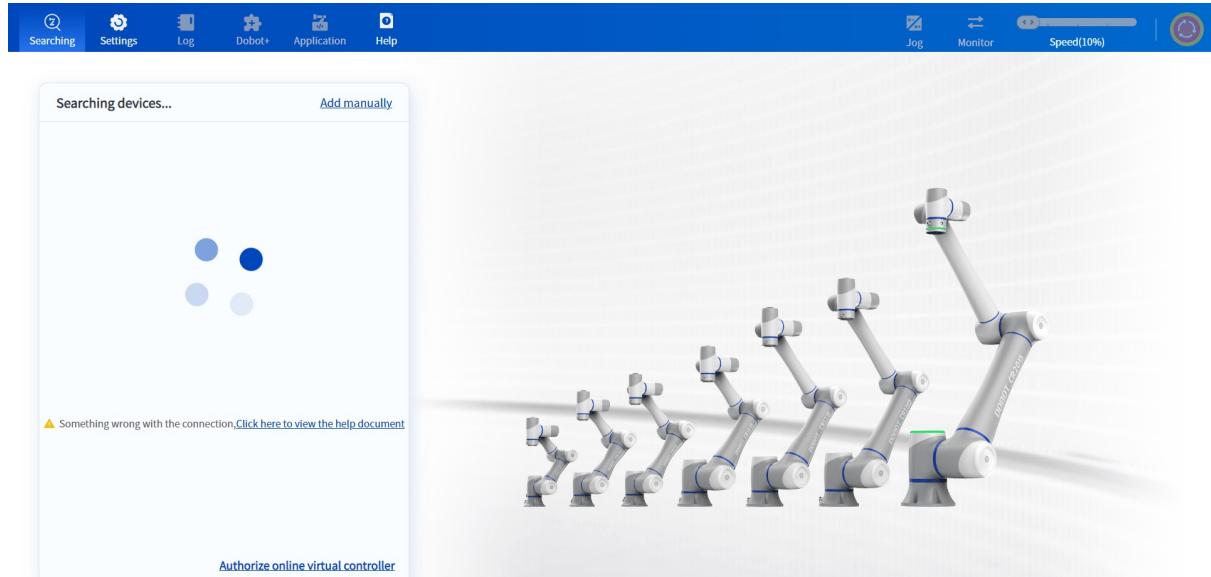
### *NOTE*

If you want to connect multiple robots simultaneously using one PC, make sure both the robots and the PC are on the same local network. You can then open multiple instances of DobotStudio

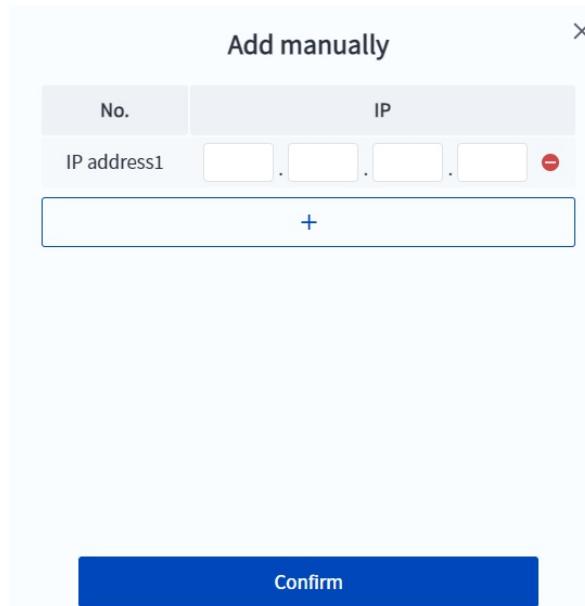
Pro on your PC and connect to different robots from each window.

## 2.4 Manual adding

If the robot cannot be found through search, you can manually add the robot's IP address.



After opening DobotStudio Pro, click **Add manually**. On the pop-up page, click + to add robot IP.



You can manually add up to 5 robot IPs at a time. After entering the IP addresses, click **Confirm** to prioritize searching for the manually added robots.

## 3 Interface Overview

- [3.1 Main interface](#)
- [3.2 Settings interface](#)
- [3.3 Log interface](#)
- [3.4 Dobot+ interface](#)
- [3.5 Application interface](#)
- [3.6 Jog panel](#)
- [3.7 Monitor panel](#)

## 3.1 Main interface

After the software is connected to the robot, the main interface appears as shown below.



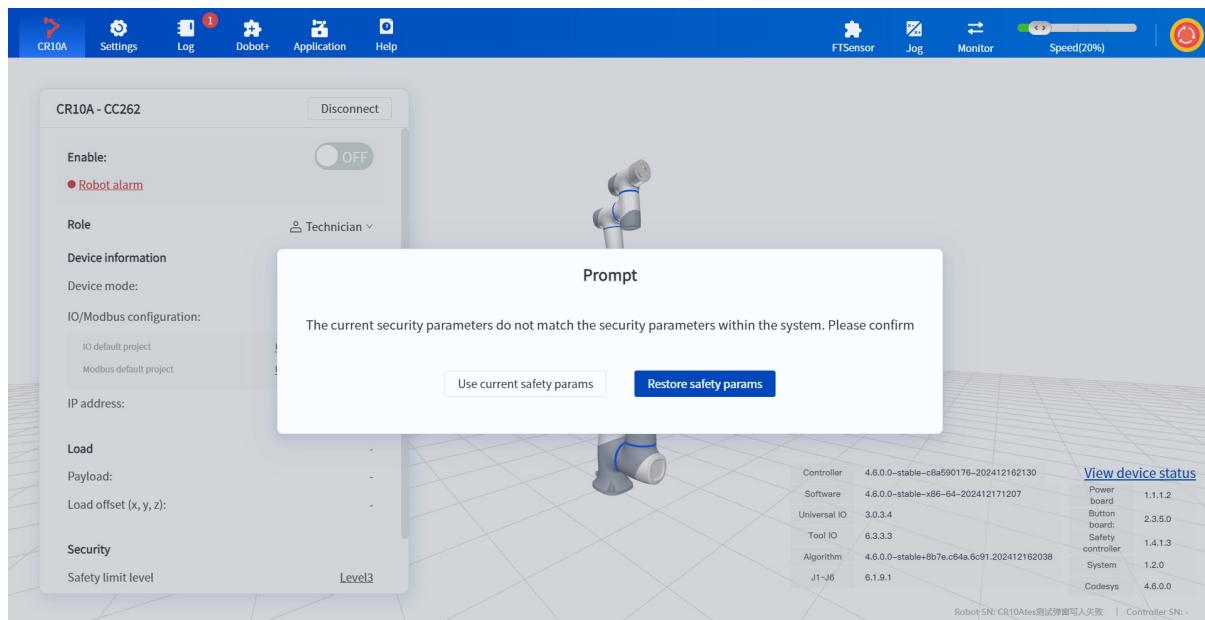
No.	NOTE
1	Click to open the main interface (current interface). The color of the robot icon synchronizes with the indicator on the robot and changes according to the robot's status. For detailed definition of the indicator colors, see the corresponding hardware guide.
2	Click to open <a href="#">Settings interface</a> to set the parameters related to robot installation, motion, safety, etc.
3	Click to open <a href="#">Log interface</a> to view alarms and logs.
4	Click to open <a href="#">Dobot+ interface</a> to install use Dobot+ plugins.
5	Click to open <a href="#">Application interface</a> for programming via Blockly or Script.
6	Click to open the help document, which can be viewed in the software interface, in a separate window, or in a web browser.
7	Click to open <a href="#">Jog panel</a> to jog or step the robot.
8	Click to open <a href="#">Monitor panel</a> to monitor the robot's I/O and global variables.
9	Used to display and set the global speed, which controls the robot speed.
10	E-Stop button. Press this button in case of an emergency, and robot will stop immediately.
11	This panel is used to enable the robot, switch user roles, set device information, display load information, and show safety information (such as safety checks and data).

12	3D model of the robot, synchronized with the real robot's posture.
13	<p>Robot S/N, software and firmware version information. When contacting technical support for troubleshooting, please send a screenshot of this information for easier troubleshooting. To upgrade the firmware version, please first use the robot maintenance tool; alternatively, you can upgrade through the <a href="#">Firmware upgrade</a> option.</p>
14	<p>Click to view the robot's current status, including voltage, current, and temperature.</p>  <p><a href="#">View device status</a></p>

## Safety checksum

The **Safety checksum** is the result of applying a specific verification algorithm to the safety parameters. If the safety parameters change, the safety checksum will also change.

DobotStudio Pro will display an error message, and the user must resolve the mismatch in the safety checksum before continuing to use the robot.



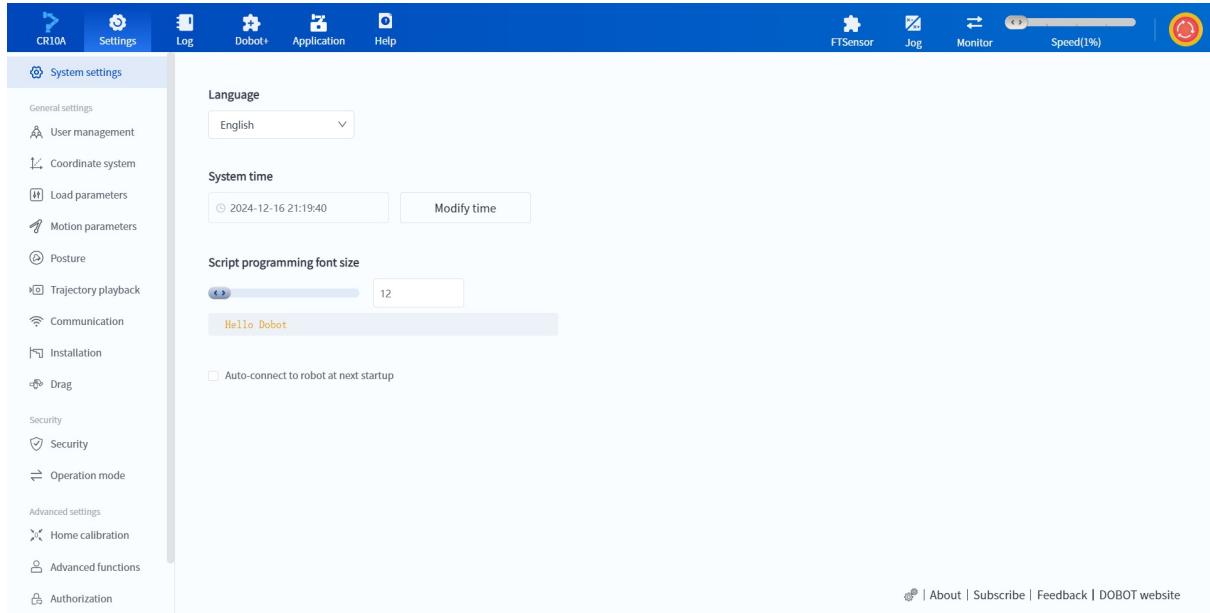
- Clicking "Use current safety parameters" will update the safety checksum based on the current system safety parameters. After a successful update, a message will indicate that the current safety parameters have been successfully set.
- Clicking "Restore safety parameters" will revert to the last correctly configured safety parameters, keeping the safety checksum unchanged. After a successful restoration, a message will indicate that the default safety parameters have been successfully restored.

Safety parameters affecting the safety checksum are as follows:

DobotStudio Pro	Parameter
Safety I/O	All parameters in the safety I/O interface
Advanced functions	Torque constraint
Safe home position	Joint coordinates
Safety zone	All parameters in the safety zone interface
Safety wall	All parameters in the safety wall interface
Collision detection (Magician E6)	All parameters in the collision detection interface
Installation settings	Inclination angle, rotation angle
Motion parameters	Playback speed
Operation mode settings	Manual mode, automatic mode
Safety limits	All parameters in the safety limits interface

## 3.2 Settings interface

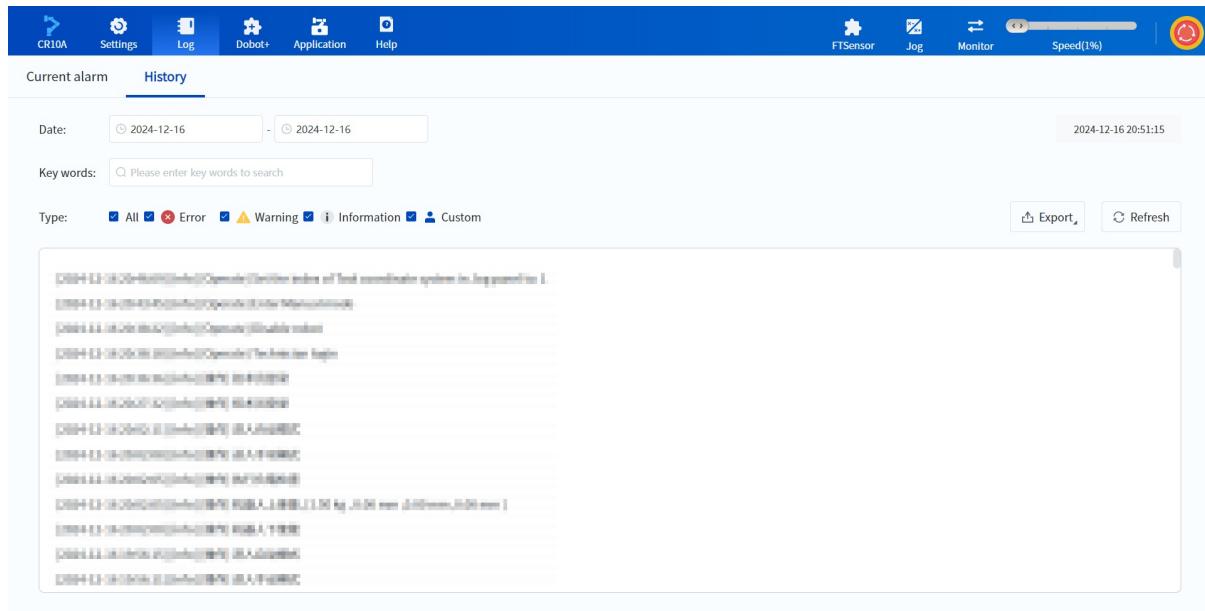
Settings interface allows you to modify software and robot-related settings, such as language, user/tool coordinate system, security settings, etc. See the instructions for each [Settings interface](#) for details.



### 3.3 Log interface

Log interface includes two sub-pages: **Current alarm** and **History**.

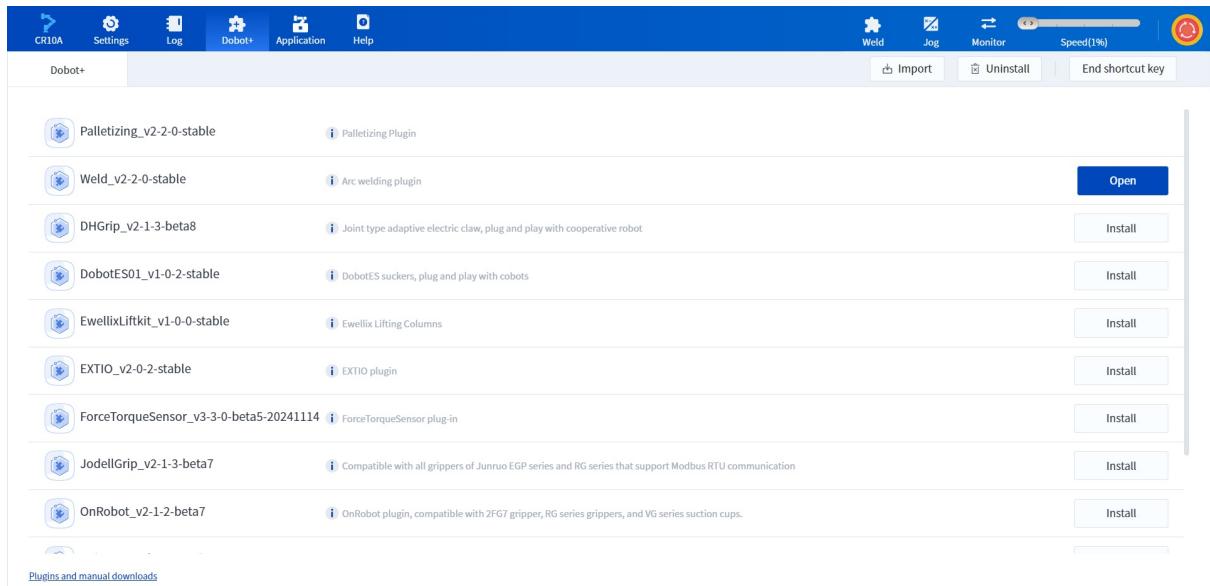
- **Current alarm:** Used to view and clear alarms. See [Alarm](#) for details.
  - **History:** Used to view and export logs. See [Log](#) for details.



## 3.4 Dobot+ interface

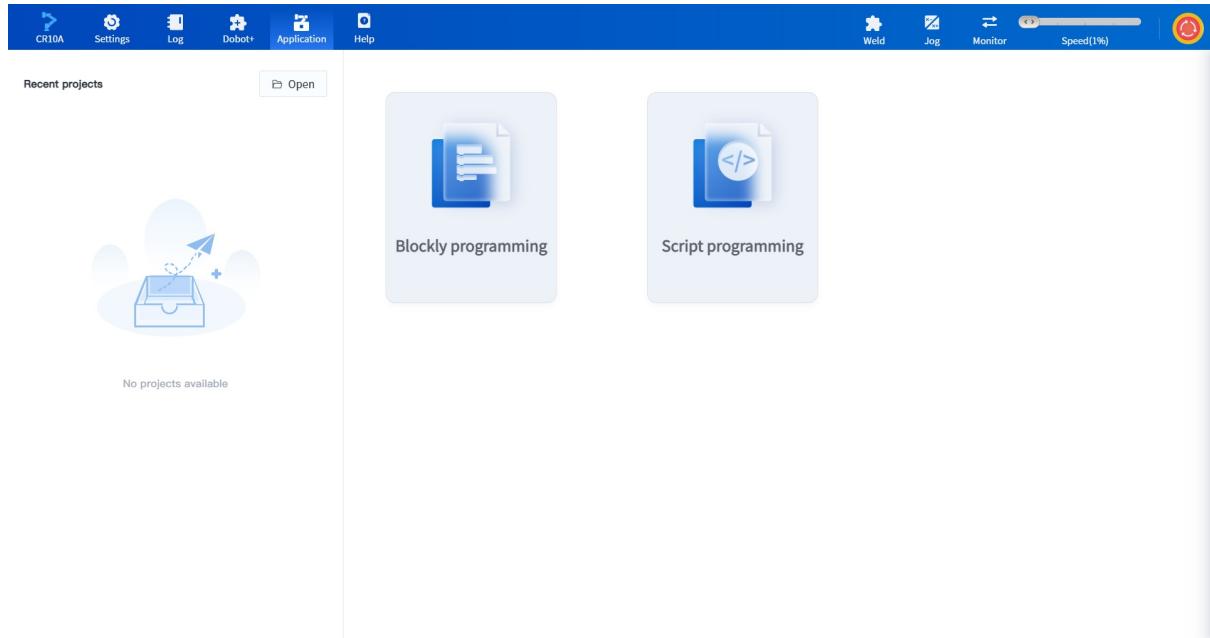
Dobot+ interface is used to manage and use Dobot+ plugins. See [Dobot+](#) for details.

Dobot+ plugins are specialized tools developed by Dobot for use with eco-accessories. These plugins allow you to configure and use the accessories, such as grippers, without the need for additional development.



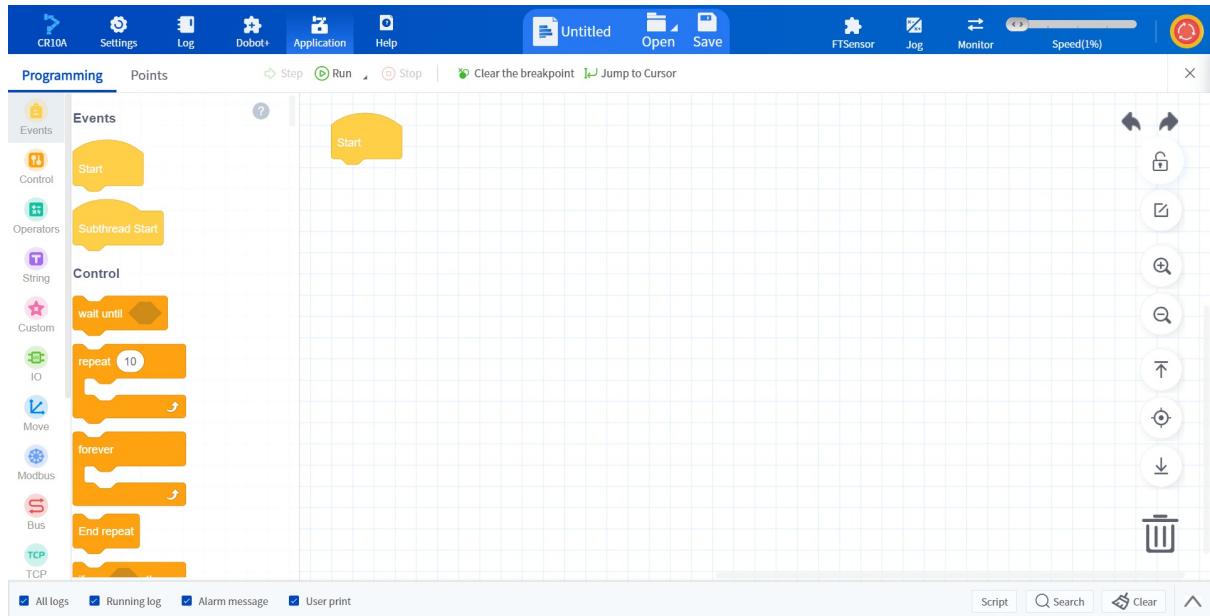
## 3.5 Application interface

Application interface allows you to create and run your projects to realize automated robot operations.



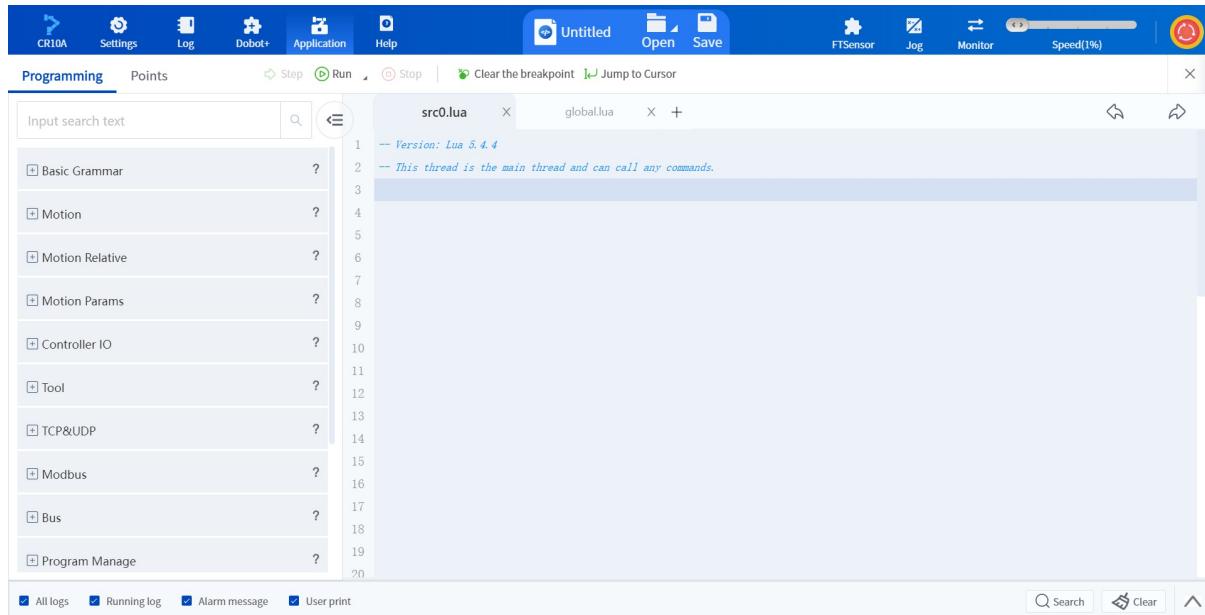
### Blockly programming

Blockly programming is a graphic programming method. You can program by dragging blocks from the left side to the programming area on the right. See [Blockly programming](#) for details.



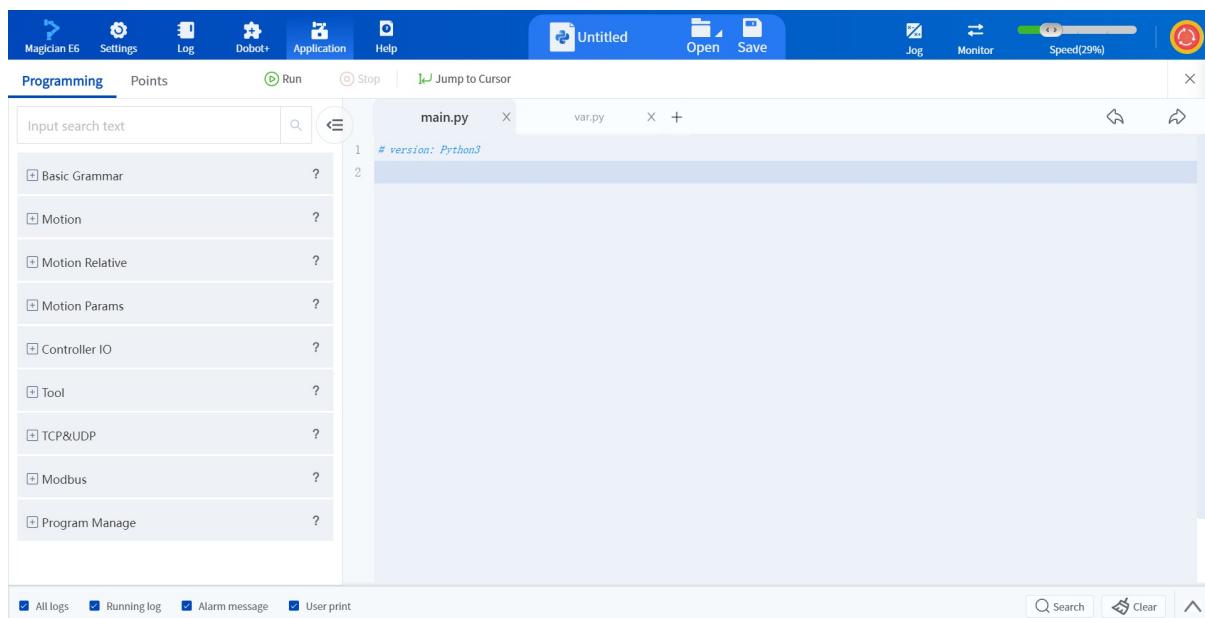
## Script programming

Script programming is a programming method based on Lua language. You can find the commands on the left side of the Programming page, configure the parameters and add them to the programming area on the right, or write the code in the programming area. See [Script programming](#) for details.



## Python programming

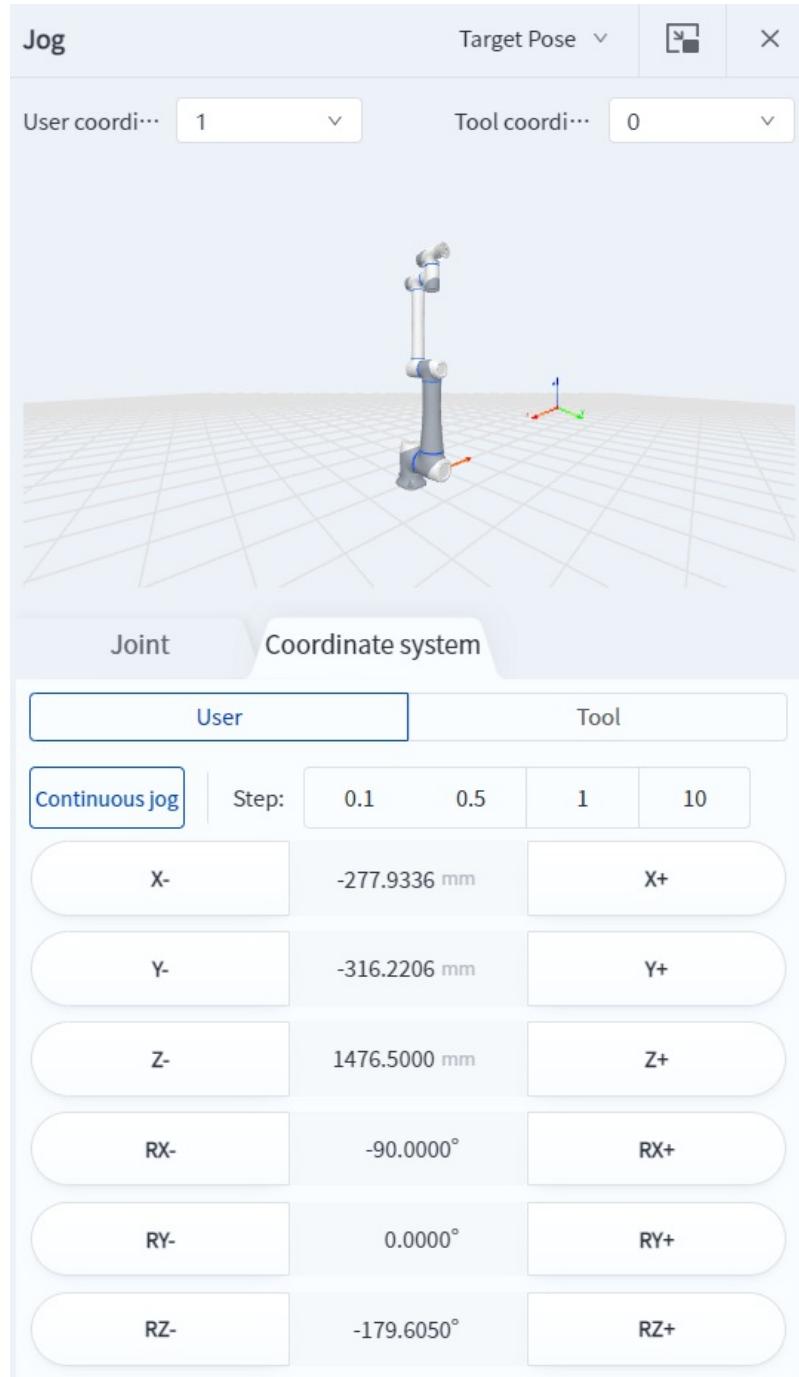
Python programming is a programming method based on Python language. **It is only available when the PC is connected to Magician E6 robot** for research and education purposes. The operation interface is similar to Script programming but does not support debugging. See [Python programming](#) for details.



## 3.6 Jog panel

Jog panel is used to control the robot to jog or step, supporting **Joint Jog** and **Coordinate Jog**. See [Jog](#) for details.

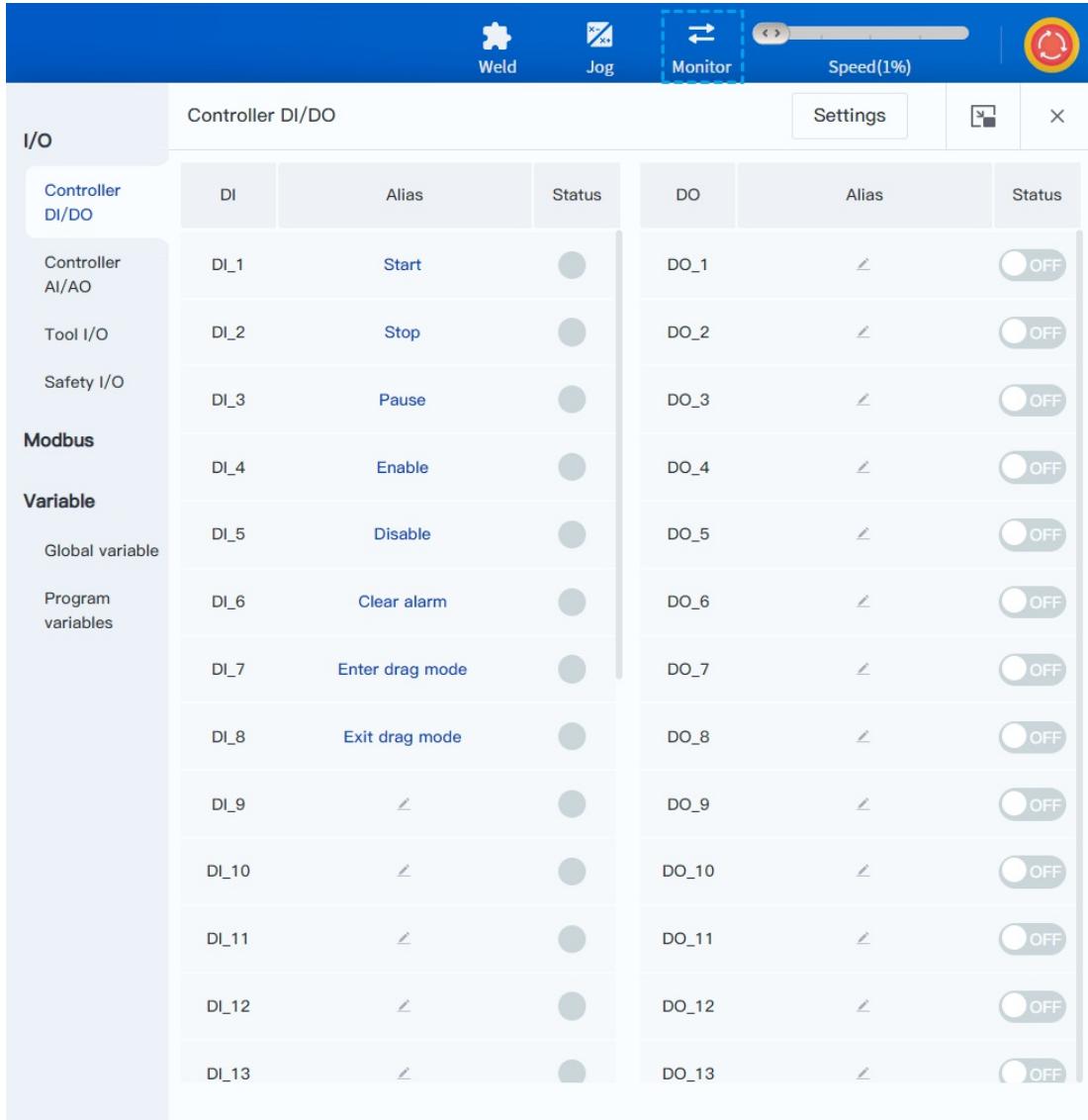
Clicking  on the upper right of the panel will turn it into a movable standalone window. Closing the standalone window and reopening the Jog page will return the panel to its embedded form.



## 3.7 Monitor panel

The **Monitor** panel allows you to monitor and configure the status and functions of the robot's IOs, as well as manage and view global variables. See [Monitor](#) for details.

Clicking  on the upper right of the panel will turn it into a movable standalone window. Closing the standalone window and reopening the Monitor page will return the panel to its embedded form.



The screenshot shows the Monitor panel with the following interface elements:

- Top Bar:** Includes icons for Weld, Jog, Monitor (highlighted with a blue dashed border), Speed(1%), and a refresh symbol.
- Header:** "Controller DI/DO" tab, "Settings" button, minimize/maximize icon, and close "X" button.
- Left Sidebar:** "I/O" tab, "Controller DI/DO" section, and a list of categories: Controller AI/AO, Tool I/O, Safety I/O, Modbus, Variable, Global variable, and Program variables.
- Table:** A grid of 13 rows for DI and DO inputs. Each row contains:
  - Controller DI/DO:** Sub-section header.
  - DI:** Input number (e.g., DI\_1 to DI\_13).
  - Alias:** Function name (e.g., Start, Stop, Pause, Enable, Disable, Clear alarm, Enter drag mode, Exit drag mode).
  - Status:** Indicator light (grey circle).
  - DO:** Output number (e.g., DO\_1 to DO\_13).
  - Alias:** Function name (e.g., OFF).
  - Status:** Indicator light (grey circle).

Controller DI/DO							
I/O	Controller DI/DO	DI	Alias	Status	DO	Alias	Status
Controller AI/AO	DI_1	Start		DO_1			
Tool I/O	DI_2	Stop		DO_2			
Safety I/O	DI_3	Pause		DO_3			
Modbus	DI_4	Enable		DO_4			
Variable	DI_5	Disable		DO_5			
Global variable	DI_6	Clear alarm		DO_6			
Program variables	DI_7	Enter drag mode		DO_7			
	DI_8	Exit drag mode		DO_8			
	DI_9			DO_9			
	DI_10			DO_10			
	DI_11			DO_11			
	DI_12			DO_12			
	DI_13			DO_13			

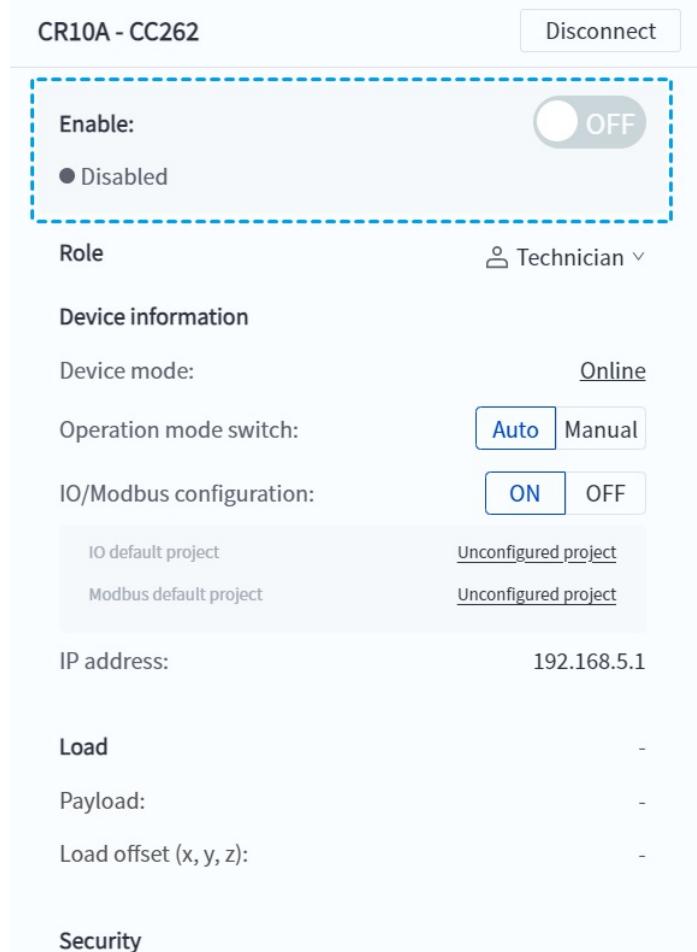
# 4 Quick Start

This chapter introduces how to create a Blockly project that allows a robot to move in a loop between two points, so as to help you quickly experience the functions of Dobot robots.

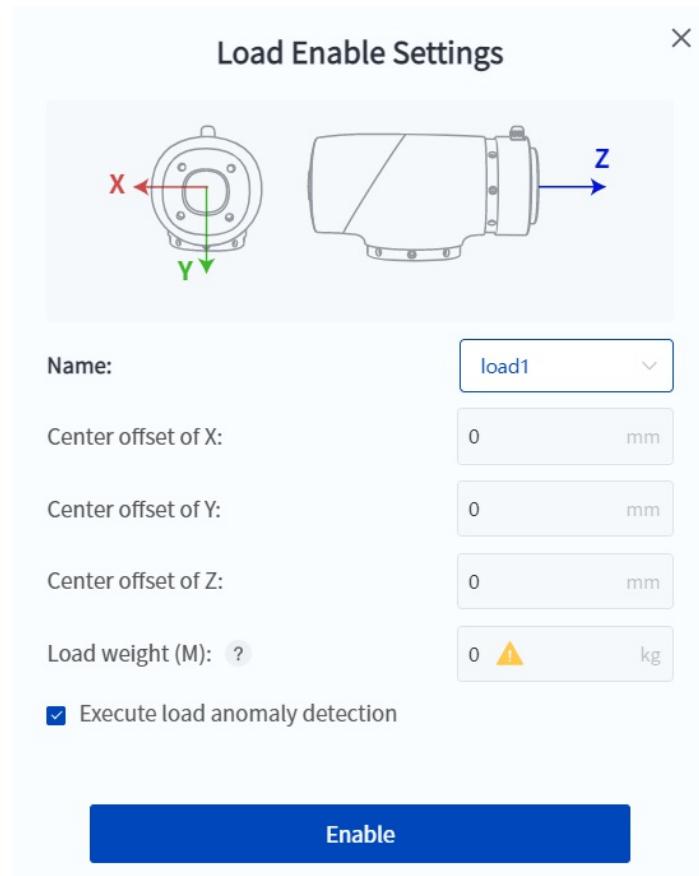
This chapter only covers the simplest steps to complete the task. For detailed explanations of each function, please refer to the subsequent chapters of this document.

## Enabling the robot

1. After connecting to the robot, click **Enable** on the information panel of the main interface.



2. The software will pop up a **Load Enable Settings** window. If no tool is installed at the end of the robot, directly click **Enable**. If a tool is installed, refer to the [Enable](#) page to set the load parameters.

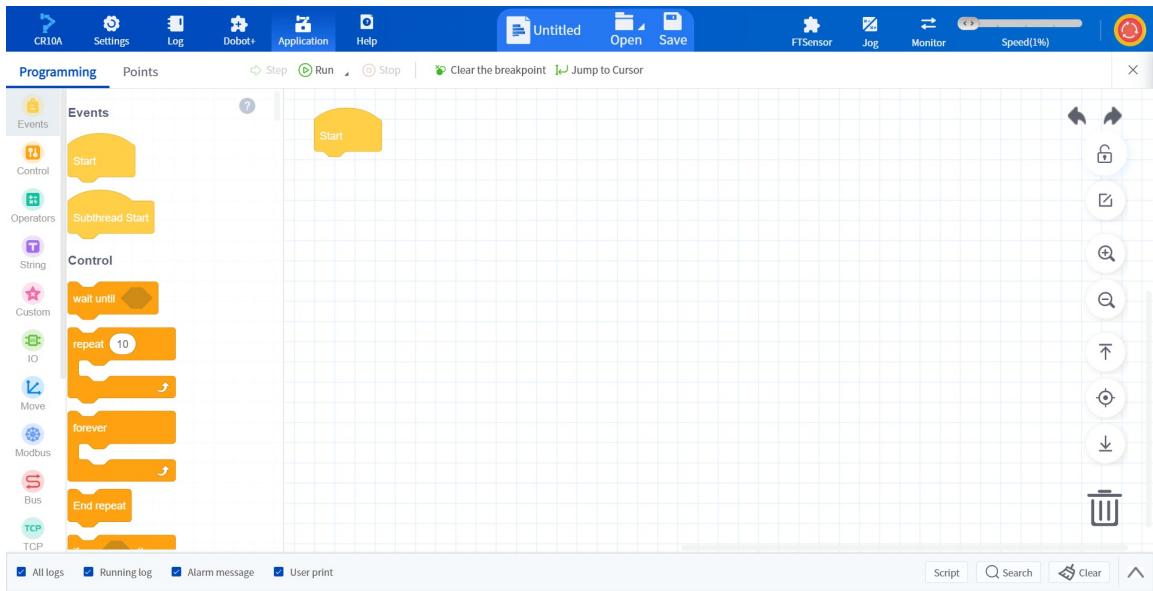


3. Once enabled successfully, the Enable button will change to **ON**.

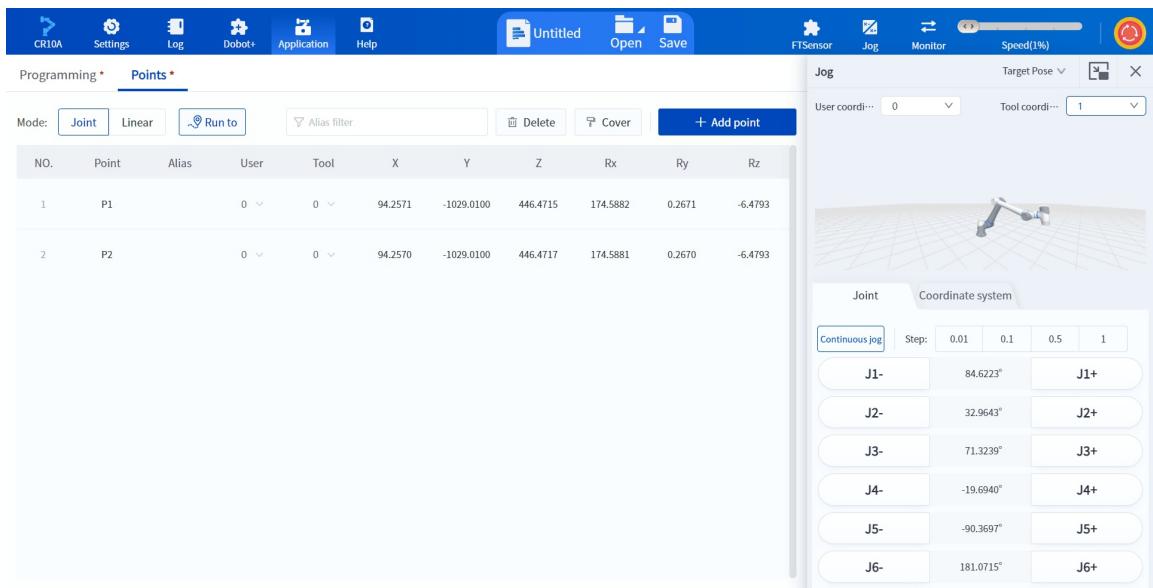


## Creating blockly project

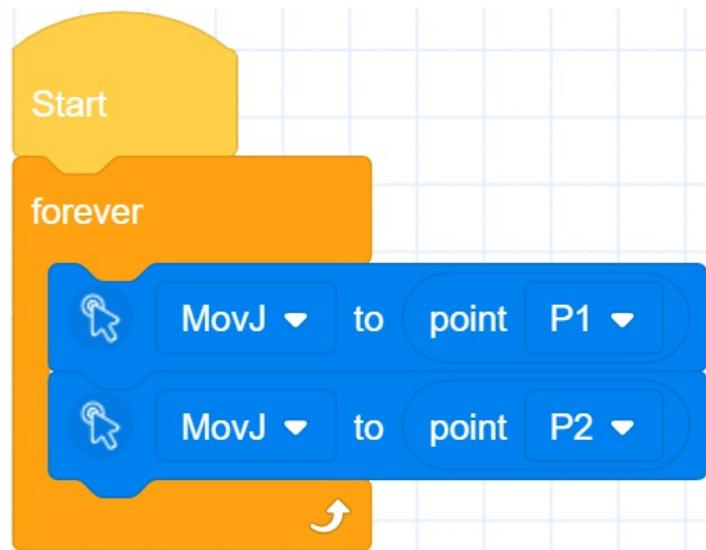
1. Open the **Application > Blockly programming** interface.



2. Open the **Points** page.



3. Click the control buttons on the **Jog** panel (e.g., **J1+** or **J1-**) to move the robot to the desired pose, then click **+ Add point** at the upper right of the **Points** page to save P1.
4. Save P2 in the same way.
5. Return to the **Programming** page. Drag the **forever** block from the **Control** block module on the left into the canvas and place it under the **Start** block.
6. Drag a **Move to Point** block from the **Motion** block module into the **forever** block, and select **P1** from the drop-down list.
7. Drag another **Move to Point** block under the previous one and select **P2** from the drop-down list. The program is shown below.



## Saving and Running

### ⚠️NOTICE

Before running the project, make sure that there are no people or obstacles within the working space of the robot.

Click **Save** at the top of the programming page, enter a project name (e.g., "test"), and save the current project. Then click **Run**. The robot will first move to P1 and then loop between P1 and P2.

# 5 Robot Basic Operations

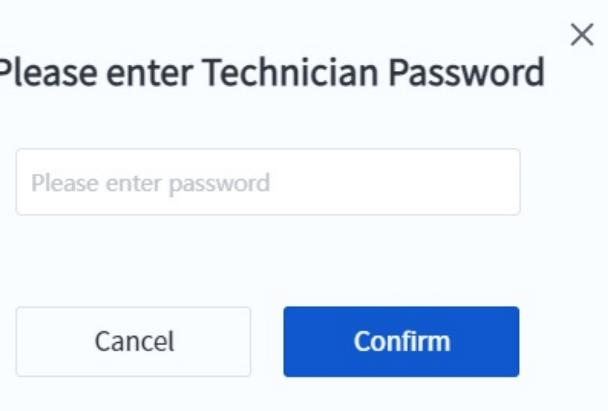
- [5.1 User login](#)
- [5.2 Enable](#)
- [5.3 Recovery mode](#)
- [5.4 Remote control](#)
- [5.5 Manual/Automatic mode](#)
- [5.6 Jog](#)
- [5.7 Drag](#)
- [5.8 Emergency stop and recovery](#)
- [5.9 Speed adjustment](#)
- [5.10 Load settings](#)
- [5.11 Collision detection settings](#)
- [5.12 Alarm](#)

## 5.1 User login

You can assign different roles to robot operators to manage permissions.

When DobotStudio Pro connects to the robot, it automatically logs in with the default role (which can be modified). If the default role does not require a password, the user will not notice the login process.

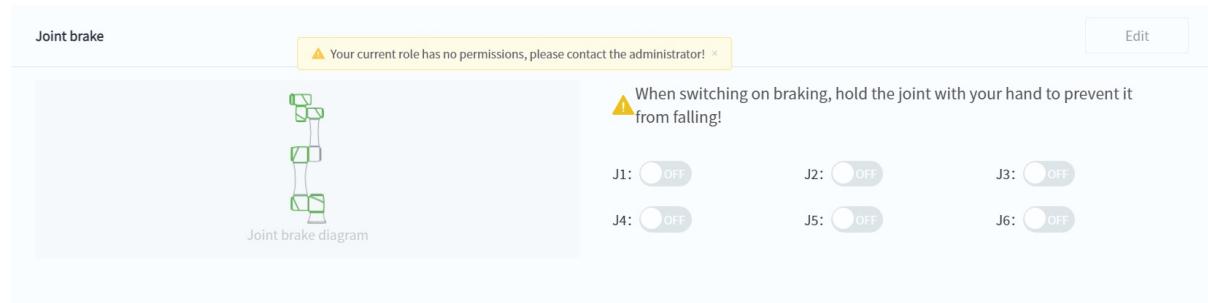
However, if a password is required, a login window will appear, prompting the user to enter the password. If you want to log in with a different role, you can switch roles in this window.



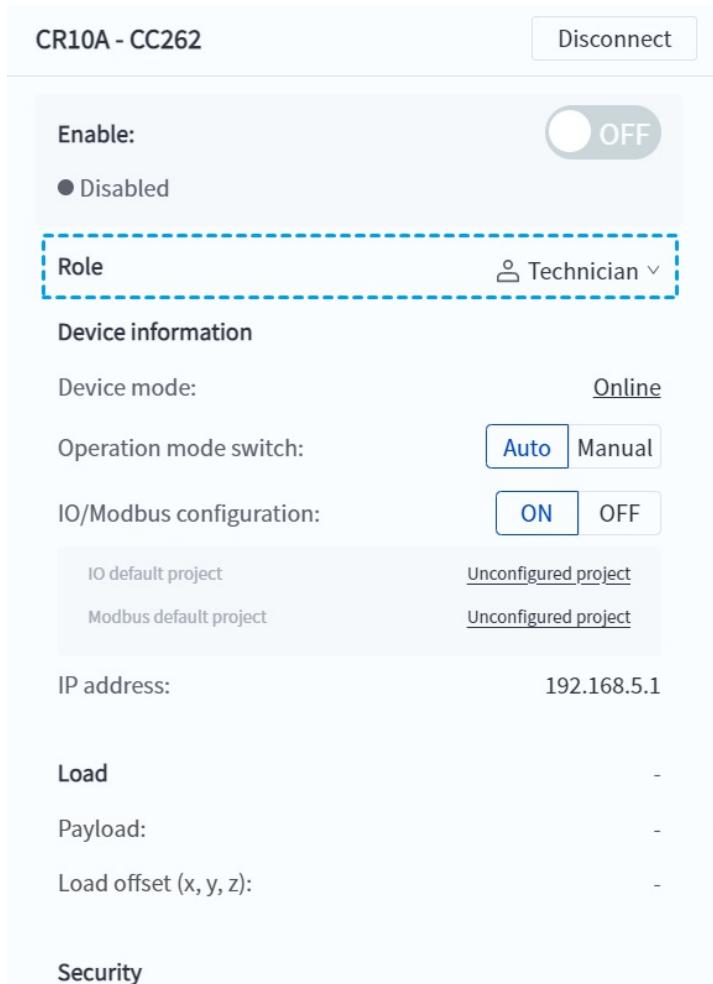
DobotStudio Pro supports four roles, each with different permissions (modifiable).

- **Administrator:** Has full access to all functions by default. Default password: 888888.
- **Technician:** The default role when the robot leaves the factory. Has access to all functions except advanced settings (related to installation and safety) by default. No password by default.
- **Operator:** Has access to basic robot operations like jog and project execution by default. No password by default.
- **Custom:** Roles with customizable names and permissions created by the administrator. If no custom roles are created, this option is not shown.

UI elements for functions that are not accessible to the currently logged-in role (such as **Edit** button) will be grayed out, and clicking them will display a message indicating insufficient permissions.



To switch to a role with higher permissions, go to the main interface and change roles via the information panel. If switching to a role with a password, you'll be prompted to enter the password. If the role has no password, the switch happens directly.



When logged in as an administrator, you can set the default login role, manage custom roles, and view or modify permissions and passwords for each role. For more details, refer to [User management](#).

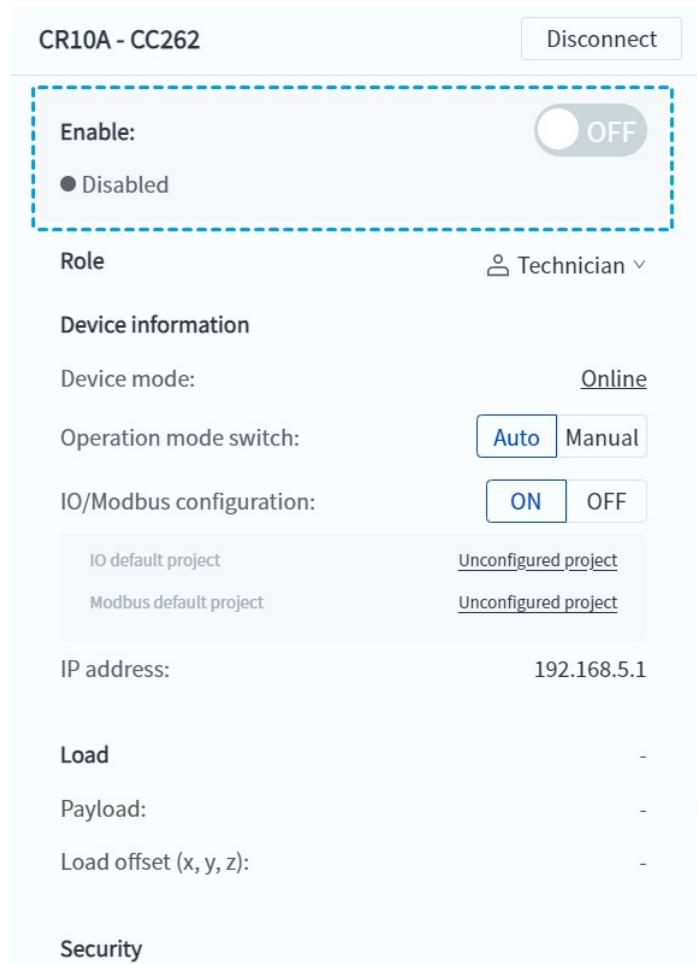
## 5.2 Enable

When the robot is powered on, it is in a disabled status by default. In this status, you can configure and program the robot, but you cannot control its motion or run projects. To control the robot to move or run projects, you must enable the robot first.

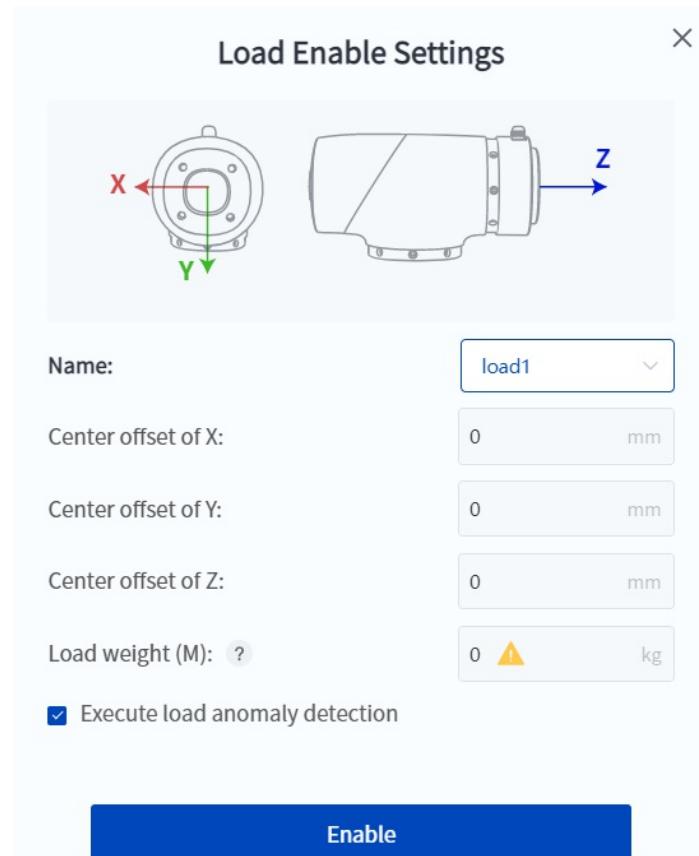
You can switch the enabling status through the Enable button on the software's main interface or the Enable button on the robot itself.

### Enable through software

The software **Enable** button is located on the main interface's information panel.



When the Enable button is **OFF**, clicking it will bring up the load settings window.



Select the load parameter group from the dropdown menu. You need to set the parameter groups in advanced on the [Load parameters](#) page and then select here. The following parameters can be manually modified by selecting **Custom**.

- **X/Y/Z-axis center offset:** The offset distance of the center of mass of load in each direction. For the direction of each axis, refer to the diagram on the interface.
- **Payload:** Total weight of the end effector and workpiece, which must not exceed the the robot's maximum load capacity.

**Execute load anomaly detection:** If this option is selected, the robot will move slightly to check for any significant discrepancy between the load parameters you set and the actual load. If the difference is too large, the robot fails to be enabled, and you need to reconfigure the load parameters or unselect this option.

#### ⚠️NOTICE

It is recommended to select **Execute load anomaly detection** when enabling the robot to ensure operational safety and protect the device.

Once the robot is enabled successfully, the enable button turns **ON**. Clicking it again can disable the robot.



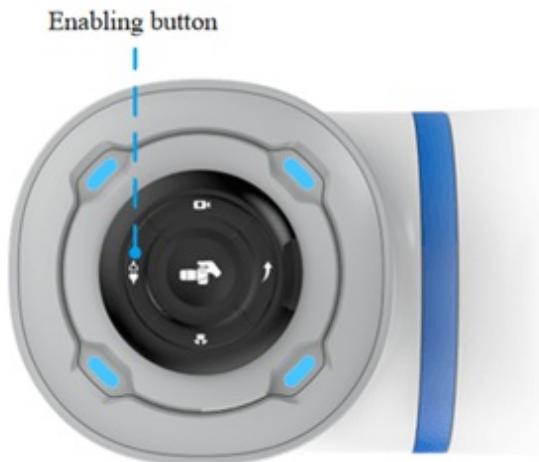
## Enable through robot button

### Robot enable button location

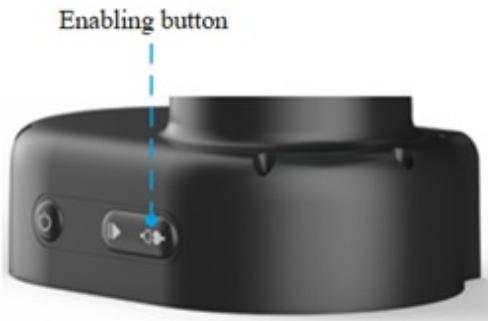
- For **CR A** series robot (except CR20A), the enable button is located at the end of the robot.

**NOTE**

The CR20A robot does not have an enable button and can only be enabled through software.



- For **Magician E6** robot, the enable button is located at the base of the robot.



### Robot enable button operation

- Enable:** When the robot is powered on but not enabled (blue indicator stays on), long-press the enable button for about 1.5 seconds and the purple indicator will flash. Once the button is released, the robot enters the enabled status (the indicator turns green and stays on).
- Disable:** When the robot is enabled, long-press the enable button for about 1.5 seconds and the purple indicator will flash. Once the button is released, the robot exits the enabled status.

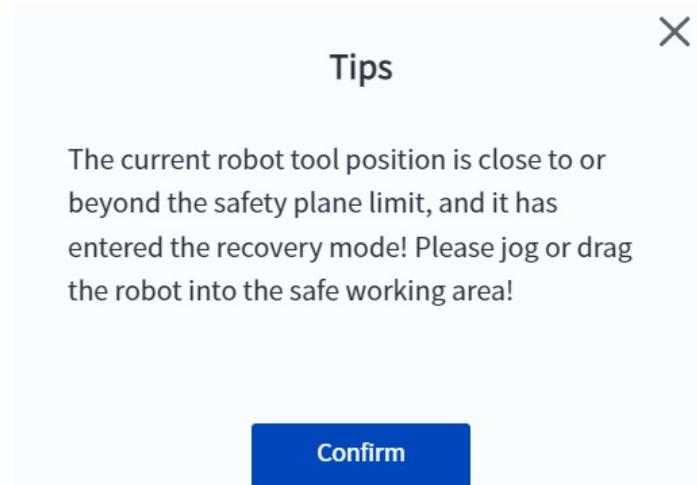
**NOTE**

If the robot is enabled through the button, it will use the last configured load settings. If it is the

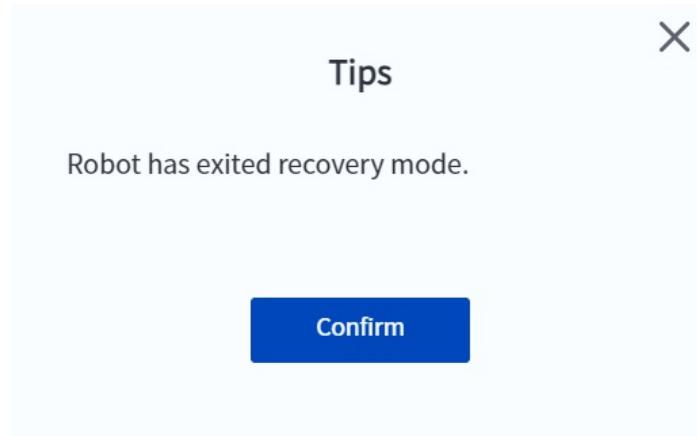
first time enabling the robot, the load parameters will default to 0.

## 5.3 Recovery mode

When the robot is enabled outside of a safe zone, the following popup will appear, and the robot's end indicator will turn orange, indicating that the robot has entered recovery mode.



In recovery mode, you can only control the robot's movement through **jog** or **drag** modes. Once the robot enters the safe zone, it will automatically exit recovery mode, and a prompt will appear.



## 5.4 Remote control

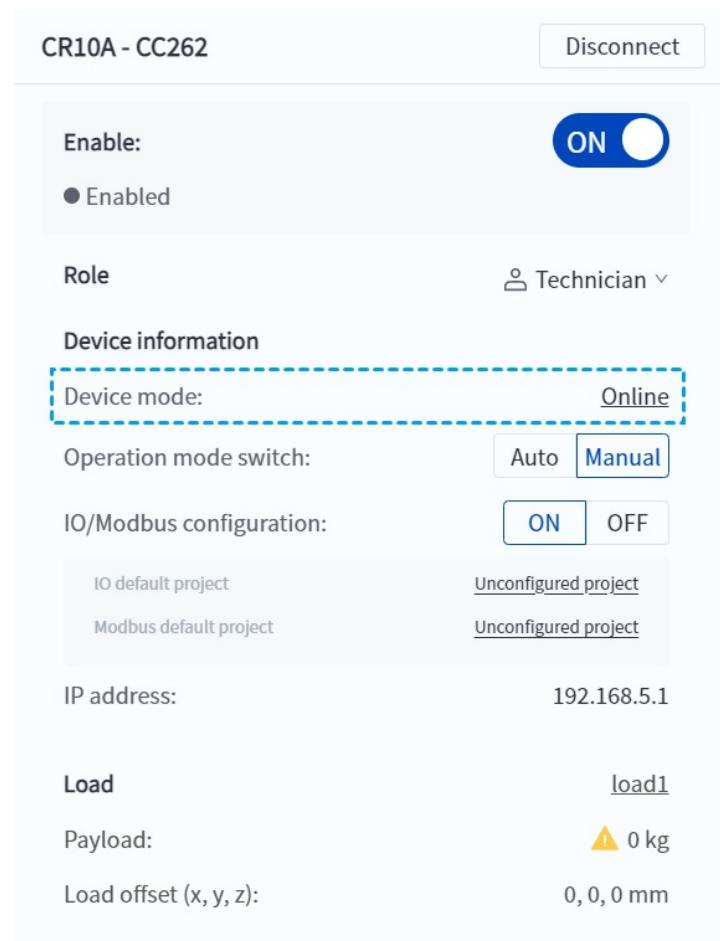
- [5.4.1 Device mode](#)
- [5.4.2 IO/Modbus configuration](#)

## 5.4.1 Device mode

The device mode indicates the robot's current control mode:

- **Online mode:** The default control mode, in which the robot can be controlled using DobotStudio Pro or be controlled remotely via I/O or Modbus.
- **TCP mode:** It is only used when users develop their own control software based on TCP communication. Except for Emergency Stop, DobotStudio Pro will not be able to operate the robot, but only to view the robot's status and related settings. If you need to develop your own control software, please contact technical support to obtain the *TCP\_IP Remote Control Interface Guide (V4)*.

You can view the current device mode on the information panel in the main interface. Clicking the underlined text allows you to switch the device mode.



### NOTE

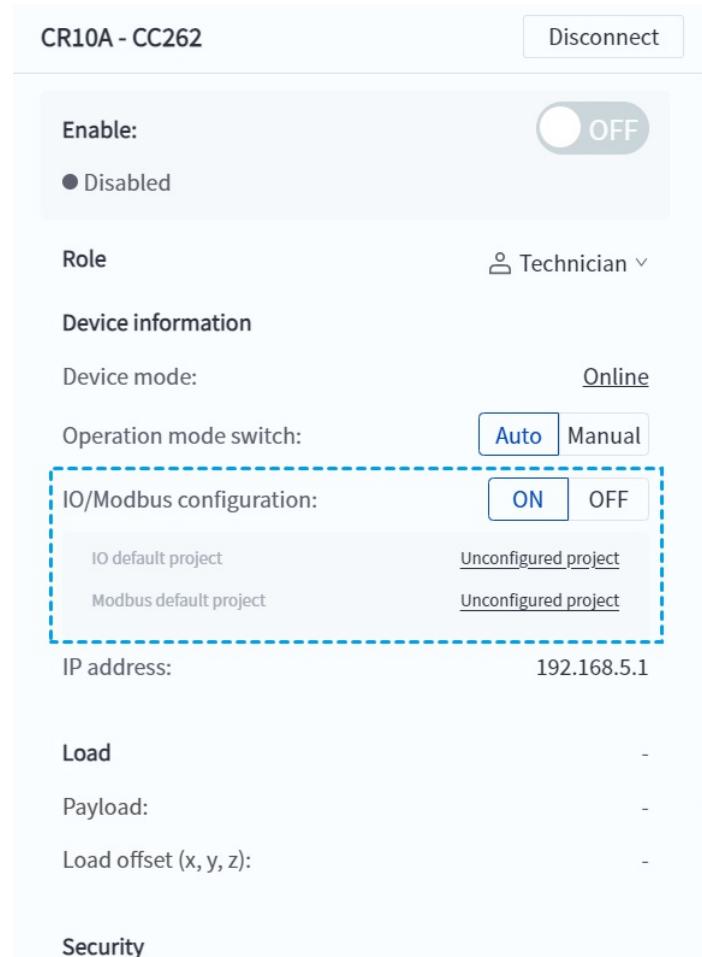
- The device mode cannot be switched when the robot is in **manual/automatic mode**. You need to disable the manual/automatic mode through **operation mode settings**.
- The device mode cannot be switched when the robot is in running or paused status.

- After switching the robot to TCP Mode, if you try to click a prohibited operation button, a prompt will appear stating that the robot is in TCP Mode and cannot be operated.

## 5.4.2 IO/Modbus configuration

The IO/Modbus configuration switch controls whether remote IO/Modbus inputs are effective. For safety reasons, it is generally recommended that the robot be controlled by only one input source. Therefore, when using DobotStudio Pro to control the robot, it is advised to turn off the IO/Modbus configuration switch.

You can toggle the IO/Modbus configuration ON or OFF by clicking it in the information panel on the main interface.



### ⚠️ NOTICE

Through [I/O settings](#) and [Modbus settings](#), you can configure the I/O and Modbus operations for your project.

The actual range of effectiveness for IO/Modbus inputs is also influenced by whether the robot is in [Manual](#) or [Automatic mode](#), as shown in the table below.

Operation mode	IO/Modbus ON	IO/Modbus OFF
Manual/Automatic mode not enabled	All IO/Modbus inputs are valid	All IO/Modbus inputs are invalid
Automatic mode	Only IO/Modbus inputs available in Automatic mode are valid	All IO/Modbus inputs are invalid
Manual mode	Only IO/Modbus inputs available in Manual mode are valid	All IO/Modbus inputs are invalid

### Remote IO/Modbus control

Function	Manual mode	Automatic mode
Start	X	√
Stop	X	√
Pause	X	√
Enable	√	√
Disable	√	√
Clear alarm	√	√
Enter drag mode	√	X
Exit drag mode	√	X
Select project	X	√

#### NOTE

In manual mode, if a pause is triggered by a collision, drag mode cannot be entered.

### Safety I/O

Function	Manual mode	Automatic mode
User E-Stop	√	√
Protective stop	X	√
Protective stop reset	X	√
Reduced mode	X	√

#### NOTE

- When manual/automatic mode is not enabled, or after a protective stop is triggered in manual mode, the robot can be jogged and dragged, but running a project and trajectory playback are not allowed.
- After a protective stop in automatic mode, all operations are not allowed, including jogging, dragging the robot, running projects, and trajectory playback.

## 5.5 Manual/Automatic mode

DobotStudio Pro supports enhancing the safety of on-site applications by setting different operation modes, which can be enabled in [Operation mode settings](#).

### NOTE

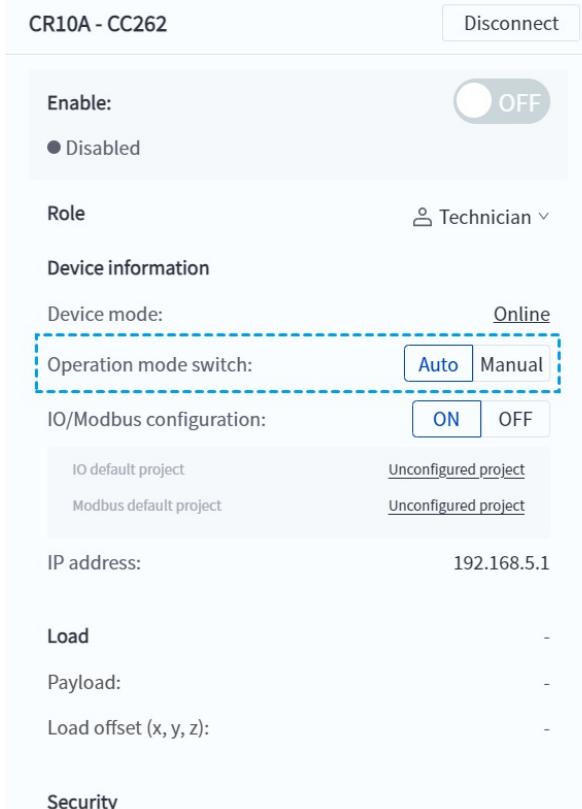
The CR20A robot is set to **Manual mode** by default, while other models are set to neither Manual nor Automatic mode by default.

- **Manual mode:** This mode is generally used for robot programming and debugging.
- **Automatic mode:** This mode is generally used for automated operations once the robot is deployed.

When the **operation mode switching** function is enabled in [Operation mode settings](#), the main interface's information panel will display a switch to toggle between manual and automatic modes. The default mode is **Manual**.

You can click the switch to change to **Automatic** mode. If a password is set for automatic mode, you will need to enter the password to switch.

When the robot is running, you cannot switch between manual and automatic modes. However, if the robot is in the paused or stopped status, you can switch between manual and automatic modes.



Allowed operations in manual/automatic mode are described in the [IO/Modbus configuration](#) section.

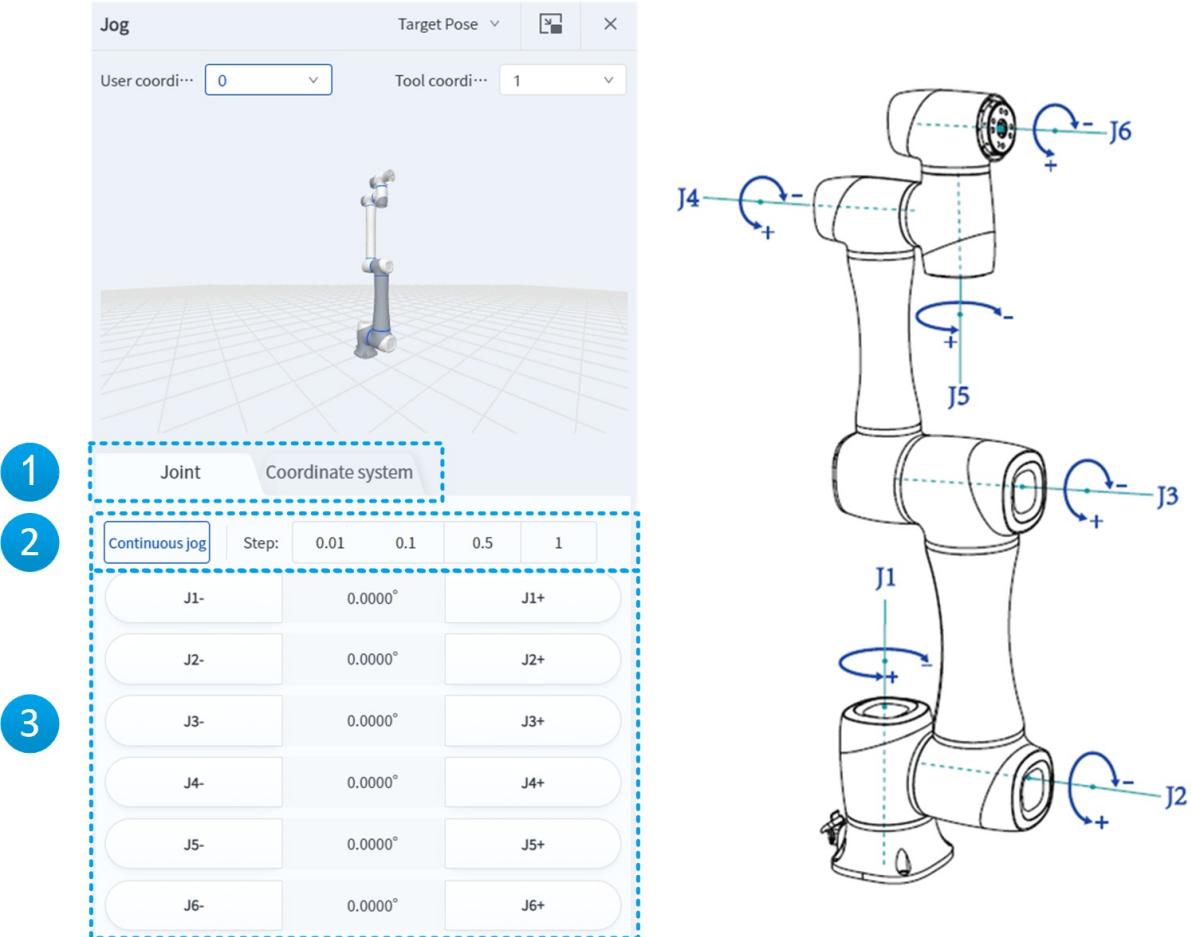
## 5.6 Jog

You can manually control the robot's motion using DobotStudio Pro, typically for point teaching.



To use the jog function, click **Jog** at the top toolbar to open the Jog panel, which supports both continuous jog and step motion based on either joint angles or the Cartesian coordinate system.

### Joint Jog



Joint jog refers to controlling the rotation of individual robot joints.

To perform joint jogging, first click the **Joint Jog** tab in area ① (as shown in the figure above), then use the buttons in area ③ to control the motion of individual joints. Refer to the diagram on the right of the figure above for the position and rotation direction of each joint axis.

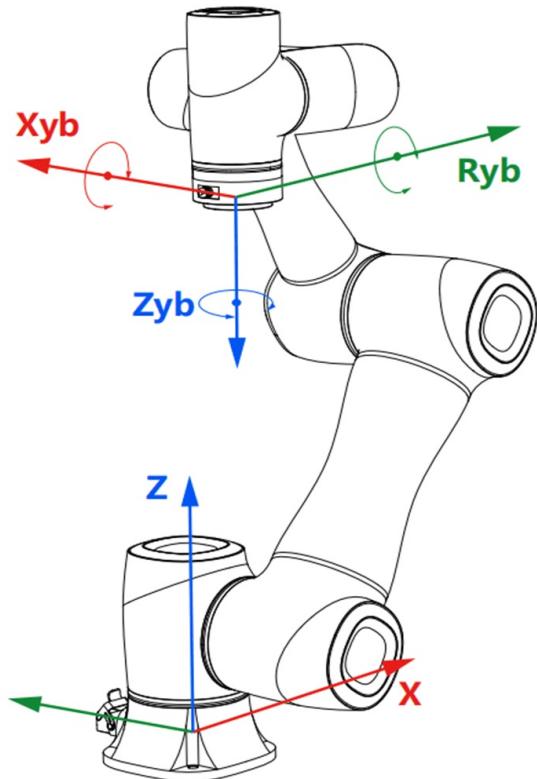
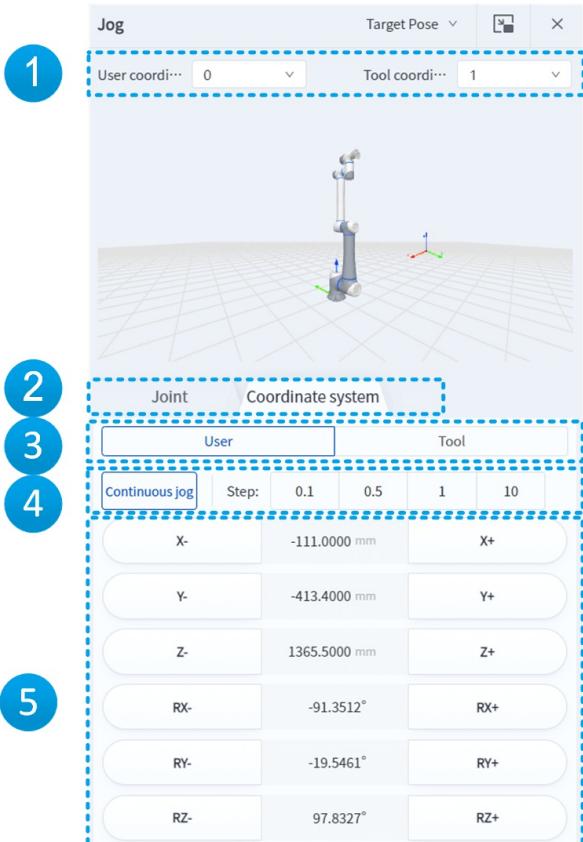
The options in area ② allow you to configure how the motion buttons in area ③ behave:

- **Continuous Jog:** If this option is selected, long-pressing the motion button can keep the robot moving, and the robot will stop as soon as you release the button.
- **Step:** If this option is selected, each click on the motion button moves the robot by a specified

distance (unit: °). Long-pressing the button will only trigger one step.

When teaching points, you can use **Continuous Jog** to move the robot near the target point, then use **Step** to fine-tune the position.

## Coordinate Jog



Coordinate jog refers to controlling the robot's TCP (Tool Center Point, the origin of the current tool coordinate system) to translate and rotate along a specified coordinate system.

To perform coordinate jogging, first click the **Coordinate Jog** tab in area ② (as shown in the figure above), then use the buttons in area ⑤ to control the motion of the TCP. The diagram on the right of the figure above, using User coordinate system 0 as an example, shows the definition of each axis and its rotation.

The options in area ④ allow you to configure how the motion buttons in area ⑤ behave:

- **Continuous Jog:** If this option is selected, long-pressing the motion button can keep the robot moving, and the robot will stop as soon as you release the button.
- **Step:** If this option is selected, each click on the motion button in area ③ moves the robot by a specified distance (X/Y/Z in mm, RX/RY/RZ in °). Long-pressing the button will only trigger one step.

When teaching points, you can use **Continuous Jog** to move the robot near the target point, then use **Step** to fine-tune the position.

### Switching the reference coordinate system

You can change the current User coordinate system and Tool coordinate system in area ①. You can manage the robot's coordinate systems in **Settings > Coordinate system management**.

In area ③, you can set the reference coordinate system for jogging:

- **User:** When jogging, the robot's TCP moves along the current user coordinate system. For example, X+ means the TCP moves in the positive direction of the X-axis of the user coordinate system.
- **Tool:** When jogging, the robot's TCP moves along the current tool coordinate system. For example, X+ means the TCP moves in the positive direction of the X-axis of the tool coordinate system.

The robot's 3D model will display the currently active coordinate system.

## Edit Jogging Value

In the joint jogging or coordinate system jogging motion control area, double-click any value can bring up the "Edit Jogging Value" window.

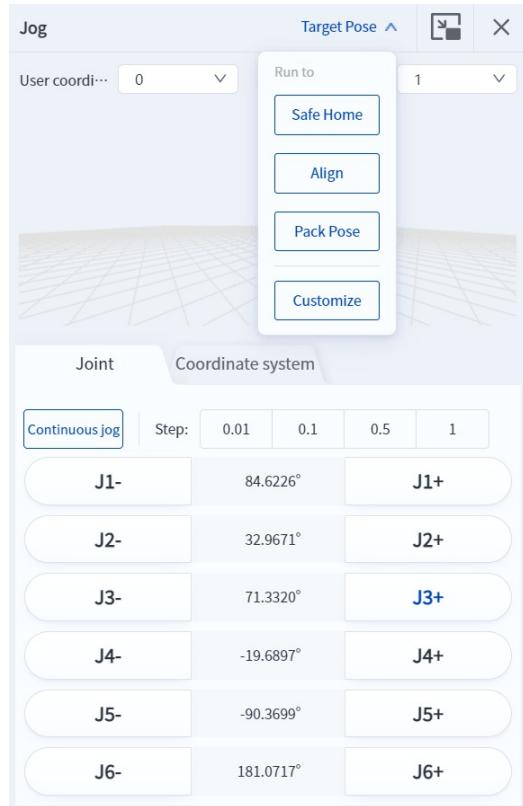


Directly input the target joint angle or coordinate system values, and the target position will be highlighted with a blue contour shadow. Long-press the **Run to** button can move the robot to the target position.

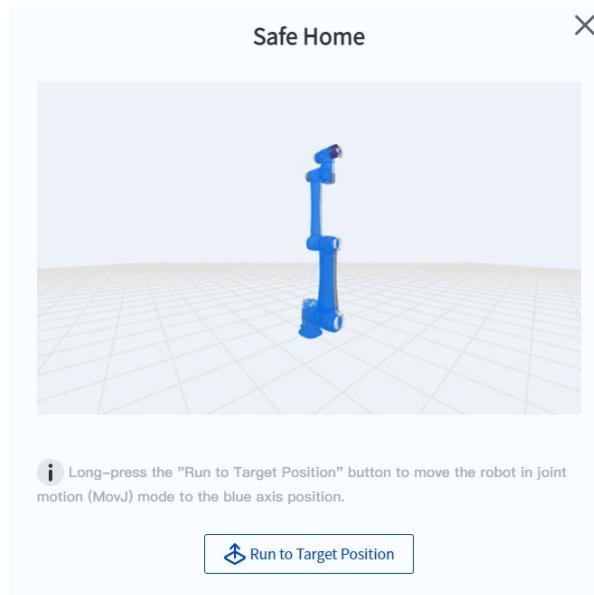
## Target Pose

The target pose allows the robot to quickly move to a specific position.

### Safe Home



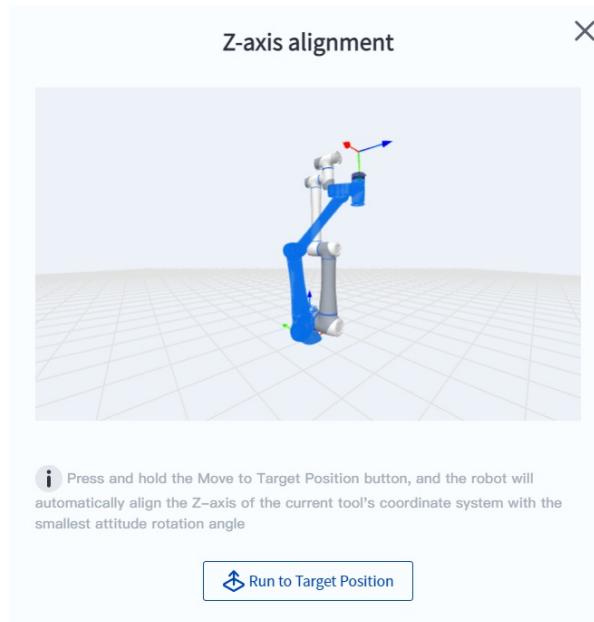
Click **Target Pose > Safe Home** will bring up the following prompt box. Long-press the **Run to Target Position** button can move the robot to the **Safe home position**, and the robot will stop as soon as you release the button.



## Z-Axis Alignment

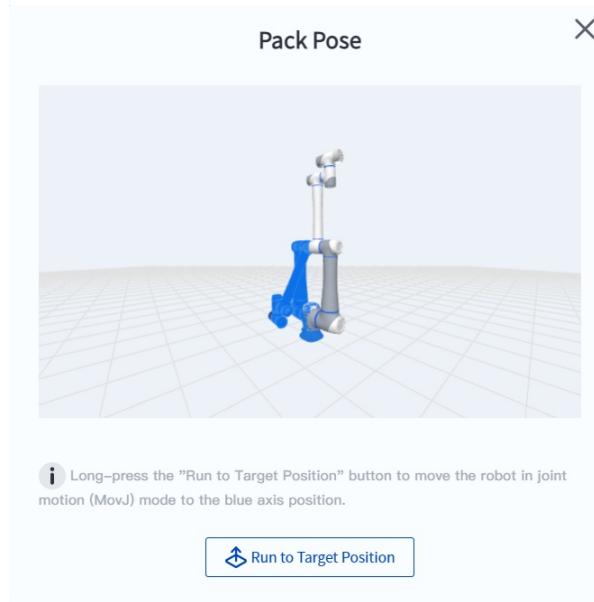
Click **Target Pose > Z-Axis Alignment** will bring up the following prompt box. Long-press the **Run to Target Position** button, the robot will automatically align the Z-axis of the current tool coordinate system to the Z-axis of the user coordinate system with the minimal rotation angle, and the robot will stop as soon

as you release the button.



### Pack Pose

Click **Target Pose > Pack Pose** will bring up the following prompt box. Long-press the **Run to Target Position** button can move the robot to the [Packing posture](#), and the robot will stop as soon as you release the button.



### Custom

Click **Target Pose > Custom**, the interface is the same as that of [Edit Jogging Value](#).

## 5.7 Drag

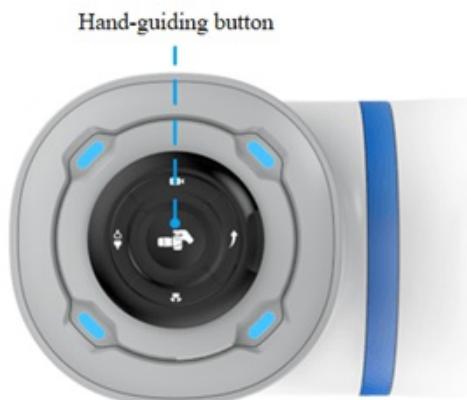
The robot can enter the drag mode by using the hand-guiding button at the end of the arm. In this mode, you can manually drag the robot arm to adjust its posture for point teaching.

In the drag mode, you can modify the resistance of each joint in [Drag settings](#) page.

**i** NOTE

Before dragging the robot, ensure that the [load parameters](#) and [installation angles](#) are correctly configured.

- For the **CR A series** robots (except CR20A), the hand-guiding button is shown in the figure below.



When the robot is enabled (green indicator steady on), long-press the hand-guiding button (more than 1.5 seconds) to enter the drag mode (green indicator flashes rapidly), and you can then drag the robot to change its posture.

To exit the drag mode, short-press the hand-guiding button (less than 1.5 seconds).

- For **CR20A** robot, the hand-guiding button is located on the side of the end effector and marked with **FREE**, as shown in the figure below.



When the robot is enabled (green indicator steady on), press this button to enter the drag mode (green indicator flashes rapidly), and you can then drag the robot to change its posture.

Releasing the hand-guiding button can exit the drag mode.

- For **Magician E6** robot, the hand-guiding button is shown in the figure below.



When the robot is enabled (green indicator steady on), long-press the hand-guiding button (more than 1.5 seconds) to enter the drag mode (green indicator flashes rapidly), and you can then drag the robot to change its posture. Long-pressing this button again in drag mode can switch the robot to trajectory recording mode.

To exit both drag mode and trajectory recording mode, short-press the hand-guiding button (less than 1.5 seconds).

## 5.8 Emergency stop and recovery

In case of an unexpected situation during robot operation, the emergency stop function can be used to immediately halt the robot.

You can trigger the emergency stop function in the following ways:

- Press the emergency stop button located on the controller.
- Trigger the user emergency stop input in [Safety I/O](#).
- Click the emergency stop button at the upper-right corner of the software interface.



If an emergency stop event is triggered, the robot will have the following two stop states:

- **If the robot stops within 500ms:** The robot will only be disabled and will not be powered off.
- **If the robot is still moving after 500ms:** The robot will be forcibly disabled and powered off.

### ⚠️ NOTICE

For Magician E6 series robots, if an emergency stop event is triggered, the robot will be immediately powered off.

Both states will trigger corresponding alarms.

After an emergency stop occurs, the emergency stop button icon will flash. To re-enable the robot, please click the emergency stop button again to reset it, clear the alarms and then re-enable it (If the robot is powered off, follow the on-screen prompts to power it back on).

### ⚠️ NOTICE

- The emergency stop button on the software interface is only a supplement to the hardware emergency stop button. In critical situations, please use the hardware emergency stop button in priority to stop the robot.
- If the emergency stop is triggered by the physical button or safety I/O, the software button's status will change, but it cannot be reset through the software interface. It must be reset via the corresponding input source.

## 5.9 Speed adjustment

The robot's speed can be adjusted using the speed slider on the right of the top toolbar. This allows users to reduce the robot's speed during program debugging to ensure safe operation.

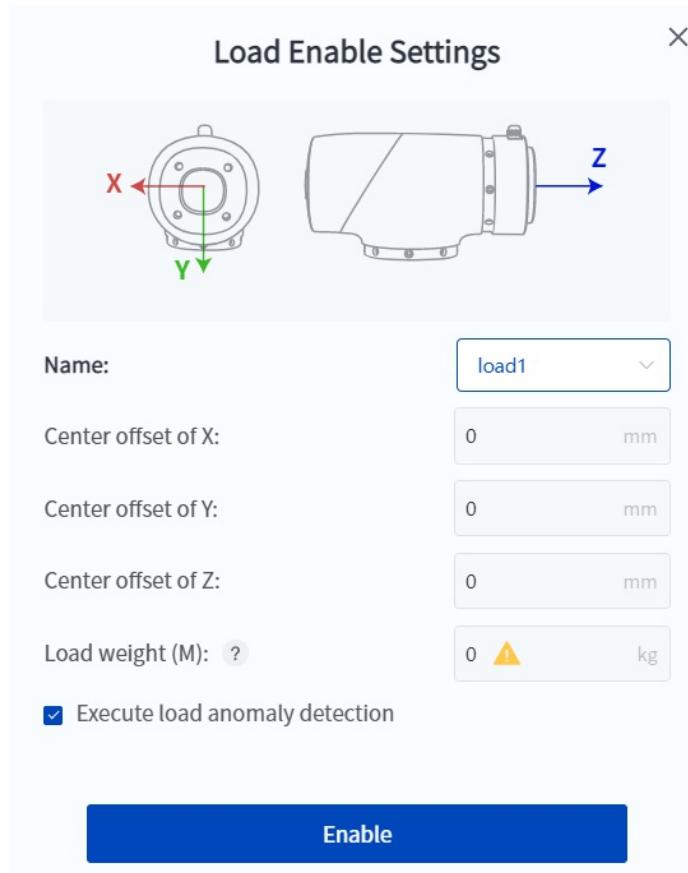


## 5.10 Load settings

The load parameters define the center of mass and weight of the robot arm's end load (including the gripper). These parameters must be set according to the actual load; incorrect settings can lead to decreased robot performance, false collision detection, and uncontrolled dragging issues.

You can set the robot's load parameters in the following ways:

- When enabling the robot through the software, load parameters must be set in the pop-up window. See [Enable](#) for details.



- During programming, you can set load parameters by using specific blocks or script commands during project execution. Refer to the command instructions in the appendix for details. These load parameters are only valid while the project is running.

**Block:**

set current payload 0 kg, offsetX 0 offsetY 0 offsetZ 0

- **Script:**

```
SetPayload(payload, {x, y, z}) -- Self-define the load parameters  
SetPayload(name) -- Use a preset load parameter group
```

Preset load parameter groups can be managed on [Load parameters](#) page.

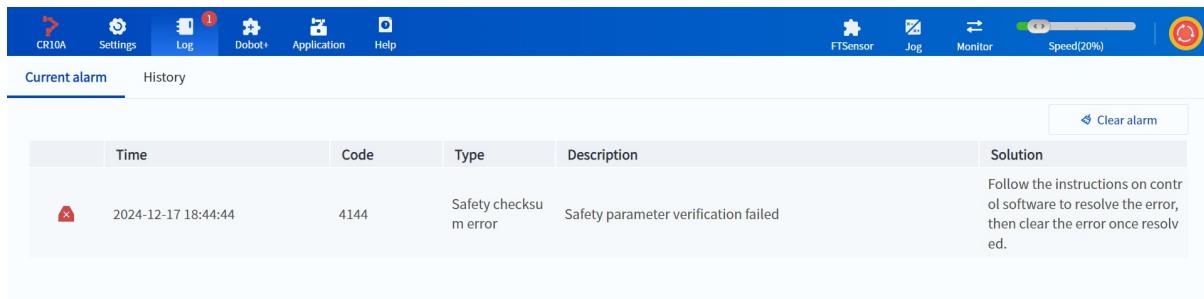
## 5.11 Collision detection settings

The robot will automatically stop when a collision is detected during its movement. You can set the sensitivity of the collision detection and specify how the system should respond after a collision. For **CRA** series robots, collision detection settings are detailed under [Safety limits](#). For **Magician E6** series robots, collision detection settings are described in the [Collision detection](#) section.

You can also set the collision detection sensitivity during operation using corresponding blocks or script commands. The collision detection parameters set in this way will only be effective while the project is running. For blockly programming, refer to the instructions for [setting collision detection](#) and [setting the collision backoff distance](#) in the Appendix. For script programming, refer to the instructions for the [SetCollisionLevel](#) and [SetBackDistance](#) commands in the Appendix.

## 5.12 Alarm

If a point is saved incorrectly or the robot is used improperly, for example, a robot moves to where a point is at a limited position or a singularity position, an alarm will be triggered. If an alarm is triggered when the robot is running, a red dot with a number will appear at the upper right of the log icon, indicating the number of current alarms.



Time	Code	Type	Description	Solution
2024-12-17 18:44:44	4144	Safety checksum error	Safety parameter verification failed	Follow the instructions on control software to resolve the error, then clear the error once resolved.

When the robot triggers an alarm, the current alarm page will display the alarm level icon, alarm time, error code, alarm type, description, and suggested solution. Please refer to the description and solution to resolve the alarm.

The meanings of the robot alarm level icons are as follows:

Alarm icon	Level	Type	Solution
	0	Fault	Error, pause, disable and power off
	1	Abnormal	Error, pause, and disable
	5	Error	Error and pause
	10	Fault	Error, stop, disable and power off
	11	Abnormal	Error, stop, and disable
	15	Error	Error and stop

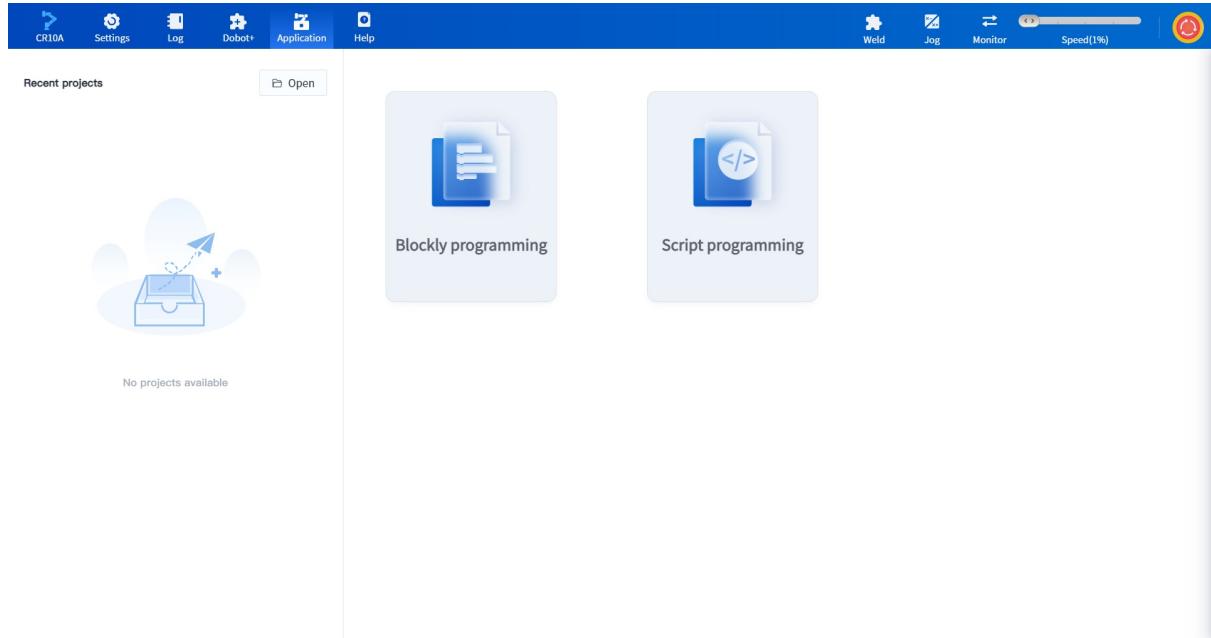
Click **Clear alarm** to clear the current alarm. For alarms that cannot be cleared directly, refer to the alarm solution for details.

# 6 Application

- [6.1 Application main interface](#)
- [6.2 Blockly programming](#)
- [6.3 Script programming](#)
- [6.4 Python programming \(Magician E6\)](#)
- [6.5 Debugging and running](#)
- [6.6 Trajectory recovery](#)

## 6.1 Application main interface

You can choose the appropriate method in the application interface to create a project for controlling the robot's automated operations.



- **Blockly programming:** A graphical programming method that is easy for programming beginners.
- **Script programming:** A method based on the Lua programming language, suitable for users with some programming background.
- **Python programming:** A method based on the Python programming language, **available only when a PC is connected to a Magician E6 robot**, mainly for educational and research purposes.



Under **Recent projects**, recently opened projects are displayed. Clicking **Open** will pop up a "Select project" page. The icons to the left of the project name indicate the type of project:

-  : Blockly projects.
-  : Script projects.
-  : Python projects.

## 6.2 Blockly programming

### 6.2.1 Overview

DobotStudio Pro provides blockly programming. You can program by dragging blocks from the sidebar into the programming area, without the need to write codes.

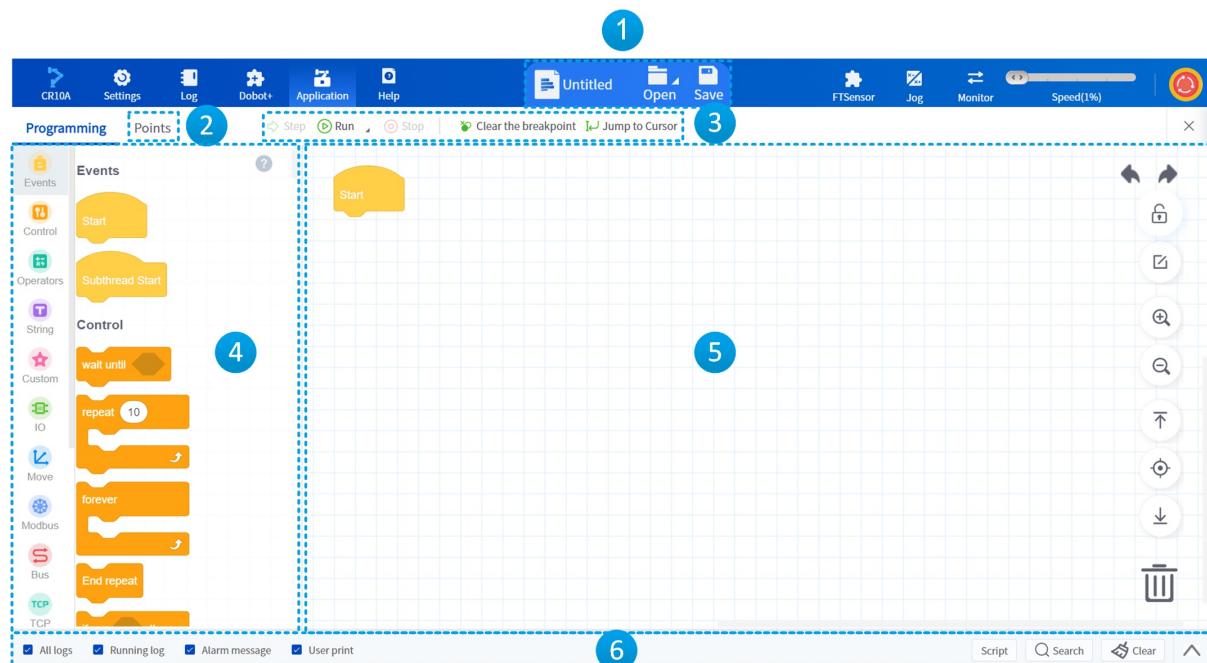
**Project:** The blockly programming is edited and run on a project basis and it supports debugging. Each time you open the software and enter the programming page, a new project is created by default. After programming, you need to name and save the project before debugging and running it. The project is saved in the robot controller and can be imported and exported.

Each project supports one main thread and up to four sub-threads.

- **Main thread:** The main thread starts with the **Start** block. All blocks can be used in the main thread.
- **Sub-thread:** The sub-thread starts with the **Subthread Start** block. Sub-threads run in parallel with the main thread and are typically used to set I/O, variables, etc., but cannot use motion blocks.

**Points:** During programming, you can move the robot using jog or drag mode, then open the Points page and save the robot's current posture as a teaching point. The saved points in the Points page are tied to the project and can be used as parameters in commands. To save points that can be used across multiple projects, use [global variables](#).

The main interface of blockly programming is shown below.

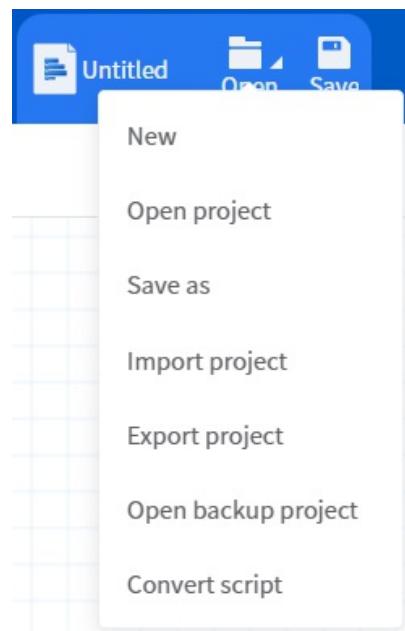


No.	NOTE
1	Display the current project name and allow project management.
2	Click to open Points page to manage the points in the project.
3	Buttons for debugging and running the project.
4	Provide blocks for programming, which can be searched by category and color. Click  at the upper right to view detailed explanations of the blocks.
5	The block programming canvas, that is, the project editing area. If the block is modified but not saved, you will see  on the left of the block to indicate changes.
6	Log panel, used to view the project's running logs. <ul style="list-style-type: none"> <li>Click the icon  on the far right to expand or collapse the log display area.</li> <li>Select the options on the left to filter the types of logs displayed.</li> <li>Click <b>Script</b> to view the script corresponding to the current block program.</li> <li>Click  <b>Search</b> to search for the specified character in the log.</li> <li>Click  <b>Clear</b> to clear the log display area.</li> </ul>

The icons on the right side of the programming area is described below.

Icon	NOTE
	Undo/Redo programming operations.
	Lock/Unlock the programming area.
	Enter editing mode. See the programming description below for details.
 	Zoom in/Zoom out the programming area.
  	Back to the top of blocks/Center the blocks/Back to the bottom of blocks.
	Drag the block to this icon to delete it. Right-click the block and select <b>Delete Block</b> to delete it.

## 6.2.2 Project management

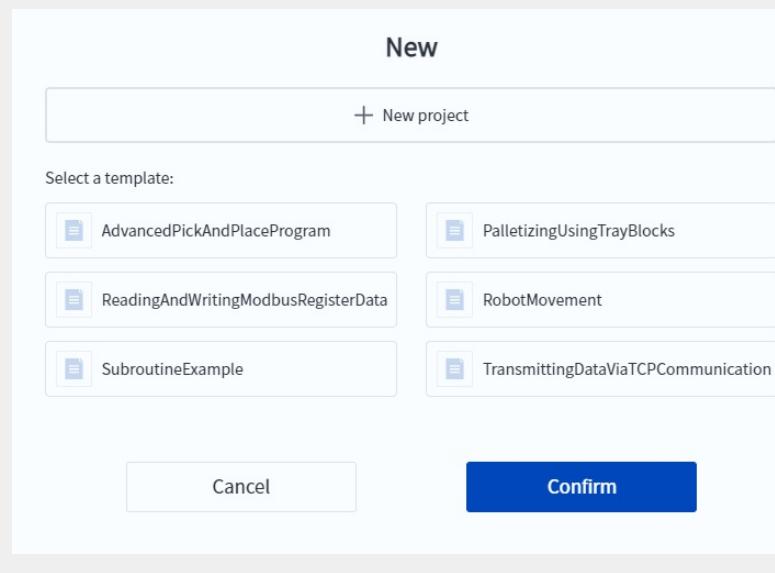


When the Blockly programming interface opens, an empty project is displayed by default with the project name displayed as **Untitled**.

Click to open the File menu, where you can create, open, save, import, export projects, and convert Blockly projects into script projects. After converting the Blockly project to script project, you can open and edit it in [Script programming](#).

### NOTE

When creating a new project, you can choose either a blank project or select a Blockly programming template.



Click  to save the current project. If the project is unnamed, you need to enter a name first.

In the following scenarios, DobotStudio Pro will automatically back up the project:

- Every ten minutes, DobotStudio Pro checks if there are any changes to the currently open project. If there are, it will automatically back up the project to the controller.
- Before starting a project, DobotStudio Pro checks if the currently open project has been modified. If so, it will automatically back up the project to the controller.
- When DobotStudio Pro disconnects from the controller (whether intentionally or due to an error), it checks for any changes to the currently open project. If there are, it will automatically back up the project to the local device (PC or tablet).

Projects backed up to the controller or the local device can be opened through the **Open backup project** option in the  menu.

## 6.2.3 Points page

You can move the robot to the desired posture using **jog** or **drag** mode and save the point in Points page.

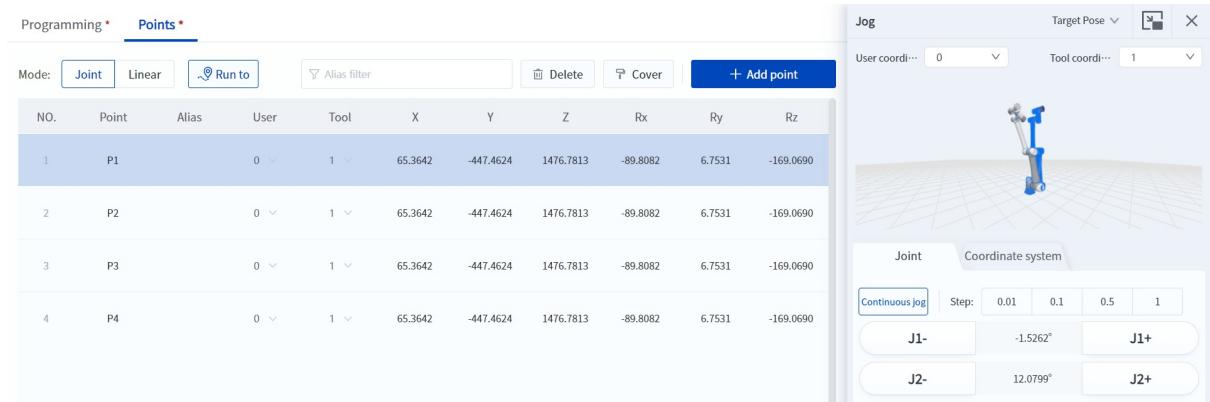
### NOTE

When the Points page is opened, you can also add teaching points by pressing the **POINT** button on the side of the CR20A end flange.



No.	NOTE
1	<ul style="list-style-type: none"><li>• Click  <b>Add point</b> to save the robot's current posture as a new point.</li><li>• After selecting a point, click  <b>Cover</b> to overwrite it with the robot's current posture.</li><li>• After selecting a point, click  <b>Delete</b> to delete the point.</li><li>• Click <b>Alias filter</b> to filter by alias and display the corresponding points in the list.</li></ul>
2	Points page. After selecting a point, clicking any value other than <b>NO.</b> and <b>Point</b> allows you to directly modify the value.
3	Control the robot to move to the selected point in the specified mode.

The posture of the selected point in the Points page will be displayed as a blue outline in the simulation area of the Jog panel, as shown in the figure below.



## 6.2.4 Programming

Before starting programming, please define the function you want the program to achieve. As an example, this section will show how to write a simple Blockly program that makes the robot move in a loop between two points.

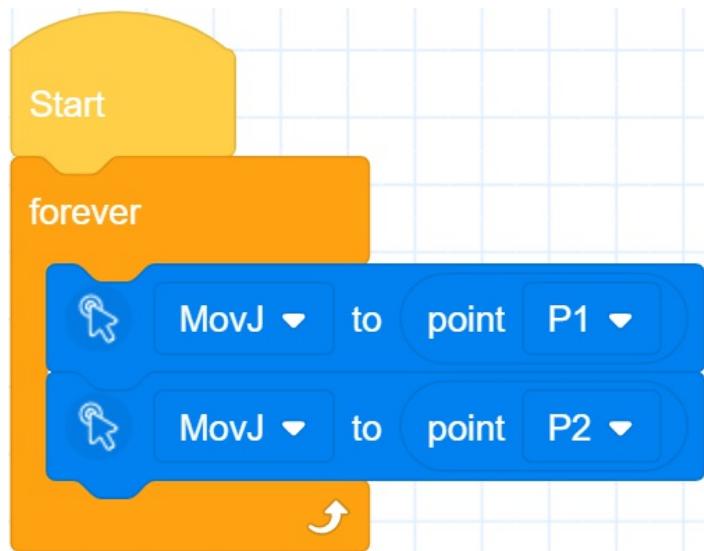
### ⚠️ NOTICE

Before debugging or running the project, make sure that there are no personnel or obstacles within the working space of the robot.

Drag the **forever** block from the **Control** block module on the left into the canvas and place it under the **Start** block.

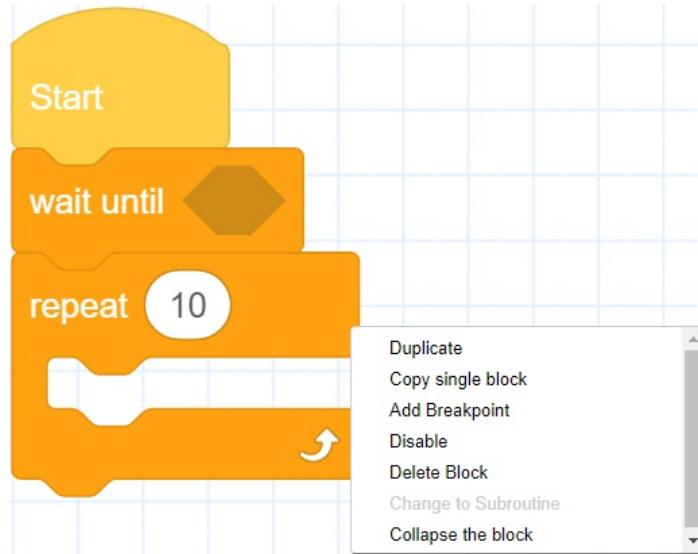
Drag a **Move to Point** block from the **Motion** block module into the **forever** block, and select **P1** from the drop-down list.

Drag another **Move to Point** block under the previous one and select **P2** from the drop-down list.



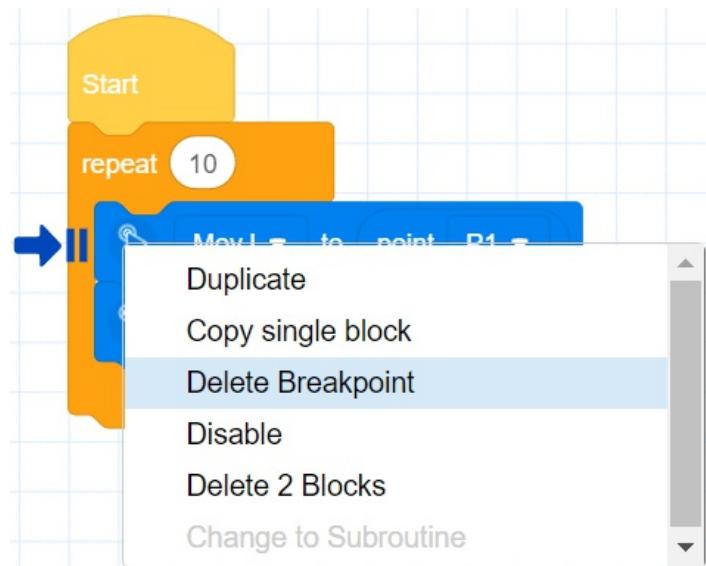
You now have a simple looping program that moves the robot between two points.

In addition to the basic programming functions described above, DobotStudio Pro also supports the following advanced operations:



- **Copy:** Right-click a block already placed in the canvas (long-press on App) to open the menu. You can copy the selected block along with all blocks connected below it, or just the selected block alone (any nested blocks will also be copied).
- **Add/delete breakpoint:** For blocks that support breakpoints, right-click and select **Add breakpoint**.

The breakpoint is marked as shown below. Then the **Add breakpoint** button will change to **Delete breakpoint**. When the program reaches a block with a breakpoint, it will pause. The cursor will stay on the breakpoint block, and the breakpoint block will not be executed. For blocks with an added breakpoint, right-click and select **Delete breakpoint** can remove the breakpoint from the current block.



- **NOTE**
  - When the program is in a paused status, breakpoints can be added or deleted.
  - Only blocks under **Start / Subroutine** in blockly programming page support adding breakpoints.
  - If a breakpoint is added to the entire subroutine (no breakpoints are added to the blocks under the subroutine), then the subroutine block will be displayed:



- If breakpoints are also added for other blocks under the subroutine, then the subroutine block will be displayed:



- For blocks with breakpoints, if they are dragged back to the menu bar, dragged away from the Start/Subroutine block, the breakpoint function will be automatically disabled.

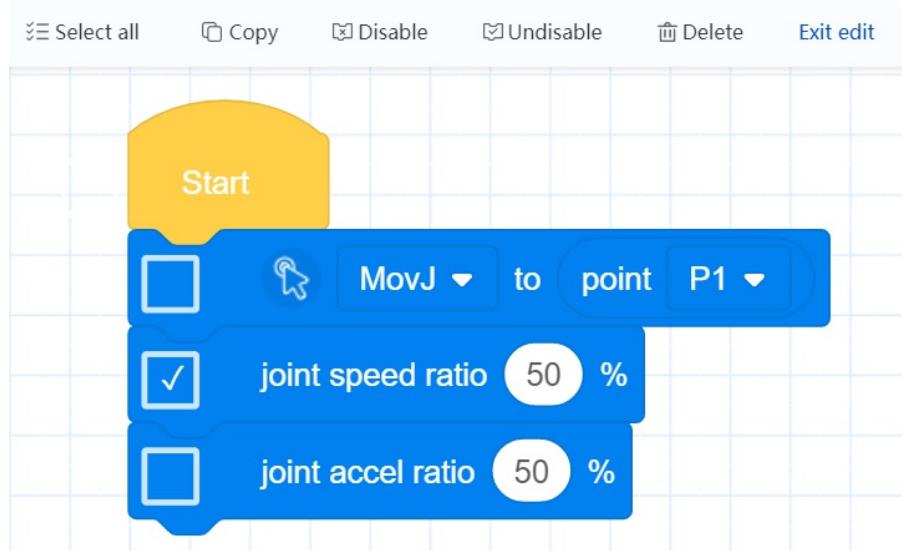
- **Disable:** Select a block and right-click to select **Disable**, the selected block will be disabled and turn gray. At the same time, by clicking the **Script** button, you will see that the script for the disabled block is turned into a comment. Once a block is disabled, breakpoints cannot be added to it. Additionally, if a block already has a breakpoint, it will be removed when the block is disabled.
- **Delete:** There are three ways to delete blocks.
  - Drag the block back into the block menu on the left and release it.
  - Drag the block into the trash icon at the bottom right of the page and release it.
  - Right-click the block and select **Delete** (any nested blocks will also be deleted).
- **Change to Subroutine:** Right-click a group of blocks not connected to the **Start** block or **Subthread Start** block, then select **Change to Subroutine**. This allows you to convert the blocks into a subroutine, which can then be reused as a single subroutine block, improving the programming efficiency. **This option is unavailable in the subroutine editing interface.**
- **Collapse/Expand blocks:** Right-click the nested blocks to collapse or expand them, making it easier to view and edit. When a block is collapsed, any breakpoints added to it will be cleared, and breakpoints cannot be added to the collapsed blocks.



Clicking on the right side of the canvas will enter the editing mode.

In edit mode, you can select multiple or all blocks to perform operations such as copying, disabling, enabling, or deleting them.

Clicking **Exit editing** or performing other operations in the programming area will exit the edit mode.



For more details on blocks, click at the upper right of the block menu, or refer to [Appendix B](#).

## 6.3 Script programming

### 6.3.1 Overview

Dobot robots provide various APIs, such as motion commands and TCP/UDP commands, using the Lua language, making it convenient for users to access them during secondary development. DobotStudio Pro offers a Lua programming environment, enabling users to write their own Lua scripts to control the robot's operations.

**Project:** The script programming is edited and run on a project basis and it supports debugging. Each time you open the software and enter the programming page, a new project is created by default. After programming, you need to name and save the project before debugging and running it. The project is saved in the robot controller and can be imported and exported.

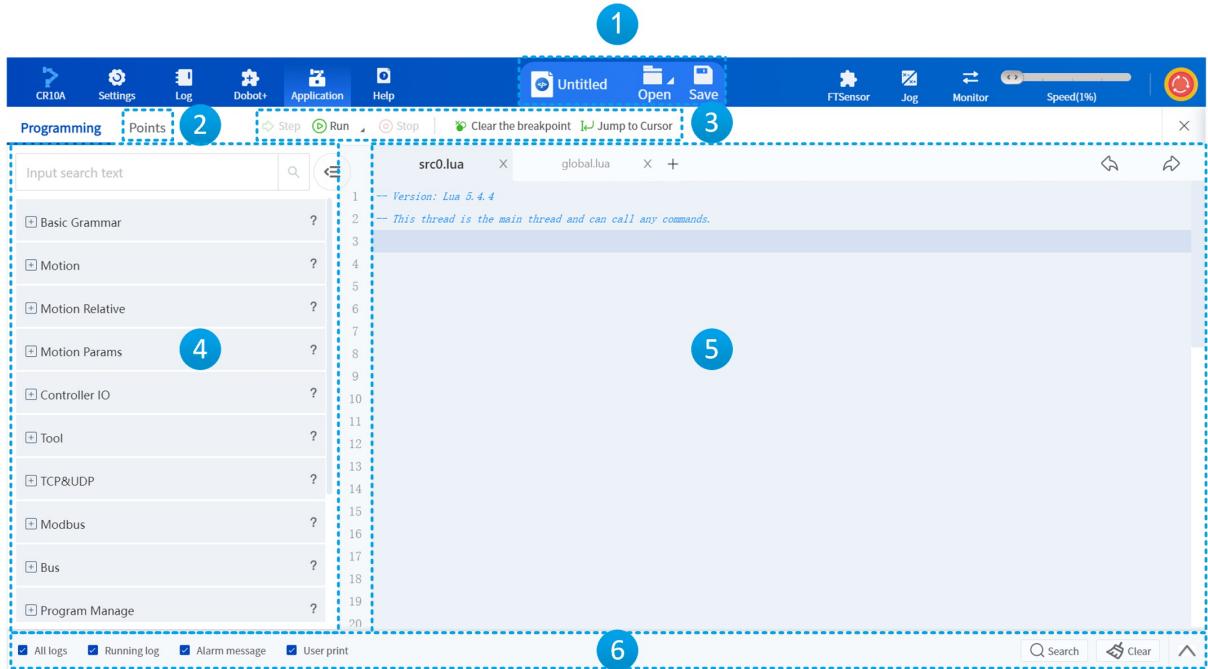
Each project consists of the following script files, displayed as tabs:

- The **src0.lua** file is the main thread and can call any command.
- The **global.lua** file is only used to define variables and sub-functions.
- 0 – 4 **sub-thread(s)**, named as **src1.lua – src4.lua**. Sub-threads run in parallel with the main program (up to 4 sub-threads) and can be used to set I/O and variables, but they cannot call motion commands.

Once the project starts, the robot executes commands in the main thread and sub-threads from top to bottom, without the need to define an entry (main) function. For more details on Lua's basic syntax, refer to [Appendix C](#).

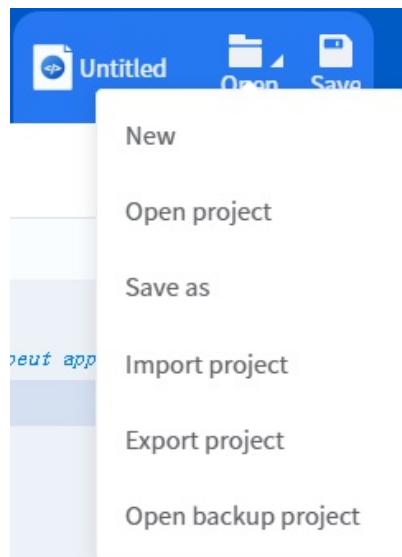
**Points:** During programming, you can move the robot using jog or drag mode, then open the Points page and save the robot's current posture as a teaching point. The saved points in the Points page are tied to the project and can be used as parameters in commands. To save points that can be used across multiple projects, use [global variables](#).

The main interface of script programming is shown below.



No.	NOTE
1	Display the current project name and allow project management.
2	Click to open Points page to manage the points in the project.
3	Buttons for debugging and running the project.
4	View and use the commands for programming. Click ? to view the relevant description on the commands.
5	Program editing area. Click the tabs to switch the script files. Click + to add sub-threads. Click ↺ ↻ on the upper right corner to undo/redo programming operations.
6	Log panel, used to view the project's running logs. <ul style="list-style-type: none"> <li>• Click the icon ▲ on the far right to expand or collapse the log display area.</li> <li>• Select the options on the left to filter the types of logs displayed.</li> <li>• Click 🔎 Search to search for the specified character in the log.</li> <li>• Click 🗑 Clear to clear the log display area.</li> </ul>

## 6.3.2 Project management

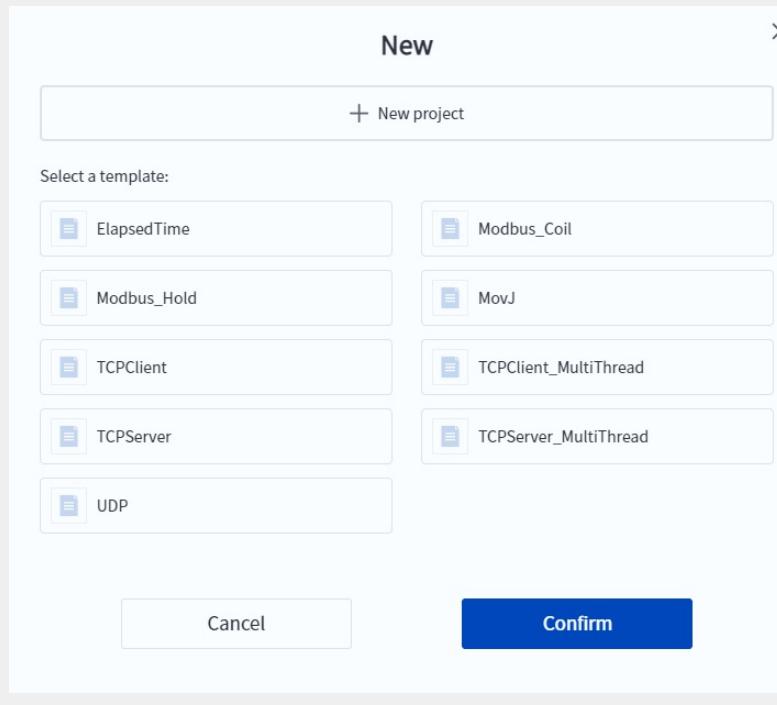


When the script programming interface opens, an empty project is displayed by default with the project name displayed as **Untitled**.

Click to open the File menu, where you can create, open, save, import, export projects, and more.

#### NOTE

When creating a new project, you can choose either a blank project or select a script programming template.



Click to save the current project. If the project is unnamed, you need to enter a name first.

In the following scenarios, DobotStudio Pro will automatically back up the project:

- Every ten minutes, DobotStudio Pro checks if there are any changes to the currently open project. If

there are, it will automatically back up the project to the controller.

- Before starting a project, DobotStudio Pro checks if the currently open project has been modified. If so, it will automatically back up the project to the controller.
- When DobotStudio Pro disconnects from the controller (whether intentionally or due to an error), it checks for any changes to the currently open project. If there are, it will automatically back up the project to the local device (PC or tablet).

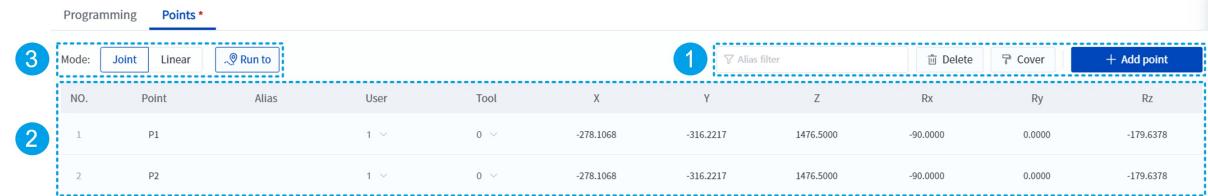
Projects backed up to the controller or the local device can be opened through the **Open backup project** option in the  menu.

### 6.3.3 Points page

You can move the robot to the desired posture using **jog** or **drag** mode and save the point in Points page.

#### NOTE

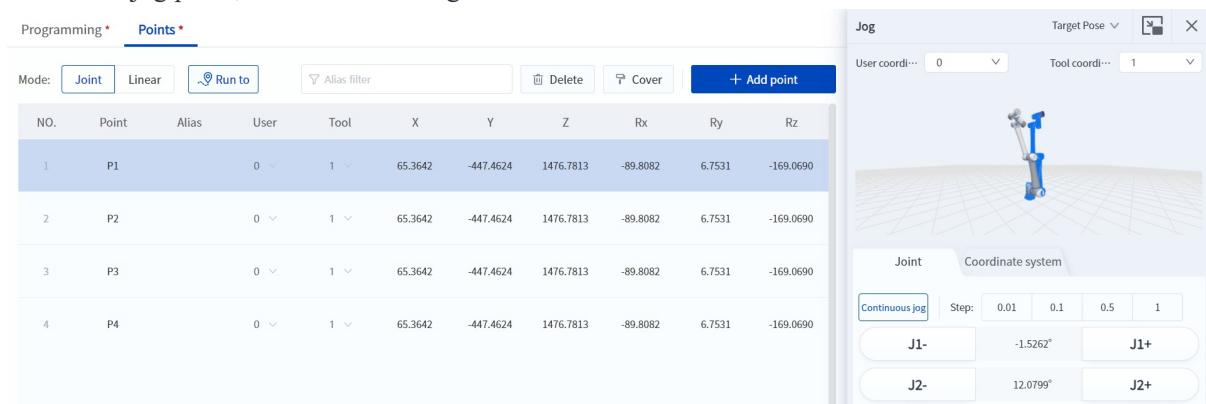
When the Points page is opened, you can also add teaching points by pressing the **POINT** button on the side of the CR20A end flange.



NO.	Point	Alias	User	Tool	X	Y	Z	Rx	Ry	Rz
1	P1		1	0	-278.1068	-316.2217	1476.5000	-90.0000	0.0000	-179.6378
2	P2		1	0	-278.1068	-316.2217	1476.5000	-90.0000	0.0000	-179.6378

No.	NOTE
1	<ul style="list-style-type: none"> <li>Click  <b>Add point</b> to save the robot's current posture as a new point.</li> <li>After selecting a point, click  <b>Cover</b> to overwrite it with the robot's current posture.</li> <li>After selecting a point, click  <b>Delete</b> to delete the point.</li> <li>Click <b>Alias filter</b> to filter by alias and display the corresponding points in the list.</li> </ul>
2	Point list. After selecting a point, clicking any value other than <b>NO.</b> and <b>Point</b> allows you to directly modify the value.
3	Control the robot to move to the selected point in the specified mode.

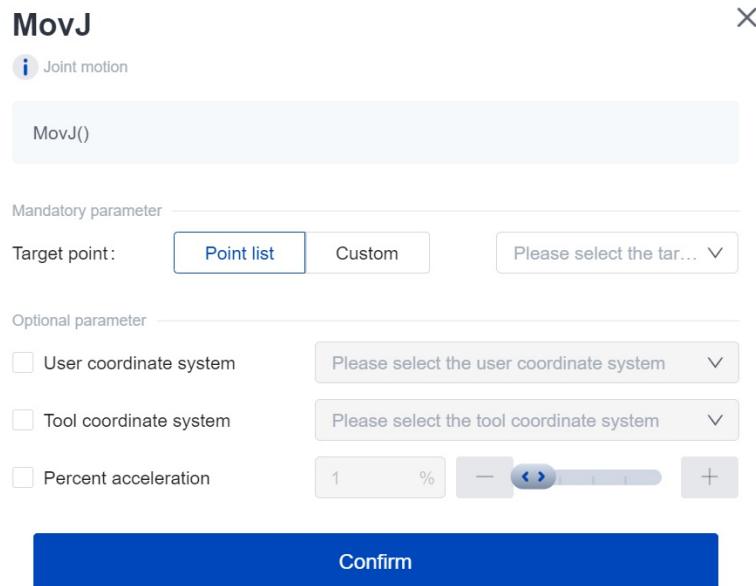
The posture of the selected point in the Points page will be displayed as a blue outline in the simulation area of the jog panel, as shown in the figure below.



### 6.3.4 Programming

You can insert commands through the following ways.

- Find the desired command in the command menu on the left and click , and a parameter settings window will pop up. Set the parameters in the window and click **OK** to add a command with parameters to the position of the cursor in the programming area.



- Find the desired command in the command menu on the left and double-click it. This will quickly insert the command with default parameters into the programming area, which you can then modify as needed.

**Programming \*** Points Step Run Stop Clear the breakpoint Jump to Cursor

Input search text	src0.lua*	global.lua
+ Basic Grammar	1 -- Version: Lua 5.4.4	x
- Motion	2 -- This thread is the main thread and can call any commands.	x +
Joint motion	3 MovJ(P1)	
MovJ	4	
Linear motion	5	
MovL	6	
	7	
	8	
	9	

- Directly type in the programming area on the right to write the code. The **TAB** key on the keyboard supports auto-completion of commands.
- Add breakpoint:** Click a code line using the mouse or touch to add a breakpoint. The breakpoint is marked as shown in the 3rd and 5th lines in the figure below. When the script reaches a code line set as a breakpoint, it will pause execution. The cursor will remain on the breakpoint line, and the line will not be executed.

```

3 MMovL(P1)while(true)
4 do
5 repeat

```

- **Delete breakpoint:** Clicking an existing breakpoint line again will remove the breakpoint, or you can click the  **Clear breakpoint** button to remove all breakpoints in the current project. The  **Clear breakpoint** button is disabled during program execution.

 **NOTE**

- Breakpoints can be added or deleted while the program is paused.
- Only "src0.lua" in script programming supports adding breakpoints.

Before starting programming, please define the function you want the program to achieve. As an example, this section will show how to write a simple script program that makes the robot move in a loop between two points.

1. Add the start point (P1) and the end point (P2) for the loop motion in turn in Points page.
2. Add `while` command to the programming area through any one way described above.
3. Add a `MovJ` command before `end`, and set P1 as the target point.
4. Add another `MovJ` command, and set P2 as the target point. The final code should look like below.

```
while(true)
do
    MovJ(P1)
    MovJ(P2)
end
```

You now have a simple looping program that moves the robot between two points.

If you need to write a sub-thread, click + on the right of the tab above the programming area to add a sub-thread. Then switch to the sub-thread to edit the program.

For more details on script programming, you can click  in the command menu, or refer to [Appendix C](#).

## 6.4 Python programming (Magician E6)

### 6.4.1 Overview

**Python programming is available only when the PC is connected to the Magician E6 robot.**

Dobot robots provide various APIs, such as motion commands and TCP/UDP commands, using the Python language, making it convenient for users to access them during secondary development. DobotStudio Pro offers a Python programming environment, enabling users to write their own Python scripts to control the robot's operations.

**Project:** The Python programming is edited and run on a project basis and it supports debugging. Each time you open the software and enter the programming page, a new project is created by default. After programming, you need to name and save the project before debugging and running it. The project is saved in the robot controller and can be imported and exported.

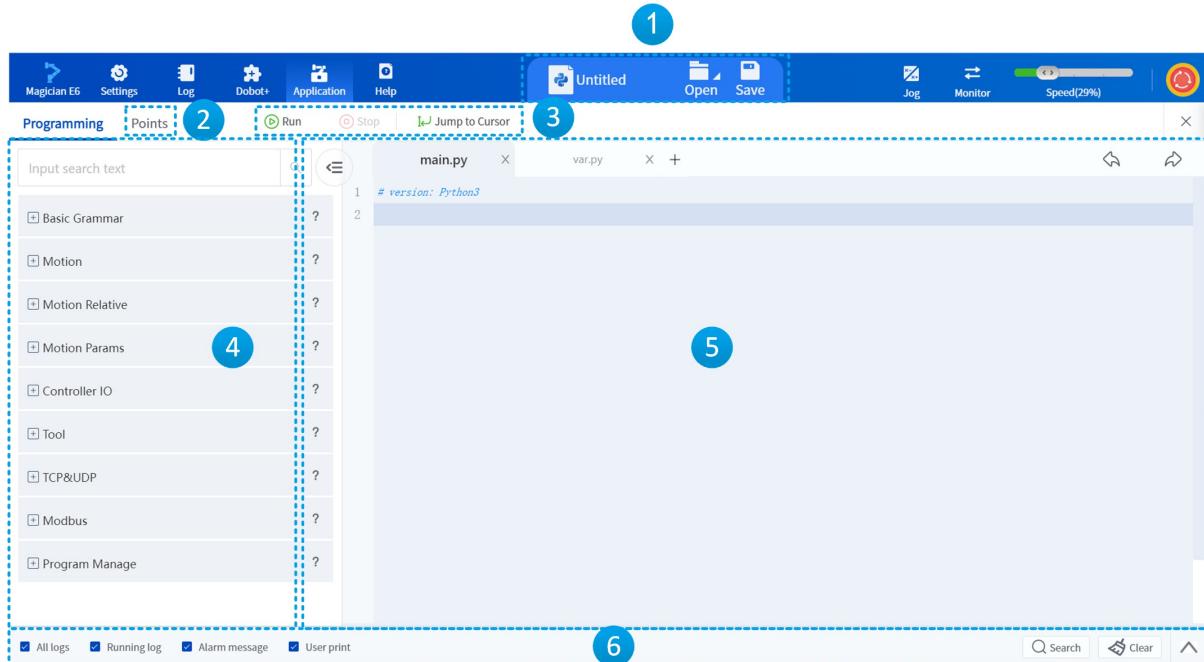
Each project consists of the following Python files, displayed as tabs:

- The **main.py** file is the **main thread** and can call any command.
- The **var.py** file is only used to define variables.
- 0 – 4 **sub-thread(s)**, named as **script1.py – script4.py**. Sub-threads run in parallel with the main program (up to 4 sub-threads) and can be used to set I/O and variables, but they cannot call motion commands.

Once the project starts, the robot executes commands in the main thread and sub-threads from top to bottom, without the need to define an entry (main) function. For more details on Python's basic syntax, refer to [Appendix E](#).

**Points:** During programming, you can move the robot using jog or drag mode, then open the Points page and save the robot's current posture as a teaching point. The saved points in the Points page are tied to the project and can be used as parameters in commands. To save points that can be used across multiple projects, use [global variables](#).

The main interface of Python programming is shown below.



No.	NOTE
1	Display the current project name and allow project management.
2	Click to open Points page to manage the points in the project.
3	Buttons running the project.
4	View and use the commands for programming. Click ? to view the relevant description on the commands.
5	Program editing area. Click the tabs to switch the Python files. Click + to add sub-threads. Click ↺ ↻ on the upper right corner to undo/redo programming operations.
6	Log panel, used to view the project's running logs. <ul style="list-style-type: none"> <li>Click the icon on the far right to expand or collapse the log display area.</li> <li>Select the options on the left to filter the types of logs displayed.</li> <li>Click <b>Search</b> to search for the specified character in the log.</li> <li>Click <b>Clear</b> to clear the log display area.</li> </ul>

## 6.4.2 Project management

When the script programming interface opens, an empty project is displayed by default with the project name displayed as **Untitled**.

Click to open the File menu, where you can create, open, save, import, export projects,

Click to save the current project. If the project is unnamed, you need to enter a name first.

In the following scenarios, DobotStudio Pro will automatically back up the project:

- Every ten minutes, DobotStudio Pro checks if there are any changes to the currently open project. If there are, it will automatically back up the project to the controller.
- Before starting a project, DobotStudio Pro checks if the currently open project has been modified. If so, it will automatically back up the project to the controller.
- When DobotStudio Pro disconnects from the controller (whether intentionally or due to an error), it checks for any changes to the currently open project. If there are, it will automatically back up the project to the local device (PC or tablet).

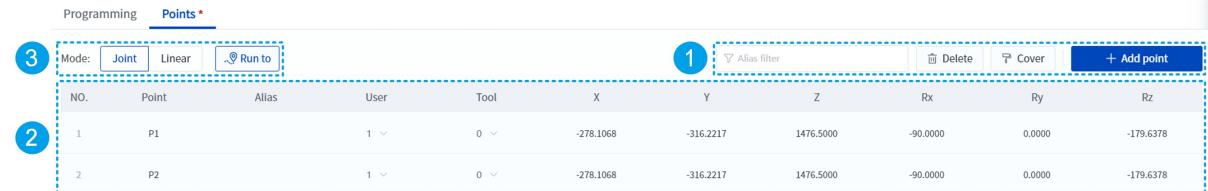
Projects backed up to the controller or the local device can be opened through the **Open backup project** option in the  menu.

### 6.4.3 Points page

You can move the robot to the desired posture using **jog** or **drag** mode and save the point in Points page.

#### NOTE

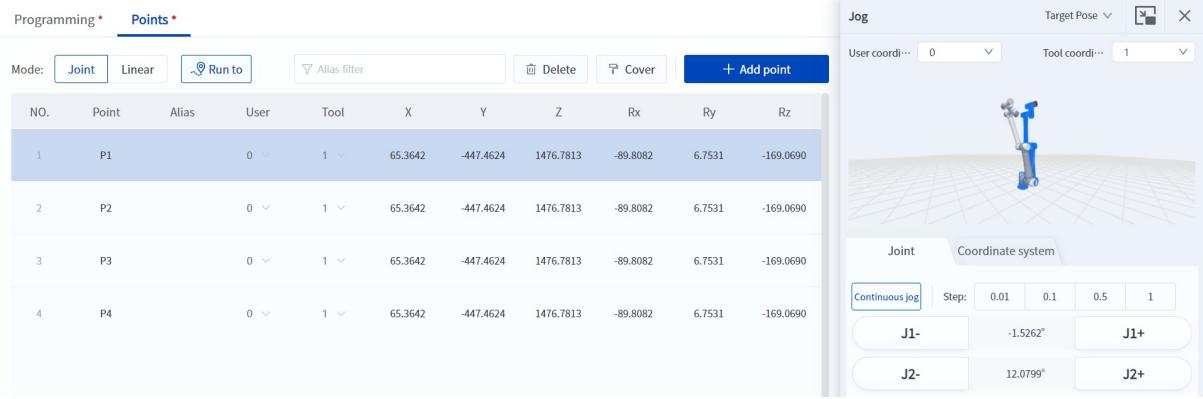
When the Points page is opened, you can also add teaching points by pressing the **POINT** button on the side of the CR20A end flange.



NO.	Point	Alias	User	Tool	X	Y	Z	Rx	Ry	Rz
1	P1	1	0	-278.1068	-316.2217	1476.5000	-90.0000	0.0000	-179.6378	
2	P2	1	0	-278.1068	-316.2217	1476.5000	-90.0000	0.0000	-179.6378	

No.	NOTE
1	<ul style="list-style-type: none"> <li>• Click  <b>Add point</b> to save the robot's current posture as a new point.</li> <li>• After selecting a point, click  <b>Cover</b> to overwrite it with the robot's current posture.</li> <li>• After selecting a point, click  <b>Delete</b> to delete the point.</li> <li>• Click <b>Alias filter</b> to filter by alias and display the corresponding points in the list.</li> </ul>
2	Point list. After selecting a point, clicking any value other than <b>NO.</b> and <b>Point</b> allows you to directly modify the value.
3	Control the robot to move to the selected point in the specified mode.

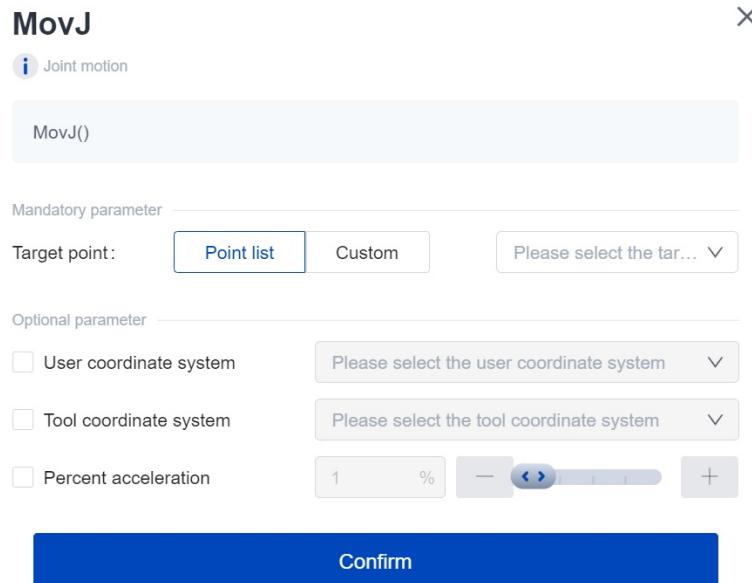
The posture of the selected point in the Points page will be displayed as a blue outline in the simulation area of the Jog panel, as shown in the figure below.



## 6.4.4 Programming

You can insert commands through the following ways.

- Find the desired command in the command menu on the left and click , and a parameter settings window will pop up. Set the parameters in the window and click **OK** to add a command with parameters to the position of the cursor in the programming area.



- Find the desired command in the command menu on the left and double-click it. This will quickly insert the command with default parameters into the programming area, which you can then modify as needed.
- Directly type in the programming area on the right to write the code. The Tab key can be used to auto-complete commands.

Before starting programming, please define the function you want the program to achieve. As an example, this section will show how to write a simple script program that makes the robot move in a loop between two points.

1. Add the start point (P1) and the end point (P2) for the loop motion in turn in Points page.
2. Add `while` command in the programming area.
3. Add a `MovJ` command, and set P1 as the target point.
4. Add another `MovJ` command, and set P2 as the target point. The final code should look like below.

```
while True:  
    MovJ(P1)  
    MovJ(P2)
```

You now have a simple looping program that moves the robot between two points.

If you need to write a sub-thread, click + on the right of the tab above the programming area to add a sub-thread. Then switch to the sub-thread to edit the program.

For more instructions on script programming, you can click  to view, which is the same as [Appendix E](#).

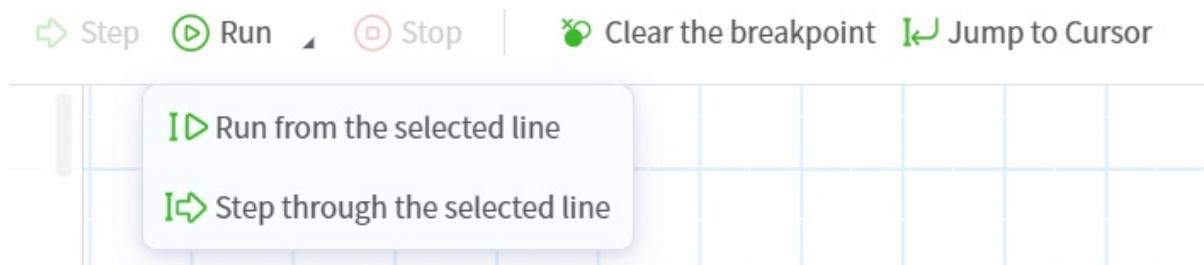
## 6.5 Debugging and running (blockly/script programming)

### ⚠️ NOTICE

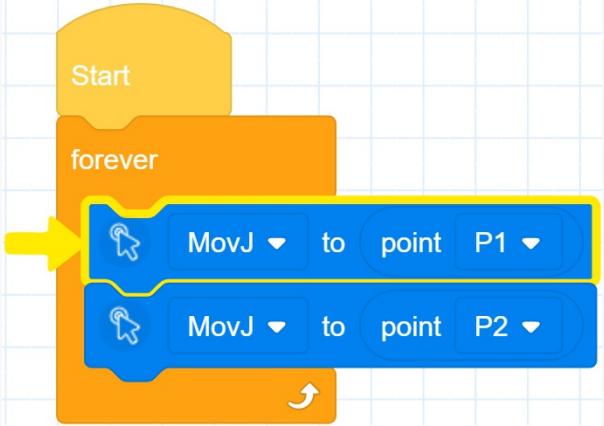
Before starting debugging or running a project, ensure that there are no people or other obstacles within the robot's working space.

The following features related to **Step**, **Run from selected line**, **Step from selected line**, and **Clear breakpoint** are applicable to both blockly programming and script programming.

The following explains the debugging and running icons with blockly programming as an example:



Icon	NOTE
👉 Step	In the paused status, click the👉 Step button to run the current highlighted line to completion. The cursor will switch to the next line or another line (e.g., function entry, return, or goto). Afterward, the script will pause, and the step command will finish.
▷ Run	Click the▷ Run button to start running the program with one click. While running, the <b>Step</b> , <b>Clear breakpoint</b> buttons are disabled, and adding or deleting breakpoints is not allowed.
>ID> Run from selected line	<ul style="list-style-type: none"><li>Start running the program from the selected line.</li><li>If the selected line cannot be jumped to (e.g., an empty line, comment line, or other non-command line), the software will pop up an error message and will not start running the program.</li></ul>
ID> Step from selected line	Start executing a single command from the selected line. After execution, the program will pause.
□ Stop	Click the <b>Stop</b> button to stop running the project.
✖ Clear breakpoint	<ul style="list-style-type: none"><li>Clear all breakpoints in the current project (refer to how to add breakpoints in <a href="#">Blockly programming</a> and <a href="#">Script programming</a>).</li><li>✖ Clear breakpoint is disabled during program execution.</li></ul>

	<ul style="list-style-type: none"> <li>Creating a new project, opening an existing project, or saving as a new project will clear breakpoints in the workspace.</li> </ul>
	<p>During debugging or running, the command being executed is highlighted in the interface, as shown in the figure below. You can click  <b>Jump to Cursor</b> to quickly locate the highlighted line.</p>  <p> <b>Jump to Cursor</b></p>

#### NOTE

- For script programming, only "src0.lua" supports **running from the selected line** and **stepping from the selected line**. The selected line refers to the line where the input cursor is located, which corresponds to the blue-highlighted program line.
- For blockly programming, only the blocks under **Start** support **running from the selected line** and **stepping from the selected line**. The selected line refers to the block that is highlighted (on App, long press for 300ms, on PC, long press for 100ms).

## Program monitoring

You can monitor the running project through [Program variables](#) or by using the **Jump to Cursor** method.

### Running with external signals (blockly/script programming)

You can also run the specified project through external signals ([IO](#) or [Modbus](#)). If you are editing a project, a prompt will appear asking if you want to back up the project. Whether you choose to back up or not, the software will return to the homepage. If the project being edited is the same one triggered by the external signal, the robot will run the last saved project. If you want to run this edited project, stop the externally triggered project, open the backup file, complete your edits and save it, then run it.

When a project is triggered by an external signal, the software will notify you upon opening the application interface or reconnecting the robot. You can then choose to either stop the running project or view and manage its status.

## 6.6 Trajectory recovery

In a paused status, the robot can automatically return to the pause point and continue running the program as originally intended using the trajectory recovery function.

The trajectory recovery function is turned on by default and is not supported to be turned off.

Advanced settings

---

Start/Stop vibration suppression  ON

**i** When it is ON, the robot's vibration suppression effect after starting or stopping will be improved.

Frequency: 8 HZ

---

Torque over-limit warning  ON

**i** When it is ON, torque over-limit warning will pop up in a speech bubble.

---

Jog while paused  ON

**i** When enabled, jogging, dragging, enabling, and disabling operations can be performed even while the robot is paused.

---

Trajectory recovery  ON

If disabled, the robot will move directly from its current position to the command endpoint at script speed, resulting in a trajectory different from the original.

**i** When enabled, if the robot moved while paused, it will first return to the paused position at jog speed and then continue along the original trajectory at script speed.(The current version does not support modification)

If the **Jog while paused** button is enabled, after a pause event is triggered, the robot is allowed to perform operations such as jogging, dragging, switching to manual mode, switching to automatic mode, enabling or disabling, etc.

## ⚠ NOTICE

The robot will enter the paused status after triggering an alarm (including alarms triggered by emergency stops).

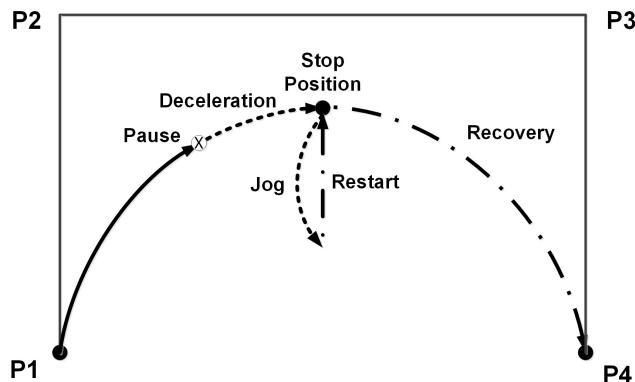
## ℹ NOTE

You can use the main interface **Application** icon to determine whether the current program is in a paused or stopped status:

-  Pausing: The program is currently paused.
-  Application: The program is currently stopped.

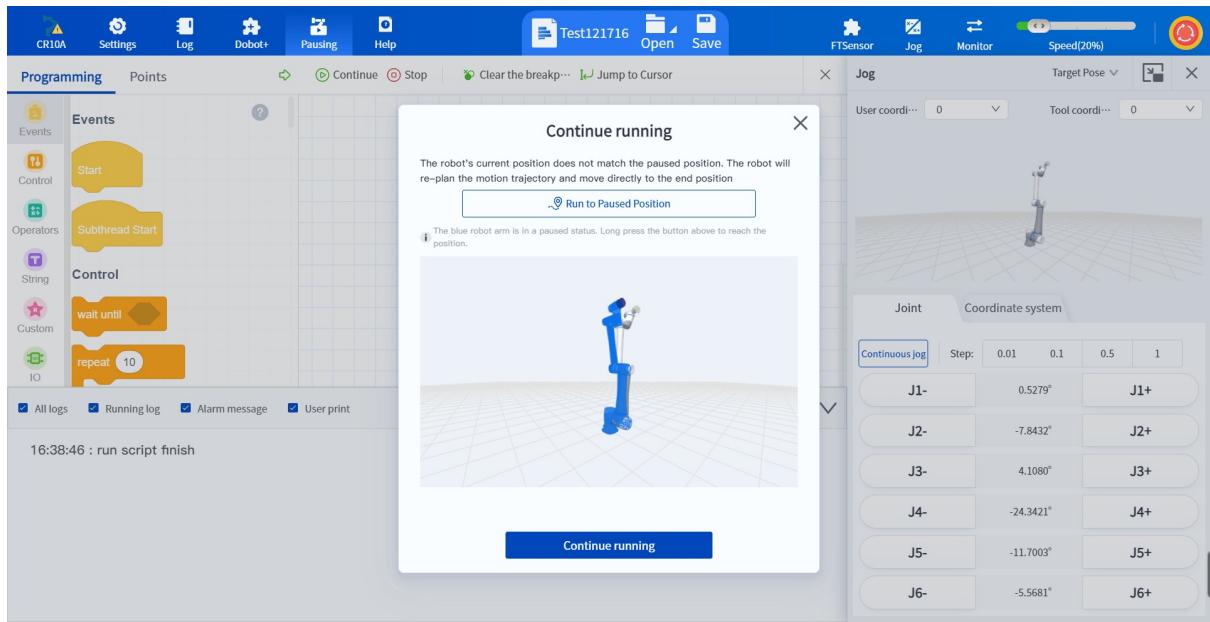
## Trajectory recovery enabled

When the robot is in a paused status, after enabling the trajectory recovery function, the robot will first move slowly to the paused position at jog speed, then resume running the program at the original script speed, ensuring no deviation in the trajectory, as shown below:



After the robot enters a paused status, click the  Continue button in the programming page. The system will automatically check if there is any deviation between the robot's current position and the position where it paused. If there is no deviation between the current position and the paused position, the robot will continue executing the program along the original trajectory.

If there is a deviation between the current position and the paused position, a popup will appear:



Clicking **Continue** will allow the robot to continue its motion based on the effect of track recovery being enabled.

For safety reasons, if the robot's current position deviates significantly from the paused position, there may be risks of self-interference or collision during the movement to the target position. A popup will appear in this case.

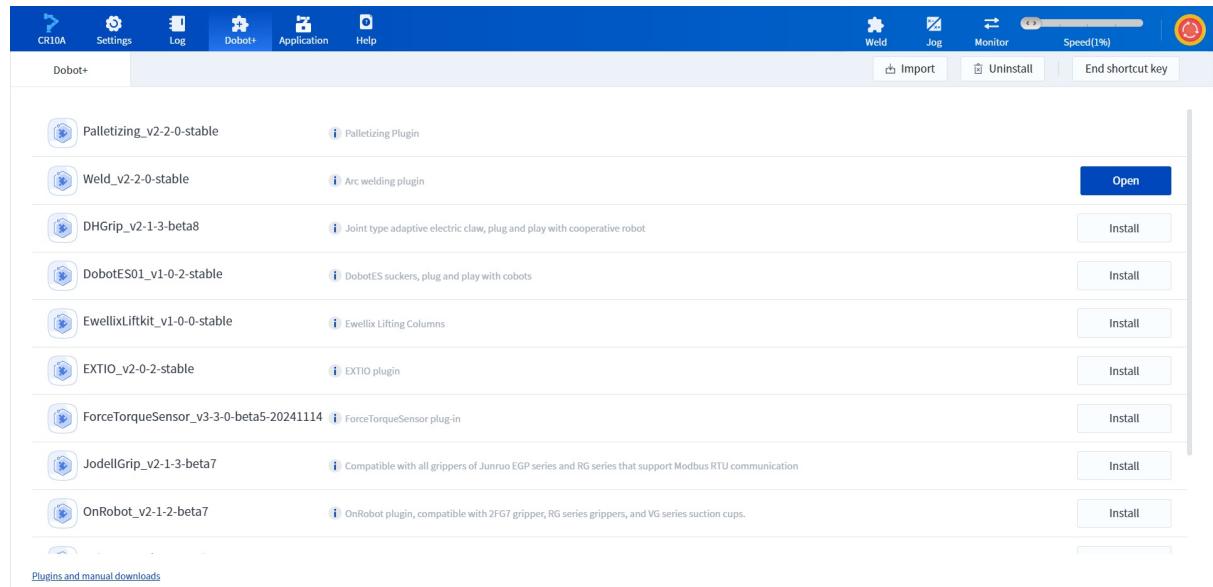
**⚠ The current position is significantly different from the paused position. Self-interference or collisions may occur while moving to the target position. Please move the robot closer to the paused position first.**

**OK, got it**

To avoid risks, long-press the **Run to Paused Position** button to move the robot to the paused position.

## 7 Dobot+

The Dobot+ page helps you quickly configure and use Dobot eco-accessories, eliminating the need for secondary development.

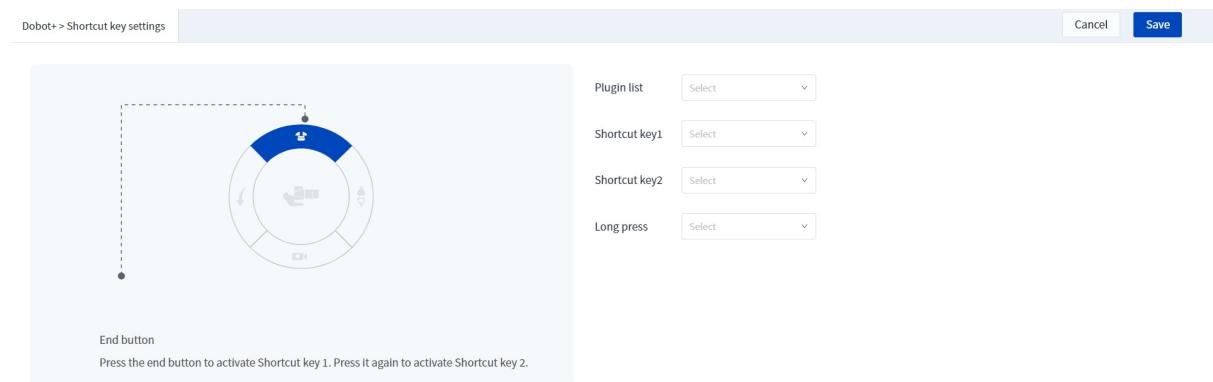


Click the **Install** button on the right of the plugin to install it, or click the **Import** button on the top title bar to upload a plugin. To uninstall a plugin, select it and click the **Uninstall** button on the top title bar.

Once the plugin is successfully installed, the **Install** button changes to **Open**. Clicking it will open the plugin in a new tab, enabling you to open multiple plugins at the same time. Different plugins have varying usage methods, which will not be detailed here.

After adding a plugin, related blocks/commands will also be available in both Blockly programming and Script programming for controlling the eco-accessories during project execution.

Click the **End shortcut key** button in the upper-right corner of the Dobot+ page to assign shortcut functions to the robot's end effector button.



First, select a plugin from the **Plugin list**, then assign it to **Shortcut key 1** or **Shortcut key 2**, or **Long press** to execute the corresponding plugin function. Press the end effector button once to execute the function assigned to **Shortcut key 1**, and press it again to execute the function for **Shortcut key 2**.

Click the **Plugins and manual downloads** option in the lower-left corner of the Dobot+ page. A prompt box will appear, allowing you to view and download the supported plugin list and their user manuals via the web or QR code.



# 8 Monitor

- [8.1 Controller DI/DO](#)
- [8.2 Controller AI/AO](#)
- [8.3 Tool I/O](#)
- [8.4 Safety I/O](#)
- [8.5 Modbus](#)
- [8.6 Global variables](#)
- [8.7 Program variables](#)

# 8.1 Controller DI/DO

## 8.1.1 DI/DO monitoring

This page is used to monitor and configure the status and functions of controller's DI (Digital Input) and DO (Digital Output).

The screenshot shows a software interface for monitoring and configuring Controller DI/DO. The top navigation bar includes icons for Weld, Jog, Monitor (which is highlighted with a dashed blue border), Speed(1%), and a refresh symbol. Below the header, there are tabs for I/O, Controller DI/DO, Settings, and a close button. The main area displays a table with columns for Controller DI/DO, DI, Alias, Status, DO, Alias, and Status. The table rows are categorized by source: Controller AI/AO, Tool I/O, Safety I/O, Modbus, Variable, Global variable, and Program variables. Each row contains a status indicator (green circle with a dot) and a toggle switch for the DO status.

Controller DI/DO						
I/O	DI	Alias	Status	DO	Alias	Status
Controller DI/DO						
Controller AI/AO	DI_1	Start	●	DO_1	↙	<input checked="" type="checkbox"/> OFF
Tool I/O	DI_2	Stop	●	DO_2	↙	<input checked="" type="checkbox"/> OFF
Safety I/O	DI_3	Pause	●	DO_3	↙	<input checked="" type="checkbox"/> OFF
Modbus	DI_4	Enable	●	DO_4	↙	<input checked="" type="checkbox"/> OFF
Variable						
Global variable	DI_5	Disable	●	DO_5	↙	<input checked="" type="checkbox"/> OFF
Program variables	DI_6	Clear alarm	●	DO_6	↙	<input checked="" type="checkbox"/> OFF
	DI_7	Enter drag mode	●	DO_7	↙	<input checked="" type="checkbox"/> OFF
	DI_8	Exit drag mode	●	DO_8	↙	<input checked="" type="checkbox"/> OFF
	DI_9	↙	●	DO_9	↙	<input checked="" type="checkbox"/> OFF
	DI_10	↙	●	DO_10	↙	<input checked="" type="checkbox"/> OFF
	DI_11	↙	●	DO_11	↙	<input checked="" type="checkbox"/> OFF
	DI_12	↙	●	DO_12	↙	<input checked="" type="checkbox"/> OFF
	DI_13	↙	●	DO_13	↙	<input checked="" type="checkbox"/> OFF

The central area of the page allows you to view and set aliases and statuses for the DI/DO.

The controller DI/DO is set as **Universal IO** by default, which can be configured as **System IO**.

- The functions of **Universal IO** are user-defined. The **Alias** is initially blank and can be modified by

clicking . You can add aliases to describe the IO function for easier identification during programming. After modification, the alias will appear as underlined black text, and you can click the text to modify it again.

- The **System IO** is the DI/DO with specific functions configured in IO settings page. The **Alias** will appear as non-underlined blue text and cannot be modified.

The circular indicator on the right of each DI refers to its current status (gray: OFF, green: ON).

The switch on the side of each DO refers to its current status. You can click it to switch between ON and OFF to control the DO status. The status of **System IO** cannot be manually switched.

Click **Settings** at the upper right of the page to access the settings page, where you can configure system IO, IO trigger modes, types, and more.

## 8.1.2 I/O settings

I/O		Controller DI/DO > Settings				Cancel	Save
	Controller DI/DO	IO settings		Select project	Advanced settings		
Controller AI/AO	DI	Feature configuration		Switch to virtual DI		DO	Feature configuration
Tool I/O	DI_1	Start	▼	<input checked="" type="checkbox"/>		DO_1	Universal DO
Safety I/O	DI_2	Stop	▼	<input checked="" type="checkbox"/>		DO_2	Universal DO
Modbus	DI_3	Pause	▼	<input checked="" type="checkbox"/>		DO_3	Universal DO
Variable	DI_4	Enable	▼	<input checked="" type="checkbox"/>		DO_4	Universal DO
Global variable	DI_5	Disable	▼	<input checked="" type="checkbox"/>		DO_5	Universal DO
Program variables	DI_6	Clear alarm	▼	<input checked="" type="checkbox"/>		DO_6	Universal DO
	DI_7	Enter drag mode	▼	<input checked="" type="checkbox"/>		DO_7	Universal DO
	DI_8	Exit drag mode	▼	<input checked="" type="checkbox"/>		DO_8	Universal DO
	DI_9	Universal DI	▼	<input checked="" type="checkbox"/>		DO_9	Universal DO
	DI_10	Universal DI	▼	<input checked="" type="checkbox"/>		DO_10	Universal DO
	DI_11	Universal DI	▼	<input checked="" type="checkbox"/>		DO_11	Universal DO

### Virtual DI

Click  on the right of the DI, and it will change to . After saving, the corresponding DI will be switched to a virtual DI. When DI is set to virtual DI, the indicator on the monitoring page changes to a switch. By clicking it, you can toggle the virtual DI between ON and OFF, allowing you to simulate external DI device inputs and debug DI-related functions, such as satisfying DI-related conditions during project execution to ensure it continues running.

### NOTE

The virtual DI remains effective after being set. During project execution, the value read by commands for the corresponding DI will be the virtual value instead of the actual one. If you want to avoid this, switch virtual DI back to real DI before running the project.

## System IO

You can configure the remote control function for DI/DO through the dropdown menu in the function configuration column. The specific functions are explained as follows:

DI function	NOTE
Start	When the robot is idle, start the specified project. See <a href="#">Select project</a> below for details. When the robot is paused, continue running the current project or other command queue.
Stop	Stop running the project (or other forms of command queue).
Pause	Pause running the project (or other forms of command queue).
Enable	Enable the robot when powered on.
Disable	Disable the robot when in enabled status.
Clear alarm	Clear the current alarms.
Enter drag mode	Control the robot to enter the drag mode when in enabled status.
Exit drag mode	Control the robot to exit the drag mode when in drag mode.

DO function	NOTE
Running status	Output 1 when the robot is executing a project, TCP command queue, or trajectory playback, and 0 otherwise. This indicates whether the robot is running a program, unrelated to the robot's motion status.
Stop status	Output 1 when the robot is stopped, and 0 otherwise.
Pause status	Output 1 when the robot is in pause status, and 0 otherwise.
Safe home status	Output 1 when the robot is in <a href="#">safe home position</a> , and 0 otherwise.
SafeSkin pause status	Output 1 when the robot is in pause status triggered by SafeSkin, and 0 otherwise.
Idle status	Output 1 when the robot is in idle status (enabled, stopped and no alarm), and 0 otherwise. This means the robot can accept and execute commands at any time.
Power-on status	Output 1 when the robot is powered on, and 0 otherwise.
Enabling status	Output 1 when the robot is enabled, and 0 otherwise.

Alarm status	Output 1 when alarm exists, and 0 otherwise.
Collision status	Output 1 when a collision is detected, and 0 otherwise.
Drag status	Output 1 when the robot is in drag mode, and 0 otherwise.
Low when not running	Output 0 when the project, TCP command queue or trajectory playback is not running, paused or stopped, and the corresponding IO status can be set through commands during running.
High when not running	Output 1 when the project, TCP command queue or trajectory playback is not running, paused or stopped, and the corresponding IO status can be set through commands during running.
Low when abnormal stop	<p>Output 0 when the robot is abnormally stopped, and the corresponding IO status can be set through commands during running.</p> <p>The following situations may cause abnormal stop:</p> <ul style="list-style-type: none"> <li>• Safety function stop (e.g., collision detection, safety wall and safety zone, safety IO).</li> <li>• Robot alarm.</li> <li>• Project execution error.</li> </ul>

The statuses **Low when not running**, **High when not running**, and **Low when abnormal stop** differ from other statuses. These statuses will only output when the condition is met, and during project or TCP mode execution, the status can be freely modified through commands, enabling users to control external devices and coordinate with the robot using a single DO, simplifying logic.

#### NOTE

- **Not running** means the program has not started. When not running, the IO status is locked and cannot be changed.
- **Abnormal stop** refers to situations where the program stops due to syntax errors, collisions, emergency stops, or other exceptions. In the event of an abnormal stop, the IO status will not be locked.

#### **Example:**

To explain how to use this type of status, take **Low when not running** as an example. In a gluing application, you can control a gluing machine through DO1 in the project, where DO1 = 1 indicates the machine is working, and DO1 = 0 indicates the machine is stopped. Setting DO1 to **Low when not running** will automatically stop the gluing machine when the project is not running, paused, or stopped. This setting will not affect command-based control of the machine during project execution.

The DI with **Backup project** function cannot be configured on this page. It must first be released on [Select project](#) page.

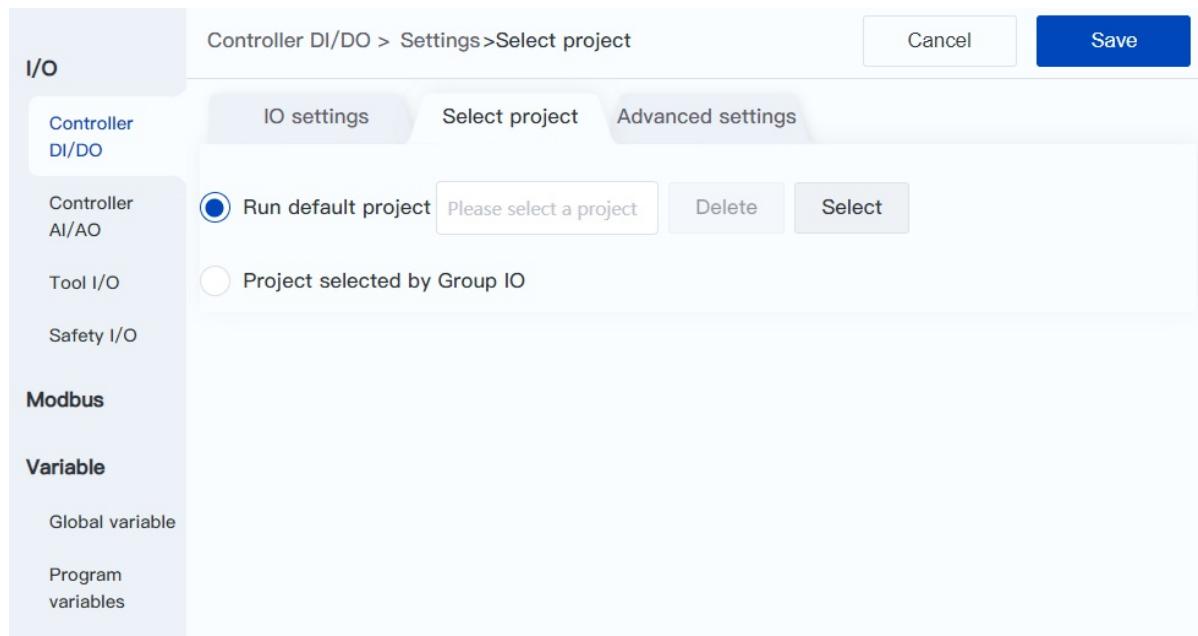
After modifying the configuration, click **Save** to apply the changes.

## ⚠️NOTICE

- All remote trigger sources take effect at the same time. For the safety of the device and production, please ensure that the robot is activated by only one control source (software/DI/Modbus).
- Whether the IO trigger takes effect is also affected by Manual/Automatic mode and IO/Modbus configuration. See the [corresponding description](#) for details.
- DO NOT send control signals before the robot is fully powered on and initialized, as this may lead to abnormal robot behavior.

## Select project

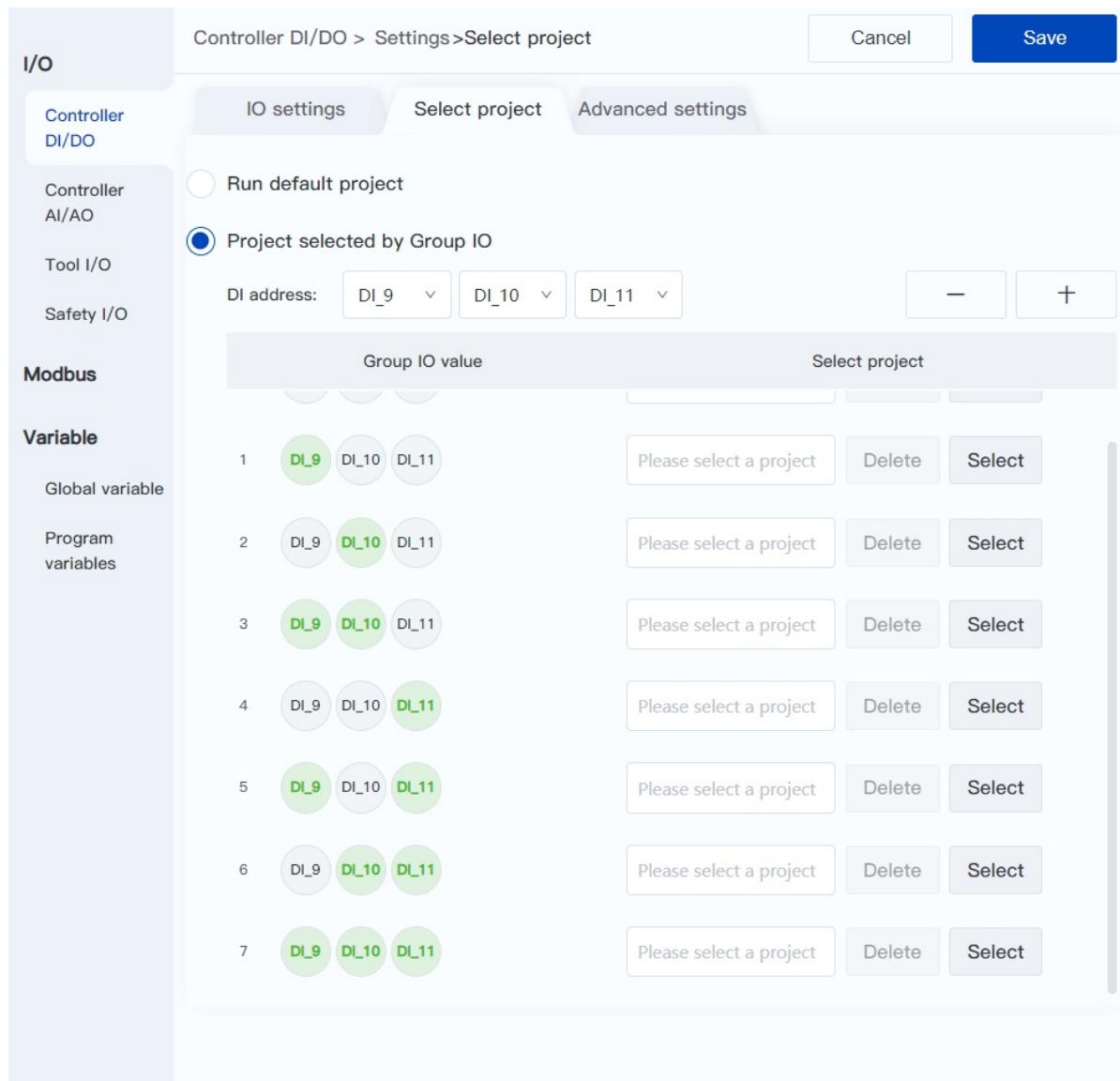
When **Run default project** is selected, the project selected here will run directly when the DI set to **Start** is triggered.



Click **Select**, the “Select project” page will pop up.

Click **Delete**, the currently selected project will be cleared.

When **Project selected by Group IO** is selected, you can configure multiple projects through group I/O.



1. Click + or - to increase or decrease the number of addresses assigned to the group I/O. The more addresses assigned (up to 4), the more projects can be configured.
  - 1 addresses: 2 projects can be configured
  - 2 addresses: 4 projects can be configured
  - 3 addresses: 8 projects can be configured
  - 4 addresses: 16 projects can be configured
2. You can modify the assigned address through the drop-down box. The addresses assigned to group I/O cannot overlap with the address of remote I/O or safety I/O (CCBOX).
3. After assigning the address, you can set the project for each group IO value. (Just set it as needed, or leave it blank).

Before running a project, the corresponding group I/O value must be set (green: ON, gray: OFF) to select the desired project. If no project is selected for the corresponding group I/O, the robot will not run when the **Start** DI is triggered, and the controller will generate an alarm.

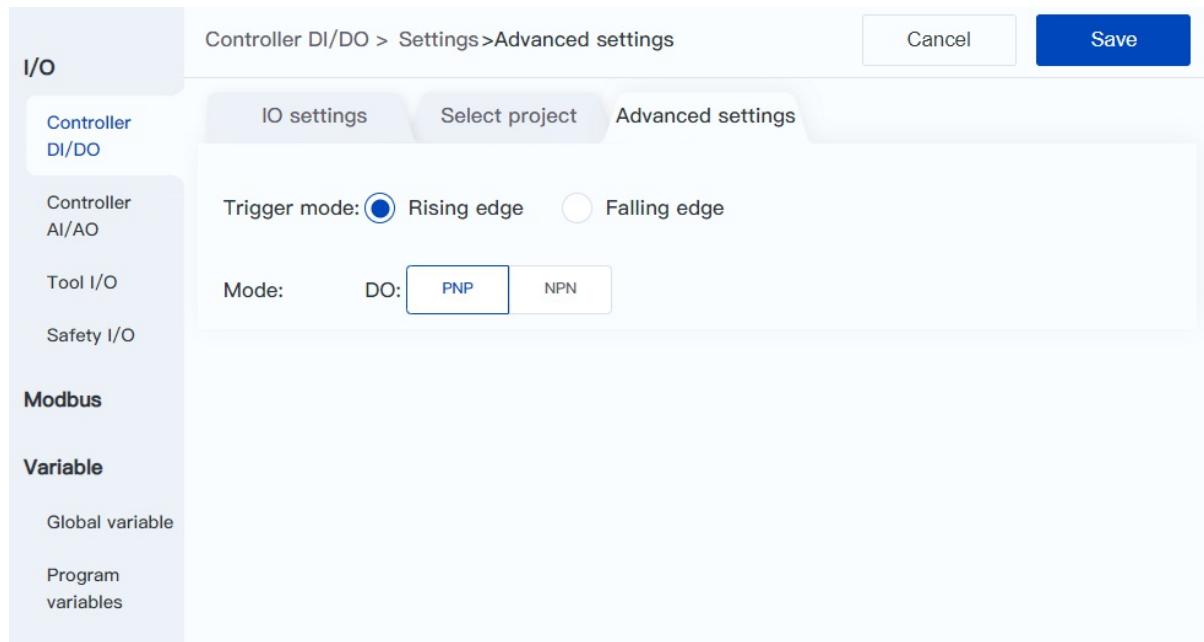
### Example:

Taking the assignment of four addresses DI1 – DI4 in the figure above as an example:

- If DI1 – DI4 are OFF, the 1st project is selected.
- If DI1 and DI2 are ON, DI3 and DI4 are OFF, the 3rd project is selected.
- If DI1 – DI4 are all ON, the 16th project is selected.

After modifying the configuration, click **Save**.

### Advanced settings



### Trigger mode

The trigger mode is used to set how the DI function is triggered. The rising edge indicates that the configured function is triggered when DI changes from OFF to ON, while the falling edge indicates that the configured function is triggered when DI changes from ON to OFF.

### Mode selection

For **CRA** series, you need to select the DO mode according to the actual hardware wiring. See the CRA series hardware guide for details.

After modifying the configuration, click **Save**.

#### ⚠ NOTICE

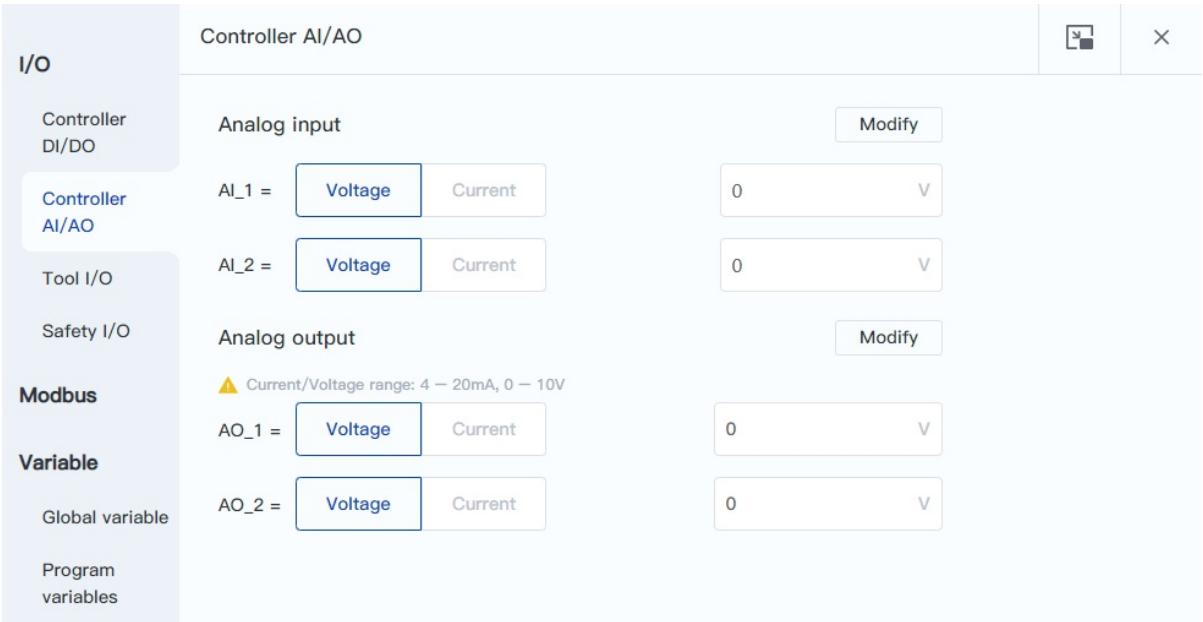
The DO mode can only be configured successfully if both the DobotStudio Pro software parameters and the hardware wiring for the CRA series are set up correctly.

## 8.2 Controller AI/AO

This page is used to monitor and configure the status and mode of controller's AI (Analog Input) and AO (Digital Output).

### NOTE

This page is not available when connected to Magician E6 robot.



Controller AI/AO		
Controller DI/DO	Analog input	
	AI_1 = <input type="button" value="Voltage"/> Current	<input type="text" value="0"/> V
Controller AI/AO	AI_2 = <input type="button" value="Voltage"/> Current	<input type="text" value="0"/> V
	Safety I/O	
Modbus	Analog output	<input type="button" value="Modify"/>
	⚠ Current/Voltage range: 4 – 20mA, 0 – 10V	
Variable	AO_1 = <input type="button" value="Voltage"/> Current	<input type="text" value="0"/> V
	AO_2 = <input type="button" value="Voltage"/> Current	<input type="text" value="0"/> V
Global variable		
Program variables		

The analog input/output is used to display the actual values of the controller's AI interfaces and supports both **Voltage** and **Current** detection modes. You can click **Modify** to switch the mode. The AO values can also be manually adjusted, but you need to click **Confirm modification** after making changes for them to take effect.

## 8.3 Tool I/O

This page is used to monitor the status and mode of the robot's Tool I/O.

### **i** NOTE

The figure below takes CR A series as an example. For models with more than one aviation connectors (such as the CR20A), multiple tabs will be displayed.

Magician E6 has only 2 DI and 2 DOs on its end, without RS485 and AI interface.

**I/O**

Controller DI/DO  
Controller AI/AO  
**Tool I/O**  
Safety I/O

**Modbus**

**Variable**

Global variable  
Program variables

**Tool I/O**

Tool cable      Tool interface pins

1: AI\_1      2: AI\_2  
3: DI\_2      4: DI\_1  
5: 24V      6: DO\_2  
7: DO\_1      8: GND

DI	Alias	Status	DO	Alias	Status
DI_1	<input type="text"/>		DO_1	<input type="text"/>	
DI_2	<input type="text"/>		DO_2	<input type="text"/>	

Analog Input · Communication Interface

RS485       Analog input

AI\_1 =  Voltage  Current 0.15 mA

AI\_2 =  Voltage  Current 0.01 V

### DI/DO

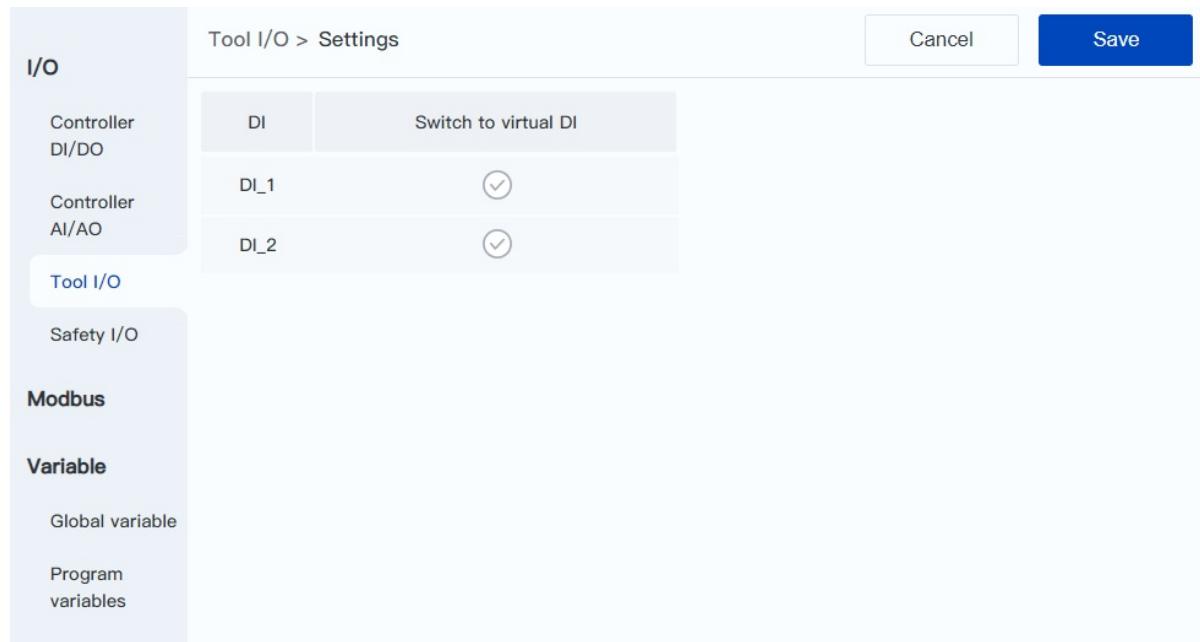
Click in the Alias column or the text to modify the alias. You can add aliases to describe the IO function for easier identification during programming.

The circular indicator on the right of each DI refers to its current status (gray: OFF, green: ON).

The switch on the side of each DO refers to its current status. You can click it to switch between ON and OFF to control the DO status.

## Virtual DI

Click **Settings** at the upper right of the page to configure the virtual DI. When DI is set to virtual DI, the indicator on the monitoring page changes to a switch. By clicking it, you can toggle the virtual DI between ON and OFF, allowing you to simulate external DI device inputs and debug DI-related functions, such as satisfying DI-related conditions during project execution to ensure it continues running.



### **i** NOTE

The virtual DI remains effective after being set. During project execution, the value read by commands for the corresponding DI will be the virtual value instead of the actual one. If you want to avoid this, switch virtual DI back to real DI before running the project.

## End working mode

- When the working mode is set to **RS485**, tool I/O pins 1 and 2 function as 485A and 485B.
- When the working mode is set to **Analog input**, tool I/O pins 1 and 2 function as AI\_1 and AI\_2, and their input values can be monitored in real-time on this page.

### **i** NOTE

When using an RS485 end-effector tool, the working mode must be set to **RS485**.

## 8.4 Safety I/O

This page is used to view the status of Safety I/O and configure its functions.

I/O	Safety I/O						Settings		
	SI	Feature configuration	Status	SO	Feature configuration	Status			
Controller DI/DO									
Controller AI/AO	SI_1 , SI_2	User E-Stop	 	SO_1 , SO_2	E-Stop status output				
Tool I/O	SI_3 , SI_4	Protective stop	 	SO_3 , SO_4	-				
Safety I/O	SI_5 , SI_6	-	 	SO_5 , SO_6	-				
Modbus	SI_7 , SI_8	-	 	SO_7 , SO_8	-				
Variable	SI_9 , SI_10	-	 	SO_9 , SO_10	-				
Global variable									
Program variables									

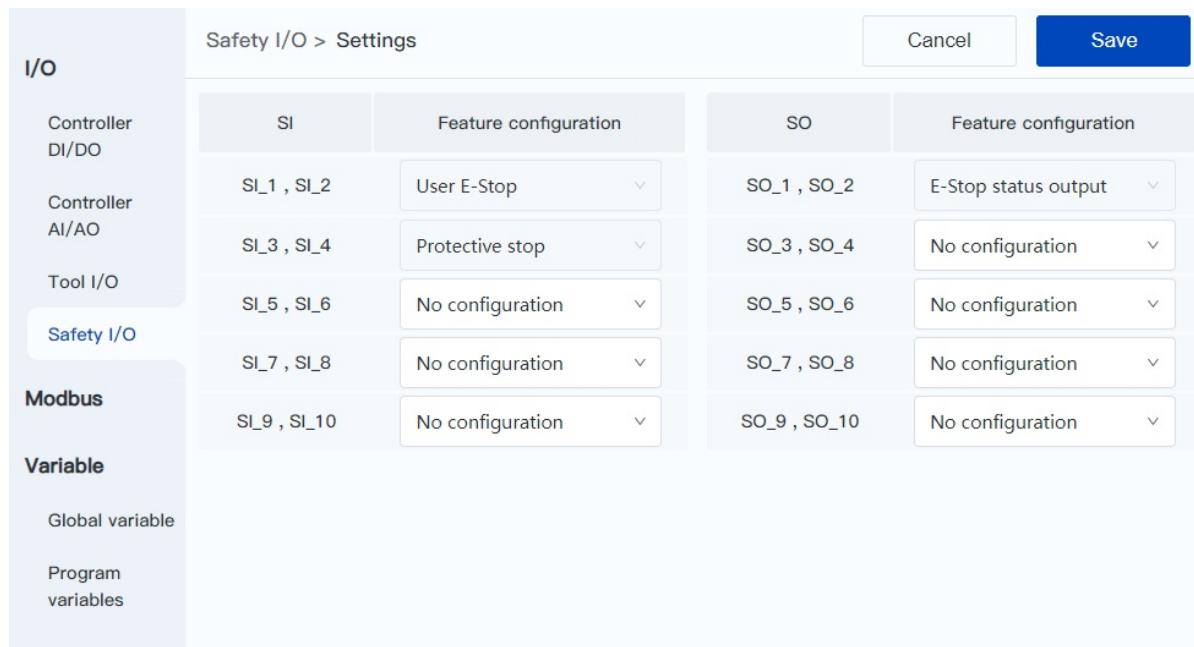
The indicators in the "Status" column refer to the Safety I/O status (green: high level, gray: low level).

The Safety I/O is configured with dual circuits. The two indicators for dual SI indicate the status of each channel, and their trigger logics vary depending on the configured functions. As the status of dual SO are synchronised, only one indicator is used to indicate the status.

### NOTE

The Safety I/O of the Magician E6 shares terminals with the [Controller DI/DO](#). You can configure it as single or dual based on your needs. Terminals configured as System I/O can no longer be configured as Safety I/O, and vice versa.

Click **Settings** to modify the Safety I/O's function configuration, and click **Save** to apply the changes.



## Configurable SI functions

Function	NOTE
User E-Stop	<p>The User E-Stop (User Emergency Stop) input is an emergency stop interface provided for users to connect external emergency stop devices. <b>It defaults to a high-level, normally closed signal input. Any low-level signal triggers the robot to enter an emergency stop status.</b> This function typically triggers the emergency stop output, which may cause self-locking in certain applications. To prevent this, you can configure the SI interface to <b>User E-Stop input (no status output)</b> in the software, and use the corresponding interface for this purpose. SI_1 and SI_2 are fixed for this function.</p>
Protective stop	<p>The Protective stop input is used for connecting external protective devices (e.g., safety gate, safety light curtains). <b>It defaults to a high-level, normally closed signal input. Any low-level signal will trigger the robot to enter a protective stop (pause status).</b></p> <ul style="list-style-type: none"> <li>If a <b>Protective stop reset</b> input is configured, the protective stop can only be released by simultaneously restoring the protective stop input signal and triggering the reset input. Once you confirm in the software to continue running, the robot will resume operation.</li> <li>If no <b>Protective stop reset</b> input is configured, restoring the protective stop input signal alone will release the protective stop status and resume the robot's operation.</li> </ul> <p>SI_3 and SI_4 are fixed for this function.</p>
Protective stop reset	<p>The Protective stop reset input is used to reset the protective stop status. <b>It defaults to a high-level, normally open signal input. The rising edge of both circuits triggers the reset of the protective stop status.</b></p>
Reduced mode	<p>The reduced mode input is used to control the robot's entry into reduced mode. It defaults to a high-level normally closed signal input, and any low-level input will trigger the robot to enter reduced mode. Once the input returns to high level, the robot will exit reduced mode and return to normal mode. When the robot enters reduced mode, the CRA series will switch its</p>

**motion speed and safety limits** to the preset values for reduced mode. For Magician E6, the global speed will be limited to 10%, and this cannot be changed to avoid safety risks associated with high-speed operation.

## ⚠️ NOTICE

Please ensure the SI signal changes at an interval of at least 150ms during operation to avoid abnormal robot behavior due to rapid signal fluctuations.

For example, if the Protective Stop Reset input is not configured, signal fluctuations (with change intervals of less than 150ms) on the protective stop input may cause the robot to pause without automatically resuming. To resolve this, you can either:

- Pause the project through the software, and then resume it.
- Trigger the protective stop input signal again, ensuring the interval is over 150ms.

## Configurable SO functions

Function	NOTE
E-Stop status output	When the robot is <b>in emergency stop status, the output voltage is low-level</b> ; otherwise, the output voltage is high-level. Any source that triggers an emergency stop will activate this output. SO_1 and SO_2 are fixed for this function.
Non-stop status output	When the robot is <b>in automatic running (non-stop status), the output voltage is low-level</b> ; otherwise, the output voltage is high-level. This status indicates whether the robot is running a program, not whether the joints are moving. For example, while running a project, if the robot is waiting for a specified DI to turn ON and not moving, the robot is in non-stop status and outputs low-level voltage. If the project is paused, the robot is in stop status and outputs high-level voltage.
Reduced mode output	When the robot is <b>in reduced mode, the output voltage is low-level</b> ; otherwise, the output voltage is high-level.
Running status output	If <b>one or more of the robot's joints move at more than 1°/s (except in drag mode)</b> , it is considered to be in <b>running status</b> , and the output voltage is low-level; otherwise, the output voltage is high-level.
Safety home status output	When the robot is <b>in safe home position, the output voltage is high-level</b> ; otherwise, the output voltage is low-level. The safe home position can be modified in the safety settings.
Protective stop status output	When the robot is <b>in protective stop status, the output voltage is low-level</b> ; otherwise, the output voltage is high-level.
System E-Stop status output	When the robot is <b>in system emergency stop status, the output voltage is low-level</b> ; otherwise, the output voltage is high-level. The system emergency stop is triggered by the emergency stop button or software emergency stop.
User E-Stop status output	When the robot is <b>in user emergency stop status, the output voltage is low-level</b> ; otherwise, the output voltage is high-level. The user emergency stop is triggered by the safety I/O.

## 8.5 Modbus

### 8.5.1 Modbus monitoring

This page is used for connecting the control software as a Modbus master to the controller's built-in Modbus slave to view and modify register values, and to configure remote Modbus control functions when the controller serves as a slave.

I/O	Modbus					Settings	Connect		X
Controller DI/DO		Alias	00000		Alias		00010		
Controller AI/AO	0								
Tool I/O	1								
Safety I/O	2								
Modbus	3								
Variable	4								
Global variable	5								
Program variables	6								
	7								
	8								
	9								

Click **Connect** at the upper right of the page and set the parameters for the slave connection.

- **Slave IP:** The address of the Modbus device. When connecting to the controller's built-in Modbus slave, enter the controller's IP address, such as 192.168.200.1.
- **Port:** The Modbus communication port. Enter 502 when connecting to the controller's built-in Modbus slave.
- **Slave ID:** The ID of the slave device.
- **Function code:** Select the function type of the slave device.
- **Address/Quantity:** Enter the register address and quantity. Refer to [Appendix A Modbus Register Definition](#) for connecting to the controller's built-in Modbus slave.
- **Scan cycle:** The time interval at which the robot scans the slave.

X

## Connect

**Connection settings:**

Slave IP:	192.168.5.1
Port:	502

**Function code definition:**

Slave ID:	1
Function code:	01: Coil register
Address:	0
Quantity:	10
Scan cycle:	1000 ms

**Connect**

After successful connection, the table in the middle of the page shows the alias and value of each address of the slave. You can double-click (on PC) or click (on App) the alias cell to modify the alias. If the register type is **coil register** or **holding register**, you can double-click (on PC) or click (on App) the value cell to modify the register value.

Click **Settings** at the upper right of the page to configure the coil register's trigger mode, view the address configuration for remote control functions, and set the project for remote start.

## 8.5.2 Modbus settings

The screenshot shows the 'Modbus > Settings' screen. On the left, a sidebar lists I/O categories: Controller DI/DO, Controller AI/AO, Tool I/O, Safety I/O, and Modbus. Under Modbus, there are sections for Variable (Global variable, Program variables) and Modbus settings. The Modbus settings section includes tabs for 'Modbus settings' (selected) and 'Select project'. It features a 'Trigger mode' switch between 'Rising edge' (selected) and 'Falling edge'. Below this, two columns of register configurations are shown:

	Coil register address configuration	Contact register address configuration	
Start	0	Running status	0
Stop	1	Stop status	1
Pause	2	Pause status	2
Enable	3	Safe home status	3
Disable	4	SafeSkin pause status	4
Clear alarm	5	Idle status	5
Enter drag mode	6	Power-on status	6
Exit drag mode	7	Enabling status	7
		Alarm status	8
		Collision status	9
		Drag status	10
		Recovery mode status	11

Buttons for 'Cancel' and 'Save' are at the top right.

**Trigger mode** is used to set how the function of the coil register is triggered. The rising edge indicates that the configured function is triggered when the coil register changes from 0 to 1, and the falling edge indicates that the configured function is triggered when the coil register changes from 1 to 0.

The address of **coil register** and **contact register** can only be viewed and cannot be modified, and the corresponding functions are described below.

Coil register function	NOTE
Start	When the robot is idle, it starts running the specified project. See <a href="#">Select project</a> below for details. When the robot is paused, it continues to run the project (or other forms of command queue).
Stop	Stop running the project (or other forms of command queue).
Pause	Pause running the project (or other forms of command queue).
Enable	Enable the robot when powered on.
Disable	Disable the robot when in enabled status.

Clear alarm	Clear the current alarms.
Enter drag mode	Control the robot to enter the drag mode when in enabled status.
Exit drag mode	Control the robot to exit the drag mode when in drag mode.

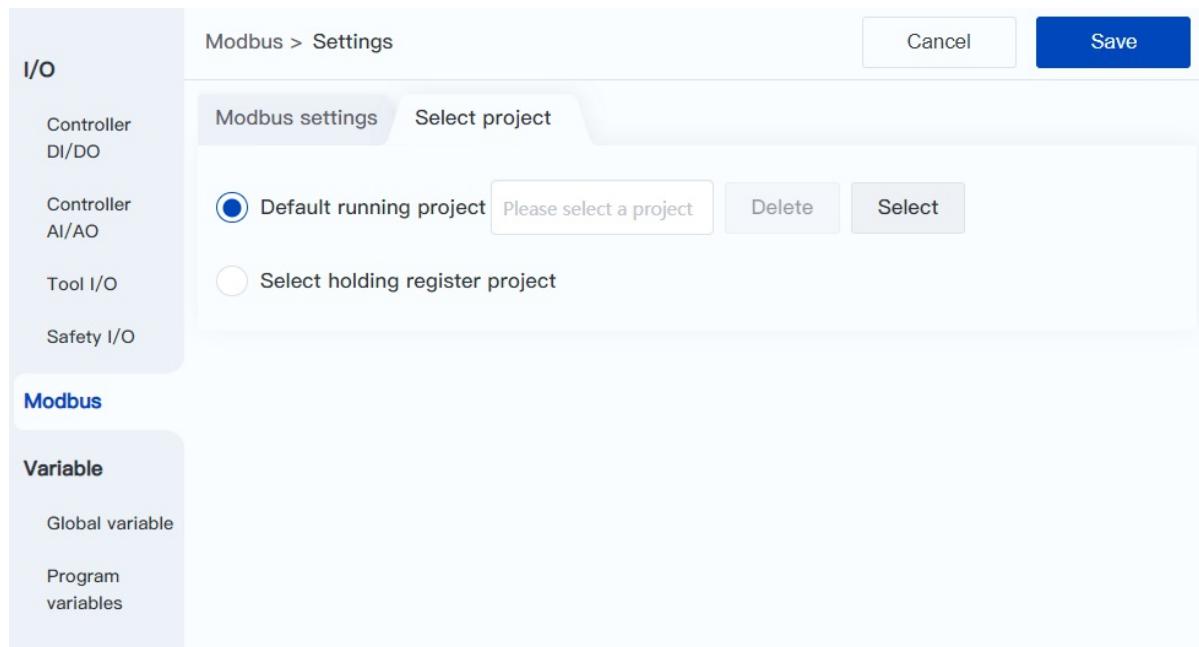
Contact register function	NOTE
Running status	Output 1 when the robot is executing a project, TCP command queue, or trajectory playback, and 0 otherwise. This indicates whether the robot is running a program, unrelated to the robot's motion status.
Stop status	Output 1 when the robot is stopped, and 0 otherwise.
Pause status	Output 1 when the robot is in pause status, and 0 otherwise.
Safe home status	Output 1 when in <a href="#">safety home position</a> , and 0 otherwise.
SafeSkin pause status	Output 1 when the robot is in pause status triggered by SafeSkin, and 0 otherwise.
Idle status	Output 1 when the robot is in idle status (enabled, stopped and no alarm), and 0 otherwise. This means the robot can accept and execute commands at any time.
Power-on status	Output 1 when the robot starts to power on (not indicate that the power on is completed), and 0 otherwise.
Enabling status	Output 1 when the robot is enabled, and 0 otherwise.
Alarm status	Output 1 when alarm exists, and 0 otherwise.
Collision status	Output 1 when a collision is detected, and 0 otherwise.
Drag status	Output 1 when the robot is in drag mode, and 0 otherwise.
Recovery mode status	Output 1 when the robot is in <a href="#">recovery mode</a> , and 0 otherwise.

After modifying the configuration, click **Save**.

### NOTICE

- All remote trigger sources take effect at the same time. For the safety of the device and production, please ensure that the robot is activated by only one control source (software/DI/Modbus).
- Modbus remote control signals may be delayed affected by the network. Please judge whether the delay is acceptable according to actual condition.
- Whether the Modbus trigger takes effect is also affected by Manual/Automatic mode and IO/Modbus configuration. See the [corresponding description](#) for details.
- DO NOT send control signals before the robot is fully powered on and initialized, as this may lead to abnormal robot behavior.

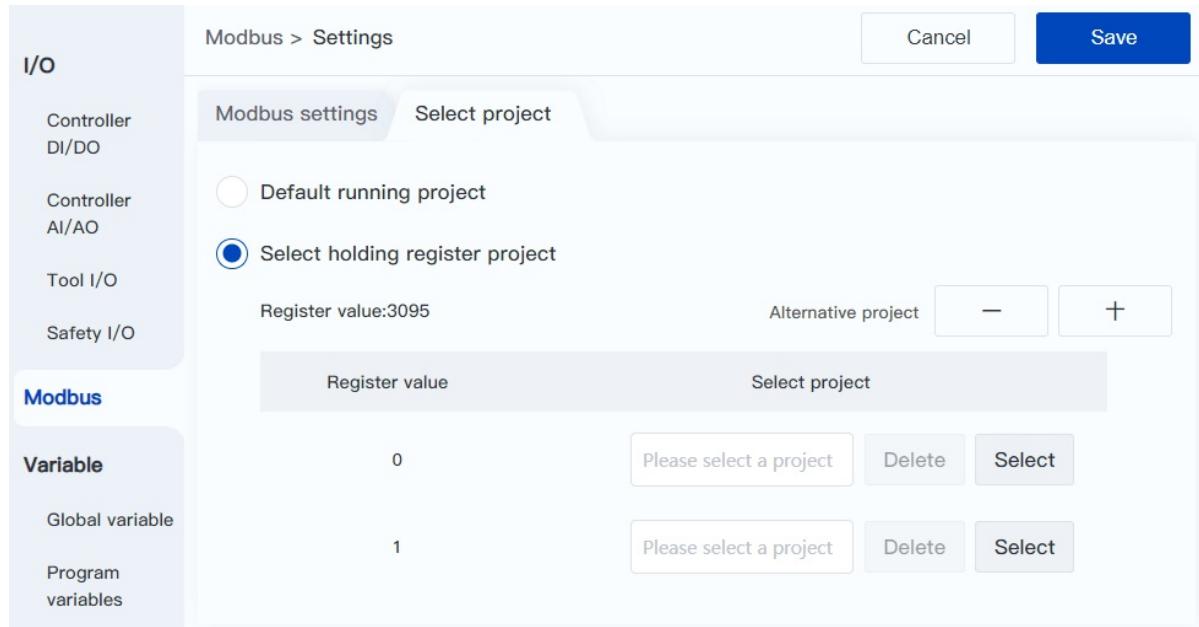
When you select **Default running project**, the project selected here will run directly if the **Start** coil register is triggered.



Click **Select**, the “Select project” page will pop up.

Click **Delete**, the currently selected project will be cleared.

Select **Select holding register project**, you can configure multiple projects.



Click + or - to increase or decrease the projects to be configured (up to 256). When the **Start** coil register is triggered, the project to be started will be determined based on the value of the holding register at the specified address (3095).

After modifying the configuration, click **Save**.

## 8.6 Global variables

This page is used to set the global variable. Global variables are persistently saved in the controller (even after power-off) and can be accessed by multiple projects.

After setting a global variable, it can be called using the related blocks in **Blockly programming** or directly called by the variable name in **Script programming**.

The screenshot shows the 'Global variable' management screen. On the left, there's a sidebar with sections for I/O (Controller DI/DO, Controller AI/AO, Tool I/O, Safety I/O), Modbus, and Variable (Global variable, Program variables). The 'Global variable' section is currently selected. The main area has a header 'Global variable' with a close button 'X'. Below the header are three buttons: 'Delete' (trash icon), 'Modify' (pencil icon), and '+ New' (plus icon). A table lists existing variables: NO 1, Variable Name var\_1, Type number, Global Hold checked (indicated by a blue checkmark), Range (empty), and Value 50. The table has columns for NO, Variable Name, Type, Global Hold, Range, and Value.

Click **+ New** to create a new global variable. Select a variable and click **Modify** to change its attributes, or click **Delete** to remove it.

The dialog box is titled 'Add Variable'. It contains fields for 'Variable Name' (set to 'var\_2'), 'Variable Type' (set to 'number'), and 'Value' (empty). Below these are 'Value Range' fields for 'Min' and 'Max' with a checkbox. A note below says 'After this parameter is selected, changes to this variable in Hold the project take effect globally'. At the bottom is a large blue 'New' button.

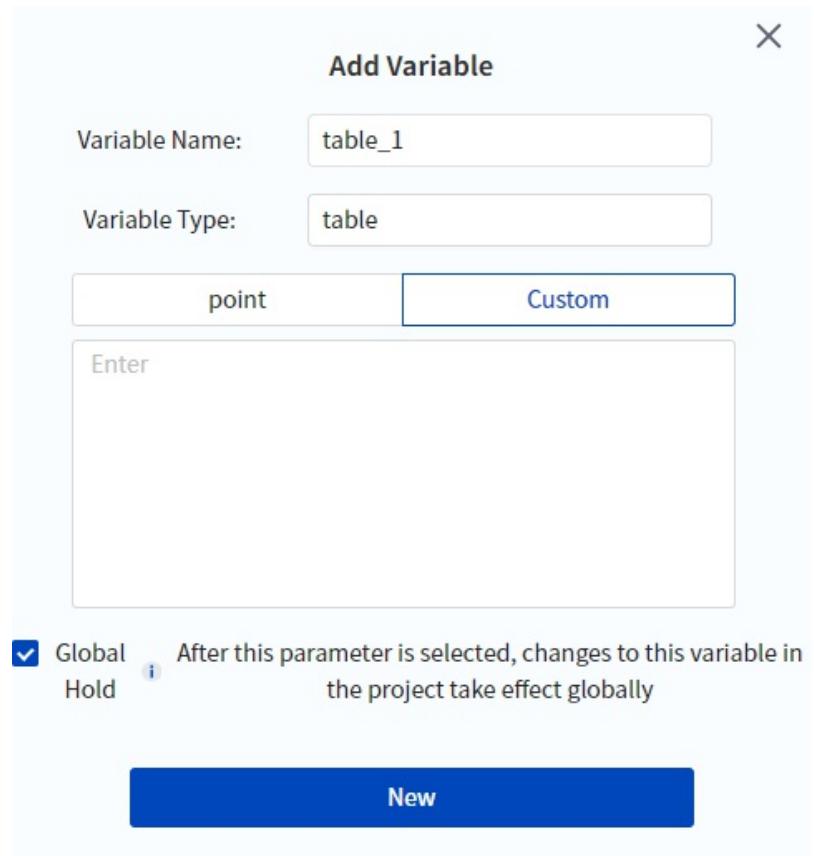
DobotStudio Pro supports the following types of global variables:

- **number:** Numeric value. You can define a value range. Values outside this range will not be saved. Assigning a value beyond this range in a project will cause the robot to report an error and stop. Only administrator can modify the value range (this permission cannot be delegated).
- **bool:** Boolean value (true/false).
- **string:** String. The string entered on this page do not require double quotation marks, but when modifying string-type global variables in a project, the string must be enclosed in double quotes.
- **table:** Table (including array). Support to set to **Point** or **Custom** format.
  - When set to **Point**, move the robot to the desired position and click **Obtain**.

**Add Variable**

Variable Name:	table_1	
Variable Type:	table	
<input checked="" type="radio"/> point		<input type="radio"/> Custom
<input type="button" value="Obtain"/> user: <input type="text" value="1"/> tool: <input type="text" value="0"/>		
X:	<input type="text" value="-277.9336"/>	Y: <input type="text" value="-316.2206"/> Z: <input type="text" value="1476.5"/>
RX:	<input type="text" value="-90"/>	RY: <input type="text" value="0"/> RZ: <input type="text" value="-179.605"/>
<input checked="" type="checkbox"/> Global <small>i After this parameter is selected, changes to this variable in Hold the project take effect globally</small>		
<input type="button" value="New"/>		

- When set to **Custom**, you need to enter the value of the variable. The format for entering and displaying variable values on this page must adhere to JSON data format restrictions, which differ from the format used in projects. Specific rules are as follows:
  - For array, use `[]` instead of `{}` on the global variables page. For example, an array in the project formatted as `{1,2,3}` should be entered and displayed as `[1,2,3]` on the global variables page.
  - For table in `{key:value}` format, the global variables page requires the format `{"key":value}`. For example, a table formatted as `{a=1, b="test"}` in the project would appear as `{"a":1, "b":"test"}` on the global variables page.



After setting, click **New** to add the global variable.

#### Global Hold:

- If this option is selected, any changes to the variable will be saved, and the modified value will persist even after exiting the script or restarting the system.
- If this option is unselected, changes to the variable will only be valid while the script is running, and will revert to the initial value after exiting the script. During script execution, the real-time value of the variable cannot be seen in the global variable monitoring list. It can only be viewed through [Program variable](#).

#### Limitations:

- Global variables only support Chinese and English.
- When using **table-type global variables**, avoid the following situations, otherwise the robot will alarm and stop the project. Both `table_1` and `table_2` below are global variables in `table` type.
  - **Nested variables**

```
-- Correct usage: Table members are assigned constants or other variables.
table_1[1] = {1,2,3}
table_1[2] = table_2
-- Incorrect usage: A table member is assigned the table itself.
table_1[1] = table_1 -- Assigning table_1 to one of its members will cause an error.
```

## 8.7 Program variables

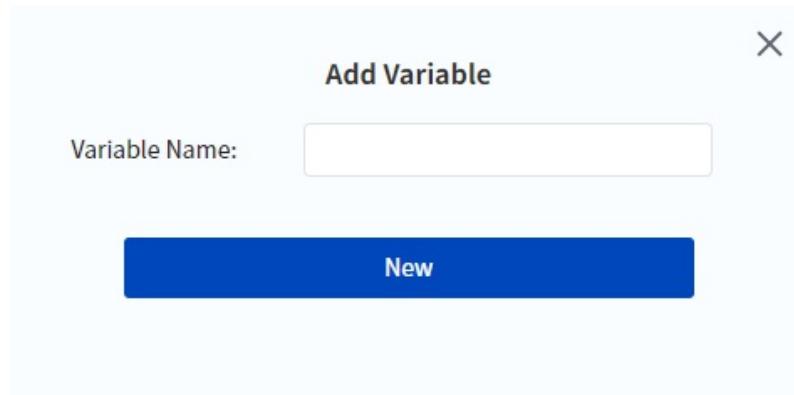
This page is used to set the variables that need to be monitored during the execution of the project. By using the variable name, you can monitor the specified variable's type and value, and you can modify the type and value of the monitored variables.

The screenshot shows a software interface titled "Program variables". On the left, there is a sidebar with categories: I/O, Modbus, Variable, Global variable, and Program variables. The "Program variables" category is currently selected. The main area displays a table with the following data:

NO	Variable Name	Type	Value
1	var_1	-	-

Below the table are three buttons: "Delete" (with a trash icon), "Modify" (with a pencil icon), and "+ New" (with a plus icon).

Click **+ New** to add a new program variable. Select a variable and click **Modify** to change its attributes, or click **Delete** to remove it.



The variable names support inputting object key values:

- For example, to monitor the joint coordinates of point P1, add the variable: P1.joint.
- To monitor the X-coordinate of point P1, add the variable: P1.pose[1].

After adding, the variable list will display the added variables.

## Program variable monitoring

Program variable monitoring starts when the script is running and polls every second. It stops polling when the script is paused or stopped. For the added program variables, if no value is available during script execution, both the type and value will display as "-".

Program variables				
I/O	NO	Variable Name	Type	Value
Controller DI/DO				
Tool I/O				
Safety I/O	1	var_1	-	-
Modbus	2	var1	NUMBER	1234
Variable	3	string1	STRING	"asdddddd"
Global variable	4	bool1	BOOL	false

## Program variable types

Program variables are categorized into three types based on their source: **local**, **upvalue**, and **global**. When there are variables with the same name, the one with the highest priority will be displayed and modified first.

The Lua native variable types that can be monitored include:

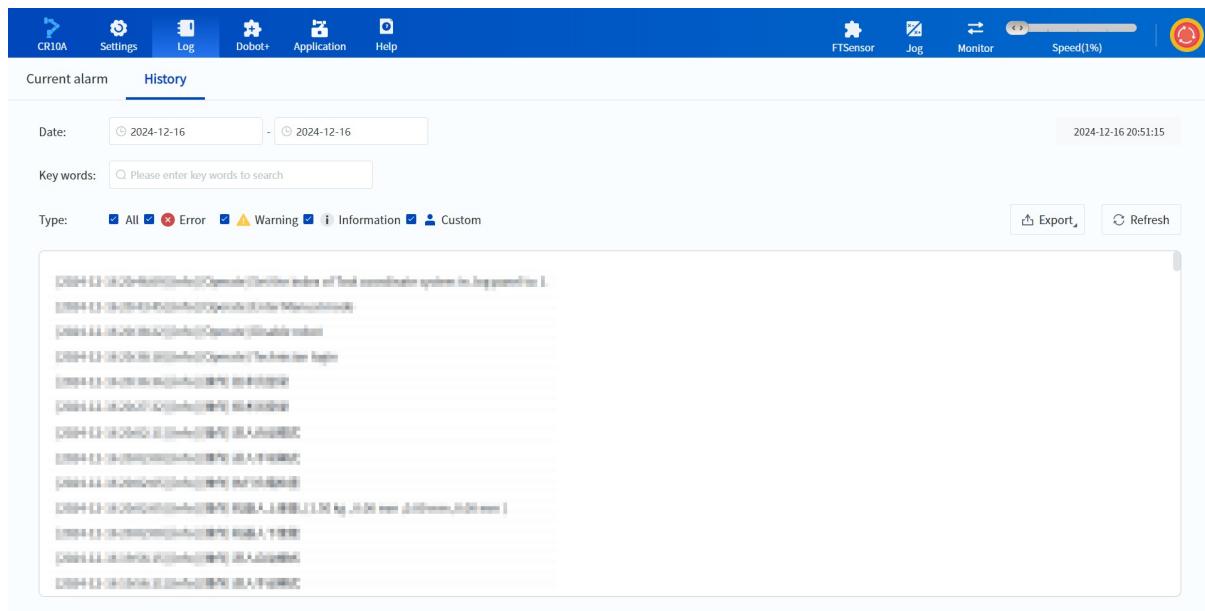
- **nil**: It means void (without any valid values). For example, if you print an unassigned variable, it will output a nil value.
- **boolean**: A boolean value, which can be either true or false. Lua treats false and nil as false, and others as true including number 0.
- **number**: A numeric value, a double-precision floating-point number that supports various operations.
- **string**: A string, which is a sequence of characters made up of numbers, letters, and underscores.
- **table**: A table (including array). The table type supports expanding to view its contents and modifying its elements.

## Limitations

- Program variables can be added, deleted, and modified while the script is running, paused, or stopped.
- A maximum of 20 program variables can be added. The length of the variable name is limited to a maximum of 256 bytes.

- Program variables only support Chinese and English.

9 Log



The Log page displays the robot's running log, which supports filtering by date, keywords and type.

The meanings of the log types are as follows:

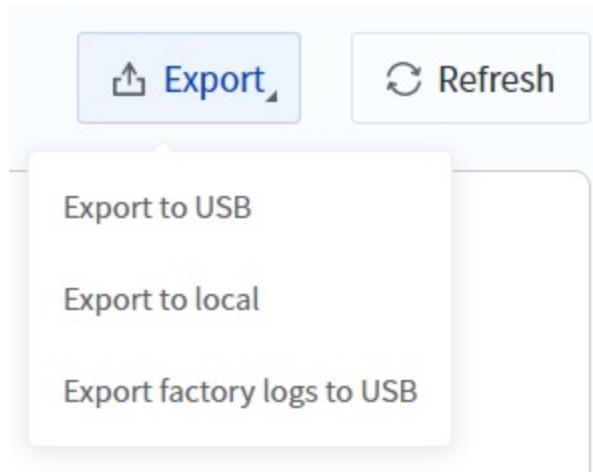
- **Error log:** Alarm information related to software errors or robot alarm status.
  - **Warning log:** Warning information related to abnormal software actions or robot status. These abnormalities do not affect subsequent actions.
  - **Operation log:** Information recorded when there are changes in the robot's status.
  - **Custom log:** Logs generated by users using the `Log(value)` command.

## Log export

**Export to USB:** Export logs matching the filter criteria to a storage device connected to the controller's USB port.

**Export to local:** Export logs matching the filter criteria to the local computer.

**Export factory logs to USB:** Export all controller logs to a USB device.



**i NOTE**

- If the connected USB storage device has multiple partitions, logs will be exported to the first partition. For some storage devices (e.g., USB drives used as boot disks), the first partition may be hidden, making the exported logs inaccessible in Windows.
- Factory logs will be saved in a folder named "logs" in the root directory of the USB drive. When exporting factory logs again, the new "logs" folder will overwrite the previous one.
- Avoid removing the USB drive during the export process, as it may cause file corruption on the drive.

In some scenarios (e.g. when the robot is running autonomously and the software is connected to the robot), the logs may not refresh automatically. You may need to manually click Refresh to get the latest logs.

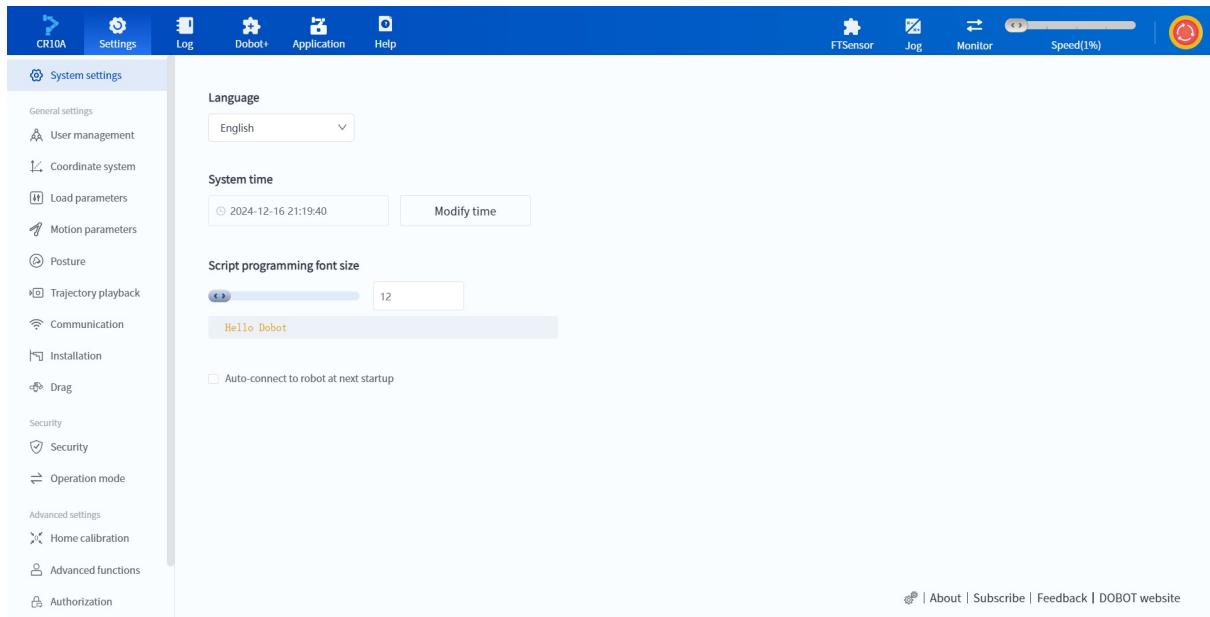
# 10 Settings

- [\*\*10.1 System settings\*\*](#)
- [\*\*10.2 User management\*\*](#)
- [\*\*10.3 Coordinate system management\*\*](#)
- [\*\*10.4 Load parameters\*\*](#)
- [\*\*10.5 Button settings \(Magician E6\)\*\*](#)
- [\*\*10.6 Motion parameters\*\*](#)
- [\*\*10.7 Posture settings\*\*](#)
- [\*\*10.8 Trajectory playback\*\*](#)
- [\*\*10.9 Communication settings\*\*](#)
- [\*\*10.10 Installation settings\*\*](#)
- [\*\*10.11 Drag settings\*\*](#)
- [\*\*10.12 Power voltage \(DC controller/Magician E6\)\*\*](#)
- [\*\*10.13 Safety settings\*\*](#)
- [\*\*10.14 Operation mode settings\*\*](#)
- [\*\*10.15 Home calibration\*\*](#)
- [\*\*10.16 Advanced functions\*\*](#)
- [\*\*10.17 File migration\*\*](#)
- [\*\*10.18 Firmware upgrade\*\*](#)

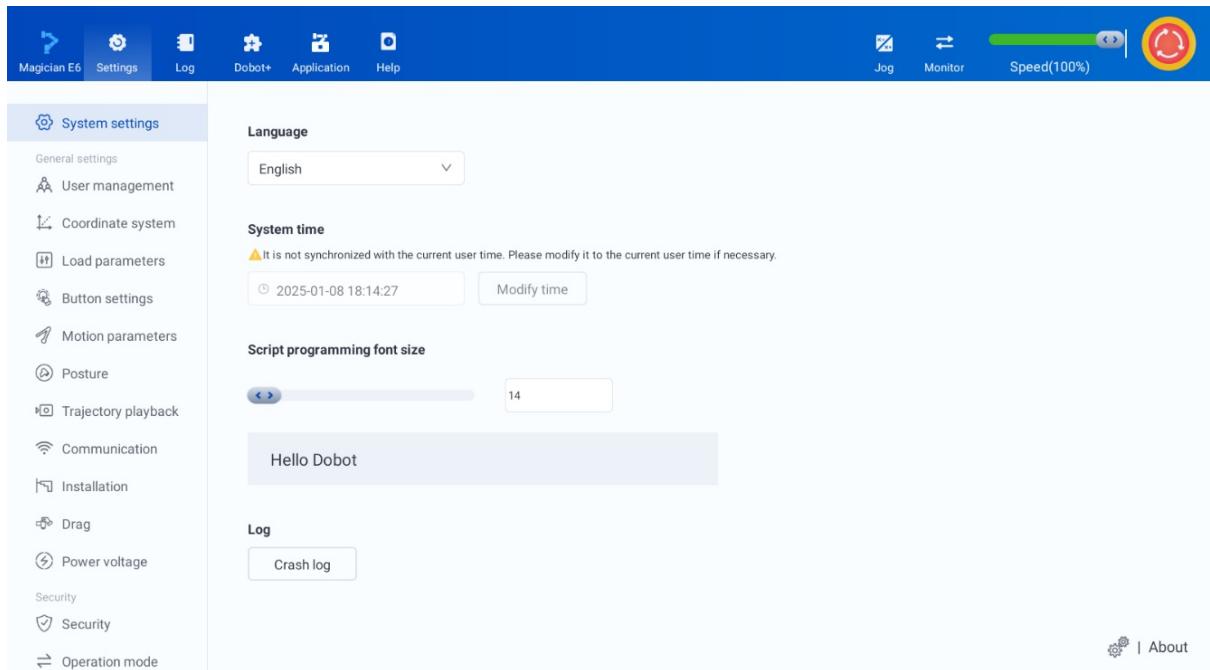
# 10.1 System settings

The System settings page is used to modify the software language, system time, script programming font size, and so on.

## For PC



## For App



**Language:** Set the display language of DobotStudio Pro. It can be modified even when the robot is not connected.

 **NOTE**

If you cannot set the language you want, please contact technical support.

**System time:** Display the current system time of the controller. It can be modified in [Default mode](#) or [Manual mode](#) when no project is running. If the system time of the current device doesn't match the controller's system time, a prompt will appear on the screen recommending that you adjust them to match.

**Script font size:** Adjust the font size in the code area of [Script programming](#). The default size is 14, with a range of [12, 50].

**Log:** Upload the crash log, supported only on App. This function can be used only when the network is connected, and can be operated when the robot is disconnected.

**Auto-connect to robot at next startup:** When selected, the software will try connecting to the currently connected robot automatically next time the software starts. Supported only on PC.

You can click the icons and text on the right bottom of the interface, as described below.



: To use the manufacturer's functions, you need to enter a manufacturer password. Please only use this feature under the guidance of technical support.

**About:** View the components of DobotStudio Pro and disclaimer.

The following functions are only available on DobotStudio Pro (PC).

**Subscribe:** Subscribe to the latest news on software updates and product releases.

**Feedback:** Feed back your problems with the software.

**DOBOT website:** Click to visit [DOBOT official website](#).

## 10.2 User management

When logged in as an administrator, this page allows you to manage permissions, passwords, as well as add or delete custom roles.

Role	Enable password	
Administrator	Yes	<button>Modify password</button>
Technician	<input type="radio"/> OFF	<button>Set password</button>
Operator	<input type="radio"/> OFF	<button>Set password</button>
custom1	<input type="radio"/> OFF	<button>Set password</button> <button>Delete</button>

### Set default role

You can set the default role (not administrator) in the upper left corner of the page, which will automatically log in each time the robot is connected. If the default role has a password enabled, you will need to enter the password or select another role to log in before performing any other operations.

### Manage custom role

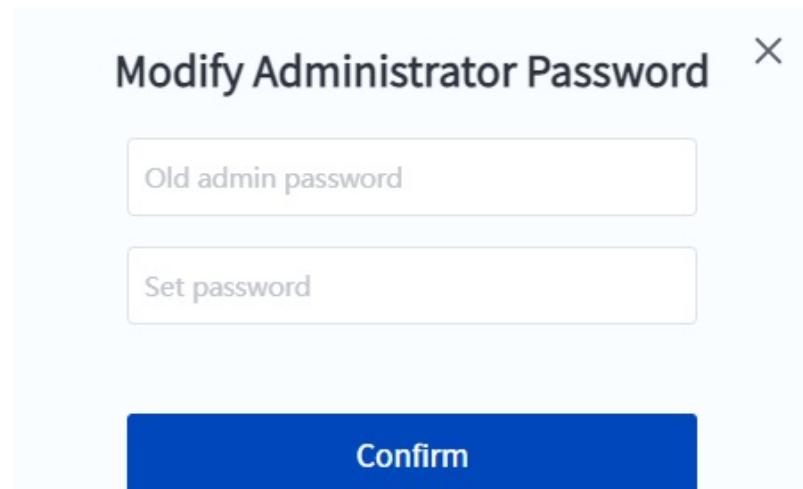
Click **+ Add role**, you can add up to two custom roles.

The custom role can be renamed or deleted. Other operations are the same as the default role, as described below.

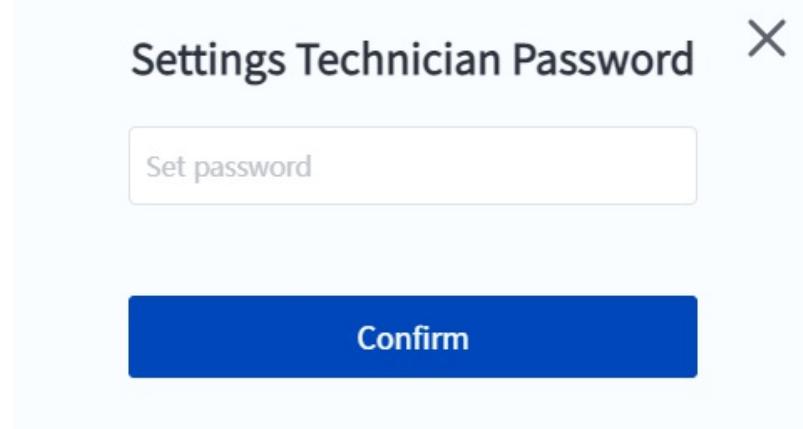
### Set role password

There is no complexity or length requirement for the role password, it supports the combination of English letters and numbers not exceeding 20 characters, please set it according to your actual need.

The administrator password must be enabled. **The default password is 888888**, which it can be changed by entering the old password.



Other roles do not have a password enabled by default. The switch in the **Enable password** column is set to **OFF**. Click the switch, the software will pop up a password setup window. Once set, the password will be enabled, and the switch will change to **ON**.



After setting a password, clicking **Set password** again will bring up the same password setting window. You can reset the password for that role by directly entering the new password without the old one.

Turning off the **Enable password** switch will clear the current password for that role. The next time you enable it, you will need to set a new password.

## Set role permissions

Click **Permission assignment** to enter the following page (custom role will only appear after being added).

User management > Permission assignment			Restore defaults	Cancel	Save
Type	Function	Administrator	Technician	Operator	custom1
	Open/Run/Stop project	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Enable, Clear alarm, Switch between Manual/Auto modes, Adjust speed ratio	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Robot Jogging	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	View/Export logs	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Online/TCP mode switching, opening/closing IO/Modbus configuration	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Basic settings	Switch between Online/TCP modes, Open/Close IO/Modbus configuration	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Modbus configuration	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Global variable settings	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	DOBOT+ plugin, end button settings	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Project and Program Management	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Point List Operation	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	System time settings	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Coordinate System Management	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Permissions for all roles can be modified. Clicking **Restore defaults** will restore the permission assignment status to the default.

After making changes, click **Save** to apply the settings.

## Default role permissions

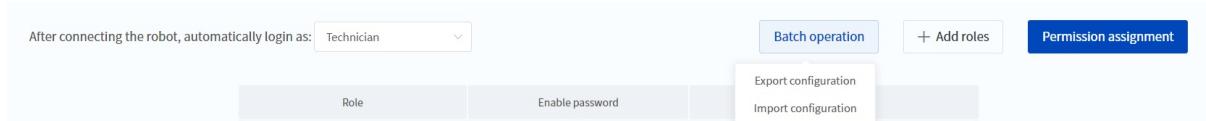
The default role permissions are shown in the table below. Note that permissions for **opening/running/stopping projects** cannot be modified.

Type	Function	Administrator	Technician	Operator
Basic operations	Open/Run/Stop project	√	√	√
	Enable, Clear alarm, Switch between Manual/Auto modes, Adjust speed ratio	√	√	√
	Robot Jogging	√	√	√
	View/Export logs	√	√	√
	Switch between Online/TCP modes, Open/Close IO/Modbus configuration	√	√	X
	IO (including Safety IO) configuration	√	√	X
	Modbus configuration	√	√	X
	Global variable settings	√	√	X
	DOBOT+ plugin, end button settings	√	√	X
	Project and program management	√	√	X

	Point list operation	√	√	X
General settings	System time settings	√	X	X
	Coordinate system management	√	√	X
	Load parameter settings	√	√	X
	Button settings	√	√	X
	Trajectory recording and playback	√	√	X
	Communication settings	√	√	X
	Power voltage	√	√	X
Advanced settings	Packing/home posture settings	√	X	X
	Motion parameter settings	√	X	X
	Installation angle settings	√	X	X
	Drag settings	√	X	X
	Security settings	√	X	X
	Manual/Auto mode settings	√	X	X
	Home calibration	√	X	X
	Advanced feature settings	√	X	X

## Batch operation

Click **Batch operation > Export configuration**, you can export the current user permission configuration as a file. Click **Batch operation > Import configuration**, you can import the user permission configuration from a file.



## 10.3 Coordinate system management

- [10.3.1 User coordinate system](#)
- [10.3.2 Tool coordinate system](#)

## 10.3.1 User coordinate system

When the position of a workpiece changes or a robot program needs to be reused in multiple processing systems of the same type, you can set up a user coordinate system. This ensures all paths update in sync with the user coordinates, greatly simplifying the teaching and programming process.

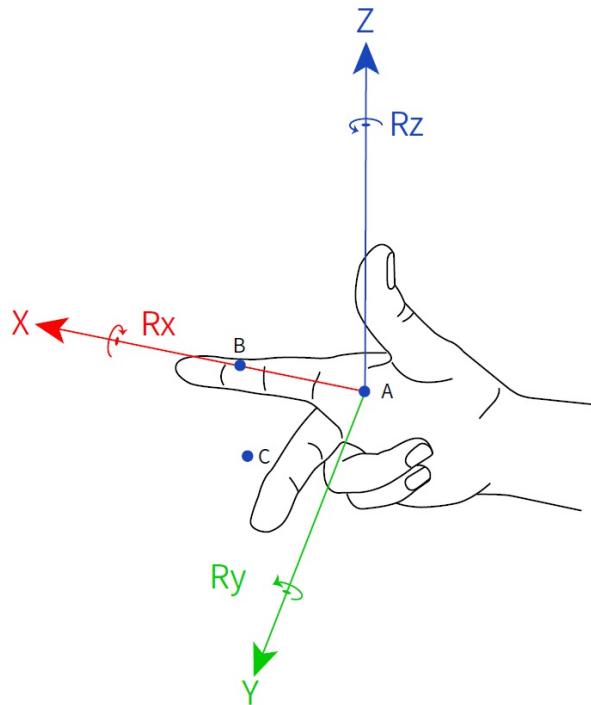
The system currently supports up to 51 [0, 50] user coordinate systems. User coordinate system **0** is defined as the base coordinate system (refer to the hardware guide for each robot), and it cannot be changed.

The origin and axis directions of the user coordinate system can be customized. The same robot posture will have different coordinate values in different user coordinate systems. The value of a user coordinate system represents the offset and rotation angle relative to User coordinate system 0.

### NOTICE

When creating a user coordinate system, ensure that the reference coordinate system is User coordinate system **0** when getting points.

The recommended method for creating a user coordinate system is the three-point teaching method. Move the robot arm to any three points: **A**, **B**, and **C**. Where, Point **A** is defined as the origin. The line between points **A** and **B** determines the positive direction of the X-axis of the user coordinate system. Point **C**, perpendicular to the X-axis, determines the positive direction of the Y-axis. The Z-axis is determined by the right-hand rule.



## Creating user coordinate system

1. Click **+ Add** on the "User coordinate system" page, as shown below.

Id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		280.107	11.897	0	0	0	-0.395
2		-	-	-	-	-	-

2. Click **Three-point setting** on the "Add user coordinate system" page.

User coordinate system > Add

index: 3 Please enter alias

Coordinates:

X: 0	Y: 0	Z: 0
Rx: 0	Ry: 0	Rz: 0

**NOTE**

You can also manually modify the X, Y, Z, Rx, Ry, and Rz values and click **Save**. X, Y, and Z refer to the position of the origin of the user coordinate system in the base coordinate system. Rx, Ry, and Rz refer to the angles by which the user coordinate system is rotated around the base coordinate system, following the order X -> Y -> Z.

3. Refer to the diagram, control the robot arm to move to the corresponding points, and click **Obtain**.

User coordinate system > Add > Three-point setting

Cancel Confirm

Three-point user coordinates diagram

Point	Operation	x	y	z	rx	ry	rz
P1	Obtain  Run to	0	0	0	0	0	0
P2	Obtain  Run to	0	0	0	0	0	0
P3	Obtain  Run to	0	0	0	0	0	0

**i NOTE**  
Long-pressing Run to can move the robot to the obtained point.

- Click **Confirm** to return to the "Add user coordinate system" page, and the coordinate values are updated to the calibrated values. You can view or modify the three-point settings used to generate the coordinate system or manually modify the values. For more details, refer to the section on modifying user coordinate system below.
- Click **Save**, and this coordinate system is added to the user coordinate system list.

## Modifying user coordinate system

- Select a coordinate system on the "User coordinate system" page and click Modify, as shown below.

User coordinate system

Hide empty ... Clear Copy Modify + Add

id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		280.107	11.897	0	0	0	-0.395
2		-	-	-	-	-	-
3		111	111	111	111	111	111

- If the values of the selected coordinate system were manually input and saved, the "Modify" page will look identical to the "Add" page. You can directly modify the values or click **Three-point setting** for calibration. If the values of the selected coordinate system were generated by three-point setting, the UI on the "Modify" page will look as shown below.

User coordinate system > Modify

index: 2 Please enter alias

Coordinates:

X: 455.672638	Y: -977.768188	Z: 133.640793
Rx: -179.972116	Ry: 0.650332	Rz: 166.14307

**Three-point setting**

3. Click **View three-point setting** at the right bottom of the coordinate values to view the three-point settings used to generate the current values, and re-obtain the points if needed.
4. Click **Modify** to directly change the coordinate values.

#### **i** NOTE

After manual modification, you can no longer view the three-point setting corresponding to the value of the coordinate system before modification.

5. Click **Save**, and this coordinate system is updated to the user coordinate system list.

## Copying user coordinate system

Select a coordinate system on the "User coordinate system" page and click  **Copy** to create a new coordinate system the same as the selected one.

User coordinate system							
Id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		280.107	11.897	0	0	0	-0.395
2		-	-	-	-	-	-
3		111	111	111	111	111	111

## Clearing user coordinate system

Select a coordinate system on the "User coordinate system" page, click  **Clear**, and confirm to clear the selected coordinate system. The cleared coordinate system still occupies its id, only the coordinate system data is cleared (e.g., coordinate system 2 in the figure below), and an error will be reported when it is called.

User coordinate system							
id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		280.107	11.897	0	0	0	-0.395
2		-	-	-	-	-	-
3		111	111	111	111	111	111

You can click **Hide empty coordinate system** to hide the empty coordinate system from the list, then the button changes to **Show empty coordinate system**, allowing you to display the empty coordinate system again.

User coordinate system							
id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		280.107	11.897	0	0	0	-0.395
2		-	-	-	-	-	-
3		111	111	111	111	111	111

The empty coordinate system can be modified and will be reassigned after modification.

## 10.3.2 Tool coordinate system

When a tool (such as a welding torch, nozzle, or gripper) is mounted on the end of the robot arm, it's necessary to set up a tool coordinate system to facilitate programming and robot operation. For example, when using multiple grippers to simultaneously handle several workpieces, you can set up a tool coordinate system for each gripper to improve the handling efficiency.

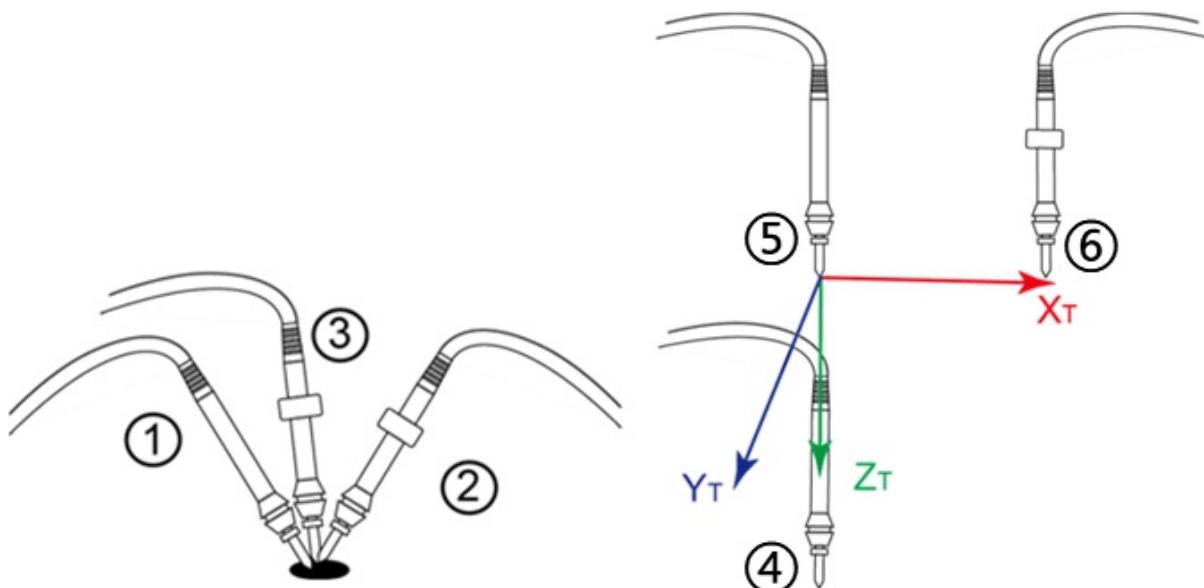
The system currently supports up to 51 [0, 50] tool coordinate systems. Tool coordinate system **0** is defined as the flange coordinate system (refer to the hardware guide for each robot), which represents no tool and cannot be changed.

The tool coordinate system is established with the TCP (Tool Center Point, typically the working point of the tool, such as the center of a suction cup or the tip of a welding torch) as the origin and is used to define the position and posture of the tool. The value of a tool coordinate system represents the offset and rotation angle relative to Tool coordinate system 0.

### ⚠️ NOTICE

When creating a tool coordinate system, ensure that the reference coordinate system is Tool coordinate system **0** when getting points.

For six-axis tools, the recommended method for creating a tool coordinate system is the six-point teaching method “TCP+ZX”. After installing the tool on the end of the robot arm, adjust the tool’s posture so that the TCP aligns with the same point (reference point) in space (①②③) from three different directions to obtain the tool’s position offset. Then, obtain the tool’s posture offset based on three additional points (④⑤⑥), where ④ aligns with the same point as ①②③.



# Creating tool coordinate system

1. Click **+ Add** on the "Tool coordinate system" page, as shown below.

Id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		111	111	111	111	111	111
2		222	222	222	222	222	222
3		-	-	-	-	-	-
4		333	333	333	333	333	333

2. Click **Six-point setting** on the "Add tool coordinate system" page.

Tool coordinate system > Add

index: 5 Please enter alias

Coordinates:

X: 0 Y: 0 Z: 0  
Rx: 0 Ry: 0 Rz: 0

**Note**

You can also manually modify the X, Y, Z, Rx, Ry, and Rz values and click **Save**. The process for **Four-point setting** and **Six-point setting** is similar and won't be repeated here.

3. Refer to the diagram, control the robot arm to move to the corresponding points, and click **Obtain**.

Tool coordinate system > Add > Six-point setting

Six-point tool coordinates diagram A, B, C: reference points

Point	Operation	x	y	z	rx	ry	rz
P1	<b>Obtain</b> <b>Run to</b>	0	0	0	0	0	0
P2	<b>Obtain</b> <b>Run to</b>	0	0	0	0	0	0
P3	<b>Obtain</b> <b>Run to</b>	0	0	0	0	0	0
P4	<b>Obtain</b> <b>Run to</b>	0	0	0	0	0	0

**Note**

Long-pressing **Run to** can move the robot to the obtained point.

4. Click **Confirm** to return to the "Add tool coordinate system" page, and the coordinate values are updated to the calibrated values. You can view or modify the three-point settings used to generate the coordinate system or manually modify the values. For more details, refer to the section on modifying tool coordinate system below.
5. Click **Save**, and this coordinate system is added to the tool coordinate system list.

## Modifying tool coordinate system

1. Select a coordinate system on the "Tool coordinate system" page and click **Modify**, as shown below.

Tool coordinate system							
Id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		111	111	111	111	111	111
2		222	222	222	222	222	222
3		-	-	-	-	-	-
4		333	333	333	333	333	333

2. If the values of the selected coordinate system were manually input and saved, the "Modify" page will look identical to the "Add" page. You can directly modify the values or click **Four/Six-point setting** for calibration. If the values of the selected coordinate system were generated by **Four/Six-point setting**, the "Modify" page will appear as shown below. The text for **View six-point setting** will change based on the generation method.

Tool coordinate system > **Modify**

index: 1	Please enter alias	Cancel	Save
Coordinates:		Four-point setting	Six-point setting
X: 0	Y: 0	Z: 150	
Rx: 0	Ry: 0	Rz: 0	

3. Click **View six-point setting** to view the six-point settings used to generate the current values, and re-obtain the points if needed. The same principle applies for four-point settings.
4. Click **Modify** to directly change the coordinate values. Note that after manual modification, you can no longer view the six/four-point setting corresponding to the value of the coordinate system before modification.
5. Click **Save**, and this coordinate system is updated to the tool coordinate system list.

## Copying tool coordinate system

Select a coordinate system on the "Tool coordinate system" page and click **Copy** to create a new coordinate system the same as the selected one.

Tool coordinate system							
id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		111	111	111	111	111	111
2		222	222	222	222	222	222
3		-	-	-	-	-	-
4		333	333	333	333	333	333

## Clearing tool coordinate system

Select a coordinate system on "Tool coordinate system" page, click **Clear**, and confirm to clear the selected coordinate system. The cleared coordinate system still occupies its id, only the coordinate system data is cleared (e.g., coordinate system 3 in the figure below), and an error will be reported when it is called.

Tool coordinate system							
id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		111	111	111	111	111	111
2		222	222	222	222	222	222
3		-	-	-	-	-	-
4		333	333	333	333	333	333

You can click **Hide empty coordinate system** to keep the empty coordinate system from being displayed in the coordinate system list, and then the button changes to **Show empty coordinate system**, which you can click to restore the display of the empty coordinate system.

Tool coordinate system							
id	Alias	X	Y	Z	Rx	Ry	Rz
0		0	0	0	0	0	0
1		111	111	111	111	111	111
2		222	222	222	222	222	222
3		-	-	-	-	-	-
4		333	333	333	333	333	333

The empty coordinate system can be modified and will be reassigned after modification.

## 10.4 Load parameters

Load parameters include the center of mass and weight of the end effector (including gripper) on the robot. Please configure these settings according to the actual load.

### **i** NOTE

This page is only used to modify the load parameter group. The changes will not take effect immediately. To make the modified parameters effective, please select the corresponding parameter group in the “Load enabling settings” page that pops up when the robot arm is enabled.

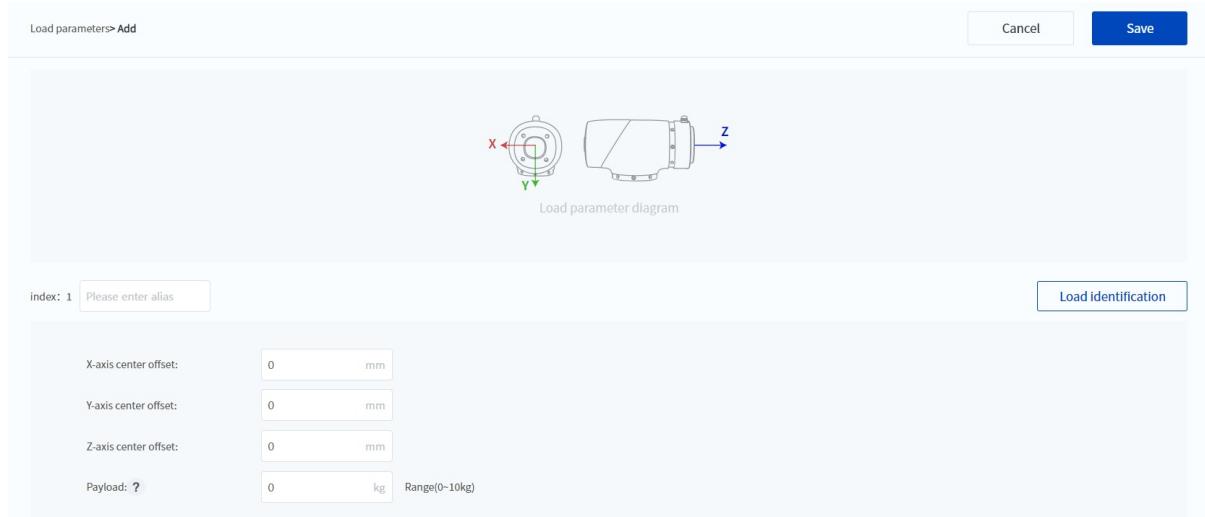
### **!** NOTICE

Incorrect load settings can reduce the robot's performance and may trigger abnormal collision detection alarms or cause the robot to become uncontrollable during dragging.

index	Alias	X-axis center offset(mm)	Y-axis center offset(mm)	Z-axis center offset(mm)	Payload(kg)
0	load1	0	0	0	0

Click **Add** to add a new group of load parameters. Click **Modify** to modify the selected group of parameters. Click **Delete** to delete the selected group of parameters.

# Adding/Modifying parameter group



The alias for the parameter group can be modified. Both the enable and programming functions will use the alias to reference the parameter group.

There are two ways to set load parameters: load identification and manual modification.

## Load identification

When the robot is enabled, without alarm and not in motion, the current load parameters of the robot can be automatically identified.

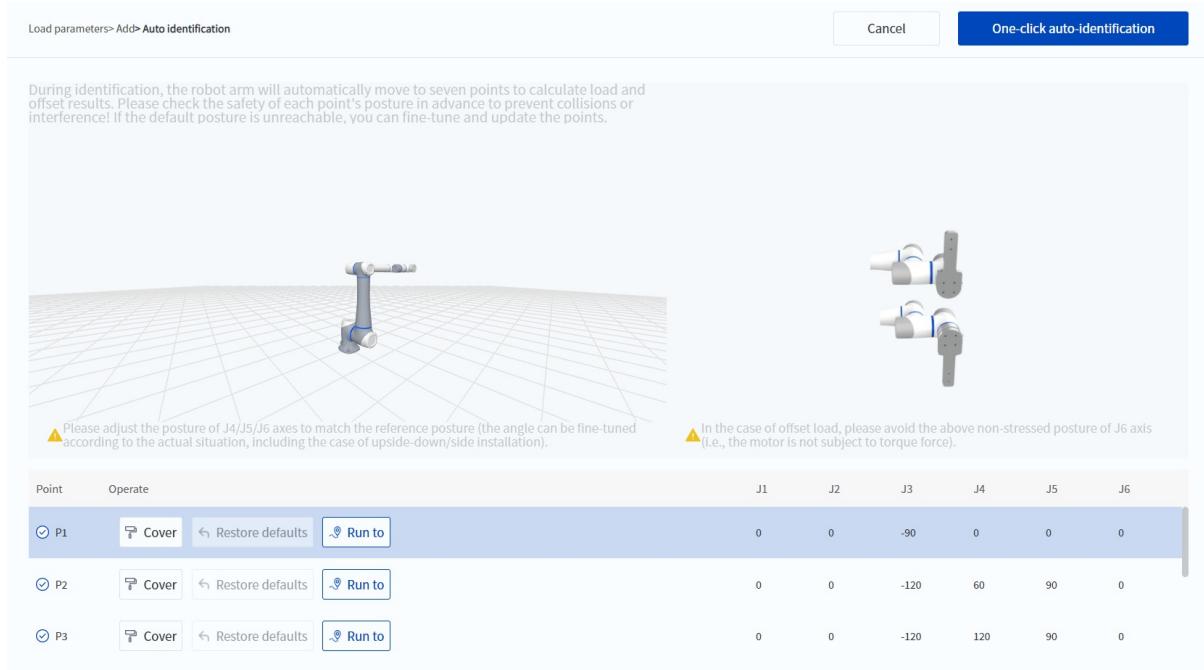
### NOTE

- Before load identification, ensure that the robots [installation angle](#) has been set correctly.
- Magician E6 does not support load identification.

1. Click **Load identification** to open the identification window.
2. The system will automatically generate 7 default points. You can select the point to view the corresponding posture diagram, and long-press **Run to** to move the robot to the corresponding point.
3. If the robot cannot move to a default point due to obstacles or other reasons, you can refer to the posture diagram and manually jog the robot to a nearby point that meets the requirements, then click **Cover** to overwrite the default point. Click **Restore defaults** to revert the overwritten point to the default point.
4. Click **One-click auto identification** and confirm, and the robot will move to each point in turn and perform identification.
  - If the identification is successful, the software will return to Add/Modify page and the load parameters will be updated to the identified parameters. Please check if these parameters are

reasonable according to the actual load. If not, re-identify the load or manually modify the parameters.

- If the identification fails, an error message will pop up and remain on the automatic identification page. Please refer to the posture diagrams for each point to ensure they meet the requirements. Modify the points that do not meet the requirements and then click **One-click auto identification** again.



## Manual modification

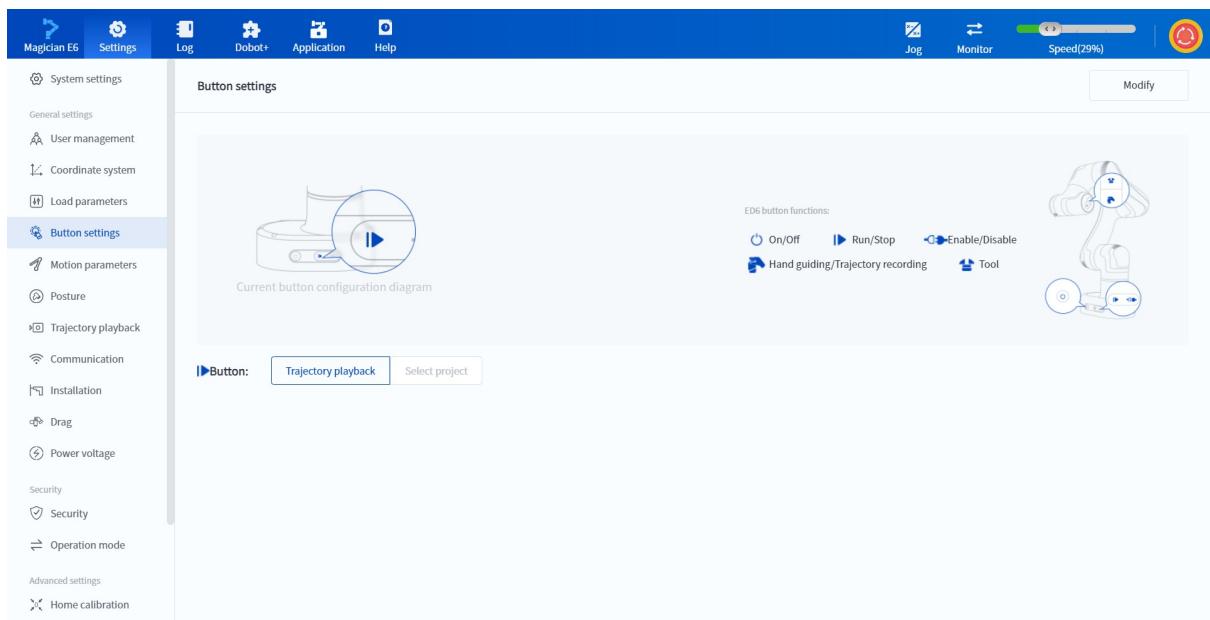
Manually enter the following load parameters:

- **X/Y/Z-axis center offset:** The offset distance of the center of mass of load in each direction. For the direction of each axis, refer to the diagram on the interface.
- **Payload:** Total weight of the end effector and workpiece, which must not exceed the the robot's maximum load capacity.

After modifying the settings, click **Save**.

## 10.5 Button settings (Magician E6)

When the robot model is Magician E6, you can configure the function of the Run/Stop button on the base.



You can click **Modify** in the upper right corner to modify the function of the Run/Stop button.

- The default function of this button is **Trajectory playback**, which plays back the most recently recorded trajectory.
- If the function is set to **Select project**, you will need to select a project.



Regardless of the assigned function, pressing this button during trajectory playback or project execution will stop the robot's operation.

## **10.6 Motion parameters**

The robot is pre-configured at the factory with optimized motion parameters for standard working conditions. It is not recommended to modify these unless there is a specific need. If you find the robot's speed too high, you can reduce the speed according to your actual needs. If you wish to increase the speed, please contact technical support for advice on speed adjustment solution.

The page for CRA series differs from the page for Magician E6.

### **10.6.1 Motion parameters (CRA)**

#### **Playback parameters**

The playback parameters refer to the motion parameters of the robot when running a project or a TCP\_IP motion command.

Playback parameters
Restore defaults
Cancel
Save

**Force torque constraint**  When torque constraint is enabled, the algorithm will adjust acceleration and jerk based on actual operating conditions to avoid torque limit violations.

**Playback speed**  Cartesian speed can be set on the "Safety limits" page.

Joint	Normal mode	Reduced mode	
J1:	<input type="range" value="161"/>	161 °/s	36.8 °/s
J2:	<input type="range" value="161"/>	161 °/s	36.8 °/s
J3:	<input type="range" value="191"/>	191 °/s	43.6 °/s
J4:	<input type="range" value="234"/>	234 °/s	54 °/s
J5:	<input type="range" value="234"/>	234 °/s	54 °/s
J6:	<input type="range" value="234"/>	234 °/s	54 °/s

**Playback acceleration**

J1:	<input type="range" value="110"/>	110 °/s <sup>2</sup>
J2:	<input type="range" value="110"/>	110 °/s <sup>2</sup>
J3:	<input type="range" value="200"/>	200 °/s <sup>2</sup>
J4:	<input type="range" value="1000"/>	1000 °/s <sup>2</sup>
J5:	<input type="range" value="1000"/>	1000 °/s <sup>2</sup>
J6:	<input type="range" value="1000"/>	1000 °/s <sup>2</sup>
X/Y/Z:	<input type="range" value="10000"/>	10000 mm/s <sup>2</sup>
RX/RY/RZ:	<input type="range" value="900"/>	900 °/s <sup>2</sup>

**Playback jerk**

J1:	<input type="range" value="1100"/>	1100 °/s <sup>3</sup>
J2:	<input type="range" value="1100"/>	1100 °/s <sup>3</sup>
J3:	<input type="range" value="2000"/>	2000 °/s <sup>3</sup>
J4:	<input type="range" value="10000"/>	10000 °/s <sup>3</sup>
J5:	<input type="range" value="10000"/>	10000 °/s <sup>3</sup>
J6:	<input type="range" value="10000"/>	10000 °/s <sup>3</sup>
X/Y/Z:	<input type="range" value="18000"/>	18000 mm/s <sup>3</sup>
RX/RY/RZ:	<input type="range" value="9000"/>	9000 °/s <sup>3</sup>

## Torque constraint

This function is crucial for ensuring the robot's reliable operation, and it is recommended to enable it. When torque constraint is enabled, the robot's algorithm adjusts the acceleration and jerk based on the actual operating conditions, preventing torque over-limit, which could lead to alarms.

## Playback speed

Set the maximum joint speed during playback motion. This should be set separately for Normal mode and Reduced mode. The maximum speed in Reduced mode cannot exceed the maximum speed in Normal mode.

This page only allows setting joint speed. The maximum TCP speed in Cartesian coordinates must be set in the **Safety limits** page.

### Playback acceleration/jerk

Set the maximum values for playback acceleration and jerk. These need to be set separately for joint motion and Cartesian motion.

## Jog parameters

Jog speed	
J1:	23 °/s
J2:	23 °/s
J3:	23 °/s
J4:	23 °/s
J5:	23 °/s
J6:	23 °/s
X/Y/Z:	160 mm/s
RX/RY/RZ:	12 °/s

Jog acceleration	
J1:	100 °/s <sup>2</sup>
J2:	100 °/s <sup>2</sup>
J3:	100 °/s <sup>2</sup>
J4:	100 °/s <sup>2</sup>
J5:	100 °/s <sup>2</sup>
J6:	100 °/s <sup>2</sup>
X/Y/Z:	300 mm/s <sup>2</sup>
RX/RY/RZ:	100 °/s <sup>2</sup>

The **Jog parameters** refer to the motion parameters when the robot is in **Jog/Step** mode or when **Running to a target position**.

### Jog speed

Set the maximum speed for Jog motion, which must be set separately for joint speed and Cartesian speed.

### Jog acceleration

Set the maximum acceleration for Jog motion, which needs to be set separately for joint acceleration and Cartesian acceleration.

## Cycle time influence factors

- In actual operation, the robot's maximum motion speed is determined by both the maximum joint speed and TCP speed. The maximum joint speed can be set through the **Motion parameters** page, while the maximum TCP speed is influenced by **Safety limits** (such as TCP speed, momentum, stop time, stop distance).
- The robot's maximum acceleration and jerk not only depend on the Jog parameters but also on whether the **Torque constraint** function is enabled, ensuring the robot's safety and stability during operation.

## 10.6.2 Motion parameters (Magician E6)

Motion parameters		<input type="button" value="Restore defaults"/>	<input type="button" value="Cancel"/>	<input type="button" value="Save"/>
<b>Teach settings:</b>		<b>Playback settings:</b>		
J1:	Speed 12 °/s	Acceleration 100 °/s <sup>2</sup>	J1: Speed 120 °/s	Acceleration 90 °/s <sup>2</sup>
J2:	Speed 12 °/s	Acceleration 100 °/s <sup>2</sup>	J2: Speed 120 °/s	Acceleration 90 °/s <sup>2</sup>
J3:	Speed 12 °/s	Acceleration 100 °/s <sup>2</sup>	J3: Speed 120 °/s	Acceleration 90 °/s <sup>2</sup>
J4:	Speed 12 °/s	Acceleration 100 °/s <sup>2</sup>	J4: Speed 120 °/s	Acceleration 90 °/s <sup>2</sup>
J5:	Speed 12 °/s	Acceleration 100 °/s <sup>2</sup>	J5: Speed 120 °/s	Acceleration 90 °/s <sup>2</sup>
J6:	Speed 12 °/s	Acceleration 100 °/s <sup>2</sup>	J6: Speed 120 °/s	Acceleration 90 °/s <sup>2</sup>
X/Y/Z:	Speed 50 mm/s	Acceleration 300 mm/s <sup>2</sup>	X/Y/Z: Speed 500 mm/s	Acceleration 1000 mm/s <sup>2</sup>
RX/RY/RZ:	Speed 12 °/s	Acceleration 100 °/s <sup>2</sup>	RX/RY/RZ: Speed 120 °/s	Acceleration 100 °/s <sup>2</sup>

The **Teach settings** refer to the motion parameters when the robot is in **Jog/Step** mode or when  **Running to** a target position. It supports setting the maximum speed and acceleration for each joint or in Cartesian coordinates.

The **Playback settings** refer to the motion parameters of the robot when running a project or a **TCP\_IP** motion command. You can set the maximum speed, acceleration, and jerk for each joint or in Cartesian coordinates.

After making changes, you can click **Save** to save the modified values. Clicking **Cancel** can discard the changes. Clicking **Restore defaults** can reset the parameters to the factory default settings.

## 10.7 Posture settings

This page is used to move the robot to various factory-preset postures.



- The **Pack Pose** minimizes the robot space, making it easier to package and transport.
- The **Home Pose** sets all joint angles to 0°.

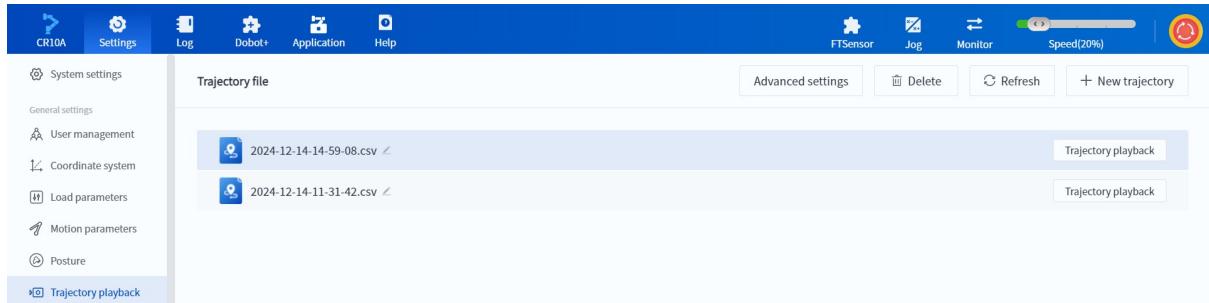
Long-pressing **Run to** can move the robot to the corresponding posture.

### NOTE

You can also move the robot to that posture through the [Jog panel](#).

## 10.8 Trajectory playback

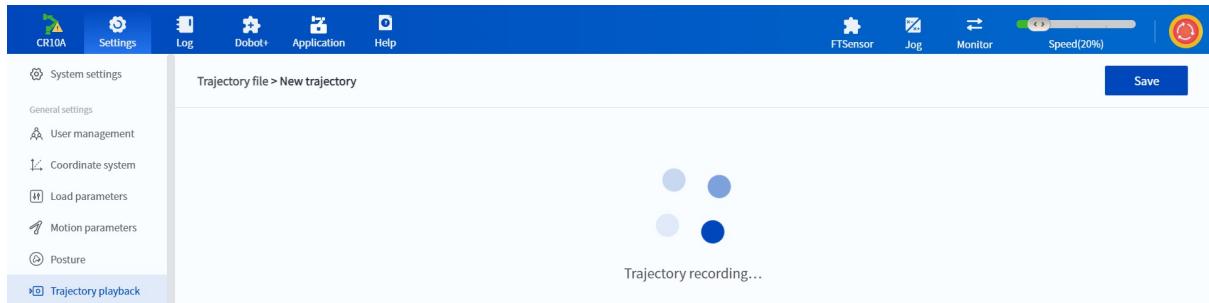
Trajectory playback is used to record and play back the movement trajectory of the robot.



Click **+ New trajectory** on the upper right corner, the robot will enter the drag mode. Now you can drag the robot, and the dragged trajectory will be recorded. During trajectory recording, a point is recorded every 50ms, with a maximum of 10,000 points per trajectory (approximately 500s).

### ⚠️ NOTICE

DO NOT switch between user or tool coordinate systems during trajectory recording, as this will prevent the recorded trajectory from being played back.



Once the desired trajectory is recorded, click **Save** to exit the drag mode, and a new record will be added to the trajectory file list.

- You can click next to the trajectory file name to modify the file name.
- You can click **Trajectory playback** to have the robot play back the recorded trajectory. You can stop the playback at any time.
- You can select a trajectory file and click **Delete** to remove it.
- You can click **Refresh** to retrieve the latest trajectory file list from the controller.

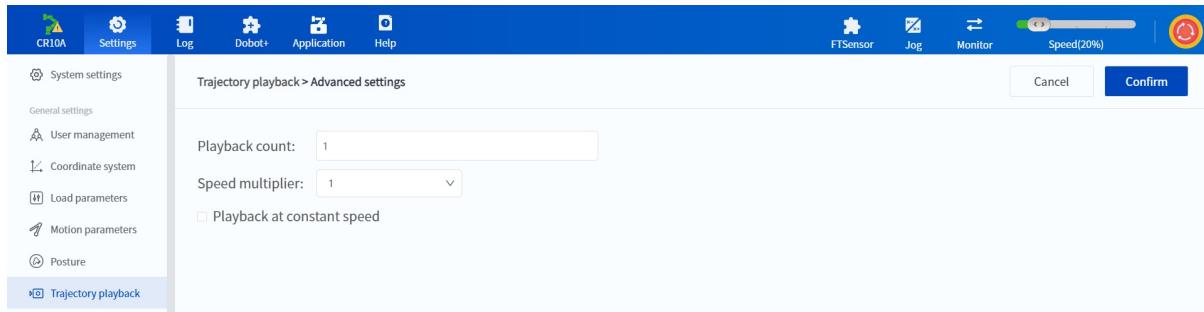
### ℹ️ NOTE

The saved trajectory files can also be called using the **Trajectory playback** block in Blockly

programming or the **Trajectory playback** command in script programming.

Clicking **Advanced settings** can set the playback method. The settings apply to trajectories started via the DobotStudio Pro interface or the robot's end button.

- **Playback count** specifies how many times the robot will play back the trajectory after clicking **Trajectory playback**. If the count is greater than 1, the robot will automatically move back to the start of the trajectory in joint mode after completing one playback, and then begin the next one.
- **Speed multiplier** is effective only when **Playback at constant speed** is unchecked. The robot will play back the trajectory at a scaled speed, proportional to the original speed (not affected by the global speed).
- When **Playback at constant speed** is checked, the robot will play back the trajectory at a constant speed, based on the global speed.



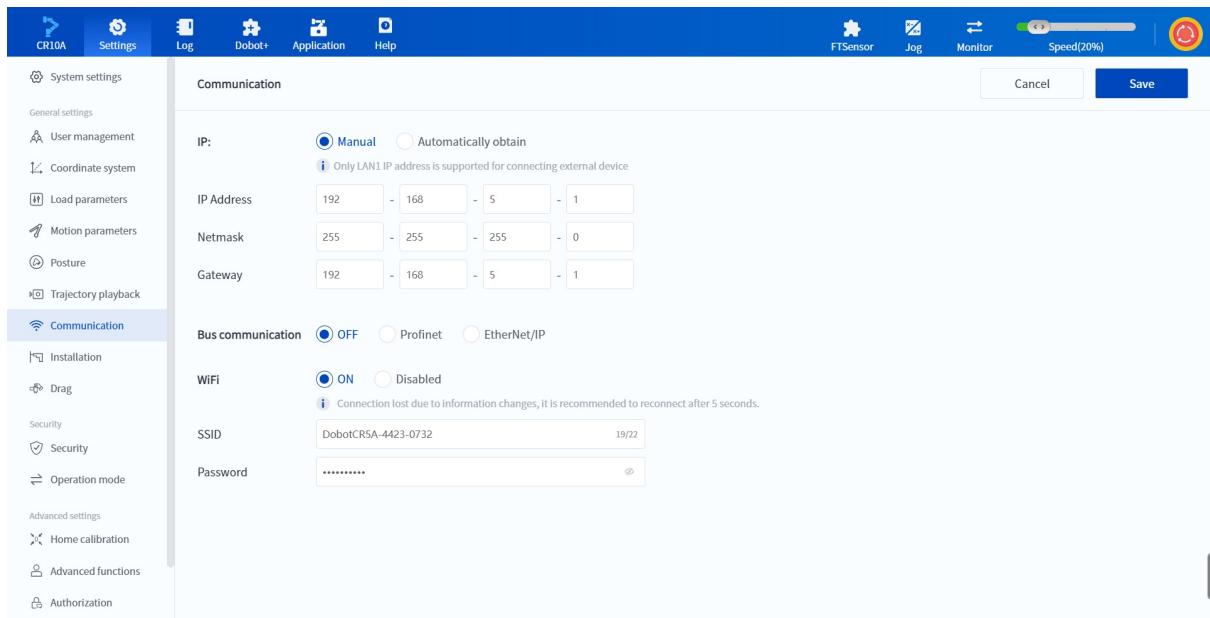
#### **NOTE**

You can also record trajectories using the robot's end-effector button (refer to the corresponding hardware guide for details). The differences between Trajectory recording and Trajectory playback via the DobotStudio Pro interface and the robot's end-effector button are as follows:

- Trajectory files generated through the DobotStudio Pro interface are named based on the save time in the format "Year-Month-Day-Hour-Minute-Second". A new file is created each time it is saved. Trajectory files generated through the robot's end-effector button are named "TrackRecord.csv". Each save overwrites the previous file.
- The DobotStudio Pro allows users to select the specific trajectory to play back. The robot's end-effector button can only play back the "TrackRecord.csv" file.

# 10.9 Communication settings

The **Communication settings** page is primarily used to configure the communication parameters of the currently connected controller, such as setting the LAN1 IP address, bus mode, and WiFi related properties.



## IP settings

The robot can communicate with external devices via the LAN interface, supporting TCP, UDP, or Modbus protocols. You can modify the robot's LAN1 IP address, subnet mask, and gateway. The IP address of the robot must be within the same network segment as that of the external equipment without conflict.

- If the robot is directly connected to an external device or via a switch, select **Manual** and modify the IP address, subnet mask (needed when connecting multiple networks), and default gateway to ensure the robot and the external device are on the same subnet.
- If the robot is connected to an external device via a router, select **Automatically obtain** to have the router automatically assign an IP address.

## Bus communication

The bus communication can be set to **OFF**, **Profinet**, or **EtherNet/IP**.

When set to **Profinet** or **EtherNet/IP**, you need to define the robot's behavior in case of bus communication loss:

- **Keep running:** No action will be taken, and the project will continue running.

- **Pause:** The project will pause if bus communication is lost.
- **Stop:** The project will stop if bus communication is lost.

For details on how to use the bus communication function, please refer to the *Dobot Bus Communication Protocol Guide (EtherNet/IP, Profinet)*.

### NOTICE

When bus communication is set to Profinet, the LAN1 interface can only be used for Profinet communication, and the IP settings cannot be modified. To use LAN1 for other communications, you need to modify and save the bus communication settings, and then restart the controller.

## WiFi settings

The robot can communicate with external devices via WiFi. You can enable or disable WiFi, as well as modify the WiFi name and password. You can click  to view the current password.

The WiFi module of Magician E6 needs to be purchased and installed by users, and is not supported to be switched on and off on this page.

### NOTICE

Modifying WiFi settings may cause the software to disconnect from the robot. If disconnected, please try reconnecting after 5 seconds.

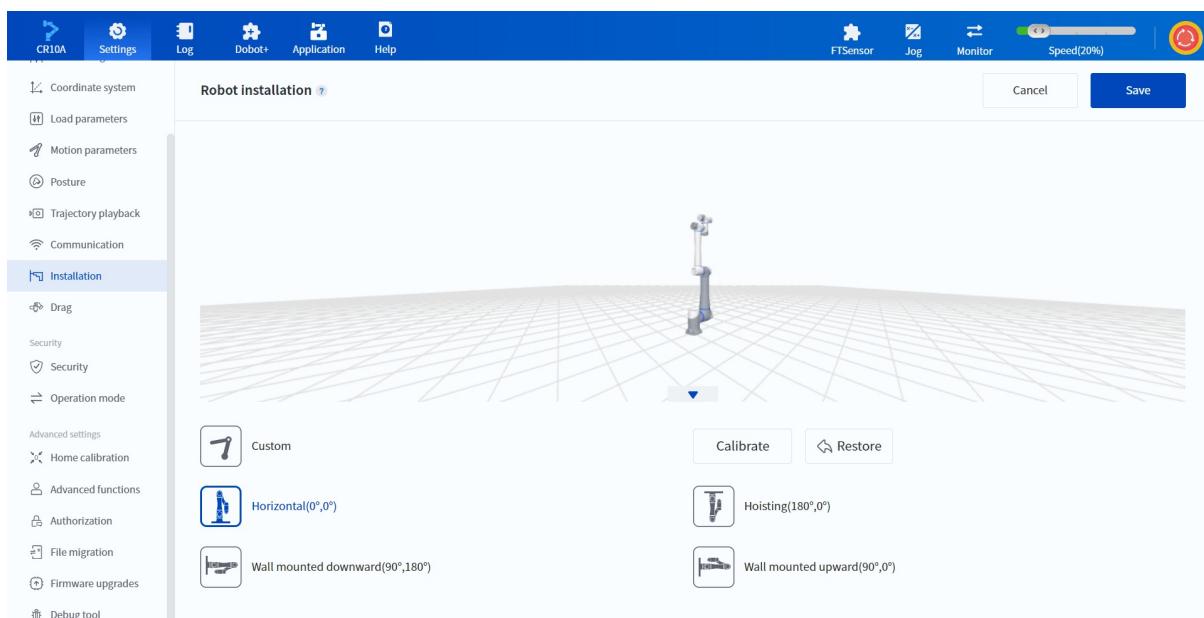
## 10.10 Installation settings

In general, the robot is installed on a flat table or floor, and in such cases, no action is required on this page. However, if the robot is installed in a ceiling-mounted, wall-mounted, or angled position, you will need to set the rotation and tilt angles while the robot is in a disabled status.

The primary purpose of installation settings is to inform the robot of the correct gravity direction and to ensure the 3D model in the software displays at the correct angle.

### ⚠ NOTICE

Incorrect installation settings may trigger abnormal collision detection alarms or cause the robot to become uncontrollable during dragging.



You can calibrate the settings manually or automatically.

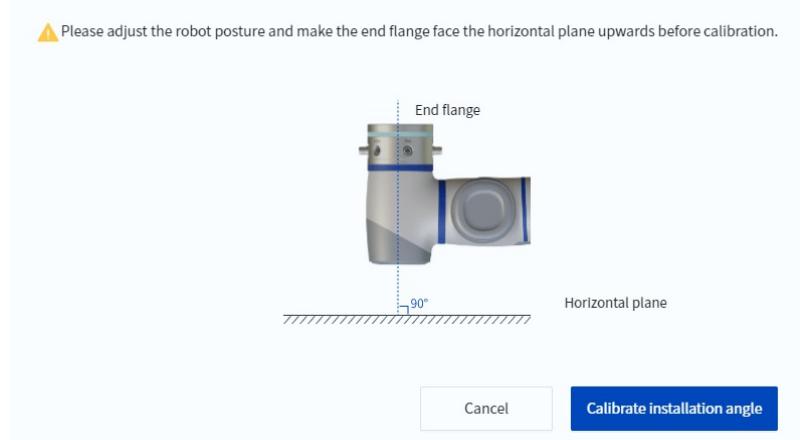
### Manual calibration

You can select a proper installation posture based on the actual condition, or select **Custom** to adjust the tilt and rotation angles below. Detailed descriptions for each posture can be viewed by clicking on the right of the page title.

- **Tilt angle:** The angle the robot rotates counterclockwise around the X-axis from its original position.
- **Rotation angle:** The angle the robot rotates counterclockwise around the Z-axis from its original position.

## Automatic calibration

After the robot is installed and enabled, click **Calibrate**, and follow the on-screen prompts to obtain the tilt and rotation angles.



Click **Restore defaults** to restore the calibrated angles to their default values.

After setting the installation angle, try entering the drag mode using the end button on the robot arm to verify if the drag function works properly. If not, reset the installation angle or contact technical support.

## 10.11 Drag settings

The Drag settings are mainly used to adjust the sensitivity of each joint during dragging operations. The lower the sensitivity, the greater resistance there is during dragging. Value range: 1% – 90%.

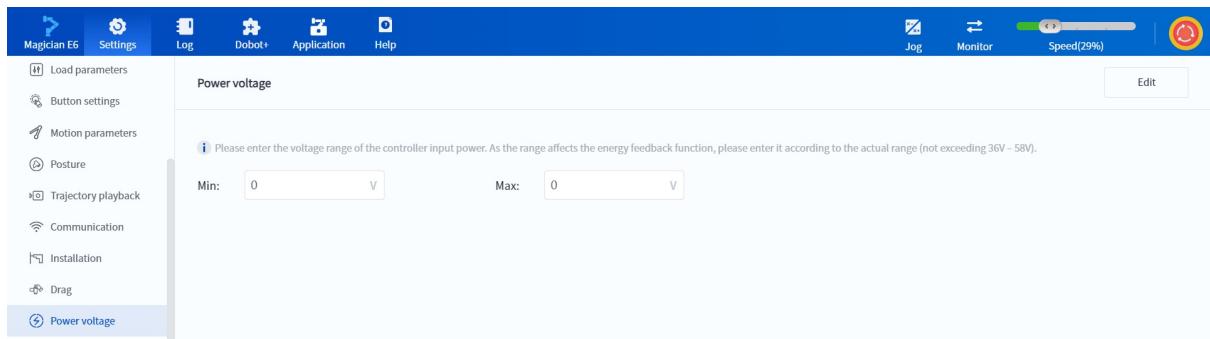


### NOTE

- The sensitivity settings only take effect when the joint dragging speed has not reached the limit. Once the dragging speed reaches the limit, reverse resistance will be applied to prevent overspeeding.
- The specific speed limits vary depending on the robot model. The speed limit for J1 to J3 is about 30°/s to 40°/s, while for J4 to J6, it is around 90°/s to 100°/s.

## 10.12 Power voltage (DC controller/Magician E6)

When the robot is connected to a DC controller or Magician E6, you need to set the power voltage in DobotStudio Pro. This voltage range is related to the energy regeneration function (used to release the electromotive force generated when the robot decelerates or brakes). Set the voltage according to the actual range of the input power to avoid overvoltage protection shutdown or damage to the controller.



Click **Settings > Power voltage**, and enter the actual input power's voltage range.

- When connected to CC262 DC controller, the allowable range is 30V – 60V, and the minimum value must be  $\leq$  the maximum value.
- When connected to Magician E6, the allowable range is 36V – 58V, and the minimum value must be  $\leq$  the maximum value.

## 10.13 Security settings

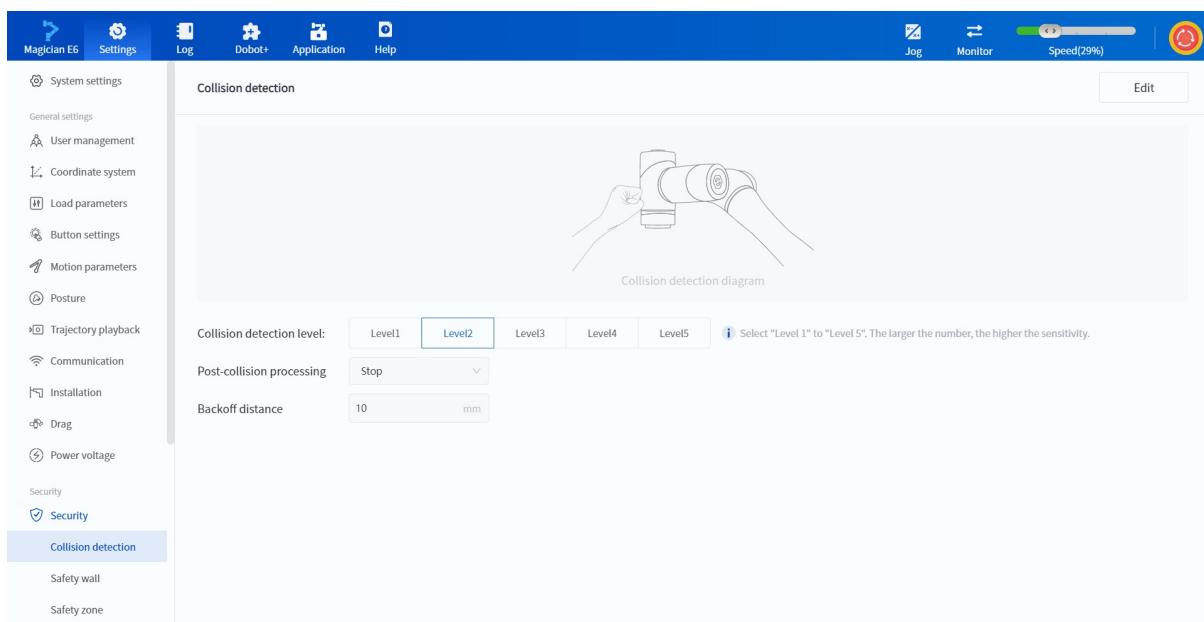
- [10.13.1 Collision detection \(Magician E6\)](#)
- [10.13.2 Safety limits \(CRA series\)](#)
- [10.13.3 Joint limits \(CRA series\)](#)
- [10.13.4 Safety wall](#)
- [10.13.5 Safety zone](#)
- [10.13.6 Safe home position](#)
- [10.13.7 Joint brake](#)

## 10.13.1 Collision detection (Magician E6)

The robot will automatically stop when a collision is detected during its movement. You can set the sensitivity of the collision detection and specify how the system should respond after a collision.

### NOTE

If you need to enable or disable collision detection, please contact technical support.



Please set the collision level according to your actual needs. The higher the level, the smaller the force required to trigger collision detection.

- Level 5 should only be used for scenarios where the robot is running at low speed. At this collision level, false triggers may occur if the robot is running with a load at high speed or high acceleration.
- When the robot is fully loaded, it is recommended to set the collision level to 3 or lower.

The response to a collision differs between jog mode and automatic running.

### Collision during jog mode

A software popup will notify you that a collision has been detected. In this case, you need to resolve the cause of the collision and click **Reset**. If you need to use the software to resolve the issue, you can click **Remind me in a minute** to temporarily close the popup (the popup will appear again after one minute).

## Collision detection

Robot arm collision detected! If not triggered by an actual collision, please check the load settings and installation angle settings.

[Remind me in a minute](#)

[Reset](#)

### Collision during automatic running

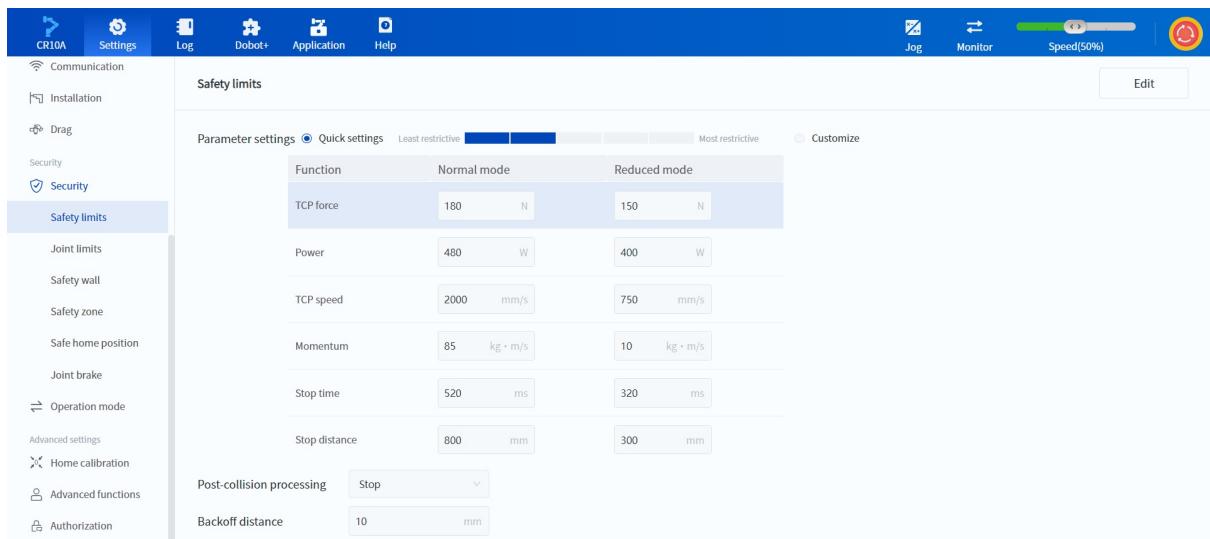
The robot will respond based on the selected **Post-collision processing** method:

- **Stop:** The robot stops running.
- **Pause:** The robot pauses, and a popup will appear (as shown in the figure below). After resolving the cause of the collision, you can choose to continue or stop based on the actual situation. While the robot is in a paused status after collision, you can also resume operation by short-pressing the hand-guiding button at the end of the robot arm.

Regardless of the handling method, after a collision, the robot will automatically retract a specified distance along the trajectory before the collision. The backoff distance can be set within the range of 0 mm to 50 mm (default: 10 mm).

## 10.13.2 Safety limits (CRA series)

To ensure the safety of both the robot and user, the robot system imposes limits on various motion parameters, and users can adjust these limits according to their needs.



### Parameter settings

Two types of settings are available: **Quick settings** and **Custom**.

- **Quick settings:** Choose from 5 preset levels (referred to as Level 1 to Level 5, where Level 1 has the loosest restrictions and Level 5 has the strictest).

#### NOTE

- Level 5 should only be used for scenarios where the robot is running at low speed. At this collision level, false triggers may occur if the robot is running with a load at high speed or high acceleration.
- When the robot is fully loaded, it is recommended to set the collision level to 3 or lower.
- The preset levels are recommendations and should not replace proper risk assessments.

- **Custom:** Enter specific values for each limit parameter. Switching back to **Quick settings** will reset the parameters to the preset values of the corresponding level.

The definition of each limiting parameter is as follows (all parameters must be set respectively for both Normal and Reduced mode):

- **TCP force:** Limits the maximum force on the robot's Tool Center Point (TCP).
- **Power:** Limits the maximum power output of the robot.
- **TCP speed:** Limits the maximum speed of the robot's TCP during motion.
- **Momentum:** Limits the robot's total momentum during motion.

- **Stop time:** Limits the maximum time required for the robot to stop after an alarm or emergency stop.
- **Stop distance:** Limits the maximum distance required for the robot to stop after an alarm or emergency stop.

When the robot plans its motion, it will adjust the maximum speed to ensure all of the above parameters remain within the set limits. If during operation, the **TCP force** or **Power** exceeds the set limits, the robot will be considered to have collided and will automatically stop while triggering a collision detection alarm.

#### NOTE

TCP speed, Momentum, Stop time, and Stop distance collectively limit the maximum TCP speed of the robot. The higher the parameters, the higher the robot's maximum speed.

The response to a collision differs between jog mode and automatic running.

#### Collision during jog mode

A software popup will notify you that a collision has been detected. In this case, you need to resolve the cause of the collision and click **Reset**. If you need to use the software to resolve the issue, you can click **Remind me in a minute** to temporarily close the popup (the popup will appear again after one minute).

#### Collision detection

Robot arm collision detected! If not triggered by an actual collision, please check the load settings and installation angle settings.

[Remind me in a minute](#)

[Reset](#)

#### Collision during automatic running

The robot will respond based on the selected **Post-collision processing** method:

- **Stop:** The robot stops running.
- **Pause:** The robot pauses, and a popup will appear (as shown in the figure below). After resolving the cause of the collision, you can choose to continue or stop based on the actual situation. While the robot is in a paused status after collision, you can also resume operation by short-pressing the hand-guiding button at the end of the robot arm.

Regardless of the handling method, after a collision, the robot will automatically retract a specified distance along the trajectory before the collision. The backoff distance can be set within the range of 0 mm to 50 mm (default: 10 mm).

You can click **Edit** to modify the parameters, and click **Save** to save the updated values after modification. Clicking **Cancel** can discard the changes. Clicking **Restore defaults** can reset the parameters to the factory default settings.

 **NOTE**

In Reduced mode, the values must always be lower than in Normal mode, otherwise the settings cannot be saved.

## 10.13.3 Joint limits (CRA series)

You can set the soft limits for each joint of the robot to restrict the range of motion for each joint.

The screenshot shows the Dobot software interface with the following details:

- Top Bar:** Includes icons for CR10A, Settings, Log, Dobot+, Application, Help, Jog, Monitor, and a speed slider set at 50%.
- Left Sidebar:** Lists various settings: Communication, Installation, Drag, Security (with a lock icon), Safety limits, Joint limits (which is selected and highlighted in blue), Safety wall, Safety zone, Safe home position, and Joint brake.
- Central Content:** A table titled "Joint limits" showing parameters for joints J1 through J6. The table has columns for Joint, Negative limit, Negative limit tolerance, Positive limit, and Positive limit tolerance.

Joint	Negative limit	Negative limit tolerance	Positive limit	Positive limit tolerance
J1	-360	+2°	360	-2°
J2	-360	+2°	360	-2°
J3	-164	+2°	164	-2°
J4	-360	+2°	360	-2°
J5	-360	+2°	360	-2°
J6	-360	+2°	360	-2°

- Bottom Note:** A small note states: "Tolerance value: When the actual value is greater than the function value plus the respective tolerance value, an alarm will be generated."

When the actual joint angle of the robot is less than the limit (Negative limit + Negative limit tolerance) or more than the limit (Positive limit + Positive limit tolerance), an alarm will be triggered and motion will stop.

The values for the negative and positive limits can be modified.

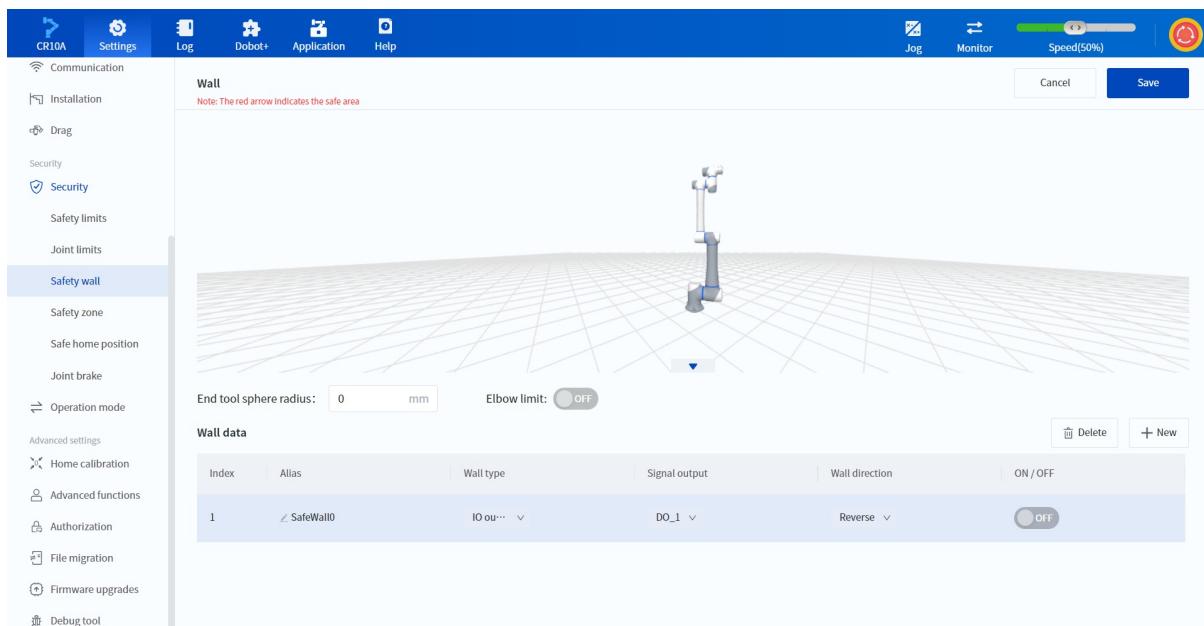
You can click **Edit** to modify the parameters, and click **Save** to save the updated values after modification. Clicking **Cancel** can discard the changes. Clicking **Restore defaults** can reset the parameters to the factory default settings.

## 10.13.4 Safety wall

DobotStudio Pro supports setting up to 8 safety walls, which define the robot's safe space based on the direction of the wall. When the tool sphere at the end of the robot arm (spherical space with a custom radius centred on the TCP) or the J3 joint (optional) approaches or exceeds the safety area defined by the safety wall, the robot arm triggers different actions based on the types of safety wall.

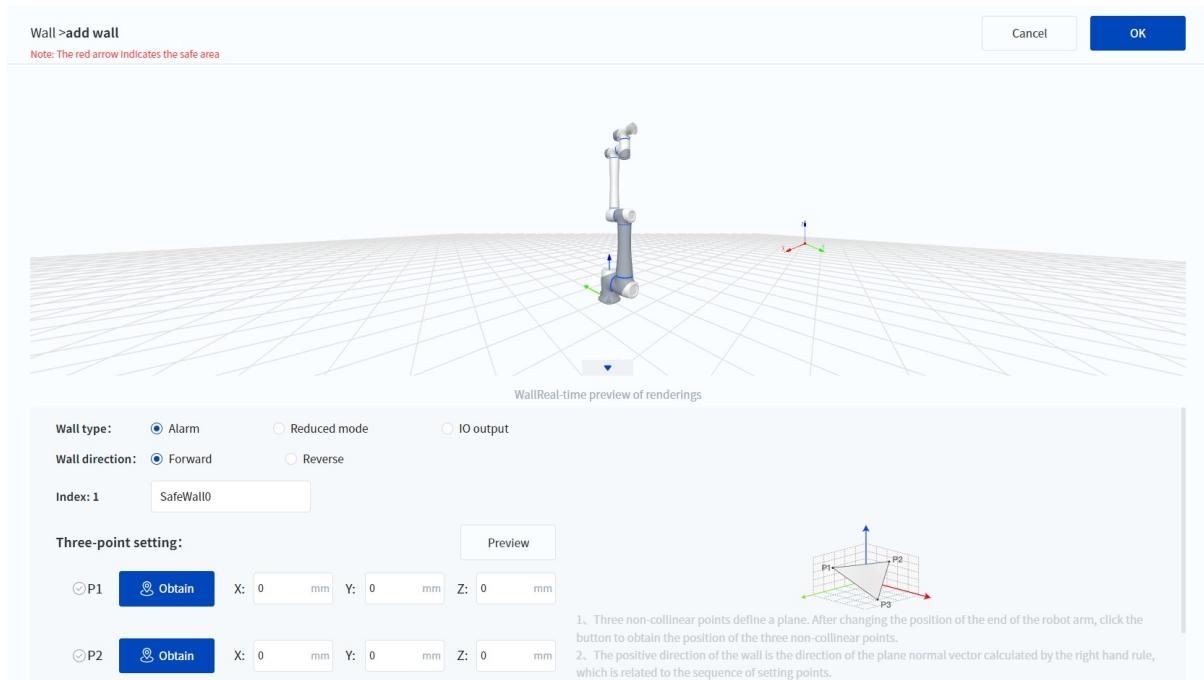
- **Alarm:** When approaching the safety boundary, if the current motion plan would cause the robot to exceed the safety zone, an alarm will be triggered, and the robot will stop.
- **Reduced mode:** When the robot exceeds the safety zone, it will enter reduced mode, operating at a slower speed.
- **IO output:** When the robot exceeds the safety zone, it will trigger a specified DO (Digital Output) without affecting its movement.

Multiple safety walls or safety zones can be active simultaneously, and the same DO can be set for all. Triggering any active safety wall or zone will initiate the corresponding action.



### Adding safety wall

Click **+ New** to add a safety wall. The safety wall is a user-defined plane, which can be determined by three non-collinear points on the plane, referred to as P1, P2, and P3.



1. Jog or drag the robot to P1, and click **Obtain** to get the coordinates of P1.
2. Use the same method to get the coordinates of P2 and P3, then click **Preview** to see the generated safety wall in the simulation area above.

#### **i** NOTE

- You can also manually enter or modify the Cartesian coordinates of the points.
- After modification, you need to click **Preview** to update the simulation display.

3. Set the wall type.
4. Set the wall direction. In the simulation area, you can view the wall direction. The arrow indicates the safe side of the wall, while the opposite side is the restricted side.

#### **i** NOTE

The positive direction of the wall is calculated based on the vector from P1 -> P2 -> P3, using the right-hand rule to determine the plane's normal vector.

5. Click **OK** to add the safety wall.

## Modifying safety wall

Wall data					
Index	Alias	Wall type	Signal output	Wall direction	ON / OFF
1	SafeWall0	IO out...	DO_1	Reverse	<input checked="" type="radio"/> OFF

You can modify all safety wall properties except for the **Index**. **Signal output** is only configurable when the wall type is **IO output**. The switch on the right controls whether the safety wall is effective (this can only be operated when the robot is in the disabled status). Only effective safety walls will interfere with the robot and be displayed in the simulation area.

Select a safety wall and click  **Delete** to remove the selected wall.

## Advanced settings



### End tool sphere radius

Specify the radius of the end tool sphere (a spherical space with the TCP as the center and a custom radius). When this spherical space interferes with the restricted zone of the safety wall, it will trigger the wall action. Setting the radius to 0 means only the TCP will interfere with the restricted zone.

### Elbow limit

When this option is set to **ON**, the J3 joint will also interfere with the restricted zone. When it is set to **OFF**, only the end tool sphere will interfere with the restricted zone.

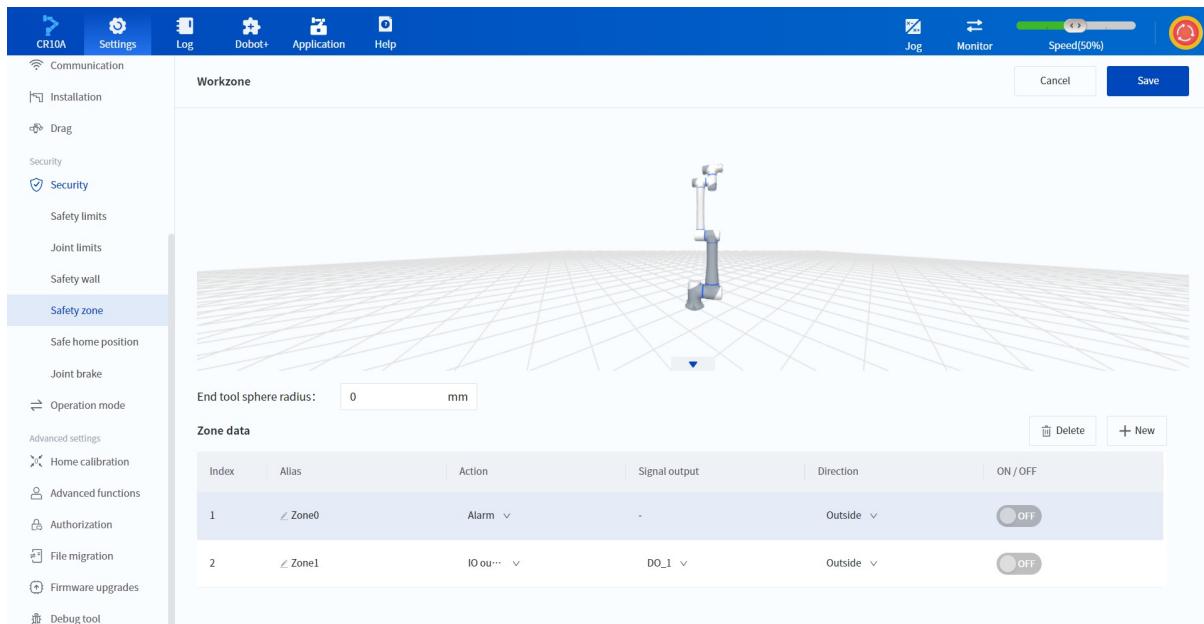
## 10.13.5 Safety zone

DobotStudio Pro supports setting up to 6 safety zones, defining either the inside or outside of the specified area as the robot's safety zone. When the **end-of-arm tool sphere (a spherical space with the TCP as its center and a custom radius)** approaches or exceeds the safety zone, different actions are triggered based on the safety zone's behavior type.

- **Alarm:** When approaching the safety boundary, if the current motion plan would cause the robot to exceed the safety zone, an alarm will be triggered, and the robot will stop.
- **Reduced mode:** When the robot exceeds the safety zone, it will enter reduced mode, operating at a slower speed.
- **IO output:** When the robot exceeds the safety zone, it will trigger a specified DO (Digital Output) without affecting its movement.

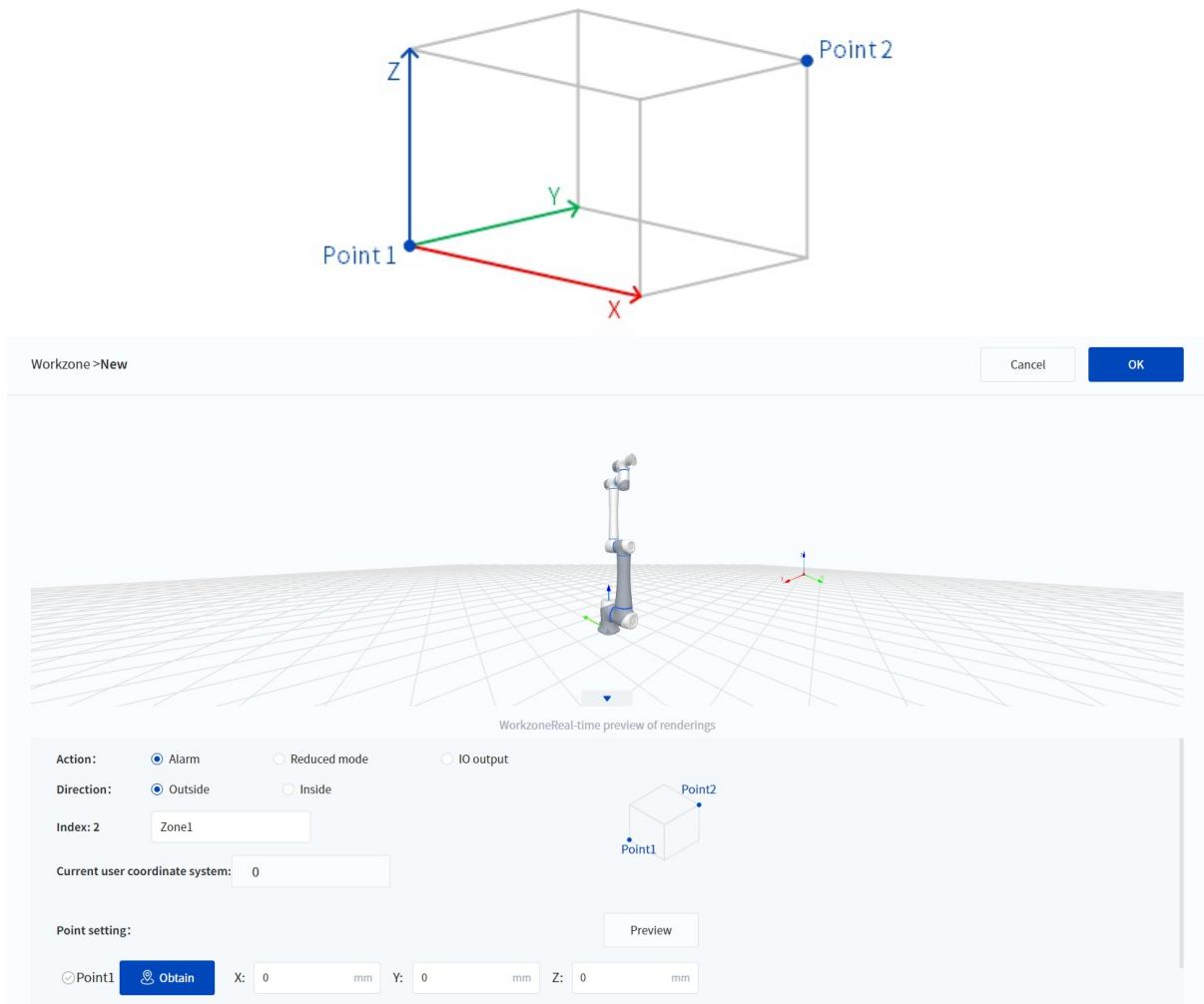
Multiple safety zones or safety walls can be active simultaneously, and the same DO can be set for all.

Triggering any active safety zone or wall will initiate the corresponding action.



### Adding safety zone

Click **+ New** to add a safety zone. The safety zone is cubic, and you need to teach the coordinates of two diagonal vertices (Point 1 and Point 2) of the safety zone. The direction of the cube's edges is determined by the specified user coordinate system, as shown in the figure below.



1. Jog or drag the robot to Point 1, and click **Obtain** to get the coordinates of Point 1.
2. Use the same method to get the coordinates of Point 2.
3. Select the **Current user coordinate system**, and click **Preview** to see the generated safety zone in the simulation area above.

**NOTE**

- You can also manually enter or modify the Cartesian coordinates of the points.
- After modification, you need to click **Preview** to update the simulation display.

4. Set the robot's behavior when the end tool sphere exceeds the safety zone.
5. Set the safety direction. **Outside** means the zone outside the cube is the safety zone. **Inside** means the zone inside of the cube is the safety zone.
6. Click **OK** to add the safety zone.

## Modifying safety zone

Zone data					
Index	Alias	Action	Signal output	Direction	ON / OFF
1	Zone0	Alarm ▾	-	Outside ▾	<input checked="" type="button"/> OFF
2	Zone1	IO out... ▾	DO_1 ▾	Outside ▾	<input checked="" type="button"/> OFF

You can modify all safety zone properties except for the **Index** and **Zone shape**. **Signal output** is only configurable for safety zones with **IO output** as the **Action**. The switch on the right controls whether the safety zone is effective (this can only be operated when the robot is in the disabled status). Only effective safety zones will interfere with the robot and be displayed in the simulation area.

Select a safety zone and click  **Delete** to remove the selected zone.

## Advanced settings

End tool sphere radius:

0

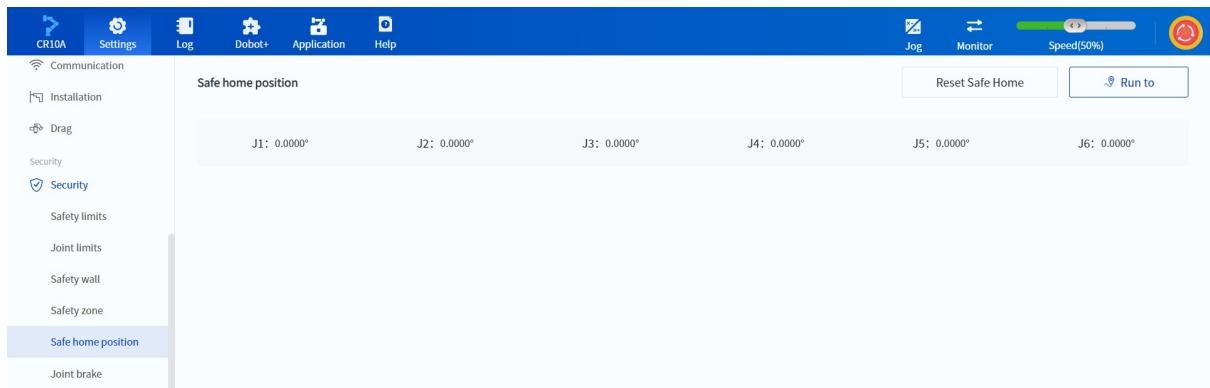
mm

### End tool sphere radius

Specify the radius of the end tool sphere (a spherical space with the TCP as the center and a custom radius). When this spherical space interferes with a non-safety zone, it will trigger the safety zone action. Setting the radius to 0 means only the TCP will interfere with the restricted zone.

## 10.13.6 Safe home position

The safe home position is a user-defined posture, with the default being the home posture, where all joint angles are set to 0. It is recommended that users modify it according to the specific application. When the robot is at the safe home position, a safe home position status signal can be output through the [safety I/O](#), [system I/O](#), or [Modbus](#). Users can use this signal to implement logic, such as allowing the project to run only when the robot is in the safe home position.



- Long-pressing **Run to** can move the robot to the safe home position.
- Clicking **Reset Safe Home** can modify the safe home position.

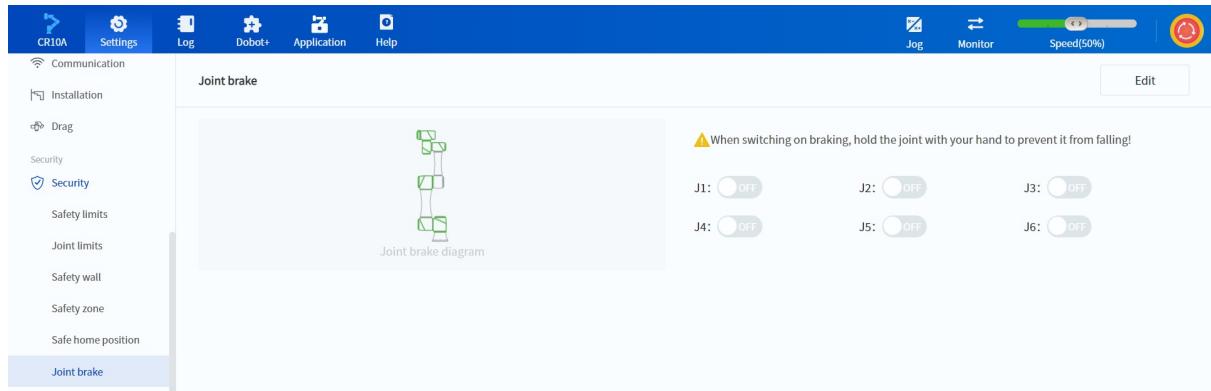
You can manually enter the angles for each joint, or move the robot to the desired posture and then click **Obtain** to get the current angles of all joints. Clicking **Restore defaults** can reset the safe home position to the default position.

After confirming all joint angles, click **Save** to update the safe home position.



## 10.13.7 Joint brake

When the robot is disabled, all joints will automatically brake to prevent the joints from moving. Joint brake keeps the motor position locked to ensure that the moving part of the machine will not move due to its self weight or external force. In an emergency, you can release the joint brakes through this page and move the corresponding joints by dragging.



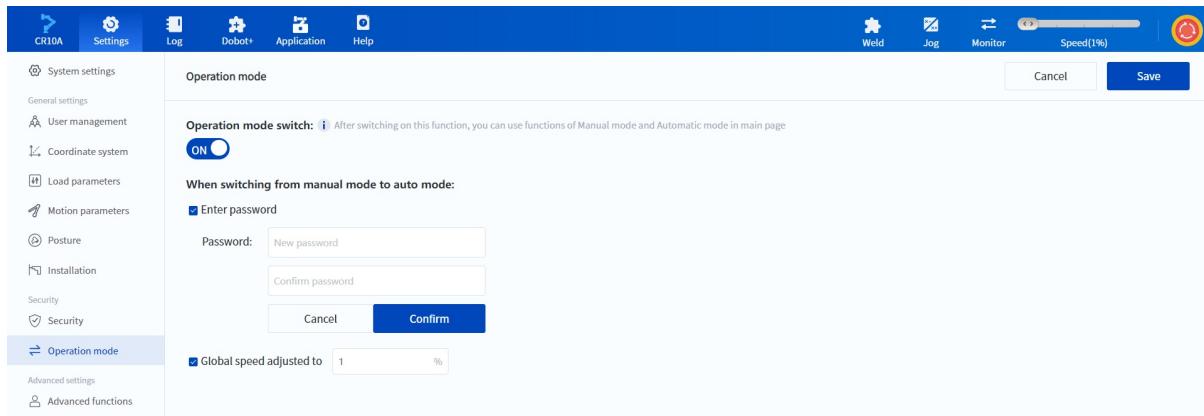
If you need to drag joints, you can switch on the joint brake function, that is, hold the joint with your hand after the robot is disabled, switch on the button for the joint you wish to move, and click **Save**.

### ⚠ WARNING

When switching on the function, be sure to hold the joint with your hand to prevent it from falling.

## 10.14 Operation mode settings

The manual/automatic mode switching function is disabled by default when the robot is shipped (CR20A is set to manual mode by default), it can be enabled on this page. The Manual/automatic mode is mainly used to enhance safety in field applications, so please decide whether to enable it based on the result of risk assessment.

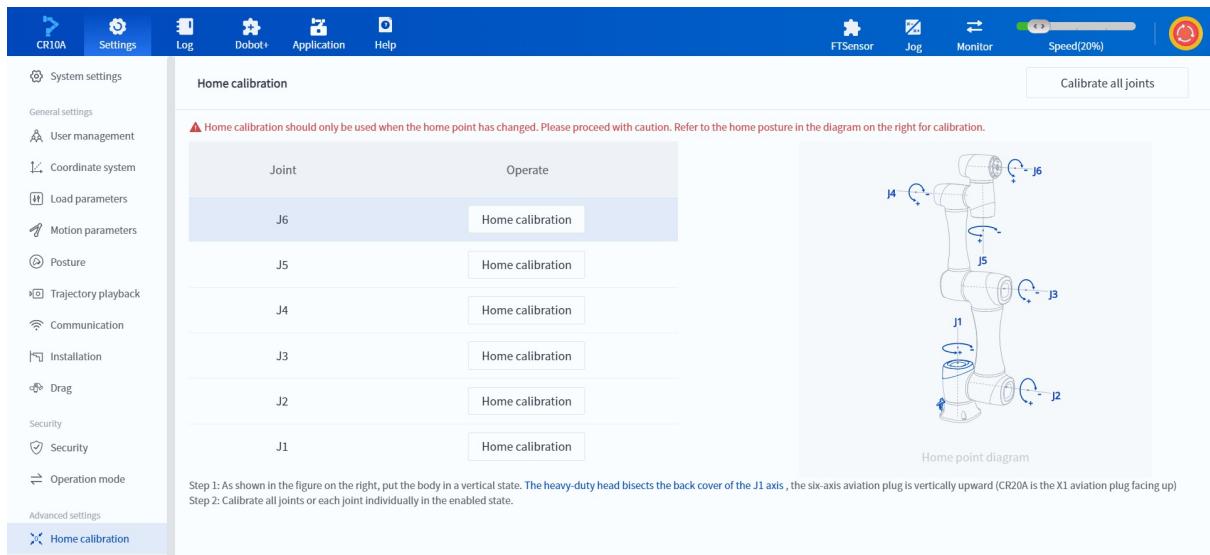


Once the operation mode switch is enabled, you can select **Enter password** when switching from Manual mode to Automatic mode. If this option is selected, you need to set a password (no initial password).

If the **Global speed adjustment** function is selected, the global speed will be automatically adjusted to the set value when switching to Automatic mode. The **Global speed adjustment** only accepts integer values between 1 and 100.

## 10.15 Home calibration

When components such as the robot's motor, reducer, or other parts are replaced, or when the robot collides with a workpiece, the home position of the robot may change. In such cases, home calibration is required.



Follow the on-screen prompts, first move the robot to the home posture (you can use the zero-point stickers on each joint of the robot to calibrate, see the corresponding hardware guide for details).

Then, while the robot is in the enabled status, click the **Calibrate all joints** button at the upper right of the page to calibrate all joints, or click the **Home calibration** button to calibrate a single joint.

### **i** NOTE

Home calibration should only be performed when the home position has changed. Please proceed with caution.

#### **Dangerous**

After calibration, the following problems may occur:

1. The point will deviate from the original teaching position
2. The robotic arm will be automatically enabled

**Cancel**

**Home calibration**

After calibration is successful, the robot must be disabled for the changes to take effect. You can then check the joint coordinates in the control panel, where the values for J1 to J6 will all be set to 0.

## **10.16 Advanced functions**

When advanced functions need to be enabled or disabled due to on-site requirements, you can configure them on this page. For explanations of each function, please refer to the on-screen prompts. If there are no specific needs, it is recommended to keep the default settings.

## Advanced settings

### Full parameter calibration

ON

-  When it is ON, TCP accuracy will be improved, especially in the case of large posture angle changes.

### Vibration suppression

OFF



-  When it is ON, the robot's vibration suppression effect will be improved.

### Start/Stop vibration suppression

ON

-  When it is ON, the robot's vibration suppression effect after starting or stopping will be improved.

Frequency:

18 HZ

### Torque over-limit warning

ON

-  When it is ON, torque over-limit warning will pop up in a speech bubble.

### Jog while paused

ON

-  When enabled, jogging, dragging, enabling, and disabling operations can be performed even while the robot is paused.

### Trajectory recovery

ON

If disabled, the robot will move directly from its current position to the command endpoint at script speed, resulting in a trajectory different from the original.

-  When enabled, if the robot moved while paused, it will first return to the paused position at jog speed and then continue along the original trajectory at script speed.(The current version does not support modification)

### Running or resuming running script

OFF

-  When running or resuming running script, the global speed is adjusted to

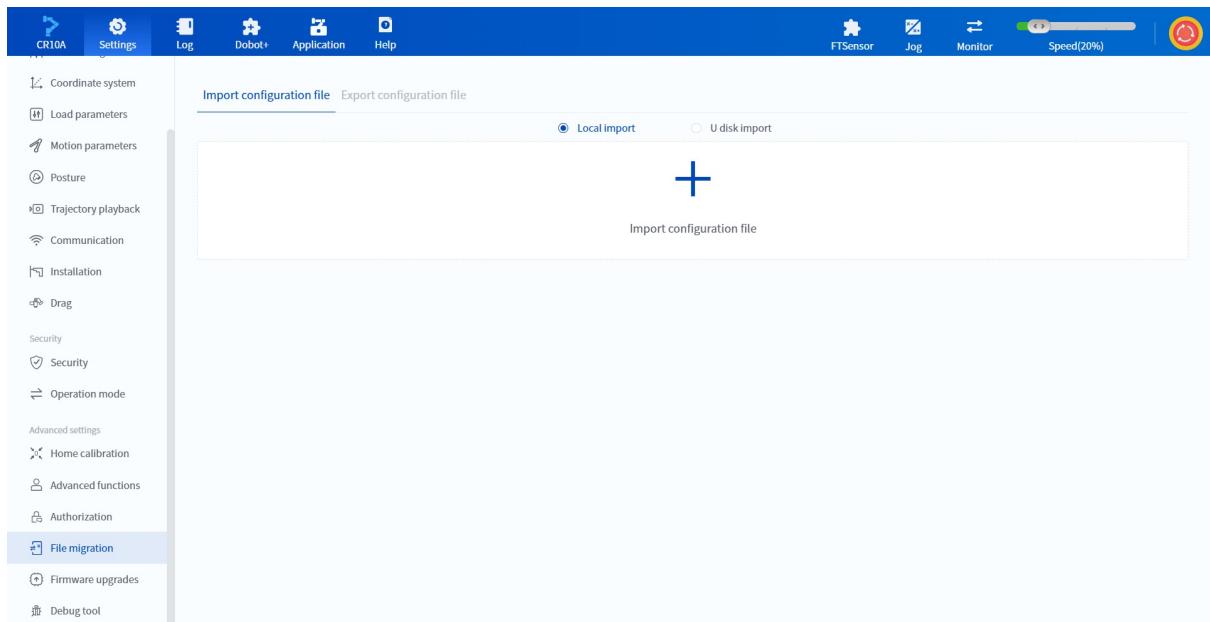
 NOTE

For details on the **Jog while paused** and **Trajectory recovery** functions, please refer to the [Trajectory recovery](#) section.

## 10.17 File migration

DobotStudio Pro supports the import and export function for the robot's internal files (such as program files, configuration files, etc.). It can be used for system backups, device replication, and other scenarios, and supports import/export through local storage or USB drives.

### Importing configuration file

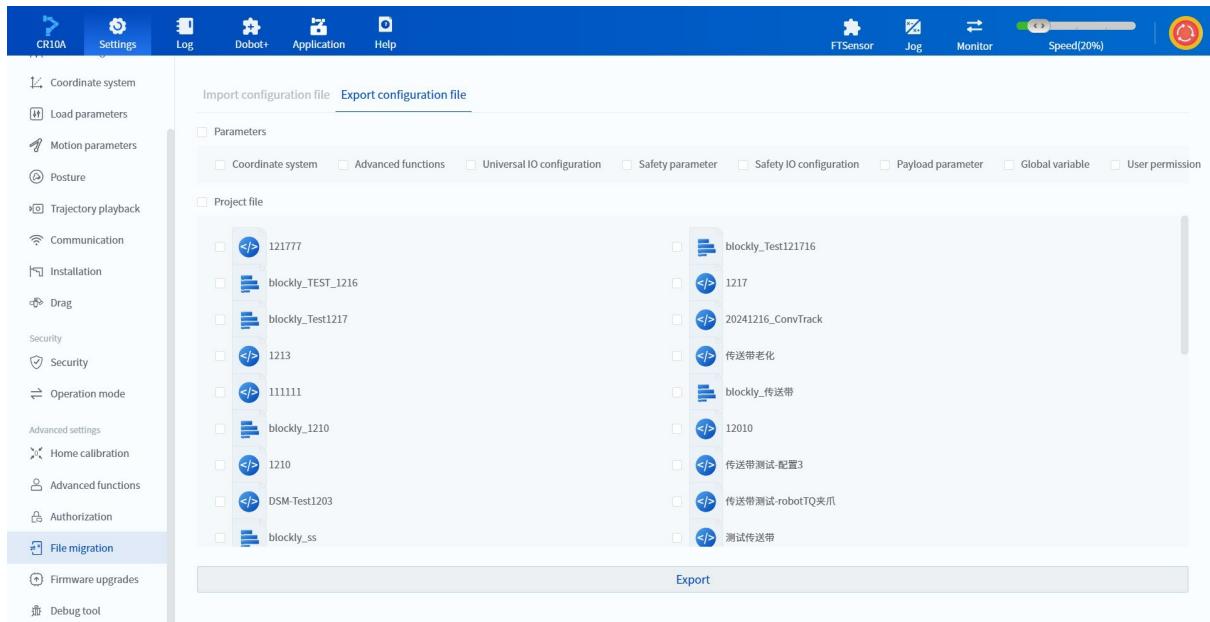


You can click **Settings > Firmware upgrade** to access the **Import configuration file** page, and import the configuration file saved on your local device or USB drive via **Local import** or **USB import**.

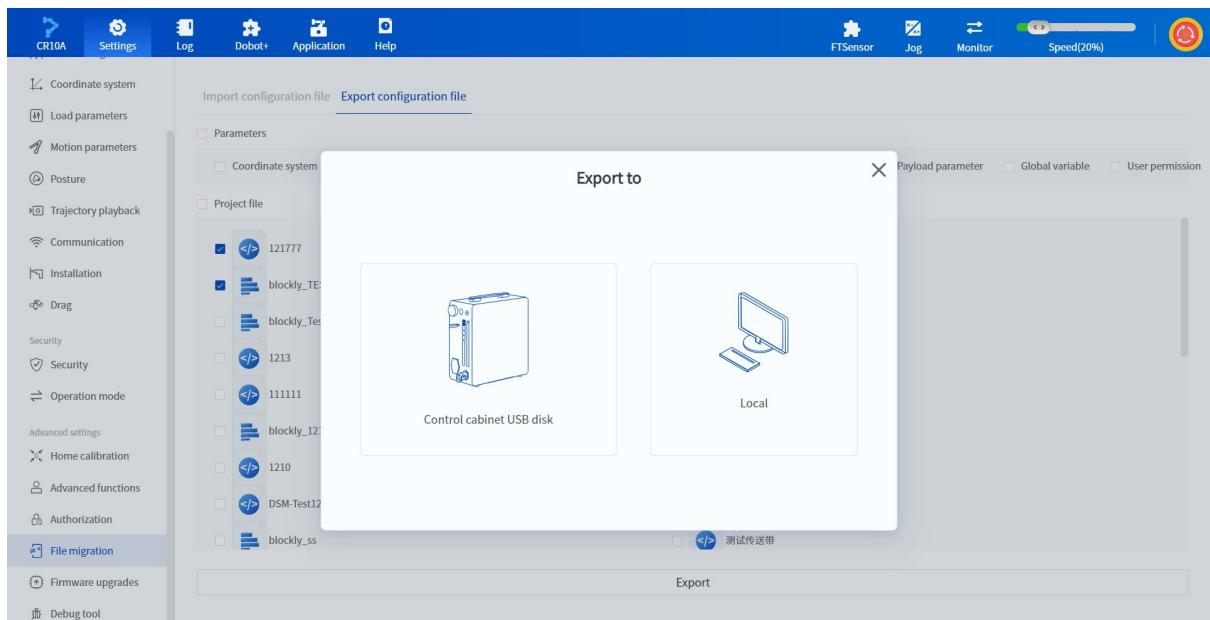
#### NOTE

- Before importing the configuration file, the robot arm must first be disabled, after which the operation can continue.
- Only compressed files in the root directory of the USB drive will be displayed during import, so please select the correct file.

### Exporting configuration file



You can click the **Export configuration file** tab, select the parameters and project files you want to export, and click the **Export** button to open the **Export to** page.



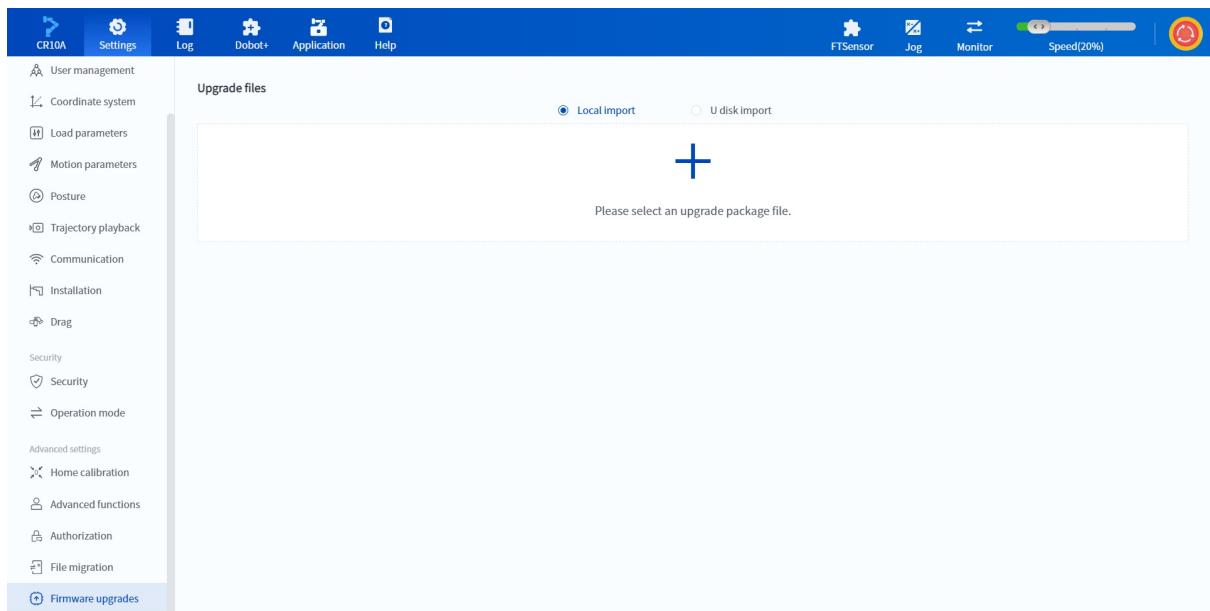
You can choose to export the system configuration file to the **Controller USB** or **Local**.

### **NOTE**

- If you choose to export to the Controller USB, the file will be exported to the root directory of the USB.
- When using the Controller USB for import/export, you can insert the USB drive into any USB port on the controller. If both USB ports have a USB drive inserted, the system will prioritize reading from the first USB drive it detects.
- Project exports are limited to 100 items.

## 10.18 Firmware upgrade

Through the **Firmware upgrade** page, you can upgrade the robot's firmware to the latest version or roll back to a previous version.

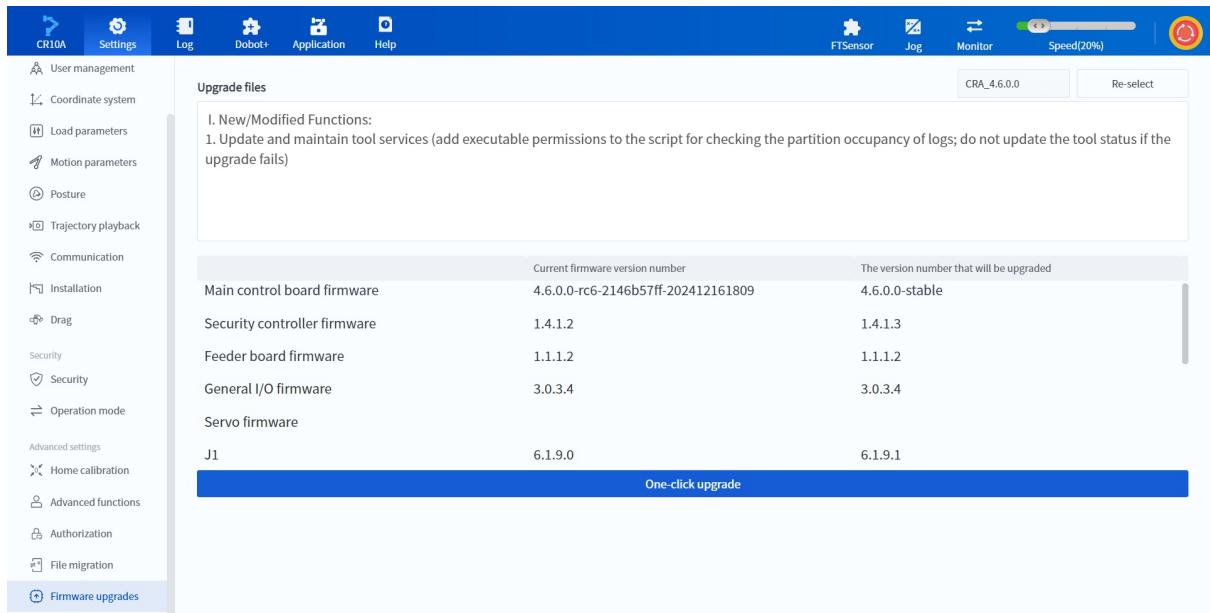


You can click **Settings > Firmware upgrade**, and import the upgrade package file saved on your local device or USB drive via **Local import** or **USB import**.

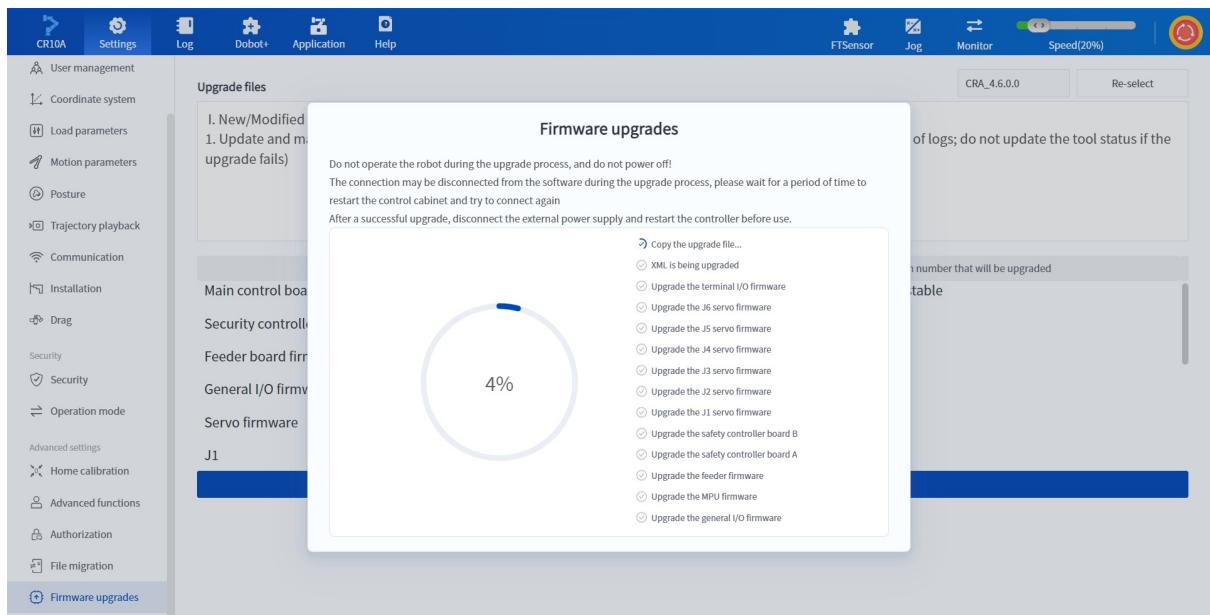
### NOTE

- Before importing the upgrade package file, the robot arm must first be disabled, after which the operation can continue. During the upgrade, the robot arm should not be powered off.
- When upgrading via **USB import**, it is recommended to insert the USB drive into a USB hub without a network interface.
- Only compressed files in the root directory of the USB drive will be displayed during import, so please select the correct file.
- Only users with administrator permission can perform the **firmware upgrade**, and this permission cannot be configured.

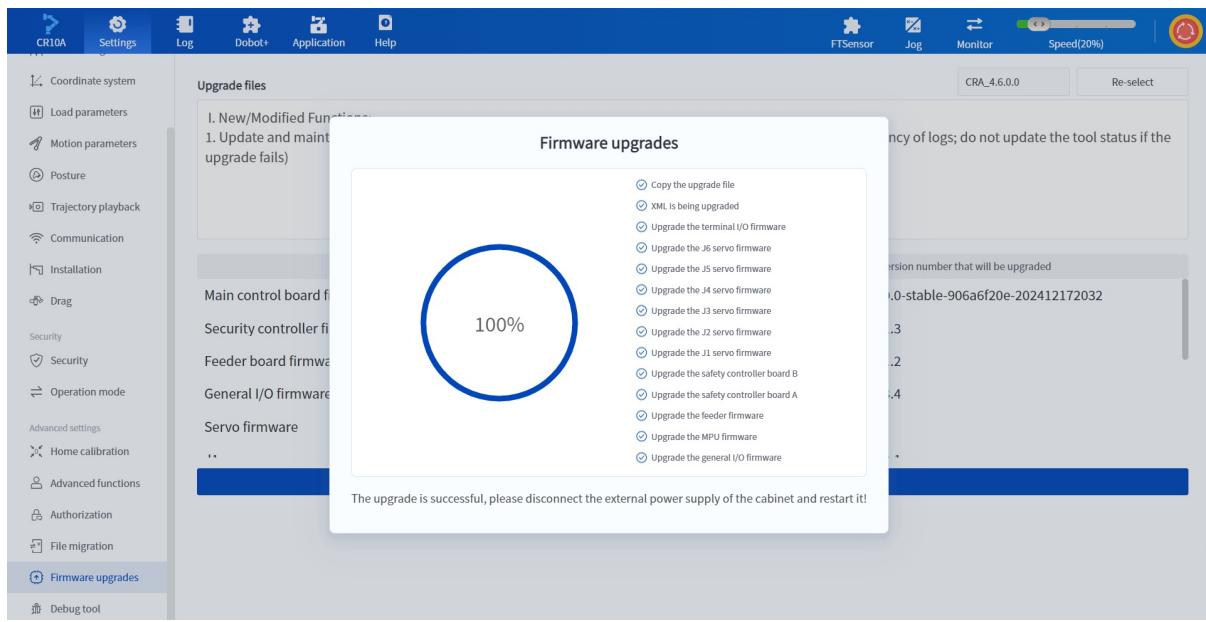
After the upgrade package file is imported, it will be decompressed locally. Once decompression is complete, the old and new firmware version information and upgrade content will be displayed.



Click **One-click upgrade** to start the firmware upgrade. The progress will be shown, and other functions will be unavailable during the upgrade.



- If "Failed to copy upgrade files!" pops up during the upgrade, click **Re-upgrade** to attempt the one-click upgrade again.
- If an error occurs (other than "Failed to copy upgrade files!") during the upgrade, refer to the error message returned for troubleshooting.
- After the upgrade is complete, a success message will be displayed. You will need to turn off the controller via the power switch and restart it. After restarting the controller, reconnect the robot to DobotStudio Pro.



### NOTE

To upgrade the firmware version, it is recommended to prioritize using the Robot Maintenance Tool.

# Appendix A Modbus Register Definition

## 1 Modbus introduction

Modbus is a serial communication protocol that allows robots to communicate with external devices. When controlling a robot via external devices (such as PLC), the external device serves as the Modbus master, while the robot serves as the Modbus slave.

Modbus Solutions:

- **Modbus-RTU:** A compact representation of data using binary format. Uses a cyclic redundancy check (CRC) for error checking.
- **Modbus-ASCII:** A representation based on ASCII characters, providing a readable and redundant format. Uses longitudinal redundancy check (LRC) for error checking.
- **Modbus-TCP:** Connects via TCP/IP using the TCP protocol. This method does not require checksum calculation.
- **Modbus-RTU-over-TCP:** A data packet carrying RTU content transmitted over TCP.

Currently, we offer solutions for Modbus-TCP and Modbus-RTU-over-TCP.

Dobot six-axis collaborative robots support the following Modbus addresses, according to the Modbus protocol.

- **00001 – 09999:** Coil register, used to control the robot; value: 0 or 1; read/write.
- **10001 – 19999:** Contact register, used to get the robot status; value: 0 or 1; read only.
- **30001 – 39999:** Input register, used to obtain real-time feedback from the robot; value: up to 64-bit double-precision floating-point numbers; read only.
- **40001 – 49999:** Holding register, used for robot PLC interaction; value: up to 64-bit double-precision floating-point numbers; read/write.

The Modbus function codes for different types of registers follow the standard Modbus protocol:

Register type	Read register	Write single register	Write multiple registers
Coil register	01	05	0F
Contact (discrete input) register	02	-	-
Input register	04	-	-
Holding register	03	06	10

The robot provides two tables for external device access. **map1** can be accessed via port 502 (Modbus-TCP) and port 503 (RTU-over-TCP). **map2** can be accessed via port 1502 (Modbus-TCP) and port 1503 (RTU-over-TCP). The resource details are as follows:

Parameter/Slave	502	503	1502	1503
Coil quantity	10000	10000	10000	10000
Discrete input quantity	10000	10000	10000	10000
Input register quantity	10000	10000	10000	10000
Holding register quantity	10000	10000	10000	10000
Located in map	map1	map1	map2	map2
Bound port	502	503	1502	1503
Modbus protocol type	TCP	RTU-over-TCP	TCP	RTU-over-TCP
Max supported masters	20	20	20	20
Slave ID	1	1	1	1

Some addresses in **map 1** are reserved by the robot system by default, as detailed in the register definition below. **map 2** is currently empty and can be used by users as needed.

## 2 Coil register (map1, control robot)

PLC address	Script address (Get/SetCoils)	Register type	Function
00001	0	Bit	Start
00002	1	Bit	Stop
00003	2	Bit	Pause
00004	3	Bit	Enable
00005	4	Bit	Disable
00006	5	Bit	Clear alarm
00007	6	Bit	Enter drag mode
00008	7	Bit	Exit drag mode
00009	8	Bit	Switch to Auto mode
00010	9	Bit	Switch to Manual mode
00011 – 01024	10 – 1023	Bit	Reserved
01025 – 09999	1024 – 9998	Bit	User-defined

### 3 Contact register (map1, robot status)

PLC address	Script address (GetInBits)	Register type	Function
10001	0	Bit	Running status
10002	1	Bit	Stop status
10003	2	Bit	Pause status
10004	3	Bit	Safe home status
10005	4	Bit	SafeSkin pause status
10006	5	Bit	Idle status
10007	6	Bit	Power-on status
10008	7	Bit	Enabling status
10009	8	Bit	Alarm status
10010	9	Bit	Collision status
10011	10	Bit	Drag status
10012	11	Bit	Recovery mode status
10013 – 19999	12 – 9998	Bit	Undefined

### 4 Input register (map1, robot real-time feedback data)

The PLC addresses **30001 – 39999** and **31722 – 39999** have undefined register functions (except for **32001 – 32067**, which stores the U16-type controller SN, robot SN, and characters for the safety checksum). The byte order is in little-endian format.

PLC address	Data type	Number of values	Size in bytes	Script address (GetInRegs)	Type	Function
31000	unsigned short	1	2	999	U16	Data validity
31001	unsigned short	1	2	1000	U16	Total message length in bytes
31002 – 31004	-	-	-	1001 – 1003	-	Reserved
31005 – 31008	uint64	1	8	1004 – 1007	U64	Digital input, see <a href="#">DI/DO description</a>
31009 – 31012	uint64	1	8	1008 – 1011	U64	Digital output, see <a href="#">DI/DO description</a>

31013 – 31016	uint64	1	8	1012 – 1015	U64	Robot mode, see <a href="#">RobotMode description</a>
31017 – 31020	uint64	1	8	1016 – 1019	U64	Unix timestamp (unit: ms)
31021 – 31024	-	-	-	1020 – 1023	-	Reserved
31025 – 31028	uint64	1	8	1024 – 1027	U64	Memory structure test standard value 0x0123 4567 89AB CDEF
31029 – 31032	-	-	-	1028 – 1031	-	Reserved
31033 – 31036	double	1	8	1032 – 1035	F64	Speed ratio
31037 – 3104	-	-	-	1036 – 1039	-	Reserved
31041 – 31044	double	1	8	1040 – 1043	F64	Control board voltage
31045 – 31048	double	1	8	1044 – 1047	F64	Robot voltage
31049 – 31052	double	1	8	1048 – 1051	F64	Robot current
31053 – 31096	-	-	-	1052 – 1095	-	Reserved
31097 – 31120	double	6	48	1096 – 1119	F64	Target joint position
31121 – 31144	double	6	48	1120 – 1143	F64	Target joint speed
31145 – 31168	double	6	48	1144 – 1167	F64	Target joint acceleration
31169 – 31192	double	6	48	1168 – 1191	F64	Target joint current
31193 – 31216	double	6	48	1192 – 1215	F64	Target joint torque
31217 – 31240	double	6	48	1216 – 1239	F64	Actual joint position
31241 – 31264	double	6	48	1240 – 1263	F64	Actual joint speed
31265 – 31288	double	6	48	1264 – 1287	F64	Actual joint current
						TCP sensor force

31289 – 31312	double	6	48	1288 – 1311	F64	(calculate through six-axis force)
31313 – 31336	double	6	48	1312 – 1335	F64	TCP actual Cartesian coordinates
31337 – 31360	double	6	48	1336 – 1359	F64	TCP actual speed in Cartesian coordinate system
31361 – 31384	double	6	48	1360 – 1383	F64	TCP force (calculate through joint current)
31384 – 31408	double	6	48	1384 – 1407	F64	TCP target Cartesian coordinates
31409 – 31432	double	6	48	1408 – 1431	F64	TCP target Cartesian speed
31433 – 31456	double	6	48	1432 – 1455	F64	Joint temperature
31456 – 31480	double	6	48	1456 – 1479	F64	Joint control mode. 8: Position mode; 10: Torque mode
31481 – 31504	double	6	48	1480 – 1503	F64	Joint voltage
31505 – 31506	-	-	-	1504 – 1505	-	Reserved
31507	char	1	1	1506 low byte	I8	User coordinate system
31507	char	1	1	1506 high byte	I8	Tool coordinate system
31508	char	1	1	1507 low byte	I8	Algorithm queue running flag
31508	char	1	1	1507 high byte	I8	Algorithm queue pause flag
31509	char	1	1	1508 low byte	I8	Joint speed ratio
31509	char	1	1	1508 high byte	I8	Joint acceleration ratio
31510	char	1	1	1509 low byte	I8	Joint jerk ratio
31510	char	1	1	1509 high byte	I8	Cartesian position speed ratio
31511	char	1	1	1510 low byte	I8	Cartesian posture speed ratio
31511	char	1	1	1510 high byte	I8	Cartesian position acceleration ratio

31512	char	1	1	1511 low byte	I8	Cartesian posture acceleration ratio
31512	char	1	1	1511 high byte	I8	Cartesian position jerk ratio
31513	char	1	1	1512 low byte	I8	Cartesian posture jerk ratio
31513	char	1	1	1512 high byte	I8	Brake status, see <a href="#">BrakeStatus description</a>
31514	char	1	1	1513 low byte	I8	Enabling status
31514	char	1	1	1513 high byte	I8	Drag status
31515	char	1	1	1514 low byte	I8	Running status
31515	char	1	1	1514 high byte	I8	Alarm status
31516	char	1	1	1515 low byte	I8	Jog status
31516	char	1	1	1515 high byte	I8	Robot type, see <a href="#">RobotType description</a>
31517	char	1	1	1516 low byte	I8	Button board drag signal
31517	char	1	1	1516 high byte	I8	Button board enabling signal
31518	char	1	1	1517 low byte	I8	Button board record signal
31518	char	1	1	1517 high byte	I8	Button board playback signal
31519	char	1	1	1518 low byte	I8	Button board gripper control signal
31519	char	1	1	1518 high byte	I8	Six-axis force online status
31520	char	1	1	1519 low byte	I8	Collision status
31520	char	1	1	1519 high byte	I8	(SafeSkin) forearm close to pause status
31521	char	1	1	1520 low byte	I8	(SafeSkin) J4 close to pause status
31521	char	1	1	1520 high byte	I8	(SafeSkin) J5 close to pause status
31522	char	1	1	1521 low byte	I8	(SafeSkin) J6 close to pause status
				1521 high		

				byte		
31523 – 31552	-	-	-	1522 – 1551	-	Reserved
31553 – 31556	double	1	8	1552 – 1555	F64	Reserved
31557 – 31560	uint64	1	8	1556 – 1559	U64	Current algorithm queue ID
31561 – 31584	double	6	48	1560 – 1583	F64	Actual torque
31585 – 31588	double	1	8	1584 – 1587	F64	Payload (kg)
31589 – 31592	double	1	8	1588 – 1591	F64	X-directional eccentric distance (mm)
31593 – 31596	double	1	8	1592 – 1595	F64	Y-directional eccentric distance (mm)
31597 – 31600	double	1	8	1596 – 1599	F64	Z-directional eccentric distance (mm)
31601 – 31624	double	6	48	1600 – 1623	F64	User coordinates
31625 – 31648	double	6	48	1624 – 1647	F64	Tool coordinates
31649 – 31652	double	1	8	1648 – 1651	F64	Trajectory playback running index
31653 – 31676	double	6	48	1652 – 1675	F64	Six-axis force original value
31677 – 31692	double	4	32	1676 – 1691	F64	[qw,qx,qy,qz] Target quaternion
31693 – 31708	double	4	32	1692 – 1707	F64	[qw,qx,qy,qz] Actual quaternion
31709	unsigned short	1	2	1708	U16	Manual/Automatic status. 1: Manual; 2: Automatic; 0: Mode switch is OFF
31710 – 31721	double	1	24	1709 – 1720	F64	Reserved
			1440			Totally 1440 bytes

PLC address	Script address (GetInRegs)	Type	Function
32001 – 32032	2000 – 2031	U16	1st to 32nd characters of the controller SN
32033 – 32065	2032 – 2064	U16	1st to 32nd characters of the robot SN
32066	2065	U16	Lower 4 characters of "Safety Checksum" (2 bytes)
32067	2066	U16	Upper 4 characters of "Safety Checksum" (2 bytes)

### Motion parameter feedback values

If the motion parameters (speed, acceleration, etc.) are set individually in the project, the relevance feedback values are not updated immediately, but only when the robot executes the next motion command.

### DI/DO description

DI/DO each occupies 8 bytes. Each byte has 8 bits (binary) and can represent the status of up to 64 ports each. Each byte from low to high indicates the status of one terminal. 1 indicates the corresponding terminal is ON, and 0 indicates the corresponding terminal is OFF or no corresponding terminal.

For example, the first byte of DI is 0x01 (00000001 in binary). The bits from low to high represent the status of D1\_1 – DI\_8 respectively, that is, DI\_1 is ON and the remaining 7 DIs are OFF.

The second byte is 0x02 (00000010 in binary). The bits from low to high represent the status of D1\_9 – DI\_16 respectively, that is, DI\_10 is ON and the remaining 7 DIs are OFF.

Different controllers vary in the number of IO terminals. The binary bits exceeding the number of IO terminals will be filled with 0.

### RobotMode description

Value	Definition	NOTE
1	ROBOT_MODE_INIT	Initialized status
2	ROBOT_MODE_BRAKE_OPEN	Brake switched on
3	ROBOT_MODE_POWEROFF	Power-off status
4	ROBOT_MODE_DISABLED	Disabled (no brake switched on)
5	ROBOT_MODE_ENABLE	Enabled and idle
6	ROBOT_MODE_BACKDRIVE	Drag mode
7	ROBOT_MODE_RUNNING	Running status (project, TCP queue motion, etc.)
8	ROBOT_MODE_SINGLE_MOVE	Single motion status (jog, RunTo, etc.)
9	ROBOT_MODE_ERROR	There are uncleared alarms. This status has the highest priority. It returns 9 when there is an alarm, regardless of the status of the robot arm.
10	ROBOT_MODE_PAUSE	Project pause status
11	ROBOT_MODE_COLLISION	Collision detection triggered status

### BrakeStatus description

This byte indicates the brake status of the each joints by bit. 1 means that the corresponding joint brake is switched on. The bits correspond to the joints in the following table:

Bit	7	6	5	4	3	2	1	0
Meaning	Reserved	Reserved	J1	J2	J3	J4	J5	J6

Example:

- 0x01 (00000001): J6 brake is switched on
- 0x02 (00000010): J5 brake is switched on
- 0x03 (00000011): J5 and J6 brakes are switched on
- 0x04 (00000100): J4 brake is switched on

### RobotType description

Value	Model
3	CR3
5	CR5
7	CR7
10	CR10
12	CR12

16	CR16
101	Nova 2
103	Nova 5
113	CR3A
115	CR5A
117	CR7A
120	CR10A
122	CR12A
126	CR16A
130	CR20A
150	Magician E6

## 5 Holding register (map1, interaction between robot and PLC)

PLC address	Script address (Get/SetHoldRegs)	Register type	Function
40001 – 41024	0 – 1023	-	Reserved
41025 – 49999	1024 – 9998	-	User-defined

# Appendix B Blockly Programming Command

- [\*\*B.1 General description\*\*](#)
- [\*\*B.2 Event\*\*](#)
- [\*\*B.3 Control\*\*](#)
- [\*\*B.4 Operator\*\*](#)
- [\*\*B.5 String\*\*](#)
- [\*\*B.6 Custom\*\*](#)
- [\*\*B.7 IO\*\*](#)
- [\*\*B.8 Motion\*\*](#)
- [\*\*B.9 Modbus\*\*](#)
- [\*\*B.10 Bus\*\*](#)
- [\*\*B.11 TCP\*\*](#)
- [\*\*B.12 Tray\*\*](#)
- [\*\*B.13 Quick start\*\*](#)

# General description

## Block types

In Blockly programming, blocks can be categorized into four main types based on their shape.

### Round-head block



This block is used at the top of a Blockly program, and all blocks to be executed must be attached below this type of block. Only the [Event](#) block group contains this type of block.

### Rectangular block

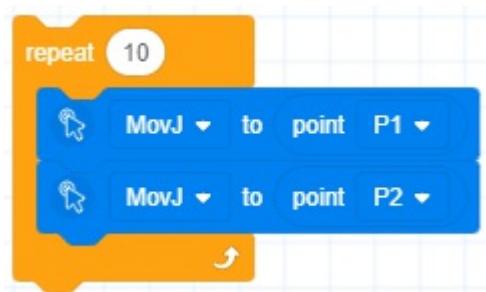


The rectangular blocks are used to execute commands and form the main part of a Blockly program. The oval and diamond-shaped parameter slots in the rectangular blocks can be filled with blocks in the corresponding shapes described below.



Rectangular blocks have a variant that allows other blocks to be nested inside, used for flow control.

Example:



### Oval block



The oval blocks return variables (numbers, strings, arrays, etc.) after execution. They are used as parameters to fill the corresponding slots in rectangular blocks.

Example:



### Diamond block



The diamond blocks are used for conditional judgments. After execution, they return either true or false. They are used as parameters to fill the corresponding slots in rectangular blocks.

Example:



## Motion mode

The robot supports the following motion modes:

### Joint motion

The robot plans the motion of each joint based on the difference between the current and target joint angles, ensuring that all joints complete their motion simultaneously. Joint motion does not constrain the TCP (Tool Center Point) trajectory, and the path is usually not a straight line.



Joint motion is not restricted by singularities (refer to the corresponding hardware guide for details). If there are no specific trajectory requirements or the target point is near a singularity, joint motion is recommended.

### Linear motion

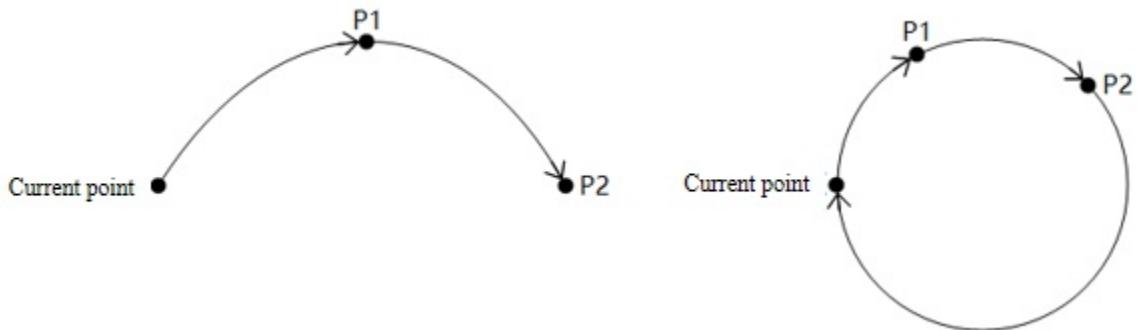
The robot plans the motion trajectory based on the current posture and the target point's posture, ensuring that the TCP moves in a straight line and the end posture changes uniformly during the motion.



When the trajectory passes through a singularity, issuing a linear motion command will result in an error. It is recommended to re-plan the point or use joint motion near the singularity.

### Arc motion

The robot determines an arc or a circle based on three non-collinear points: the current position, P1, and P2. During the motion, the end posture of the robot is interpolated between the current point and the posture at P2, while the posture at P1 is not considered (i.e., when the robot reaches P1 during the motion, its posture may differ from the taught posture).



When the trajectory passes through a singularity, issuing an arc motion command will result in an error. It is recommended to re-plan the point or use joint motion near the singularity.

## Coordinate system parameters

For motion blocks, you can specify the user and tool coordinate systems via advanced settings, with the following priority:

1. When the coordinate system is specified in advanced settings, the specified system is used. If the point parameter is a taught point, the posture coordinates of the taught point are converted into the values of the specified coordinate system.
2. If no coordinate system is specified in advanced settings, and the point parameter is a taught point, the system uses the coordinate system index attached to the taught point.
3. If no coordinate system is specified in advanced settings, and the point parameter is a joint variable or posture variable, the global coordinate system set in the **Control** block module is used (see **Set user coordinate system** and **Set tool coordinate system** blocks for details; if no such blocks are defined, the default coordinate system is 0).

### **i** NOTE

When starting a project, the default global coordinate system is set to 0, regardless of the values set in

the Jog panel before running the project.

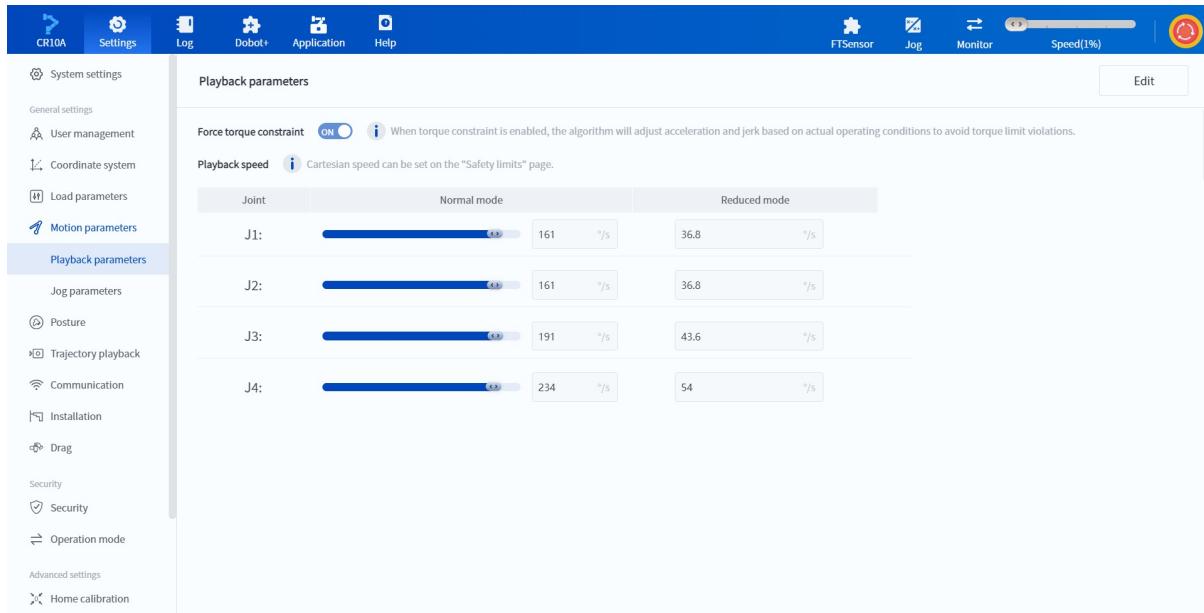
## Speed parameters

### Relative speed

You can specify the acceleration (Accel) and velocity (V) ratio for the robot when executing the motion command via advanced settings.

```
Robot actual speed = Maximum speed x Global speed x Command speed  
Robot actual acceleration = Maximum acceleration x Command speed
```

The maximum speed/acceleration is controlled by **Playback settings**, which can be viewed and modified on the "Motion parameters" page of DobotStudio Pro.



The global speed can be adjusted via DobotStudio Pro (speed slider at the top right of the figure above) or the **SpeedFactor** command.

The command speed is the ratio specified in the advanced settings of the motion block. If no acceleration/velocity ratio is specified in the advanced settings, the default ratio set by the **joint/linear speed/acceleration ratio** block is used. If no such block is defined, the default value is 100.

### Absolute speed

The advanced settings of the linear and arc motion blocks allow you to specify the absolute speed (Speed) of the robot when executing the motion command.

The absolute speed is not affected by the global speed but is limited by the maximum speed set in **Playback settings** (or reduced maximum speed if the robot is in reduced mode). If the absolute speed exceeds this maximum, the robot will operate at the maximum speed.

For example, if the absolute speed for linear motion is set to 1000 mm/s, which is less than the maximum speed of 2000 mm/s in the playback settings, the robot will move at a target speed of 1000 mm/s, regardless of the global speed. However, if the robot is in reduced mode (with a reduction rate of 10%), the maximum speed becomes 200 mm/s, which is lower than 1000 mm/s, so the robot will move at a target speed of 200 mm/s.

Absolute speed and velocity ratio cannot be set simultaneously.

## Smooth transition

In some cases, the robot needs to pass through multiple transition points before reaching the target point. These transition points are typically used to adjust the robot's motion path to avoid obstacles and do not require the robot to reach them precisely. By setting CP parameters, the robot will begin turning before reaching the transition point, making its speed and trajectory smoother during the turn.

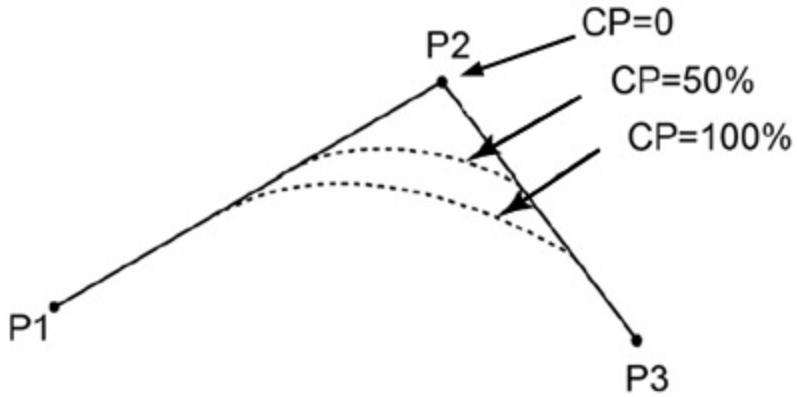
You can specify the **CP (Continuous Path Ratio)** or **R (Continuous Path Radius)** in the advanced settings of the motion command to configure how the robot transitions from one motion command to the next.

### **i** NOTE

- Joint motion mode does not support setting a continuous path radius (R).
- Smooth transitions cannot be applied if the user-specified path points are based on different tool coordinate systems.

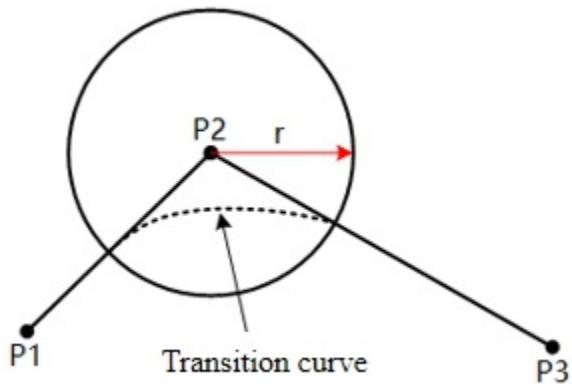
## CP

When setting the CP, the system automatically calculates the curvature of the transition curve. The larger the CP value, the smoother the curve, as shown in the figure below. The CP transition curve is affected by speed and acceleration. Even if the path points and CP values are the same, different speeds/accelerations will result in different curvatures.

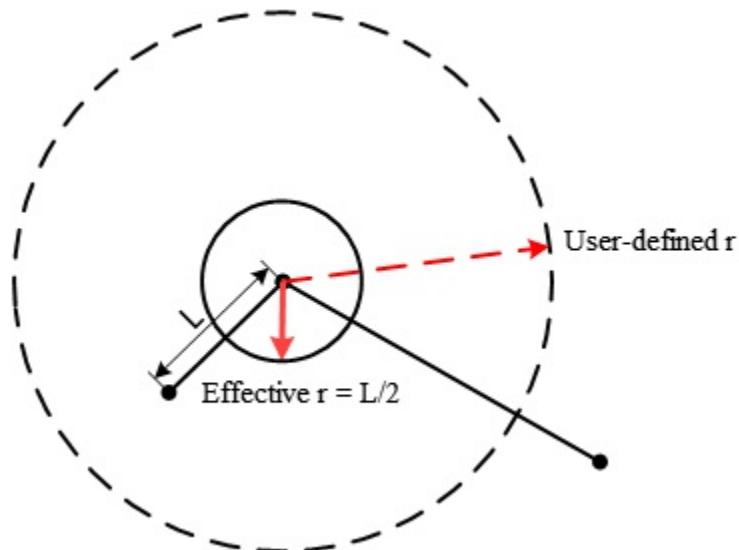


## R

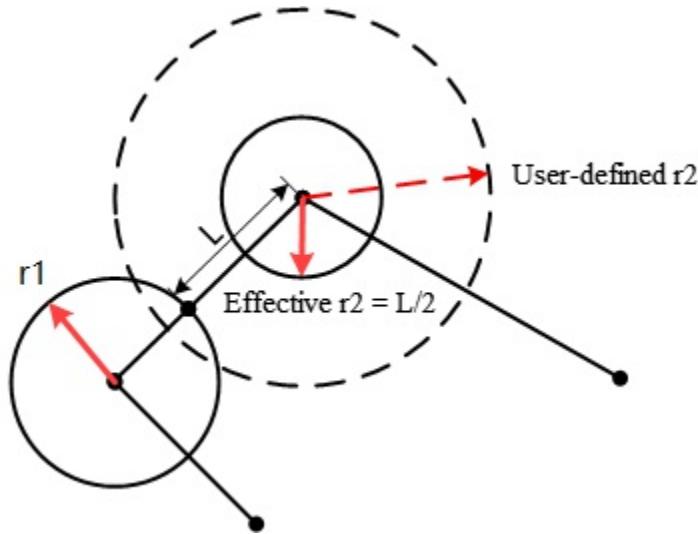
When setting the R, the system calculates the transition curve using the transition point as the center and the specified radius. The R transition curve is not affected by speed or acceleration, but is determined by the path points and the specified radius.



If the radius you set is too large (i.e., it exceeds the distance between the start/end point and the transition point), the system will automatically use half of the shorter distance between the start/end point and the transition point ( $L$  in the figure below) as the radius to calculate the transition curve.



If two consecutive transition radii ( $r_1$  and  $r_2$  in the figure below) overlap, the system will use the point where the first transition ends as the start point of the second motion, and the second transition radius will be recalculated according to the rules for large transition radii.



### Default values

If no CP or R is specified in the advanced settings, the default value set by the **Continuous path ratio** block is used. If no such block is defined, the default value is 0.

#### NOTE

As CP causes the robot to bypass intermediate points, if CP is set, any IO signal outputs or function settings (e.g., enabling/disabling SafeSkin) between two motion commands will be executed during the transition.

If you want the robot to execute commands precisely when it reaches the intermediate point, set the CP parameters of the previous command to 0.

## Stop condition

Some motion blocks support specifying the stop condition through advanced settings. While executing a motion command, if the specified stop condition is met, the robot will stop the current motion and proceed directly to the next command.

- The supported conditions include IO, variables, and any expression that complies with Lua syntax.
- A maximum of 3 stop conditions can be set, and the conditions can be linked using the keywords **And** (all conditions must be met) / **Or** (any condition can be met). Mixing **And** and **Or** is not allowed.

Stop Condition

 When the condition is met, the robot skips the current movement

When	DI	1	==	ON
And	DI	1	==	ON

+

 NOTE

- Specifying a stop condition will invalidate the R parameter.
- After a stop condition is set, the CP parameter can still be applied, but the transition segment (the curved part) will not be included in the stop condition range.
- If the previous command before the stop condition includes CP, the stop condition will only be triggered after exiting the transition segment.

# Event

The event blocks are used as a trigger to start the program. Only the blocks connected below an event block will be executed.

## Start command



Start

**Description:** It is the mark of the main thread of a program. After creating a new project, there is a **Start** block in the programming area by default. Please attach other non-event blocks under the **Start** block to program.

**Limitation:** Only one **Start** block can be used per project.

## Sub-thread start command



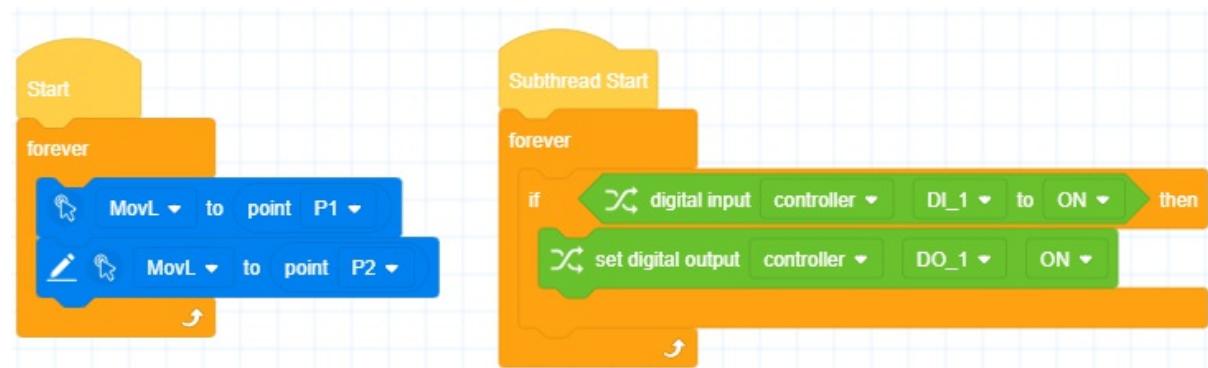
Subthread Start

**Description:** It is the mark of the sub-thread of a program. The sub-thread runs synchronously with the main thread, but it cannot call robot control commands. It can only perform variable operations or I/O control. Please use sub-threads as needed based on program logic.

**Limitation:** A maximum of four sub-threads can be used per project.

### Example:

In the example below, when the project runs, the main thread and sub-thread will start simultaneously. The motion commands in the main thread and the IO commands in the sub-thread will run independently, each following its own execution queue.



# Control

The control blocks are used to control the running path of the program.

## Wait until...

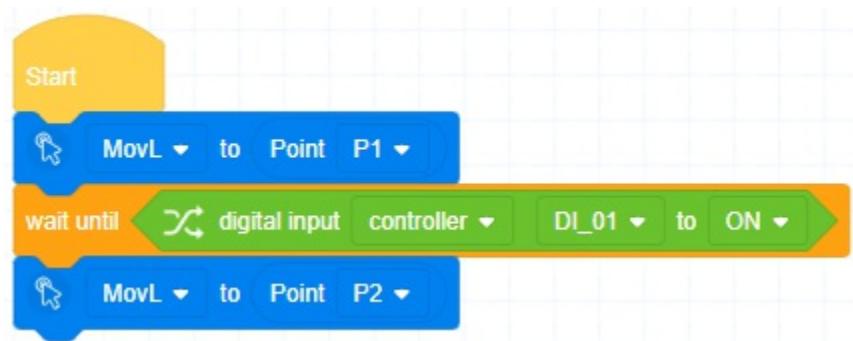


**Description:** The program pauses until the parameter is **true**, then continues.

**Parameter:** Use other hexagonal blocks as the parameter.

### Example:

The robot moves to P1, waits until DI1 is ON, then moves to P2.



## Repeat n times



**Description:** Nest other blocks inside this block. The nested block will be repeated a specified number of times.

**Parameter:** The number of repetitions.

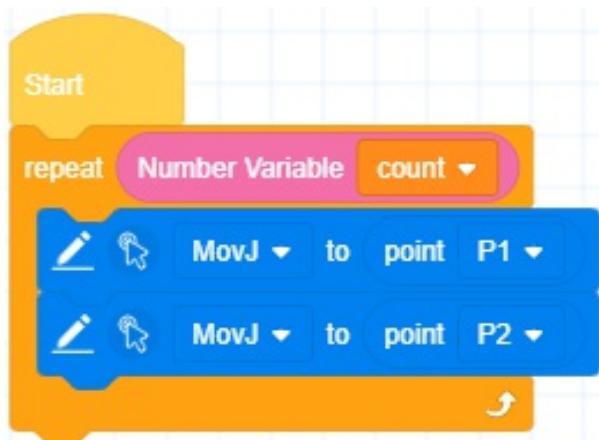
### Example 1:

The robot repeats two motion commands 10 times.



### Example 2:

The robot loops through two motion commands n times, where n is the value of the numeric variable `count`.



## Repeat continuously



**Description:** Nest other blocks inside this block. The nested block will be repeated continuously until the **End repeat** block is encountered. No other blocks can be attached below this block.

## End repeat

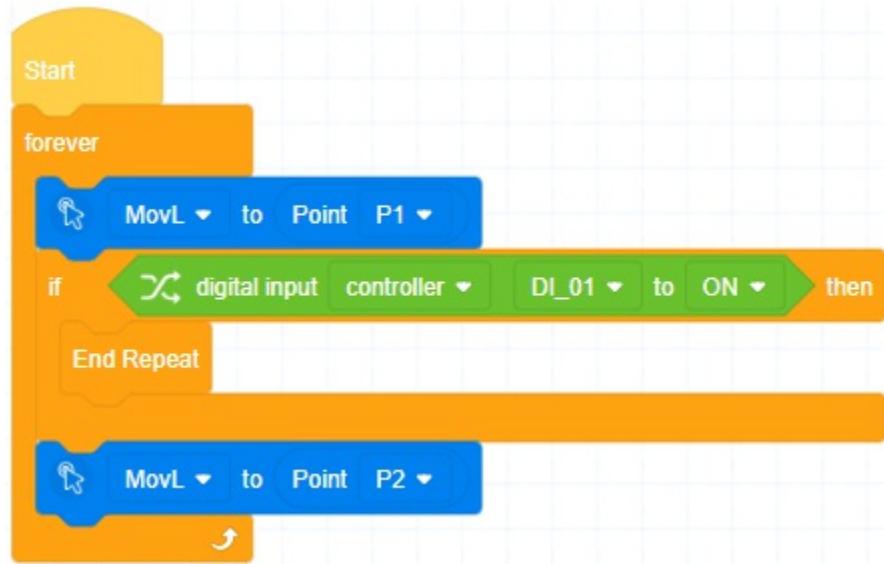


**Description:** Used to be nested inside the following repeat blocks. When the program reaches this block, it will immediately stop repeating and execute the blocks after the repeat block.



### Example:

During the repeated motions between P1 and P2, if the robot reaches P1 and DI1 is ON, the repetition will end immediately.



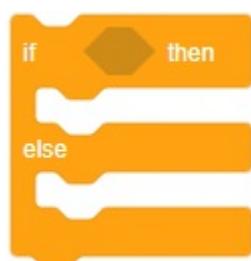
### if...then...



**Description:** If the parameter is **true**, the nested block will be executed. If the parameter is **false**, the program will skip to the next block.

**Parameter:** Use other hexagonal blocks which return a Boolean value (true or false) as the parameter.

### if...then...else...

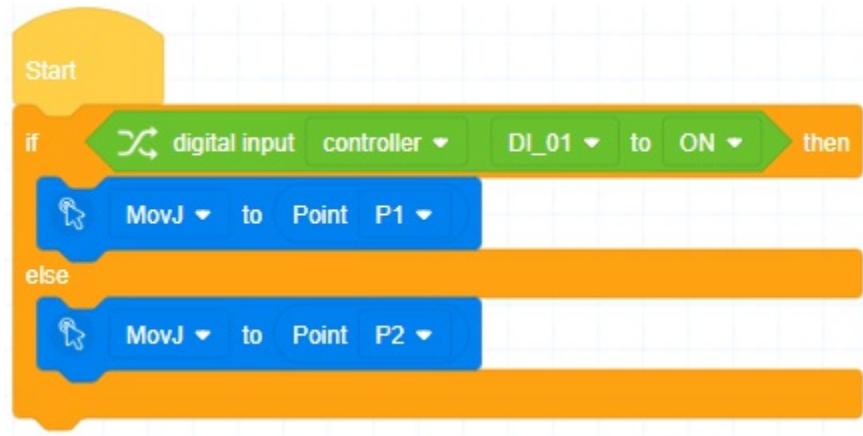


**Description:** If the parameter is **true**, the nested block before "else" will be executed. If the parameter is **false**, the nested block after "else" will be executed.

**Parameter:** Use other hexagonal blocks which return a Boolean value (true or false) as the parameter.

### Example:

If DI1 is ON, the robot moves to P1; otherwise, it moves to P2.



### Repeat until...



**Description:** Repeat the nested block until the parameter is **true**.

**Parameter:** Use other hexagonal blocks which return a Boolean value (true or false) as the parameter.

### Set label

A yellow "Set label name as" control block with a dropdown menu showing "label1".

**Description:** Set a label. After setting the label, the program can jump to it through "Goto label" block.

**Parameter:** The label name, which must start with a letter and cannot contain spaces or special characters.

### Goto label

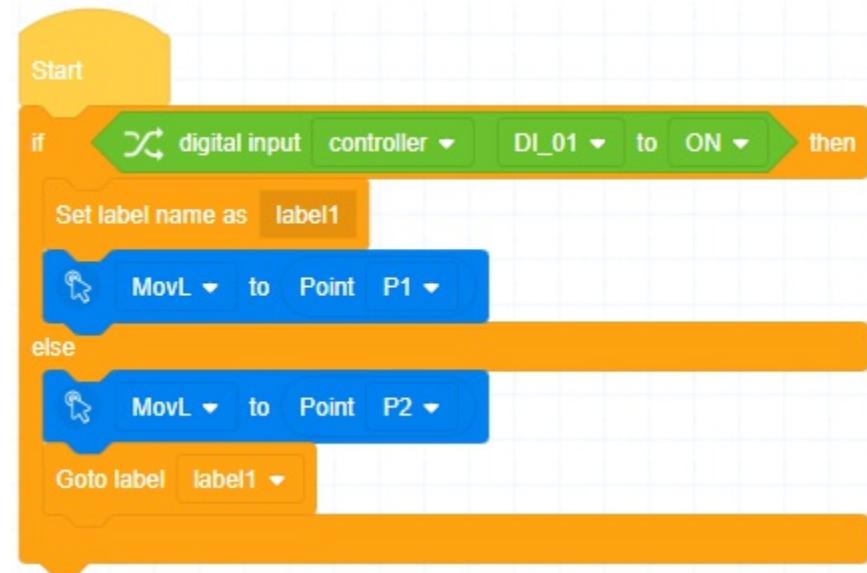
A yellow "Goto label" control block with a dropdown menu showing "label1".

**Description:** When the program reaches this block, it will jump directly to the specified label and execute the blocks after the label.

**Parameter:** The name of the label that already set.

**Example:**

If DI1 is ON, the robot moves directly to P1; otherwise, it moves to P2 first, then to P1.



**i** **NOTE**

This block can only jump to labels within the same column (including subroutines), and cannot jump across columns. For example, it cannot jump from a custom function to the main thread.

## Fold command



**Description:** Allow nested blocks to be collapsed for display. This has no control effect and is only used to make the program more organized and readable. Click the double arrow in the bottom right of the block to toggle the fold status.

**Parameter:** Provide a description for the folded block, preferably an indirect and intuitive name.

## Pause



**Description:** When the program reaches this block, it will automatically pause and can only continue running through the software or remote control.

# Stop program



**Description:** The program automatically stops and exits execution when it reaches this block.

## Custom log



**Description:** Output the custom log message, which can be viewed and exported on the Log page of the software.

# Set collision detection



**Description:** Set the collision detection function. The collision detection level set by this block is only effective during the running of the project. Once the project stops, the value will revert to its previous status.

**Parameter:** Select the sensitivity of the collision detection function. Options range from **OFF** to **Levels 1 to 5**. The higher the level, the more sensitive the detection.

## Set collision backoff distance



**Description:** Set how far the robot should retreat along its path after a collision is detected. The value set by this block is only effective during the current project. After the project stops, it will revert to its original value.

**Parameter:** Collision backoff distance, range: [0,50], unit: mm.

## Modify user or tool coordinate system



**Description:** Modify the specified user or tool coordinate system.

### Parameter:

- Specify to modify the user coordinate system or tool coordinate system.
- Specify the index of the coordinate system to be modified.
- Specify the parameter of the coordinate system that has been modified.
- Specify the scope of the modification:
  - Script only:** The modified coordinate system will only be effective during the current project, and will revert to its original value after the project stops.
  - Global save:** The modified coordinate system will be saved globally and will retain its modified value even after the project stops.

### Example:



## Calculate and update user coordinate system



**Description:** Calculate and update the specified user coordinate system. The modification will only be effective during the current project, and the coordinate system will revert to its original value after the project stops.

**Parameter:**

- Specify the index of the user coordinate system to be used as the calculation reference. The initial value for User coordinate system 0 is the base coordinate system.
- Specify the calculation direction.
  - **Left-multiply:** The previous parameter's coordinate system rotates relative to the base coordinate system.
  - **Right-multiply:** The previous parameter's coordinate system rotates relative to itself.
- Specify the offset values for the coordinate system.
- The system will calculate a new user coordinate system based on the above parameters. You need to specify which user coordinate system to update with the result.

**Example:**

- Calculate **User 1, Left-multiply**, Offset: X 10 Y 10 Z 10 RX 10 RY 10 RZ 10, result updates to 1

This command indicates that a coordinate system initially aligned with User coordinate system 1 is translated by {x=10, y=10, z=10} and rotated by {rx=10, ry=10, rz=10} relative to the base coordinate system. The resulting new coordinate system is assigned to User coordinate system 1.

- Calculate **User 1, Right-multiply**, Offset: X 10 Y 10 Z 10 RX 10 RY 10 RZ 10, result updates to 1

This command indicates that a coordinate system initially aligned with User coordinate system 1 is translated by {x=10, y=10, z=10} and rotated by {rx=10, ry=10, rz=10} relative to User coordinate system 1. The resulting new coordinate system is assigned to User coordinate system 1.

## Calculate and update tool coordinate system



**Description:** Calculate and update the specified tool coordinate system. The modification will only be effective during the current project, and the coordinate system will revert to its original value after the project stops.

**Parameter:**

- Specify the index of the tool coordinate system to be used as the calculation reference. The initial value for Tool coordinate system 0 is the flange coordinate system.
- Specify the calculation direction.
  - **Left-multiply:** The previous parameter's coordinate system rotates relative to the flange coordinate system.
  - **Right-multiply:** The previous parameter's coordinate system rotates relative to itself.

- Specify the offset values for the coordinate system.
- The system will calculate a new tool coordinate system based on the above parameters. You need to specify which tool coordinate system to update with the result.

#### Example:

- Calculate **Tool 1, Left-multiply**, Offset: X **10** Y **10** Z **10** RX **10** RY **10** RZ **10**, result updates to **1**

This command indicates that a coordinate system initially aligned with Tool coordinate system 1 is translated by {x=10, y=10, z=10} and rotated by {rx=10, ry=10, rz=10} relative to the flange coordinate system. The resulting new coordinate system is assigned to Tool coordinate system 1.

- Calculate **Tool 1, Right-multiply**, Offset: X **10** Y **10** Z **10** RX **10** RY **10** RZ **10**, result updates to **1**

This command indicates that a coordinate system initially aligned with Tool coordinate system 1 is translated by {x=10, y=10, z=10} and rotated by {rx=10, ry=10, rz=10} relative to Tool coordinate system 1. The resulting new coordinate system is assigned to Tool coordinate system 1.

## Set user coordinate system

**Set user coordinate system to** 0 ▾

**Description:** Set the user coordinate system used by the current project. After the project stops, it will revert to its original value.

**Parameter:** The index of the user coordinate system.

## Set tool coordinate system

**Set tool coordinate system to** 0 ▾

**Description:** Set the tool coordinate system used by the current project. After the project stops, it will revert to its previous value.

**Parameter:** The index of the tool coordinate system.

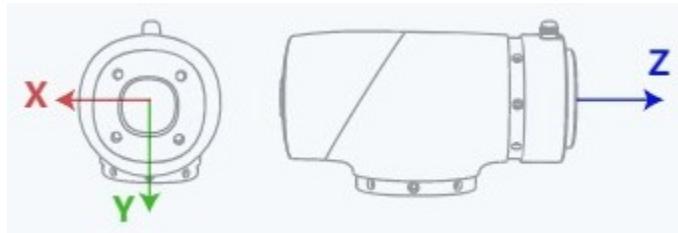
## Set payload parameters

**set current payload** 0 kg, offsetX 0 offsetY 0 offsetZ 0

**Description:** Set the weight and eccentric coordinates of the end effector's payload. After the project stops, it will revert to its previous value.

**Parameter:**

- Enter the weight of the current payload. Unit: kg.
- Enter the eccentric coordinates of the current payload. Refer to the figure below for coordinate axis directions. Unit: mm.



## Wait



**Description:** Wait for a specified time before executing the next command.

**Parameter:** The time delay before issuing the command. The maximum wait time is 2,147,483,647 ms. Setting a value exceeding the maximum will invalidate the command.

## Set safety wall switch



**Description:** Set the ON/OFF status of a single safety wall. After the project stops, it will revert to its previous value.

**Parameter:**

- Select the safety wall index. The currently selected safety wall will be displayed as a "?" mark after it is deleted.
- Select ON/OFF status of safety wall.

## Set safety zone switch



**Description:** Set the ON/OFF status of a single safety zone. After the project stops, it will revert to its previous value.

**Parameter:**

- Select the safety zone index. The currently selected safety zone will be displayed as a "?" mark after it

is deleted.

- Select ON/OFF status of safety zone.

## Get system time

get system time

**Description:** Get the current system time.

**Return:** The current system time's Unix timestamp in milliseconds, which is the number of milliseconds since 00:00:00 on January 1, 1970, GMT. This command is generally used to calculate time differences. To get the local time, please convert the obtained GMT according to your local time zone.

**Example:**

If the obtained value is **1686304295963**, converting to Beijing time yields **2023-06-09 17:35** (plus 963 milliseconds).

If the obtained value is **1686304421968**, converting to Beijing time yields **2023-06-09 17:41** (plus 968 milliseconds).

Subtracting multiple obtained values allows you to calculate time differences.

## Start timing

start timing

**Description:** Start timing when the project runs to this block. It needs to be used combined with the **Get timing result** block below.

## Get timing result

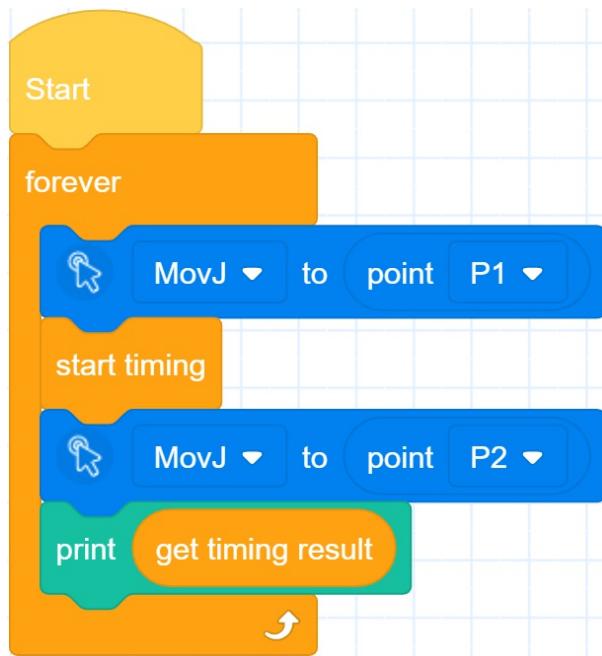
get timing result

**Description:** End timing and return the time difference.

**Return:** The time difference from the start to the end of timing, unit: ms. The maximum measurable time is 4,294,967,295 ms (about 49.7 days). After exceeding this time, it will reset and start counting from 0.

**Example:**

Print the time taken for the robot to move from P1 to P2 through joint motion.



## Set tool mode



**Description:** For robot models where the end effector's AI1 and AI2 interfaces share terminals with the 485 interface (CR and CR A series), you can use this command to set the mode of the shared terminals. The default mode is **485 mode**.

### NOTE

For robots that do not support tool mode switching, calling this command will have no effect.

### Parameter:

- Select tool mode, supporting **485 mode** and **analog input mode**. For 485 mode, there is no need to set the following two parameters.
- Select the **analog input mode** for AI1, supporting **0~10V voltage input mode**, **current mode**, and **0~5V voltage input mode**.
- Select the **analog input mode** for AI2, supporting the same modes as AI1.

## Set data type for tool RS485



**Description:** Set the data type for the RS485 interface of the end effector.

**Parameter:**

- Enter the baud rate of the RS485 interface.
- Select whether to use a parity bit.
- Select the length of the stop bit.

## Set tool power status



**Description:** Set the power status of the end effector, generally used to restart the end power, such as reinitializing the gripper by repowering it. It is recommended to wait at least **4 ms** between consecutive calls to this interface.

**NOTE**

After turning off the end power, the tool DO will also become inactive.

**Parameter:** Select the power status.

## Custom popup

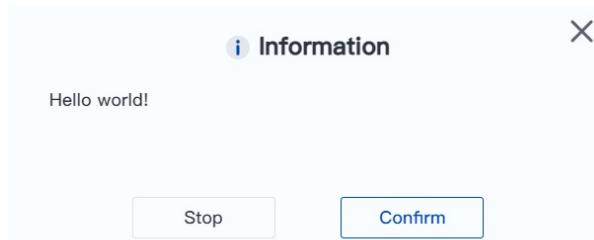


**Description:** Used to display a user-defined information window during script execution. DobotStudio Pro will only display one popup at a time, showing the most recent popup information.

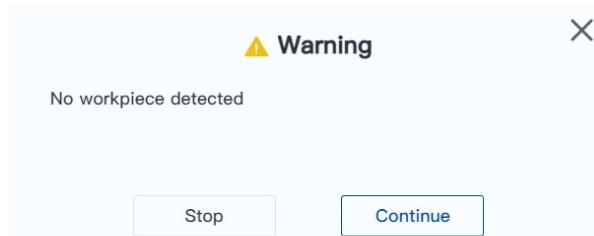
If DobotStudio Pro is not open while the project is running (e.g., when starting the project via IO or Modbus), the popup will not appear. However, warning and error-type popup instructions will still pause the project.

**Parameter:**

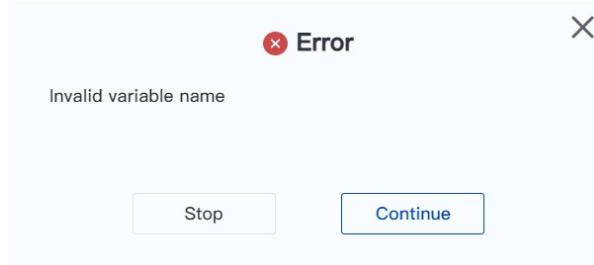
- Message type.
  - **Information:** This type of popup does not affect project execution. Click **Stop** to stop the project, or click **Confirm** to close the popup. Once the popup appears, you can also stop the project via IO or Modbus, and the popup disappears automatically after the project is stopped.



- **Warning:** This type of popup pauses the project. Click **Stop** to stop the project, or click **Continue** to resume the project. Once the popup appears, you can also stop or resume the project via IO or Modbus, and the popup disappears automatically after the project is stopped or resumed.



- **Error:** This type of popup pauses the project. Click **Stop** to stop the project, or click **Continue** to resume the project. Once the popup appears, you can also stop or resume the project via IO or Modbus, and the popup disappears automatically after the project is stopped or resumed.



- Popup title. Only supports strings, with a maximum length of 128 characters.
- Message content. Supports various variable types, including strings. Strings must not exceed 128 characters. Other variable types will be converted to strings, with the resulting string size not exceeding 512 bytes.
- Whether the popup content is written to the log.
  - No: The popup content is not written to the log.
  - Yes: The popup content is written to the corresponding log type (information popups are logged as "Custom" type). The log entry is formatted as "Popup Title: Popup Content".

Current alarm [History](#)

Date:  -  2024-12-11 17:50:00

Key words:

Type:  All  Error  Warning  Information  Custom

[Export](#) [Refresh](#)

## Comment



**Description:** Add a comment. Comments do not affect project execution and are primarily used to help viewers understand the project logic.

**Parameter:** Comment content.

# Operator

The Operator blocks are used for performing operations on variables or constants.

## Arithmetic command



**Description:** Perform addition, subtraction, multiplication or division to the parameters.

**Parameter:**

- Fill in both blanks with variables or constants. You can use oval blocks that return numerical values or directly fill in the values.
- Select the arithmetic operator in the middle.

**Return:** The result after operation.

## Comparison command



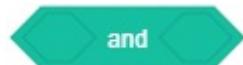
**Description:** Perform a comparison operation on the parameters.

**Parameter:**

- Fill in both blanks with variables or constants. You can use oval blocks that return numerical values or directly fill in the values.
- Select the comparison operator in the middle.

**Return:** It returns **true** if the comparison result is true, and **false** if the result is false.

## AND command



**Description:** Perform **and** operation on the parameters.

**Parameter:** Fill in both blanks with variables using other hexagonal blocks.

**Return:** It returns **true** if both parameters are true, and **false** if either one is false.

## OR command



**Description:** Perform **or** operation on the parameters.

**Parameter:** Fill in both blanks with variables using other hexagonal blocks.

**Return:** It returns **true** if either parameter is true, and **false** if both are false.

## NOT command

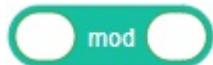


**Description:** Perform **not** operation on the parameters.

**Parameter:** Fill in the blank with a variable using a hexagonal block.

**Return:** It returns **false** if the parameter is true, and **true** if the parameter is false.

## Modulo command



**Description:** Perform modulo operation on the parameters.

**Parameter:** Fill in both blanks with variables or constants. You can use oval blocks that return numerical values or directly fill in the values.

**Return:** The result after operation.

## Rounding command



**Description:** Perform rounding operation on the parameters.

**Parameter:** Fill in the blank with a variable or constant. You can use an oval block that returns numerical values or directly fill in the value.

**Return:** The result after operation.

## Unary operation



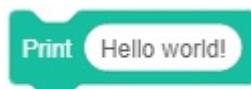
**Description:** Perform various unary operations on the parameters. The input values for the trigonometric function (sin/cos/tan) and the return values for the inverse trigonometric function (asin/acos/atan) are all in "degree".

**Parameter:**

- Select an operator.
  - abs (absolute value)
  - floor (round down)
  - ceiling (round up)
  - sqrt (square root)
  - sin
  - cos
  - tan
  - asin
  - acos
  - atan
  - ln
  - loh
  - e<sup>^</sup>
  - 10<sup>^</sup>
- Fill in the blank with a variable or constant. You can use oval blocks that return numerical values or directly fill in the values.

**Return:** The result after operation.

## Print command



**Description:** Output the parameter to the console for debugging purposes.

**Parameter:**

Enter the variable or constant to be output to the console. You can use other oval blocks or directly fill in the values.

# String

The string blocks include general functions for strings and arrays.

## Get the Nth character of a string



**Description:** Get the Nth character from a string variable.

**Parameter:**

- 1st parameter: String. You can use other oval blocks or fill in directly.
- 2nd parameter: Specify which character in the string to return.

**Return:** The character at the specified position in the string.

## Check if String A contains String B



**Description:** Check whether the first string contains the second string.

**Parameter:** The two strings to be checked. You can use oval blocks that return string, or fill in directly.

**Return:** It returns **true** if String A contains String B, otherwise returns **false**.

## Concatenate two strings



**Description:** Concatenate two strings, with the second string appended to the first.

**Parameter:** The two strings to be concatenated. You can use oval blocks that return string, or fill in directly.

**Return:** The concatenated string.

## Get length of string or array



**Description:** Get the length of the specified string or array. The length of a string is the number of characters it contains. The length of an array is the number of elements it contains.

**Parameter:** A string or array. You can use oval blocks that return string or array.

**Return:** The length of the string or array.

## Compare two strings

String comparison



**Description:** Compare two strings based on ASCII codes.

**Parameter:** The two strings to be compared. You can use oval blocks that return string, or fill in directly.

**Return:** It returns **0** if String 1 equals String 2, **-1** if String 1 is less than String 2, and **1** if String 1 is greater than String 2.

## Convert Array to String

Convert array to string

Array:



Separator:



**Description:** Convert a specified array into a string, with the elements separated by a specified delimiter. For example, if the array is {1, 2, 3} and the delimiter is |, the converted string will be "1|2|3".

**Parameter:**

- The array to be converted to string. You can use oval blocks that return string.
- The delimiter to use for the conversion.

**Return:** The converted string.

## Convert String to Array

Convert string to array

String:



Separator:



**Description:** Convert a specified string into an array by splitting it based on a specified delimiter. For example, if the string is "1|2|3" and the delimiter is |, the converted array will be {[1]=1, [2]=2, [3]=3}.

**Parameter:**

- The string to be converted to array. You can use oval blocks that return string, or fill in directly.
- The delimiter to use for the conversion.

**Return:** The converted array.

## Get Element at a specific Index in an Array

Array:

Index:

1

**Description:** Get the element at the specified index in the array. The index represents the position of the element in the array. For example, in the array {7, 8, 9}, the index of 8 is 2.

**Parameter:**

- The target array, using oval blocks that return arrays.
- The index of the specified element.

**Return:** The value of the element at the specified position in the array.

## Get multiple Elements from an Array



**Description:** Get multiple elements from an array at specified index positions, based on the range of a start index and an end index, using a step value.

**Parameter:**

- The target array, using oval blocks that return arrays.
- The range of elements to get, specified by the start index and end index.
- The step value determines how frequently to get elements. A step of 1 gets all elements, while a step of 2 gets every second element, and so on.

**Return:** A new array composed of the specified elements.

## Set Element at a specific Index in an Array



**Description:** Set the value of the element at the specified index in the array.

**Parameter:**

- The target array, using oval blocks that return arrays.
- The index of the specified element.
- The value of the specified element.

# Custom

The custom commands are used to create and manage custom blocks, as well as to call global variables.

## Call global variable



global variable var\_1 ▾

**Description:** Call a global variable set in the control software.

**Parameter:** Select the name of the global variable. The currently selected global variable will be displayed as a "?" mark after it is deleted.

**Return:** The value of the global variable.

## Set global variable



set var\_1 ▾ to 0

**Description:** Set the value of a specified variable. Please note that the blocks for setting global and custom variables have the same shape, but their functions differ slightly.

### Parameter:

- Select the name of the variable to modify. The currently selected global variable will be displayed as a "?" mark after it is deleted.
- The modified value, which can be filled in directly or by using another oval block.

## Create custom variable



Make a Variable

Click to create a custom variable. You can select the variable type as either a number or a string. The variable name must begin with a letter and cannot contain spaces or special characters. After creating at least one variable, the following blocks related to custom variables will appear in the block list.

## Custom number variable



Number Variable a ▾

**Description:** A newly created custom number variable, with a default value of nil. It is recommended to assign a value before use. You can use the dropdown menu to rename the current variable or delete the variable.

**Return:** The value of the variable.

## Set value of custom number variable



**Description:** Set the value of a specified number variable. Please note that the blocks for setting global and custom variables have the same shape, but their functions differ slightly.

**Parameter:**

- Select the name of the variable to modify.
- The modified value, which can be filled in directly or by using another oval block.

## Add value of number variable



**Description:** Add specified value to a number variable.

**Parameter:**

- Select the name of the variable to modify.
- The value to add, which can be filled in directly or by using another oval block. To decrease the value, set a negative number.

## Custom string variable



**Description:** A newly created custom string variable, with a default value of nil. It is recommended to assign a value before use. You can use the dropdown menu to rename the current variable or delete the variable.

**Return:** The value of the variable.

## Set value of custom string variable



**Description:** Set the specified string variable.

**Parameter:**

- Select the name of the variable to modify.
- The modified value, which can be filled in directly with a string.

## Create array

Make a Array

Click to create a custom array. The array name must begin with a letter and cannot contain spaces or special characters. After creating at least one array, the following array-related blocks will appear in the block list.

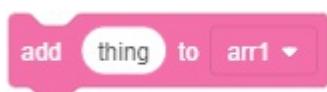
## Custom array



**Description:** A newly created custom array, initially empty. It is recommended to assign values before use. Right-click (PC) / long-press (App) the block in the block list to rename or delete the array. You can also rename or delete the array through the dropdown menu in other array blocks. The checkbox in front of the array block is currently non-functional and can be ignored.

**Return:** The value of the array.

## Add variable to array



**Description:** Add a variable to the specified array. The newly added variable will become the last item in the array.

**Parameter:**

- The variable to add, which can be filled in directly or by using another oval block.
- Select the array to modify.

## Delete item from array



**Description:** Delete a specific item from the specified array.

#### **Parameter:**

- Select the array to modify.
- The index of the item to delete, which can be filled in directly or by using another oval block that returns a numeric value.

## **Delete all items from array**



delete all of [arr1 v]

**Description:** Delete all items from the specified array.

**Parameter:** Select the array to modify.

## **Insert item into array**



insert [thing] at [1] of [arr1 v]

**Description:** Insert a variable at the specified position in the array.

#### **Parameter:**

- Select the array to modify.
- The position to insert, which can be filled in directly or by using another oval block that returns a numeric value.
- The variable to add, which can be filled in directly or by using another oval block.

## **Replace item in array**



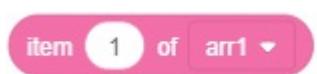
replace item [1] of [arr1 v] with [thing]

**Description:** Replace a specific item in the array with a specified variable.

#### **Parameter:**

- Select the array to modify.
- The index of the item to replace, which can be filled in directly or by using another oval block that returns a numeric value.
- The new variable, which can be filled in directly or by using another oval block.

## **Get item from array**



item [1] of [arr1 v]

**Description:** Get the value of a specified item from the array.

**Parameter:**

- Select an array.
- The item index, which can be filled in directly or by using another oval block that returns a numeric value.

**Return:** The value of the specified item.

## Get total number of items in array



length of arr1 ▾

**Description:** Get the total number of items in the array.

**Parameter:** Select an array.

**Return:** The total number of items in the specified array.

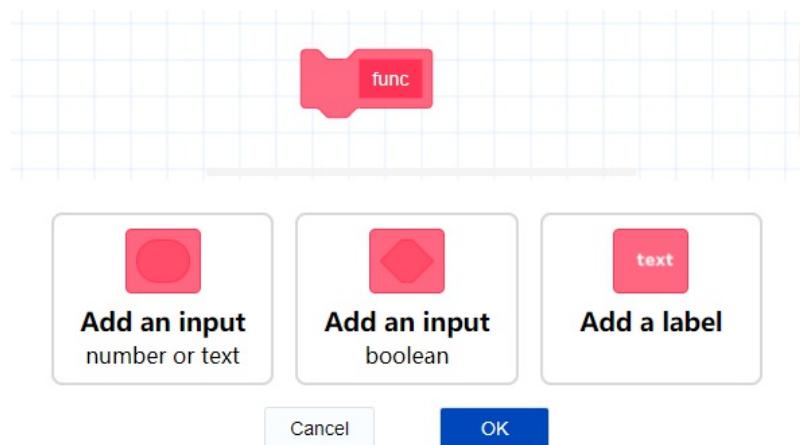
## Create a function



Make a Function

Click to create a function. A function is a fixed segment of code that allows you to define a group of blocks that perform a specific task. Once defined, you can simply call the function whenever needed, avoiding the need to repeatedly build the same blocks. When creating a new function, you must declare and define it. Once the function is created, the corresponding function block will appear in the block list.

### 1. Declare function



In this interface, you need to define the function's name, the input (parameter) types, the number of inputs, and their names. The function and parameter names cannot contain spaces or special characters. You can also add labels to the function, which serve as comments or provide additional details about the function or its inputs.

## 1. Define function

After declaring the function, the function header block will appear in the programming area.



You need to attach blocks below the function header to define the function.

The inputs defined in the function header can be dragged into the blocks below, meaning that the actual values passed when the function is called will be used as parameters.

## Custom function



**Description:** A user-defined function block, where both the function name and input parameters are defined by the user. It is used to call the previously defined function. Right-click (PC) / long-press (App) the block in the block list to modify the function declaration. To delete a function, delete its function header block.

## Create a subroutine

Make a subroutine

Click to create a new subroutine.

### NOTE

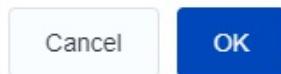
The difference between a subroutine and a function is that a subroutine is not encapsulated.

Essentially, it is a reusable block of code. Calling a subroutine is equivalent to copying and pasting the subroutine's code into the corresponding position.

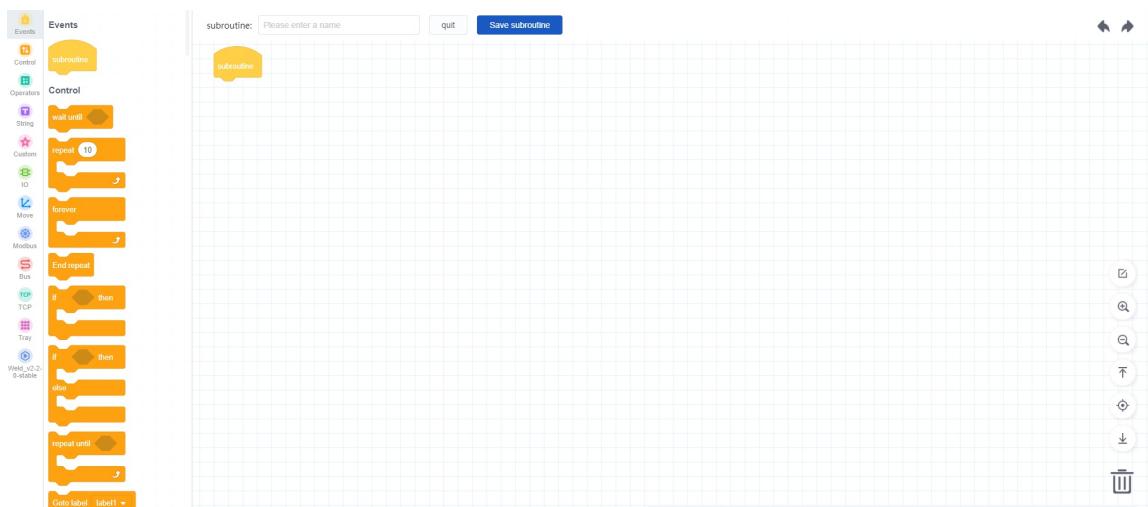
Blockly programming supports nested subroutine calls, allowing up to two layers of nesting. Subroutines can be used in both Blockly and Script programming. After successfully creating a subroutine, the corresponding subroutine block will appear in the block list.

Please select the new subroutine type

- Blockly programming     Script programming



- After selecting **Blockly programming**, the page switches to subroutine block programming, allowing you to set a description and write the subroutine.



- After selecting **Script programming**, a subroutine script programming window will pop up, allowing you to set a description and write the subroutine.

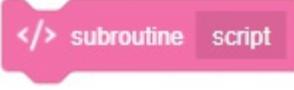


## Subroutine

- Blockly subroutine



- Script subroutine



</> subroutine script

**Description:** A subroutine block, which is defined by the user when creating the subroutine, is used to call saved subroutines. Right-click (PC) / long-press (App) the block in the block list to modify or delete the subroutine.

# IO

The IO blocks are used to manage the input/output of the robot's IO terminals. The value range of the input/output ports is determined by the number of corresponding terminals on the robot. Please refer to the hardware guide of the respective robot.

## Set digital output



**Description:** Set the ON/OFF status of the specified DO.

**Parameter:**

- Select the position of DO port, either the controller or tool.
- Select the DO port index.
- Select the output status (ON or OFF).

## Check digital output status



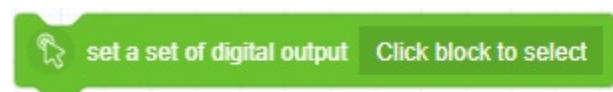
**Description:** Determine if the current status of the specified DO meets the condition.

**Parameter:**

- Select the position of DO port, either the controller or tool.
- Select the DO port index.
- Select the status that will evaluate to **true**.

**Return:** It returns **true** if the specified DO's current status meets the condition, otherwise it returns **false**.

## Set a group of digital outputs



**Description:** Set a group of DOs.

**Parameter:** Drag the block into the programming area and click to set its DO and status.

## Set up a set of digital outputs

X

Assign controller DO index:



DO_1	v	ON	v
DO_2	v	ON	v

Cancel

Save

- Click + or - to add or remove the number of DOs to be set.
- Select the DO port index from the left dropdown menu.
- Select the output status (ON or OFF) from the right dropdown menu.

## Wait for a group of digital outputs



wait a set of digital output

Click block to select

to ON

0 s

**Description:** Wait until the status of a group of DOs meets the condition before continuing to the next command.

**Parameter:**

- Select the status to wait for (ON or OFF).
- Set the timeout for waiting; if set to 0, it will wait indefinitely until the condition is met.
- After dragging the block into the programming area, click to set the DOs to wait for.

## Get a set of digital outputs

X

Assign controller DO index:



DO_1	v
DO_2	v

Cancel

Save

- Click + or - to add or remove the number of DOs to wait for.
- Select the DO port index from the dropdown menu.

## Wait for digital input



**Description:** Wait until the specified DI meets the condition or until a timeout occurs, then execute the subsequent commands.

### Parameter:

- Select the position of DI port, either the controller or tool.
- Select the DI port index.
- Select the status to wait for (ON or OFF).
- Set the timeout for waiting; if set to 0, it will wait indefinitely until the condition is met.

## Wait for a group of digital inputs



**Description:** Wait until the status of a group of DIs meets the condition before continuing to the next command.

### Parameter:

- Select the status to wait for (ON or OFF).
- Set the timeout for waiting; if set to 0, it will wait indefinitely until the condition is met.
- After dragging the block into the programming area, click to set the DIs to wait for.

Get a set of digital inputs X

### Assign controller DI index:

-
+

DI\_1

DI\_2

Cancel
Save

- Click + or - to add or remove the number of DIs to wait for.

- Select the DI port index from the dropdown menu.

## Set analog output



**Description:** Set the value of the specified AO. The meaning of the value (Voltage/Current) can be viewed and modified in DobotStudio Pro under **Monitor > Controller AI/AO**.

**Parameter:**

- Select the AO port index.
- The value to be output, which can be filled in directly or by using another oval block that returns a numeric value.

## Get analog output



**Description:** Get the value of the specified AO. The meaning of the value (Voltage/Current) can be viewed and modified in DobotStudio Pro under **Monitor > Controller AI/AO**.

**Parameter:** Select the AO port index.

**Return:** The current voltage or current value set for the AO.

## Check digital input status



**Description:** Determine if the current status of the specified DI meets the condition.

**Parameter:**

- Select the position of DI port, either the controller or tool.
- Select the DI port index.
- Select the status that will evaluate to **true**.

**Return:** It returns **true** if the specified DI's current status meets the condition, otherwise it returns **false**.

## Get analog input



**Description:** Get the value of the specified AI.

The meaning of the controller's analog input (Voltage/Current) can be viewed and modified in DobotStudio Pro under **Monitor > Controller AI/AO**.

To get the analog input at the end effector, set the tool mode to analog input mode using the [Set tool mode](#) block.

**Parameter:**

- Select the position of AI port, either the controller or tool.
- Select the AI port index.

**Return:** The value of the specified AI.

# Motion

The motion commands are used to control the robot's motion and set motion-related parameters.

The point parameters can be selected here after being added to the "Points" page of the project. The motion blocks also support dragging out the default variable block and replacing it with other oval blocks that return points.

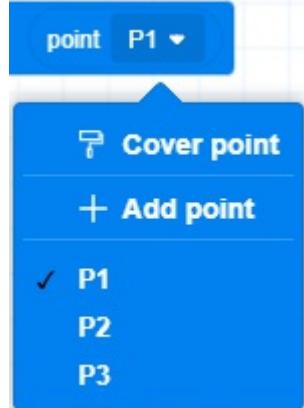
## Move to target point



**Description:** Control the robot to move from the current position to the target point. After dragging the block to the programming area, you can click it to configure advanced settings.

### Parameter:

- Select the motion mode, supporting joint motion (MovJ) and linear motion (MovL).
- Target point. The default point variable block supports the following functions:



- Select a point from the **Points** page.
- Add the robot's current posture as a new point and automatically select it.
- Overwrite the currently selected point with the robot's current posture.

### Advanced configuration

MovJ has fewer configurable parameters than MovL. The figure below shows the popup for MovL. Differences are explained further in the text.

## Motion block settings

The dialog box contains the following settings:

- V:** Velocity ratio, range: 1 – 100. Currently set to 1%.
- Absolute speed:** Available only for MovL. Set to 1 mm/s.
- Acceleration:** Acceleration ratio, range: 1 – 100. Currently set to 1%.
- CP:** Continuous path ratio, range: 0 – 100. Currently set to 0%.
- Transition Radius:** Radius of the transition curve, incompatible with CP. Available only for MovL. Set to 0 mm.
- User Coordinate:** The index of the user coordinate system. Set to 0.
- Tool Coordinate:** The index of the tool coordinate system. Set to 0.

**Advanced setting**

**Save**

The following parameters need to be selected to take effect. Detailed explanations of the parameters can be found in [General description](#).

- **V:** Velocity ratio, range: 1 – 100.
- **Speed:** Absolute speed, incompatible with V. Available only for MovL.
- **Accel:** Acceleration ratio, range: 1 – 100.
- **CP:** Continuous path ratio, range: 0 – 100.
- **Transition Radius (R):** Radius of the transition curve, incompatible with CP. Available only for MovL.
- **User coordinate:** The index of the user coordinate system.
- **Tool coordinate:** The index of the tool coordinate system.
- **Process I/O settings:**

**Advanced setting**

Process I / O settings ?

DO Index	DO_01	<span style="color: red;">-</span>
DO Status	<input type="radio"/> OFF	
Trigger mode	Percentage	Distance
Distance	0	mm

**+**

### **Process I/O settings and Stop condition cannot be selected at the same time.**

It is used to set the specified DO status when the robot reaches a specified distance or percentage.

A positive distance refers to the distance from the starting point, while a negative distance refers to the distance from the target point. If a CP is set, the robot will not reach the target point, which may affect the timing of the DO output.

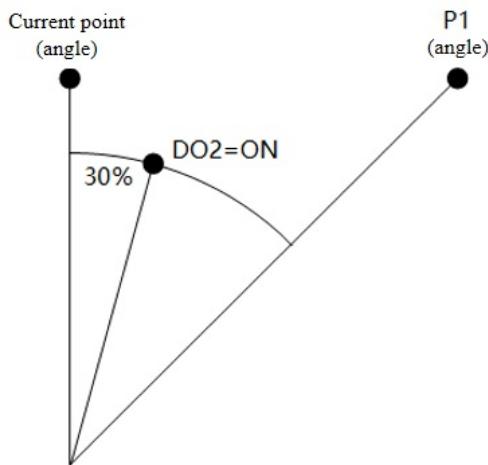
For joint motion (MovJ), the distance refers to the composite angular vector of the joints, making the calculation more complex. Therefore, it is recommended to use the Percentage mode.

You can click + on the bottom to add process I/O, and click - on the right to delete it.

Here are some examples of process I/O settings:

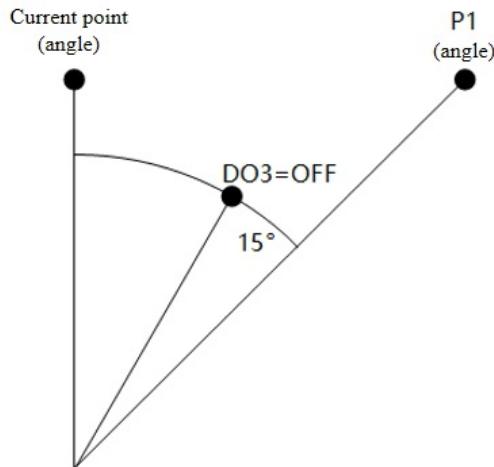
**Example 1:** Move to P1 in MovJ mode, DO index: DO\_02, DO status: ON, Trigger mode: Percentage, Distance: 30%.

This means DO2 will be set to ON when the robot reaches 30% of the distance from the starting point in MovJ mode.



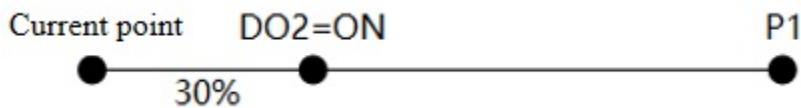
**Example 2:** Move to P1 in MovJ mode, DO index: DO\_03, DO status: OFF, Trigger mode: Distance, Distance: -15°.

This means DO3 will be set to OFF when the robot is 15° away from the target point in MovJ mode.



**Example 3:** Move to P1 in MovL mode, DO index: DO\_02, DO status: ON, Trigger mode: Percentage, Distance: 30%.

This means DO2 will be set to ON when the robot reaches 30% of the distance from the starting point in MovL mode.



**Example 4:** Move to P1 in MovL mode, DO index: DO\_03, DO status: OFF, Trigger mode: Distance, Distance: -15mm.

This means DO3 will be set to OFF when the robot is 15mm away from the target point in MovL mode.



- **Stop condition:**

Stop Condition When the condition is met, the robot skips the current movement

When	DI	1	=	ON
And	DI	1	=	ON

+

**Process I/O settings and Stop condition cannot be selected at the same time.**

It is used to set the stop condition for the motion. When the condition is met, the robot will end the current motion and execute the next command.

**Example 1:** Set only one condition: "When DI1 == ON," the robot will skip the current motion when DI1 is detected to be ON during execution.

**Example 2:** Set two conditions: "When DI1 == ON, and DI2 ~= ON," the robot will skip the current motion when DI1 is ON and DI2 is not ON.

**Example 3:** Set two conditions: "When DI1 == ON, or var <= 10," the robot will skip the current motion when DI1 is ON or the global variable (var) is less than or equal to 10.

## Relative motion along coordinate system



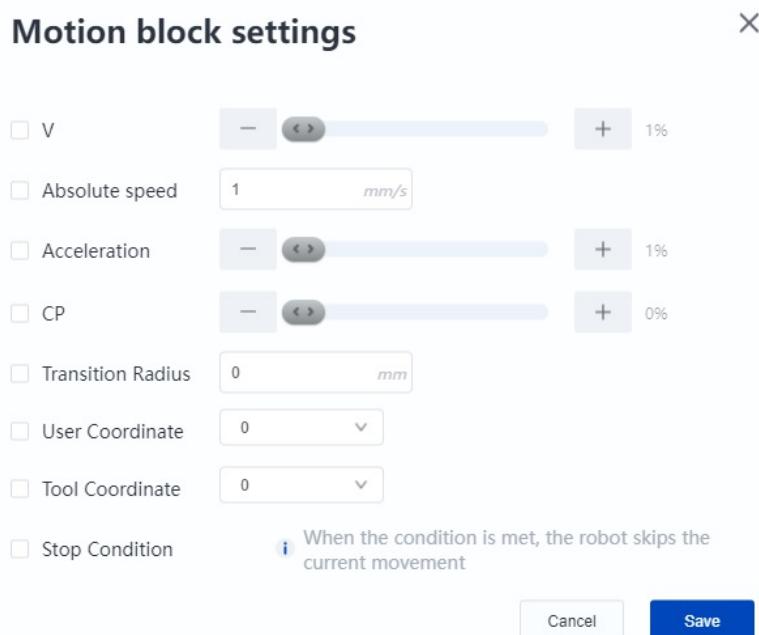
**Description:** Control the robot to move a specified distance along the selected coordinate system from its current position. After dragging the block to the programming area, you can click it to configure advanced settings.

### Parameter:

- Select the motion mode, including relative joint motion (RelMovJ) and relative linear motion (RelMovL).
- Specify whether the robot moves relative to the user coordinate system or the tool coordinate system.
- The offset in the specified coordinate system. **x, y, z:** Spatial offset values in millimeters (mm). **rx, ry, rz:** Angular offset values in degrees (°).

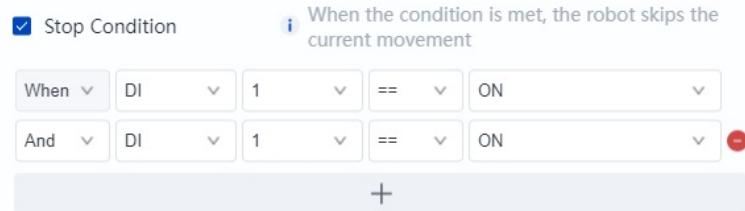
### Advanced configuration

MovJ has fewer configurable parameters than MovL. The figure below shows the popup for MovL. Differences are explained further in the text.



The following parameters need to be selected to take effect. Detailed explanations of the parameters can be found in [General description](#).

- **V:** Velocity ratio, range: 1 – 100.
- **Speed:** Absolute speed, incompatible with V. Available only for MovL.
- **Accel:** Acceleration ratio, range: 1 – 100.
- **CP:** Continuous path ratio, range: 0 – 100.
- **Transition Radius (R):** Radius of the transition curve, incompatible with CP. Available only for MovL.
- **User coordinate:** The index of the user coordinate system.
- **Tool coordinate:** The index of the tool coordinate system.
- **Stop condition:**



It is used to set the stop condition for the motion. When the condition is met, the robot will end the current motion and execute the next command.

## Get point after offsetting along coordinate system

**Get offset based on** point P1 **along** User **coordinate** Δx 30 Δy 0 Δz 0 rx 0 Δry 0 Δrz 0

**Description:** Offset the specified point by a specified distance along the selected coordinate system, and return the new point.

### Parameter:

- Select the point to offset.
- Select the coordinate system based on the taught point, either the user coordinate system or the tool coordinate system.
- Enter the offset distance in each direction.

**Return:** The new point after offset.

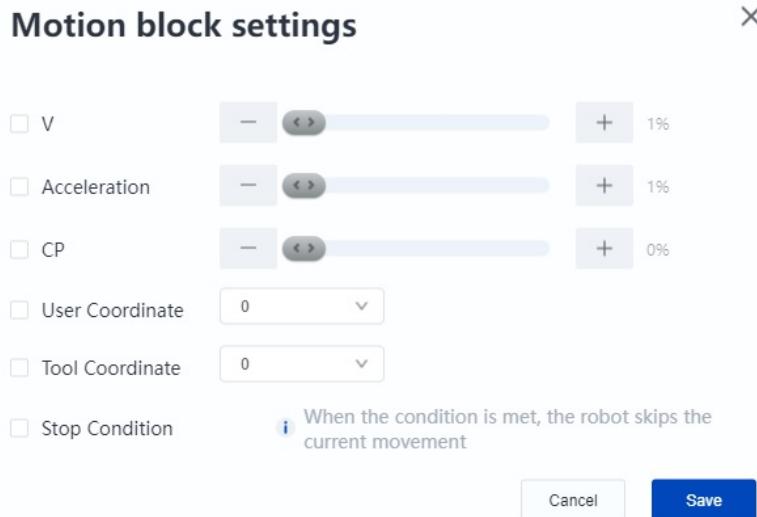
## Joint offset motion



**Description:** Control the robot to move each joint by a specified offset from its current position. After dragging the block to the programming area, you can click it to configure advanced settings.

**Parameter:** The offset for each joint, unit: °.

#### Advanced configuration



The following parameters need to be selected to take effect. Detailed explanations of the parameters can be found in [General description](#).

- **V:** Velocity ratio, range: 1 – 100.
- **Accel:** Acceleration ratio, range: 1 – 100.
- **CP:** Continuous path ratio, range: 0 – 100.
- **User coordinate/Tool coordinate:** Not applicable for this block.

## Get point after joint offset



**Description:** Offset the specified point by a specified angle along each joint, and return the new point.

**Parameter:**

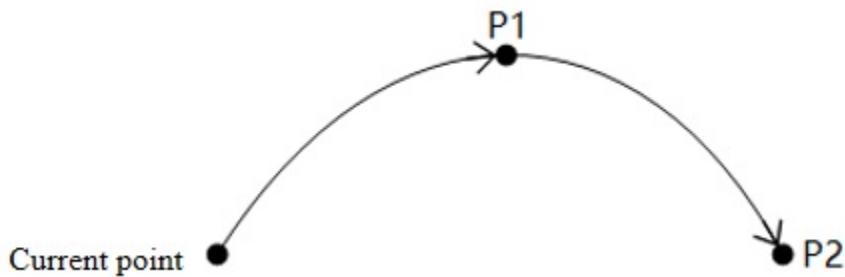
- Select the point to offset.
- Enter the offset angles for each joint.

**Return:** The new point after offset.

## Arc motion



**Description:** Control the robot to move from the current position to a target point in the Cartesian coordinate system using arc interpolation. The coordinates of the current position must not lie on the line determined by the intermediate point and the end point. After dragging the block to the programming area, you can click it to configure advanced settings.



During the motion, the end posture of the robot is interpolated between the current point and the posture at P2, while the posture at P1 is not considered (i.e., when the robot reaches P1 during the motion, its posture may differ from the taught posture).

#### Parameter:

- **Middle point:** The point used to define the arc's intermediate point.
- **End point:** The target point.

#### Advanced configuration

### Motion block settings

V - + 1%

Absolute speed 1 mm/s

Acceleration - + 1%

CP - + 0%

Transition Radius 0 mm

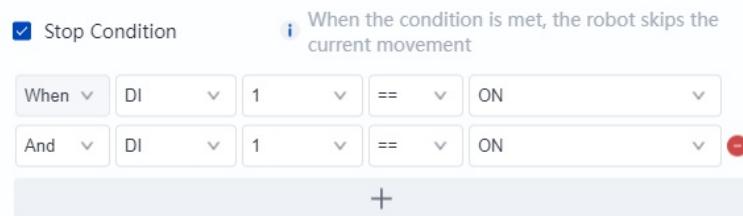
User Coordinate 0

Tool Coordinate 0

Stop Condition When the condition is met, the robot skips the current movement

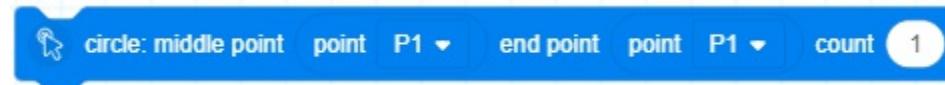
The following parameters need to be selected to take effect. Detailed explanations of the parameters can be found in [General description](#).

- **V:** Velocity ratio, range: 1 – 100.
- **Speed:** Absolute speed, incompatible with V.
- **Accel:** Acceleration ratio, range: 1 – 100.
- **CP:** Continuous path ratio, range: 0 – 100.
- **Transition Radius (R):** Radius of the transition curve, incompatible with CP.
- **User coordinate:** The index of the user coordinate system.
- **Tool coordinate:** The index of the tool coordinate system.
- **Stop condition:**

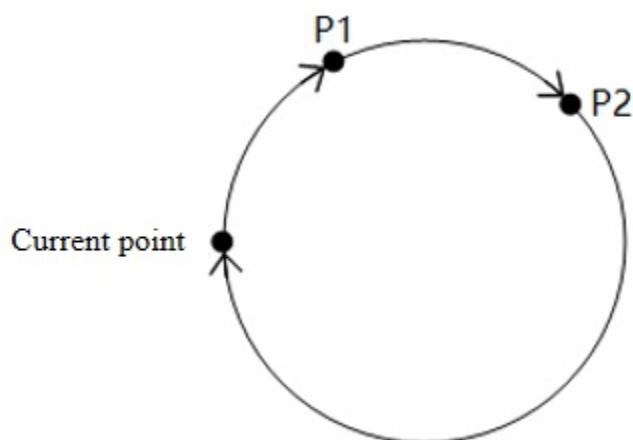


It is used to set the stop condition for the motion. When the condition is met, the robot will end the current motion and execute the next command.

## Circle motion



**Description:** Control the robot to perform full-circle interpolation from its current position, and return to the starting position after completing the specified number of rotations. The coordinates of the current position must not lie on the line determined by the intermediate point and the end point. After dragging the block to the programming area, you can click it to configure advanced settings.

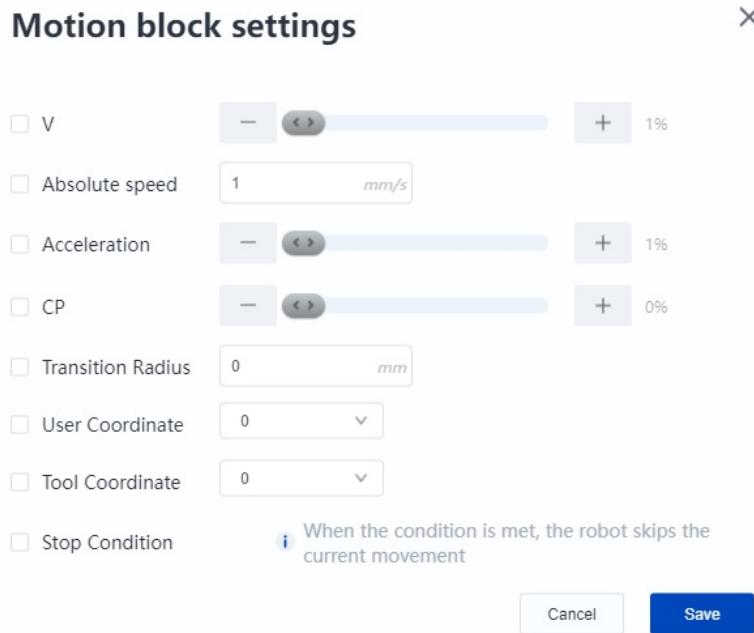


During the motion, the end posture of the robot is interpolated between the current point and the posture at P2, while the posture at P1 is not considered (i.e., when the robot reaches P1 during the motion, its posture may differ from the taught posture).

#### Parameter:

- **Middle point:** The positioning point 1 used to define the circle.
- **End point:** The positioning point 2 used to define the circle.
- **Count:** Enter the number of circles to perform, range: 1 – 999.

#### Advanced configuration



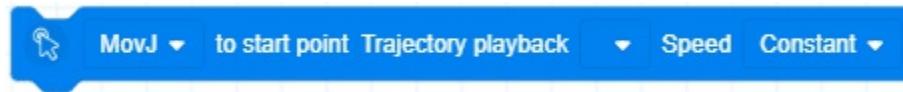
The following parameters need to be selected to take effect. Detailed explanations of the parameters can be found in [General description](#).

- **V:** Velocity ratio, range: 1 – 100.
- **Speed:** Absolute speed, incompatible with V.
- **Accel:** Acceleration ratio, range: 1 – 100.
- **CP:** Continuous path ratio, range: 0 – 100.
- **Transition Radius (R):** Radius of the transition curve, incompatible with CP.
- **User coordinate:** The index of the user coordinate system.
- **Tool coordinate:** The index of the tool coordinate system.
- **Stop condition:**



It is used to set the stop condition for the motion. When the condition is met, the robot will end the current motion and execute the next command.

## Trajectory playback



**Description:** The robot moves to the starting point of the trajectory and then performs trajectory playback. The trajectory file must be recorded during the trajectory playback process. After dragging the block to the programming area, you can click it to configure advanced settings.

### Parameter:

- Select the motion mode for moving to the trajectory starting point, supporting joint motion (MovJ) and linear motion (MovL).
- Select the trajectory file for playback. The currently selected trajectory file will be displayed as a "?" mark if it is deleted.
- Select the playback speed:
  - Constant speed: The robot will play back the trajectory at a constant speed.
  - 0.25x speed: The playback speed is scaled to 0.25 times the original recorded speed, unaffected by the global speed. The same for the followings.
  - 0.5x speed
  - 1x speed
  - 2x speed

### Advanced configuration

## Motion block settings

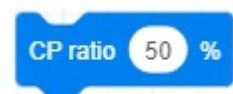
<input type="checkbox"/> Playback interval	50	ms	Range(8-1000)
<input type="checkbox"/> Filter coefficient	1		Range(0-1)
<input type="checkbox"/> User Coordinate	0	▼	
<input type="checkbox"/> Tool Coordinate	0	▼	

Cancel Save

The following parameters need to be selected to take effect.

- **Playback interval:** The sampling interval between trajectory points, i.e., the time difference between two adjacent points when generating the trajectory file. Range: [8, 1000], unit: ms, 50 by default (the sampling interval when the controller records the trajectory file).
- **Filter coefficient:** The smaller the value, the smoother the trajectory curve during playback, but the greater the distortion relative to the original trajectory. Set an appropriate filter coefficient based on the smoothness of the original trajectory. Range: (0,1], 1 means filtering is OFF, 0.2 by default.
- **User coordinate:** Specify the user coordinate system index for the trajectory points. If not specified, the user coordinate system recorded in the trajectory file will be used.
- **Tool coordinate:** Specify the tool coordinate system index for the trajectory points. If not specified, the tool coordinate system recorded in the trajectory file will be used.

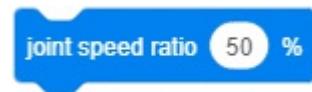
## Set CP ratio



**Description:** Set the continuous path (CP) ratio for the motion. This setting takes effect only in the current project. For more details on continuous path, see [General description](#).

**Parameter:** CP ratio, range: 0 – 100.

## Set joint speed ratio



**Description:** Set the speed ratio for joint motion. This setting takes effect only in the current project. For more details on speed calculation, see [General description](#).

**Parameter:** Joint speed ratio, range: 0 – 100.

## Set joint acceleration ratio

joint accel ratio 50 %

**Description:** Set the acceleration ratio for joint motion. This setting takes effect only in the current project. For more details on acceleration calculation, see [General description](#).

**Parameter:** Joint acceleration ratio, range: 0 – 100.

## Set linear speed ratio

linear speed ratio 50 %

**Description:** Set the speed ratio for linear and arc motions. This setting takes effect only in the current project. For more details on speed calculation, see [General description](#).

**Parameter:** Linear and arc speed ratio, range: 0 – 100.

## Set linear acceleration ratio

linear accel ratio 50 %

**Description:** Set the acceleration ratio for linear and arc motions. This setting takes effect only in the current project. For more details on acceleration calculation, see [General description](#).

**Parameter:** Linear and arc acceleration ratio, range: 0 – 100.

## Set global speed ratio

global speed ratio 50 %

**Description:** Set the global speed ratio for the robot. This setting takes effect only in the current project. For more details on speed calculation, see [General description](#).

**Parameter:** Global speed ratio, range: 0 – 100.

## Modify coordinates of a specified point

set X ▾ value of point P1 ▾ to 0

**Description:** Modify the value of a specified Cartesian coordinate axis for a selected point.

**Parameter:**

- Select a point.
- Select a coordinate axis.
- Enter the new value.

## Get coordinates of a specified point

point P1 ▾

**Description:** Get coordinates of a specified point.

**Parameter:** Select a point to get the coordinates.

**Return:** The coordinates of the specified point.

## Check motion feasibility

Check if MovJ ▾ P1 ▾ is reachable

**Description:** Check the feasibility of the robot moving from the current point to the specified point using the selected motion mode. The system calculates the entire motion trajectory and checks if any point in the path is unreachable.

**Parameter:**

- Select the motion mode, supporting joint motion (MovJ) and linear motion (MovL).
- Select the target point.

**Return:** The check result.

- 0: No error.
- 16: End point closed to shoulder singularity.
- 17: End point inverse kinematics error with no solution.
- 18: Inverse kinematics error with result out of working area.
- 22: Arm orientation error.
- 26: End point closed to wrist singularity.
- 27: End point closed to elbow singularity.
- 29: Speed parameter error.
- 30: Full parameter inverse kinematics error with no solution.
- 32: Shoulder singularity in trajectory.
- 33: Inverse kinematics error with no solution in trajectory.
- 34: Inverse kinematics error with result out of working area in trajectory.
- 35: Wrist singularity in trajectory.
- 36: Elbow singularity in trajectory.
- 37: Joint angle is changed over 180 degrees.

## Get coordinates of a specified axis

get CurrentPoint ▾ X ▾ value

**Description:** Get the value of a specified Cartesian coordinate axis for a selected point.

**Parameter:**

- Select a point to get the coordinates.
- Select a coordinate axis.

**Return:** The value of the specified Cartesian coordinate axis for the selected point.

## Get joint angle of specified point

get CurrentPoint ▾ J1 ▾ value

**Description:** Get the angle of the specified joint for the selected point.

**Parameter:**

- Select a point to get the joint angle.
- Select a joint to get the angle.

**Return:** The angle of the specified joint for the selected point.

## Forward kinematics (joint angles to posture)

Get pose by converting joint angles of P1 ▾ using User 0 ▾ and Tool 0 ▾ (forward kinematics)

**Description:** Calculate the robot's end-effector posture in the specified Cartesian coordinate system based on the given joint angles.

**Parameter:**

- Select a point, where the joint angles will be used for the forward kinematics calculation.
- User coordinate system index.
- Tool coordinate system index.

**Return:** The pose variable obtained through the forward kinematics calculation, format: `{pose = {x, y, z, rx, ry, rz} }`.

## Inverse kinematics (posture to joint angles)

Get joint angles by converting pose of P1 ▾ using User 0 ▾ and Tool 0 ▾ (inverse kinematics)

**Description:** Calculate the robot's joint angles based on the given coordinates in the specified Cartesian coordinate system. Since the Cartesian coordinates only define the spatial position and tilt angle of the TCP, the robot can reach the same pose through different configurations, meaning one pose can correspond to multiple sets of joint angles. This block will return the set of joint angles closest to the robot's current configuration.

**Parameter:**

- Select a point, where the pose will be used for the inverse kinematics calculation.
- User coordinate system index.
- Tool coordinate system index.

**Return:** Two variables, which can be printed for review. The first variable is the error code. 0 indicates that the inverse kinematics was successful, -1 indicates failure (no solution). The second variable is the joint angles obtained from the inverse kinematics, format: `{joint = {j1, j2, j3, j4, j5, j6} }`. If the inverse kinematics fails, all values for J1 to J6 will be 0.

## Get encoder value

get ABZ value

**Description:** Get the current value of the ABZ encoder.

**Return:** The current value of the encoder.

# Modbus

The Modbus blocks are used for operations related to Modbus communication. You can refer to the [corresponding DEMO](#) for a quick experience of these blocks.

The Modbus function codes for different types of registers follow the standard Modbus protocol:

Register type	Read register	Write single register	Write multiple registers
Coil register	01	05	0F
Contact (discrete input) register	02	-	-
Input register	04	-	-
Holding register	03	06	10

## Create Modbus master



**Description:** Create a Modbus TCP master based on the controller's network interface and establish a connection with the slave. A maximum of 15 slaves can be connected at the same time.

### Parameter:

- Enter the IP address of the Modbus slave.
- Enter the port of the Modbus slave.
- Select the Modbus slave ID.

When connecting to the robot's built-in slave, set the IP to the robot's IP (default 192.168.5.1, modifiable) and the port to 502 (map1) or 1502 (map2). See [Appendix A Modbus Register Definition](#) for details.

When connecting to a third-party slave, refer to the instructions of the corresponding Modbus register address for the read/write register address range and definition.

## Create tool-RS485-based Modbus master



**Description:** Create a Modbus TCP master based on the end-effector RS485 interface and establish a connection with the slave. A maximum of 15 devices can be connected at the same time.

### Parameter:

- Enter the IP address of the Modbus slave.
- Enter the baud rate of the RS485 interface.
- Select the Modbus slave ID.
- Select whether to use a parity bit.
- Select the length of the stop bit.

When connecting to the robot's built-in slave, set the IP to the robot's IP (default: 192.168.5.10, modifiable) and the port to 60000 (this cannot be set or modified).

When connecting to a third-party slave, refer to the instructions of the corresponding Modbus register address for the read/write register address range and definition.

## Create RS485-based Modbus master



**Description:** Create a Modbus RTU master based on the controller's RS485 interface and establish a connection with the slave. A maximum of 15 devices can be connected at the same time.

### Parameter:

- Enter the baud rate of the RS485 interface.
- Select the Modbus slave ID.
- Select whether to use a parity bit.
- Enter the length of the data bit, currently only supports 8.
- Select the length of the stop bit.

## Get result of creating Modbus master

**get result of creating modbus master**

**Description:** Get the result of creating the Modbus master.

### Return:

- 0: Modbus master has been created successfully.
- 1: Failed to create a new master, the maximum of 15 masters has been reached.
- 2: Failed to initialize the master, check if the IP, port, and network are working properly.
- 3: Failed to connect to the slave, check if the slave is functioning and the network is normal.

## Wait for input register



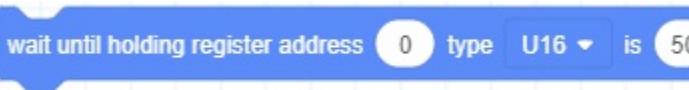
```
wait until input register address 0 type U16 is 50
```

**Description:** Wait until the value at the specified address of the input register meets the condition before executing the next command.

**Parameter:**

- The starting address of the input register.
- Data type:
  - U16: 16-bit unsigned integer (two bytes, occupy one register).
  - U32: 32-bit unsigned integer (four bytes, occupy two register).
  - F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
  - F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).
- The condition that the value at the specified address in the input register must satisfy.

## Wait for holding register



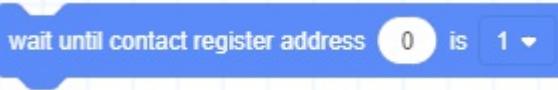
```
wait until holding register address 0 type U16 is 50
```

**Description:** Wait until the value at the specified address of the holding register meets the condition before executing the next command.

**Parameter:**

- The starting address of the holding register.
- Data type:
  - U16: 16-bit unsigned integer (two bytes, occupy one register).
  - U32: 32-bit unsigned integer (four bytes, occupy two register).
  - F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
  - F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).
- The condition that the value at the specified address in the holding register must satisfy.

## Wait for contact register



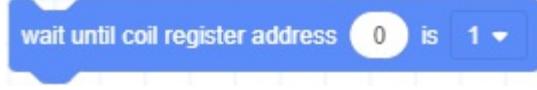
```
wait until contact register address 0 is 1
```

**Description:** Wait until the value at the specified address of the contact register meets the condition before executing the next command.

**Parameter:**

- The starting address of the contact register.
- The condition that the value at the specified address in the contact register must satisfy.

## Wait for coil register



wait until coil register address 0 is 1 ▾

**Description:** Wait until the value at the specified address of the coil register meets the condition before executing the next command.

**Parameter:**

- The starting address of the coil register.
- The condition that the value at the specified address in the coil register must satisfy.

## Get input register



get input register address 0 type U16 ▾

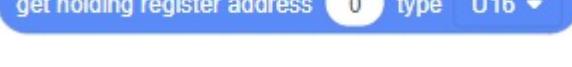
**Description:** Get the value from the specified address of the input register.

**Parameter:**

- The starting address of the input register.
- Data type:
  - U16: 16-bit unsigned integer (two bytes, occupy one register).
  - U32: 32-bit unsigned integer (four bytes, occupy two register).
  - F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
  - F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

**Return:** The value from the specified address of the input register.

## Get holding register



get holding register address 0 type U16 ▾

**Description:** Get the value from the specified address of the holding register.

**Parameter:**

- The starting address of the holding register.
- Data type:
  - U16: 16-bit unsigned integer (two bytes, occupy one register).
  - U32: 32-bit unsigned integer (four bytes, occupy two register).
  - F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
  - F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

**Return:** The value from the specified address of the holding register.

## Get contact register

get contact register address 0

**Description:** Get the value from the specified address of the contact register.

**Parameter:** The starting address of the contact register.

**Return:** The value from the specified address of the contact register.

## Get coil register

get coil register address 0

**Description:** Get the value from the specified address of the coil register.

**Parameter:** The starting address of the coil register.

**Return:** The value from the specified address of the coil register.

## Get multiple values of coil register

get coils register array address 0 bits 1

**Description:** Get multiple values from the specified address of the coil register.

**Parameter:**

- The starting address of the coil register.
- Number of register bits.

**Return:** Coil register values stored in table. The first value in table corresponds to the value of coil register at the starting address.

## Get multiple values of holding register

set holding register address 0 data 50 type U16 ▾

**Description:** Get multiple values from the specified address of the holding register.

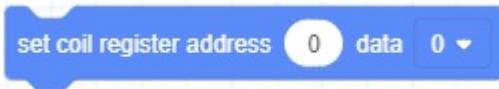
**Parameter:**

- The starting address of the holding register.
- Number of values to be read.
- Data type:
  - U16: 16-bit unsigned integer (two bytes, occupy one register).

- U32: 32-bit unsigned integer (four bytes, occupy two register).
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

**Return:** Coil register values stored in table. The first value in table corresponds to the value of coil register at the starting address.

## Set coil register



**Description:** Write a specified value to the specified address of the coil register.

**Parameter:**

- The starting address of the coil register.
- Select the value to write to the register (either 0 or 1).

## Set multiple coil registers



**Description:** Write multiple values to the specified address of the coil register.

**Parameter:**

- The starting address of the coil register.
- Enter the values to write to the register, separated by commas. Each value must be either 0 or 1.

## Set holding register



**Description:** Write a specified value to the specified address of the holding register.

**Parameter:**

- The starting address of the holding register.
- Select the value to write to the register, which must match the chosen data type.
- Data type:
  - U16: 16-bit unsigned integer (two bytes, occupy one register).
  - U32: 32-bit unsigned integer (four bytes, occupy two register).
  - F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
  - F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

## **Close Modbus master**

close modbus master

**Description:** Close the Modbus master and disconnect from all slaves.

# Bus

The Bus blocks are used to read and write Profinet or Ethernet/IP bus registers. For details on how to use the bus communication function, please refer to the *Dobot Bus Communication Protocol Guide (EtherNet/IP, Profinet)*.

## NOTE

Magician E6 does not support this set of commands.

## Get bus register value



**Description:** Get the value of the specified bus register.

### Parameter:

- Select the register type, supporting **Bus-input register** and **Bus-output register**.
- Select the data type of the register, supporting **bool**, **int**, or **float**.
- Select the register address.

**Return:** The value of the specified register. For **bool** type, the value will be either **0** or **1**.

## Set bus register value



**Description:** Set the value of the specified bus register.

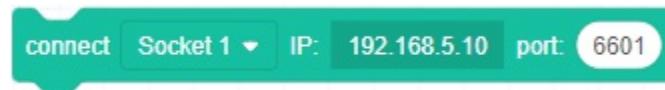
### Parameter:

- Select the register type, supporting **Bus-output register**.
- Select the data type of the register, supporting **bool**, **int**, or **float**.
- Select the register address.
- Enter the value to set. For **bool** type, the value will be either **0** or **1**.

# TCP

The TCP blocks are used for operations related to TCP communication. You can refer to the [corresponding DEMO](#) for a quick experience of these blocks.

## Connect SOCKET



**Description:** Create a TCP client to communicate with the specified TCP server.

**Parameter:**

- Select the SOCKET index (4 TCP communication links at most can be established).
- The IP address of the TCP server.
- The port of the TCP server.

## Get result of connecting SOCKET

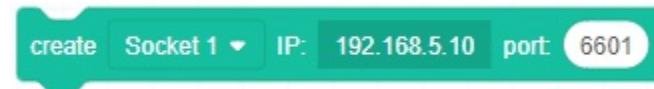


**Description:** Get the result of TCP communication connection.

**Parameter:** Select the SOCKET index.

**Return:** It returns **0** for successful connection, **1** for failed connection.

## Create SOCKET



**Description:** Create a TCP server and wait for a client to connect.

**Parameter:**

- Select the SOCKET index (4 TCP communication links at most can be established).
- The IP address of the TCP server.
- The port of the TCP server. Avoid using the following ports, as they are occupied by the system, which may cause server creation to fail.  
7, 13, 22, 37,

139, 445, 502, 503,  
1501, 1502, 1503, 4840, 8172, 9527,  
11740, 22000, 22001, 29999, 30004, 30005, 30006,  
60000 – 65504, 65506, 65511 – 65515, 65521, 65522

## Get result of creating SOCKET

get result of creating **Socket 1** ▾

**Description:** Get the result of creating TCP server.

**Parameter:** Select the SOCKET index.

**Return:** It returns **0** for successful creation, **1** for failed creation.

## Close SOCKET

close **Socket 1** ▾

**Description:** Close the specified SOCKET and disconnect the communication link.

**Parameter:** Select the SOCKET index.

## Get variable

get variable **Socket 1** ▾ type: **string** ▾ name:  waittime **0** s

**Description:** Get variable via TCP communication and save it.

**Parameter:**

- Select the SOCKET index.
- Select the variable type (string or number).
- name: Specify the variable to store the received data using a previously created variable block.
- wait time: Set the timeout period. If set to 0, there will be no timeout, and it will wait indefinitely until the variable is received.

## Get result of reading variables

Get **Socket 1** ▾ read variable result

**Description:** Get the result of reading the variable via TCP.

**Parameter:** Select the SOCKET index.

**Return:** It returns **0** for successful reading, **1** for failed reading.

## Send variable



send variable    Socket 1 ▾    Hello world

**Description:** Send variable via TCP communication.

**Parameter:**

- Select the SOCKET index.
- The data to be sent, you can use another oval block that returns string or numeric value, or directly fill in the blank.

## Get result of sending variable



get result of sending variables from    Socket 1 ▾

**Description:** Get the result of sending the variable via TCP.

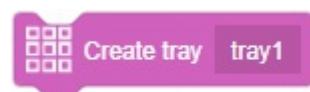
**Parameter:** Select the SOCKET index.

**Return:** It returns **0** for successful sending, **1** for failed sending.

# Tray

The tray is a carrying device for placing batch materials in a regular arrangement, commonly used in automated loading and unloading. There are usually many grooves distributed in an array in the tray, each of which can place one material. Using tray commands allows you to create a full array of tray points from a few taught points, enabling quick implementation of automated loading and unloading for robots.

## Create tray



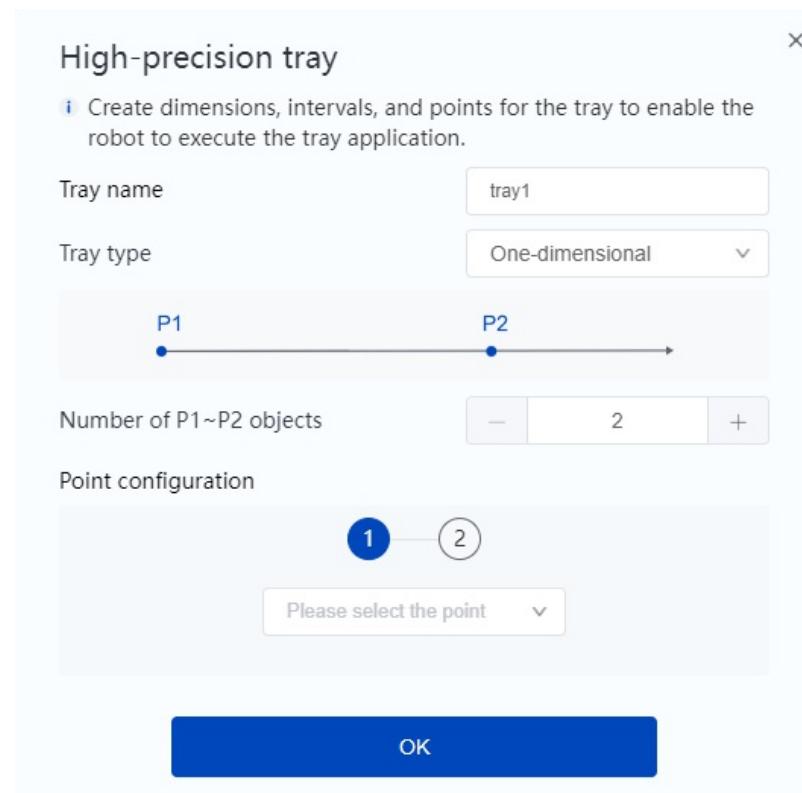
**Description:** Create tray, e.g. 1D, 2D and 3D trays. You can create up to 20 trays. Creating a tray with an existing name will overwrite the current tray without increasing the tray count.

### Parameter:

Enter the tray name in the block and click the block to open the settings window.

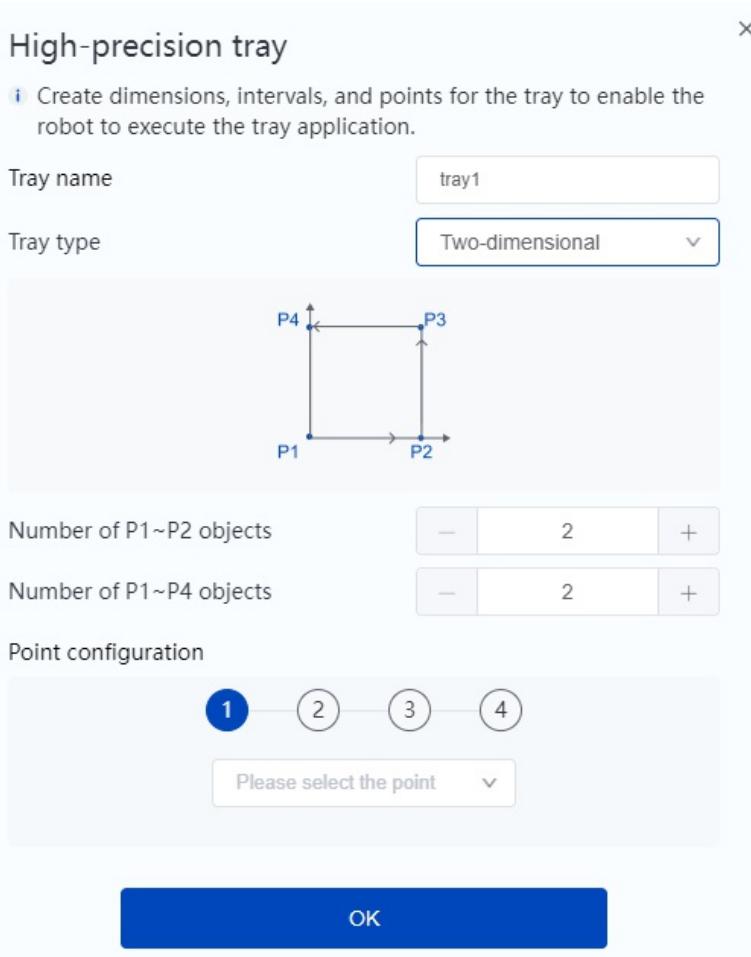
Select the tray dimension first, and configure the parameters accordingly.

- 1D tray



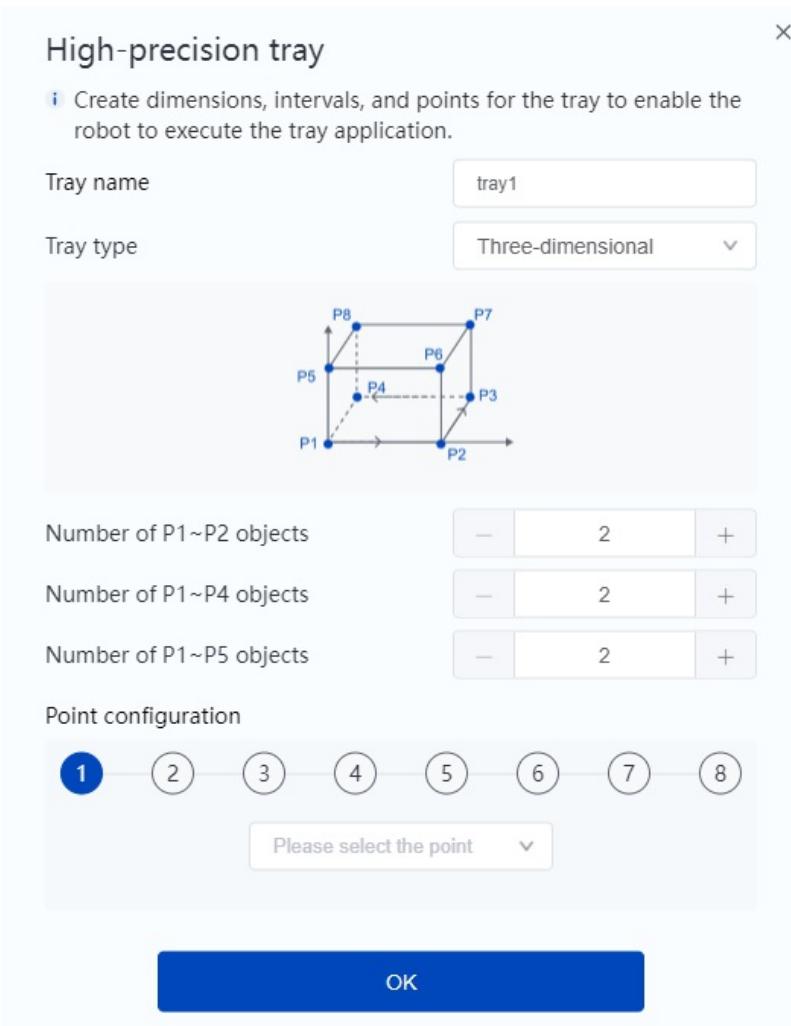
1D tray is a set of points arranged in a straight line, evenly spaced.

- Enter the number of objects between P1 and P2, which represents the total number of points in the 1D tray.
- Configure points P1 and P2 by selecting from the saved points list.
- 2D tray



2D tray is a set of points arranged in a grid on a flat plane.

- Enter the number of objects between P1 and P2 (rows) and between P1 and P4 (columns). The total number of points is the product of these two values.
- Configure points P1 to P4 by selecting from the saved points list.
- 3D tray



3D tray is a set of points distributed in a three-dimensional space, resembling multiple 2D trays stacked vertically.

- Enter the number of objects between P1 and P2 (rows), P1 and P4 (columns), and P1 and P5 (layers). The total number of points is the product of these three values.
- Configure points P1 to P8 by selecting from the saved points list.

#### **⚠️NOTICE**

If an end tool is used, please make sure that the tool coordinate system for the end tool is selected when teaching points.

## Get total tray points

Get **tray1** **tray point counts**

**Description:** Get the total number of points in the created tray.

**Parameter:** Select the name of the created tray.

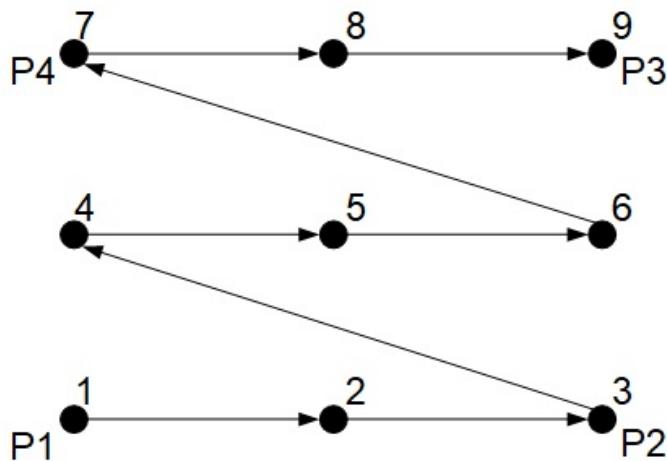
**Return:** The total number of points in the specified tray.

## Get tray point

Get tray1 ▾ tray point 1

**Description:** Get the point of the specified index from the specified tray. The point index is determined based on the order in which points were set when creating the tray.

- 1D tray: The index of P1 is 1, the index of P2 is the same as the number of points, and so on.
- 2D tray: The following figure takes a 3x3 tray as an example to illustrate the relationship between taught point and point index.



- 3D tray: Similar to 2D tray, the index of the first point on the second layer is the index of the last point on the first layer plus one, and so on.

**Parameter:**

- Select the name of the created tray.
- Enter the index of the point to obtain.

**Return:** The coordinates of the specified point.

# Quick start

- **Reading and writing Modbus register data**
- **Transmitting data via TCP communicationn**

# Reading and writing Modbus register data

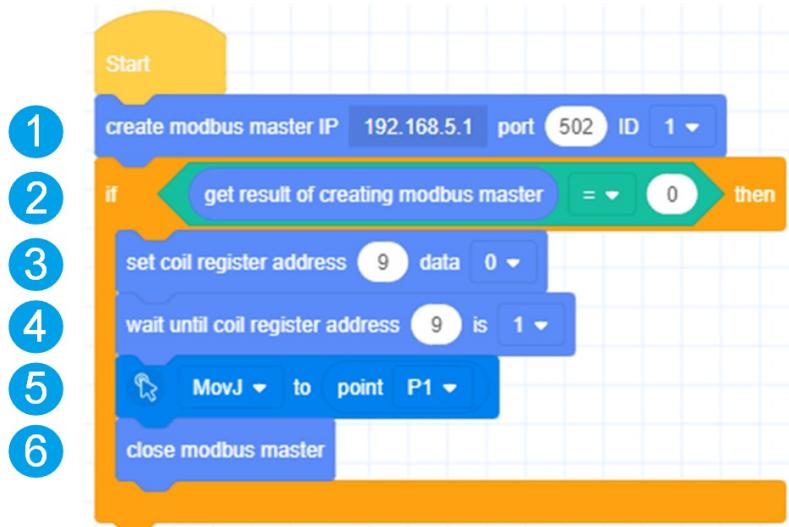
## Scenario description

To demonstrate how to read and write Modbus data using Blockly programming, let's assume the following scenario:

The robot creates a Modbus master, connects to an external slave, and reads the value at the specified coil register address. If the value at this address is 1, the robot moves to P1.

## Programming steps

To achieve the scenario described above, we need to write a program shown in the figure below.



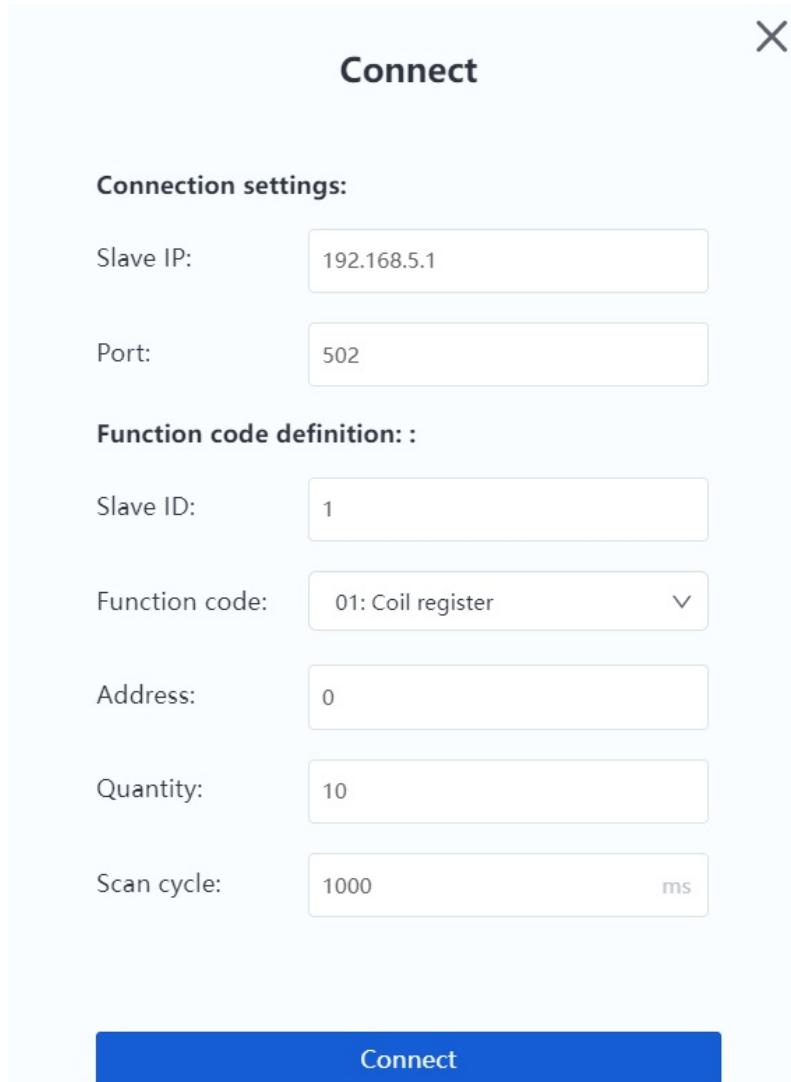
1. Create a Modbus master. Set the IP address to the address of the slave device, and the port and ID to their default values. In this demo, for quick verification, we will use the robot's built-in slave, so the IP address is set to the robot's IP (default: 192.168.5.1, but can be modified).
2. Check if the master is successfully created. The subsequent steps will only be executed if the master is successfully created. Otherwise, the program will end.
3. Set coil register 0 to 0. The value of coil register 9 may have been modified before, which could affect the program logic. Therefore, it is necessary to set the value of coil register 0 to 0 here.
4. Wait for the value of coil register 9 to change to 1.
5. Control the robot to move to P1, which is a user-defined point.
6. Close the Modbus master.

If you need to connect to a third-party slave, modify the IP and port settings in the "create modbus master" block to the address of the third-party slave. For the range and definition of the register address when reading and writing the register, please refer to the instructions of the corresponding slave.

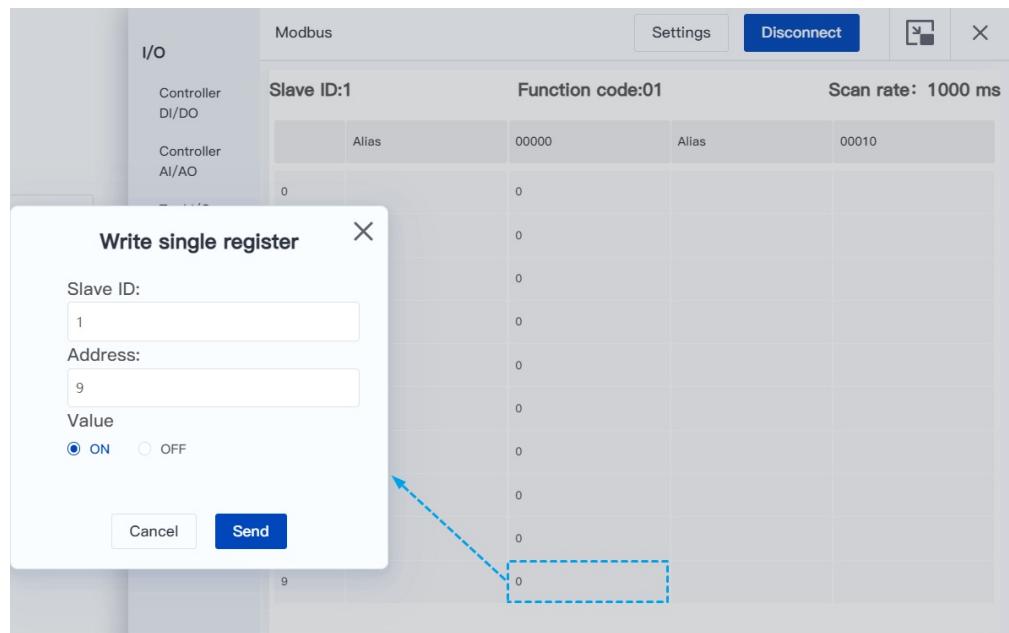
## Running the program

To quickly run this program, you can use a [Modbus monitoring](#) tool to modify the value of the coil register.

1. Open **Monitor > Modbus** page and click **Connect** in the upper right corner.
2. The default connection settings are shown in the figure below, and no changes are required, just click **Connect**.



3. Once the Modbus master is successfully created, double-click (on PC) or click (on App) on the cell with the corresponding value of coil register 9, and the "Write Single Coil" window pops up.
4. Change the **Value** of the coil to **ON** and click **Send**.



5. Observe whether the robot moves to P1.

# Transmitting data via TCP communication

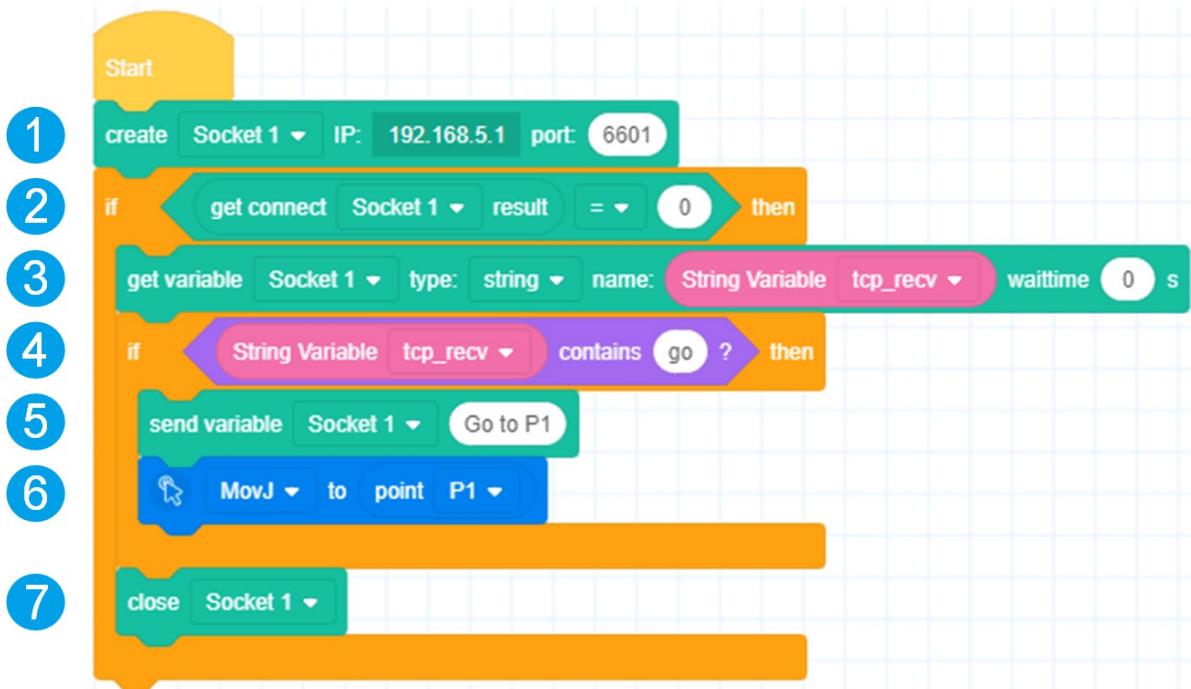
## Scenario description

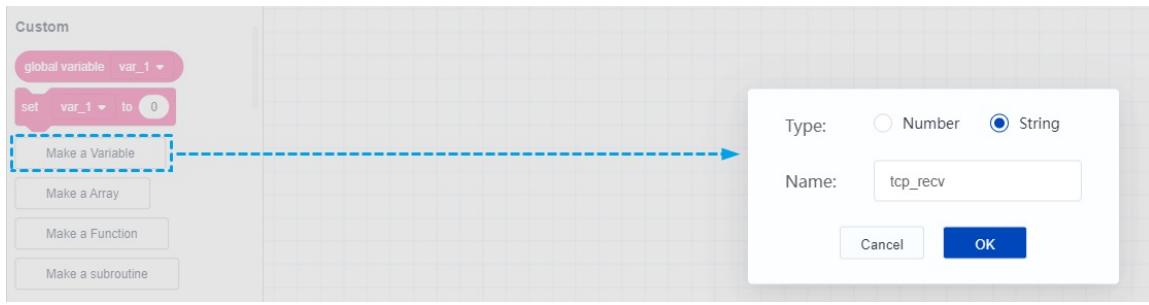
To demonstrate how to use Blockly programming for TCP communication, let's assume the following scenario:

The robot creates a TCP server, waits for a client connection, and sends the "go" command. Upon receiving the command, the robot responds with the message "Go to P1" and then moves to P1.

## Programming steps

To achieve the scenario described above, we need to write a program shown in the figure below.





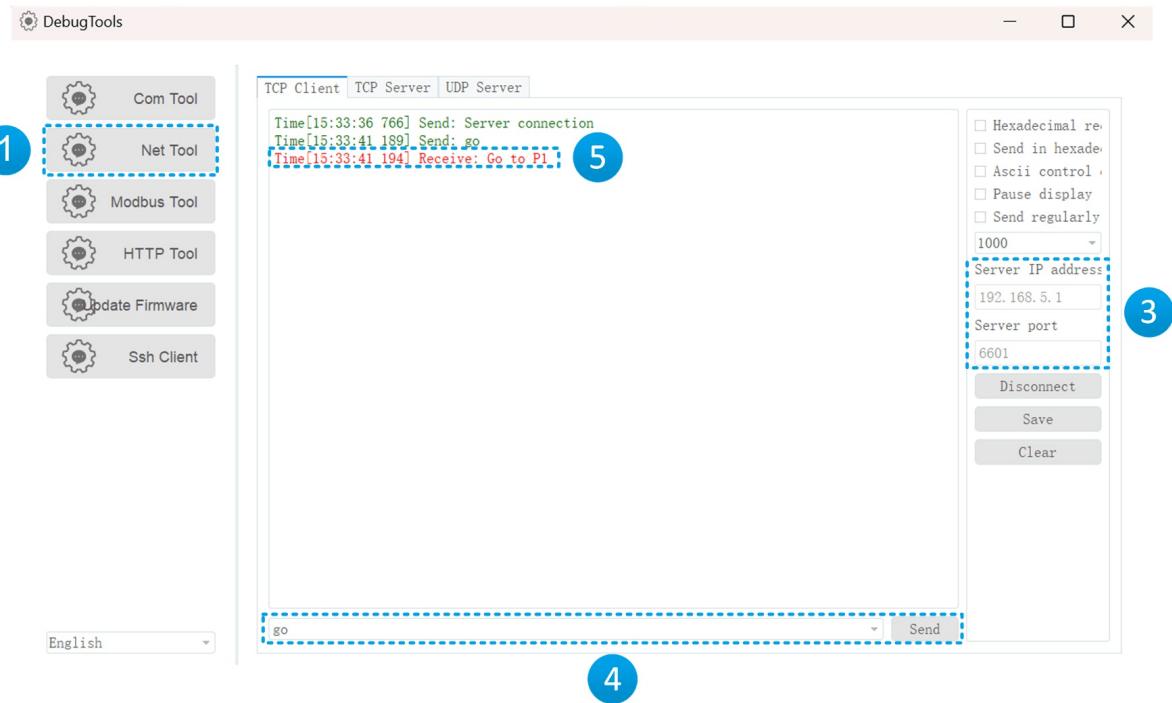
4. Check if the received string contains "go". If the string contains "go", proceed to steps 5 and 6. Otherwise, proceed to step 7.
5. Send the string "Go to P1" to the client.
6. Control the robot to move to P1, which is a user-defined point.
7. Close the TCP server.

## Running the program

To quickly run the program, you can use the built-in debugging tool of DobotStudio Pro as the TCP client.

1. Open **Settings > Debug Tool**, then navigate to **Net Tool > TCP Client**.
2. Move the robot to a point other than P1 (for observing whether it executes the motion command later). Then save and run the program.
3. Once the program logs show that the TCP server is successfully created, modify the server's IP address and port in DebugTools page, and click **Connect**.
4. After a successful connection, enter "go" in the input field at the bottom of DebugTools page and click **Send**.
5. Observe whether the debugging tool receives the message "Go to P1" and if the robot moves to P1.

The figure below shows the interface of the debugging tool. The marked numbers correspond to the steps above.



# Appendix C Script Programming Command

- [C.1 Basic syntax](#)
- [C.2 General description](#)
- [C.3 Motion command](#)
- [C.4 Relative motion command](#)
- [C.5 Motion parameter](#)
- [C.6 IO](#)
- [C.7 Tool](#)
- [C.8 TCP&UDP](#)
- [C.9 Modbus](#)
- [C.10 Bus register](#)
- [C.11 Program control](#)
- [C.12 Tray](#)
- [C.13 SafeSkin](#)

# Basic syntax

- Basic concepts
- Variable and data type
- Operator
- Flow control
- Functions
- General mathematical functions
- General string-processing functions
- General table (array) functions

# Basic concepts

## NOTE

If you want to systematically learn about Lua programming, please search for Lua tutorials online. This guide only lists some basic Lua syntax for quick reference.

## Identifiers

Identifiers are used to define variables, functions, or other user-defined items. An identifier must start with a letter (A-Z or a-z) or an underscore (\_), followed by zero or more letters, underscores, or digits (0-9).

It is best to avoid using identifiers that start with an underscore followed by uppercase letters, as Lua's reserved words also follow this pattern.

Lua does not allow the use of special characters such as @, \$, and % to define identifiers.

**Lua is case-sensitive, so all identifiers in your program must match the case used in the examples provided in this guide.**

## Keywords

Here are the reserved keywords in Lua that cannot be used as constants, variables, or user-defined identifiers:

**and, break, do, else, elseif, end, false ,for, function, if , in , local, nil, not, or, repeat, return, then, true, until, while, goto**

Generally, names that start with an underscore followed by uppercase letters (like \_VERSION) are reserved for Lua's internal global variables.

## Comments

Comments do not affect the execution of the program and are primarily used to help readers understand the code.

### Single-line comment

In a single line of code, anything after two dashes (--) is treated as a comment. Comments can be on a separate line or at the end of a line of code.

```
-- Single-line comment
print("Hello World!") -- Single-line comment
```

## **Multi-line comment**

Multi-line comments start with “--[ [” and end with “--] ]”. Everything between these markers is treated as a comment.

```
--[ [  
    Multi-line comment  
    Multi-line comment  
--]]
```

# Variable and data type

Variables are used to store values, pass values as parameters, or return results. Values are assigned to variables using the "`=`" operator.

In Lua, a variable is explicitly declared as a local variable using **local** (valid within the current function), with its scope extending from the point of declaration until the end of the function.

Variables that are not explicitly declared as **local** default to script-level variables (valid within the current script file), with a scope limited to a single script file.

In addition, you can set controller-level global variables via **Monitor > Global variables** page in DobotStudio Pro, which can be called directly from different script files within the same controller.

The screenshot shows the 'Global variable' configuration screen. On the left, there's a sidebar with sections for I/O (Controller DI/DO, Controller AI/AO, Tool I/O, Safety I/O), Modbus, and Variable (Global variable, Program variables). The 'Global variable' section is selected. The main area has a table with columns: NO, Variable Name, Type, Global Hold, Range, and Value. One row is present: NO 1, Variable Name var\_1, Type number, Global Hold checked, Range empty, and Value 50. There are buttons for Delete, Modify, and New at the top right of the table.

NO	Variable Name	Type	Global Hold	Range	Value
1	var_1	number	<input checked="" type="checkbox"/>		50

**Example 1:** Mainly illustrates the differences of the functioning scope between local variables and script-level variables.

```
function func()      -- Define a function, and print a and b when running
    local a = 1      -- Local variable
    b = 2            -- Script-level variable

end

func()              -- Execute a function, print a and b as 1 and 2
print(a)            --> nil (Called outside the functioning scope of the variable and printed as
nil)
print(b)            --> 2
```

**Example 2:** Mainly illustrates that local variables can have the same name as script-level variables, but only take effect within functions. The code blocks for flow control (loop, conditional judgment) such as "do/end" also belong to functions.

```
a = "a"

for i=10,1,-1 do
do
    local a = 6      -- Local variable
    print(a)         --> 6, the variable called within the statement block is the local variabl
e "a"
end

print(a)           --> a, the variable called outside the statement block is the script-level var
iable "a"
```

**Example 3:** Mainly illustrates the differences of the functioning scope between global variables and another two variables, as well as the global hold function of global variables.

```
-- src0.lua file for project 1
-- Assume two global variables g1 and g2 have been added to the global variables page
-- g1 is non-global hold variable, value: 10
-- g2 is global hold variable, value: 20

local a = 1
b = 2
print(a)      --> 1
print(b)      --> 2

print(g1)      --> 10
print(g2)      --> 20
SetGlobalVariable("g1",11) -- Assign a value to non-global hold variable
SetGlobalVariable("g2",22) -- Assign a value to global hold variable
print(g1)      --> 11
print(g2)      --> 22

-- src0.lua file for project 2
-- Project 2 runs after Project 1

print(a)      --> nil
print(b)      --> nil
print(g1)      --> 10 (non-global hold variable restores to the value before modification of
project 1)
print(g2)      --> 22 (global hold variable becomes the value after the modification of proje
ct 1)
```

Variable names can be a string made up of letters, underscores and numbers, which cannot start with a number. The keywords reserved by Lua cannot be used as a variable name.

For Lua variables, you do not need to define their types. After you assign a value to the variable, Lua will automatically judge the type of the variable according to the value. Assigning a different type of value from the current type to a variable in the script will change the type of the variable. However, the type of variables set on **Monitor > Global variables** page cannot be changed, and the type of the assigned value must be the same as the type when the variable was created, otherwise the controller will report an error when running the script.

Lua supports a variety of data types, including number, boolean, string and table. The array in Lua is a type of table.

There is also a special data type in Lua: nil, which means void (without any valid values). For example, if you print an unassigned variable, it will output a nil value.

## Number

The number in Lua is a double precision floating-point number and supports various operations. The following format are all regarded as a number:

- 2
- 2.2
- 0.2
- 2e+1
- 0.2e-1
- 7.8263692594256e-06

## Boolean

The boolean type has only two optional values: true and false. Lua treats false and nil as false, and others as true including number 0.

## String

A string is a sequence of characters consisting of numbers, letters, and underscores. Strings in Lua can be represented in three ways:

- Characters between single quotes.
- Characters between double quotes.
- Characters between [[ and ]].

When performing arithmetic operations on a numeric string, Lua will attempt to convert the numeric string into a number.

```
-- Defining a string with single quotes
local str1 = 'Hello, Lua!'
print(str1) -- Output: Hello, Lua!
```

```

-- Defining a string with double quotes
local str2 = "Hello, Lua!"
print(str2) -- Output: Hello, Lua!

-- Defining a multi-line string using [[ and ]]
local str3 = [[
This is a multi-line
string in Lua.
]]
print(str3)
-- Output:
-- This is a multi-line
-- string in Lua.

-- Lua automatically converts a numeric string to a number and performs arithmetic operations
local numStr = "10"
local result = numStr + 20 -- Automatically converts "10" to number 10
print(result) -- Output: 30

```

## Table

A table is a group of data with indexes.

- The simplest way to create a table is using {}, which creates an empty table, or you can initialize the table directly.
- In Lua, a table is essentially an associative array, meaning you can use any type of value as an index, but this value cannot be **nil**.
- Tables do not have a fixed size and can be resized as needed.
- You can get the length of a table using the # symbol.

```

-- Initialize a table with sequential indices
local tbl = {[1] = 2, [2] = 6, [3] = 34, [4] = 5}
print("tbl length ", #tbl) -- Output: 4

```

In this example, the indices [1] to [4] of the table `tbl` are sequential, so `#tbl` correctly returns the length 4.

## Array

Arrays in Lua refer to collections of elements arranged in a specific order, and they can be one-dimensional or multi-dimensional.

In Lua, arrays belong to the `table` type, and index keys can be integers. The size of the array is not fixed.

- One-dimensional array: The simplest array with a logical structure of a linear table.
- Multi-dimensional array: An array contains an array or the index of a one-dimensional array

corresponds to an array.

In Lua, the array's indexes start from 1, but you can also use 0 or negative numbers. When accessing an element at a non-existent index, it will return **nil**.

**Example 1:** One-dimensional array using a **for** loop to iterate through the elements.

```
-- Create a one-dimensional array
local array = {"Lua", "Tutorial"}

-- Iterate over the array using a for loop, starting from index 1
for i = 1, #array do
    print(array[i])      -- Output: Lua Tutorial
end
```

**Example 2:** One-dimensional array with numerical elements

```
-- Create a one-dimensional array with numerical elements
local numbers = {10, 20, 30, 40, 50}

-- Iterate over the array using a for loop, starting from index 1
for i = 1, #numbers do
    print(numbers[i])      -- Output: 10 20 30 40 50
end
```

**Example 3:** Array with custom indices

```
-- Create an array and use negative and positive numbers as indices
local array = {}
for i = -2, 2 do
    array[i] = i * 2 + 1 -- Assign values to the array
end

-- Iterate over the array using a for loop, starting from index -2 to 2
for i = -2, 2 do
    print(array[i])    -- Output: -3 -1 1 3 5
end
```

**Example 4:** A multi-dimensional array with three rows and three columns

```
-- Initialize an array
array = {}
for i=1,3 do
    array[i] = {}
    for j=1,3 do
        array[i][j] = i*j
    end
end

-- Access an array
```

```
for i=1,3 do
    for j=1,3 do
        print(array[i][j])      -- Output: 1 2 3 2 4 6 3 6 9
    end
end
```

# Operator

## Arithmetic operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Floating point division
//	Floor division
%	Remainder division
^	Exponentiation

Example:

```
a=20
b=5
print(a+b)           -- Print the result of a plus b: 25
print(a-b)           -- Print the result of a minus b: 15
print(a*b)           -- Print the result of a times b: 100
print(a/b)           -- Print the result of a divided by b: 4
print(a//b)          -- Print the result of a divisible by b: 4
print(a%b)           -- Print the remainder of a divided by b: 0
print(a^b)           -- Print the result for the b-power of a: 3200000
```

## Bitwise operators

Operator	Description
&	AND operator
\	OR operator
~	XOR operator
<<	Left shift operator
>>	Right shift operator

Example:

```
print(a&b)           -- Print the result of a AND b: 4
print(a|b)           -- Print the result of a OR b: 21
print(a~b)           -- Print the result of a XOR b: 17
```

```

print(a<<b)          -- Print the result of a shift left b: 640
print(a>>b)          -- Print the result of a shift right b: 0

```

## Relational operators

Operator	Description
==	Equal
~=	Not equal to
<=	Less than or equal to
>=	Greater than or equal to
<	Less than
>	Greater than

Example:

```

a=20                  -- Create variable a
b=5                   -- Create variable b
print(a==b)           -- Determine whether a is equal to b: false
print(a~=b)           -- Determine whether a is not equal to b: true
print(a<=b)           -- Determine whether a is less than or equal to b: false
print(a>=b)           -- Determine whether a is greater than or equal to b: true
print(a<b)            -- Determine whether a is less than b: false
print(a>b)            -- Determine whether a is greater than b: true

```

## Logical operators

Operator	Description
and	Logical AND operator. Returns true only if both sides are true. Otherwise, false.
or	Logical OR operator. Returns true if either side is true. If both sides are false, returns false.
not	Logical NOT operator. Inverts the truth value of the expression.

```

a=true
b=false
print(a and b)        -- Output: false, both a and b must be true for it to return true
print(a or b)         -- Output: true, returns true if either a or b is true
print(not a)          -- Output: false, a is true, so not a is false
print(not b)          -- Output: true, b is false, so not b is true
print(not (20>5))    -- Output: false, 20 > 5 is true, so not (20 > 5) is false

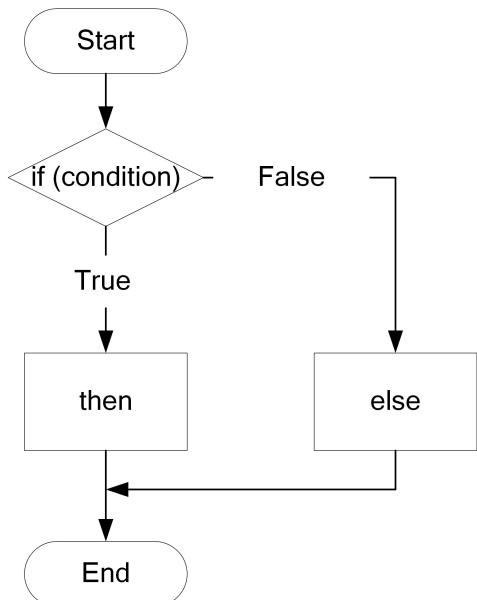
```

# Flow control

Command	NOTE
if...then... else...end	Conditional command (if). Conditions are evaluated from top to bottom. If a condition is <b>true</b> , the corresponding code block is executed, and all subsequent conditions are ignored.
while... do...end	Loop command (while). Repeat a code block as long as the condition is <b>true</b> . The condition is checked for true before the code block is executed.
for...do... end	Loop command (for). Repeat a code block a specified number of times, as controlled by the <b>for</b> statement.
repeat... until()	Loop command (repeat). Repeat a code block until a specified condition becomes <b>true</b> .

## Conditional statement (if)

The condition inside the parentheses after **if** is an expression, and its result can be any value. Lua considers **false** and **nil** as **false**, everything else is **true**, including the number 0. If the expression is **true**, the code block after **then** is executed. If the expression is **false** and an **else** block is present, the code inside the **else** block is executed. If there's no **else** block, the code proceeds to execute the statements following the **end**.



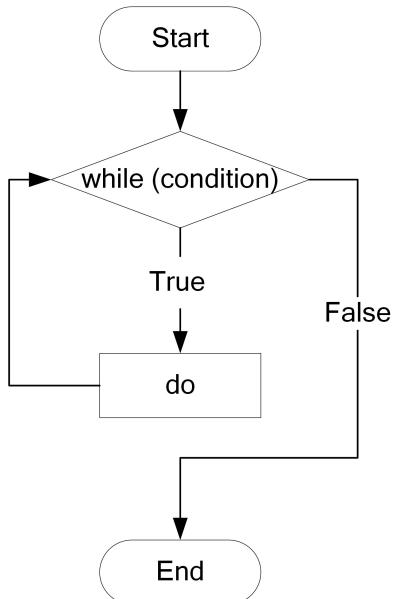
The **if** statement can be nested, the following is a typical example.

```
a = 100;  
b = 200;  
--[ Check conditions --]  
if(a == 100)  
then
```

```
--[ If the `if` condition is true, check the following `if` condition --]
if(b == 200)
then
    --[ If the `if` condition is true, execute this block --]
    print("value of a:", a ) -- value of a: 100
    print("value of b:", b ) -- value of b: 200
end
else
    --[ If the first `if` condition is false, execute the following block --]
    print("a≠100")
end
```

## Loop command (while)

The condition inside the parentheses after **while** is evaluated. If it is **true**, the **do** statement block is executed, and the condition is checked again. If it is **false**, the code proceeds to the statements after **end**.



Example:

```
a=10
while( a < 20 )
do
    print("value of a:", a) -- Execute 10 times, the output value is 10 to 19
    a = a+1
end
```

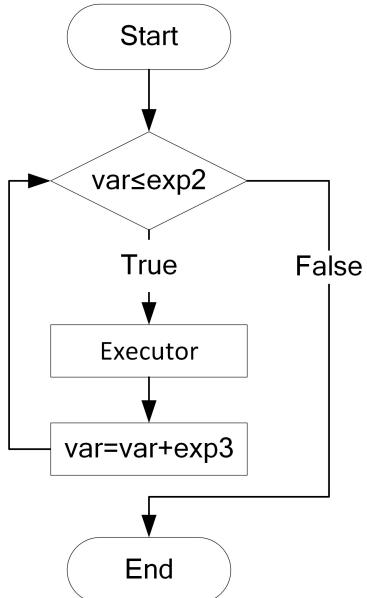
## Loop command (for)

The syntax for a **for** loop is as follows:

```
for var=exp1,exp2,exp3 do
<execution body>
```

```
end
```

The variable **var** starts with **exp1**. After each execution of \, **var** is incremented by **exp3** (**exp3** can be a negative value; if it is not specified, it defaults to 1). The loop continues until **var** exceeds **exp2**.

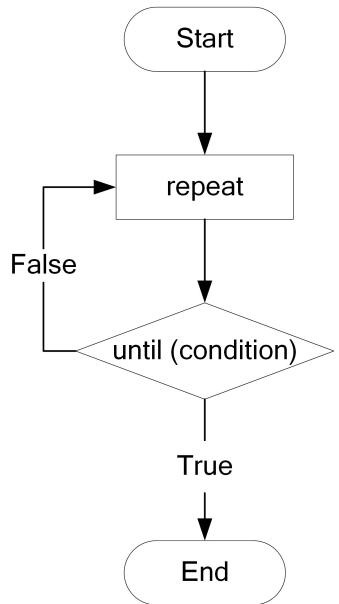


Example:

```
for i=10,1,-1 do
    print(i) -- Execute 10 times, the output value is 10 to 1
end
```

## Loop command (repeat)

The **repeat** loop is similar to the **while** loop, but with a key difference: in a **while** loop, the condition is checked before the loop body is executed. In a **repeat** loop, the condition is checked after the body has been executed. If the condition is **false**, the loop continues.



Example:

```
a = 10
repeat
    print("value of a:", a) -- Execute 5 times, the output value is 10 to 15
    a = a + 1
until(a > 15)
```

# Functions

Functions are the primary means of abstracting statements and expressions. The definition format is as follows:

```
function function_name(argument1, argument2, argument3..., argumentn)
    function_body
    return result_params_comma_separated
end
```

- **function\_name**: The name of the function. Functions can be defined first and then called by name, or defined directly during the call (in which case the name can be omitted).
- **argument1, argument2, argument3..., argumentn**: Function parameters, separated by commas. Functions can also have no parameters.
- **function\_body**: The function body refers to the code block that is executed within a function.
- **result\_params\_comma\_separated**: The return values of the function. Lua functions can return multiple values, separated by commas. Functions can also have no return values.

**Example 1:** Function without input parameters or return values

```
function greet()
    print("Hello, Lua!") -- Function body
end

greet() -- Calling the function, output: Hello, Lua!
```

**Example 2:** Function with input parameters, but no return value

```
function printSquare(number)
    print("Square of " .. number .. " is " .. (number * number)) -- Function body
end

printSquare(5) -- Calling the function, output: Square of 5 is 25
```

**Example 3:** Function with input parameters and return values

```
function maximum(a)
    local mi = 1 -- Index of the maximum value
    local m = a[mi] -- Maximum value
    for i, val in ipairs(a) do
        if val > m then
            mi = i
            m = val
        end
    end
end
```

```
    return m, mi -- Returning the maximum value and its index
end

local maxVal, maxIndex = maximum({8, 10, 23, 12, 5})
print("Max value:", maxVal) -- Output: Max value: 23
print("Index of max value:", maxIndex) -- Output: Index of max value: 3
```

### NOTE

This guide primarily explains how to use the pre-defined functions provided by Dobot. Users can also define their own functions. To create custom functions, please ensure that the function definition is written in the **global.lua** file or at the top of the **src\*.lua** file where the function is being called. Otherwise, an error will occur during runtime.

# General mathematical functions

Lua provides some basic math functions to complement arithmetic operators.

## **math.abs(X)**

Return the absolute value of X. Example:

```
print("math.abs(-10):", math.abs(-10)) -- Output: 10
```

## **math.floor(X)**

Return the largest integer less than or equal to X (round down). Example:

```
print("math.floor(3.7):", math.floor(3.7)) -- Output: 3
```

## **math.ceil(X)**

Return the smallest integer greater than or equal to X (round up) . Example:

```
print("math.ceil(3.2):", math.ceil(3.2)) -- Output: 4
```

## **math.sqrt(X)**

Return the square root of X. Example:

```
print("math.sqrt(16):", math.sqrt(16)) -- Output: 4
```

## **math.rad(X)**

Return the radian value of X (convert degrees to radians). Example:

```
print("math.rad(180):", math.rad(180)) -- Output: 3.1415926535898
```

## **math.deg(X)**

Return the degree value of X (convert radians to degrees). Example:

```
print("math.deg(math.pi):", math.deg(math.pi)) -- Output: 180
```

## **math.sin(X)**

Return the sine of X, where X is in radians.

If you want to use degrees as input, you can convert them using **math.rad**. Example:

```
print("math.sin(math.rad(30)):", math.sin(math.rad(30))) -- Output: 0.5
```

## **math.cos(X)**

Return the cosine of X, where X is in radians.

If you want to use degrees as input, you can convert them using **math.rad**. Example:

```
print("math.cos(math.rad(60)):", math.cos(math.rad(60))) -- Output: 0.5
```

## **math.tan(X)**

Return the tangent of X, where X is in radians.

If you want to use degrees as input, you can convert them using **math.rad**. Example:

```
print("math.tan(math.rad(45)):", math.tan(math.rad(45))) -- Output: 1
```

## **math.asin(X)**

Return the arcsine of X in radians. You can convert the result to degrees using **math.deg**. Example:

```
print("math.deg(math.asin(0.5)):", math.deg(math.asin(0.5))) -- Output: 30 (the return value  
is in degrees)
```

## **math.acos(X)**

Return the arccosine of X in radians. You can convert the result to degrees using **math.deg**. Example:

```
print("math.deg(math.acos(0.5)):", math.deg(math.acos(0.5))) -- Output: 60 (the return value  
is in degrees)
```

## **math.atan(X)**

Return the arctangent of X in radians. You can convert the result to degrees using **math.deg**. Example:

```
print("math.deg(math.atan(1)):", math.deg(math.atan(1))) -- Output: 45 (the return value is  
in degrees)
```

## math.log(X)

Return the natural logarithm of X. Example:

```
print("math.log(10):", math.log(10)) -- Output: 2.302585092994
```

## math.exp(X)

Return e (natural exponent) raised to the power of X. Example:

```
print("math.exp(1):", math.exp(1)) -- Output: 2.718281828459
```

# General string-processing functions

Lua provides general functions for string manipulation, such as searching and replacing strings.

## string.sub(s, i, j)

Used to extract a string.

### Required parameter

- **s:** The string to be extracted.
- **i:** The starting position for extraction (1-based index).

### Optional parameter

- **j:** The ending position for extraction, default is -1, which refers to the last character.

### Return

The extracted string.

### Example

```
sub1 = string.sub("abcde", 3) -- Extract from the 3rd bit: cde
sub2 = string.sub("abcde", 1, 3) -- Extract from the 1st to the 3rd bit: abc
sub3 = string.sub("abcde", 3, 3) -- Extract from the 3rd bit: c
```

## string.find (s, sub, i, plain)

Search for a substring within a specified string and return the found index.

### Required parameter

- **s:** The string to be searched.
- **sub:** The substring to be searched.

### Optional parameter

- **i:** The starting position for the search, default is 1.
- **plain:** Specify whether to use plain text mode. If this parameter is set, **i** must also be specified.
  - **true:** Use plain text; sub will be treated as a plain text string.
  - **false:** Default value; sub supports [pattern matching](#).

### Return

- The starting and ending indices of the first matched substring in the original string.
- If captures are defined in the pattern, the captured values will be returned after the two indices.
- Return **nil** if the substring is not found.

### Example

```

i,j = string.find("abcde", "cd") -- Search for "cd" in "abcde"; return: i = 3, j = 4

if string.find(str1, str2) == nil -- If "str1" contains str2, then execute the contents of TODO
0.
then
--TODO
end

-- Advanced usage
i,j = string.find("abcabc", "ab", 3) -- Search for "ab" in "abcabc" starting from the 3rd bit
; return: i = 4, j = 5.

i,j = string.find("123%abc", "%a") -- Support pattern matching by default; "%a" is interpreted
as a wildcard (any character), matching the first letter 'a', i and j both = 5.
i,j = string.find("123%abc", "%a", 1, true) -- Use plain text mode; "%a" is treated as plain
text, matching "%a" in the original string, i = 4, j = 5.

i,j,sub = string.find("abc 10 edf 100", "(%d+)") -- Search for and capture the first sequence
of digits; return indices and captured result: i = 5, j = 6, sub = 10.

```

## string.match(s, sub, i)

Search for a substring in a specified string and return the captured result or substring.

### Required parameter

- **s:** The string to be searched.
- **sub:** The substring to be searched, supports [pattern matching](#).

### Optional parameter

- **i:** The starting position for the search, default is 1.

### Return

- The matched substring.
- If captures are defined in the pattern, all captured results will be returned.
- Return **nil** if the substring is not found.

### Example

```

sub1 = string.match("abcde", "cd") -- Search for "cd" in "abcde"; return the matched substrin
g: "cd".

sub2 = string.match("abc 10 edf 100", "%a+", 4) -- "%a+" represents consecutive letters; sear
ch starting from the 4th bit; return: "edf".

sub3 = string.match("abc 10 edf 100", "%a+ (%d+) %a+") -- Capture the number between two sequ
ences of letters; return: 10.

```

## string.gmatch(s, sub)

Loop through a specified string to find substrings, returning the captured results or substrings.

### Required parameter

- **s:** The string to be searched.
- **sub:** The substring to be searched, supports [pattern matching](#).

### Return

- An iterator function that returns a match result each time it is called.
- If no match is found, the iterator function returns **nil**.

### Example

```
for word in string.gmatch("Hello world from dobot", "%a+")
do
    print(word)
end
--[ [
The output will print each word line by line:
Hello
world
from
dobot
--]]
```

## string.gsub(s, find, repl, n)

Used to replace specified parts of a string.

### Required parameter

- **s:** The string to be replaced.
- **find:** The characters to be replaced; sub supports [pattern matching](#).
- **repl:** The characters to replace with.

### Optional parameter

- **n:** The maximum number of replacements. If not specified, all occurrences will be replaced.

### Return

The replaced string and the number of replacements.

### Example

```
str, n = string.gsub("aaaa","a","z"); -- Replace all 'a' with 'z'; return: str = "zzzz", n = 4.

str, n = string.gsub("aaaa","a","z",3); -- Replace the first 3 'a' with 'z'; return: str = "zz
za", n = 3.

str, n = string.gsub("a1b2c3","%d","z"); -- Replace all digits with 'z'; return: str = "azbzcz
", n = 3.
```

## Pattern matching mechanism

Lua's pattern matching mechanism is similar to regular expressions and can be used with **string.find**, **string.gmatch**, **string.gsub**, **string.match** for fuzzy matching.

### Character

Character classes are the basic units of pattern matching, representing specific sets of characters. The main character classes are:

- Single character (except `^$().%.[ ]\*+-?`): Match the character itself.
- `.` : Match any character.
- `%a` : Match any letter.
- `%c` : Match any any control character (e.g., `\n`).
- `%d` : Match any digit.
- `%l` : Match any lowercase letter.
- `%p` : Match any punctuation.
- `%s` : Match any whitespace character (space).
- `%u` : Match any uppercase letter.
- `%w` : Match any alphanumeric character.
- `%x` : Match any hexadecimal digit.
- `%x` (where x is a non-alphanumeric character): Match the character x, mainly used for matching special characters (`^$().%.[ ]\*+-?`), e.g., `%%` matches `%`.
- `[multiple character classes]` : Match any character within the brackets (`[]`). For example, `[%w_]` matches any alphanumeric character (`%w`) or underscore (`_`).
- `[^multiple character classes]` : Match any character not in the brackets (`[]`). For example, `[^%s%p]` matches any non-whitespace and non-punctuation character.

All single-letter categories (like `%a`, `%c`, etc.) become their corresponding complements when capitalized. For example, `%S` matches any non-space character.

### Pattern entries

Pattern entries are combinations of character classes and special symbols used to define how characters are matched. Common pattern entries include:

- A single character class matches any single character from that class.
- A single character class followed by `*` matches zero or more characters from that class. This entry always matches the longest possible string.
- A single character class followed by `+` matches one or more characters from that class. This entry always matches the longest possible string.
- A single character class followed by `-` matches zero or more characters from that class. Unlike `*`, this entry always matches the shortest possible string.
- A single character class followed by `?` matches zero or one character from that class.

- `%n`, where n is an integer from 1 to 9, matches the same substring captured by the nth capture (see **Captures** section for details).

## Mode

A pattern is a sequence of pattern entries. For example:

- To match dates in the format **dd/mm/yyyy**, you can use the pattern `%d%d/%d%d/%d%d%d%` .
- To match words starting with an uppercase letter, you can use the pattern `%u%l+` .

## Captures

A pattern can enclose sub-patterns in parentheses, which are called captures.

When a match is successful, the substring matched by the capture is saved and can be returned or used later for further operations. Captures are numbered in the order of their opening parentheses. For example, in the pattern `(a*(.)%w(%s*))` :

- The substring that matches the entire pattern `a*(.)%w(%s*)` is Capture 1.
- The character that matches `.` is Capture 2.
- The substring that matches `%s*` is Capture 3.
- Captures 2 and 3 are substrings of Capture 1.

As a special case, an empty pair of parentheses `()` will capture the position of the corresponding character in the string. For instance, if you apply the pattern `()aa()` to the string `flaaap`, two captures will be generated: 3 and 5.

## Other common methods

Function	NOTE
<code>string.upper(argument)</code>	Convert the entire string to uppercase letters
<code>string.lower(argument)</code>	Convert the entire string to lowercase letters
<code>string.reverse(arg)</code>	Reverse the string
<code>string.format(...)</code>	Return a formatted string similar to <b>printf</b>
<code>string.len(arg)</code>	Calculate the length of a string
<code>string.rep(string, n)</code>	Return n copies of the string

### Example:

```

str = "Lua"
print(string.upper(str))      -- Convert to uppercase letters, result: LUA
print(string.lower(str))      -- Convert to lowercase letters, result: lua
print(string.reverse(str))    -- Reverse the string, result: auL
print(string.len("abc"))      -- Calculate the length of the string "abc", result: 3
print(string.format("the value is: %d",4))  -- Result: the value is:4
print(string.rep(str,2))       -- Copy the string twice, result: LuaLua

```

## Other operators

Command	NOTE
..	Concatenate two strings
#	Return the length of string or table

```
a = "Hello "
b = "World"
c = {1,2,3}

print(a..b) -- Print the concatenated string: Hello World

print(#b) -- Print the length of string b: 5

print(#c) -- Print the length of table c: 3
```

# General table (array) functions

Lua provides general functions for table (array) operations, such as inserting, sorting, and more. The tables being operated on must be a contiguous one-dimensional array, with indices from Lua's default range of 1 to n (where n is the length of the array).

## table.concat (table, sep, start, end)

Concatenate elements of the table into a string, in the order of their indices, with options to specify a separator and start/end positions.

### Required parameter

- **table:** The table to operate on.

### Optional parameter

- **sep:** Separator. No separator by default.
- **start:** Starting position. 1 by default.
- **end:** Ending position. Default is the length of the array.

### Return

The concatenated string. If the starting position is greater than the ending position, an empty string is returned.

### Example

```
fruits = {"banana", "orange", "apple"}  
  
print(table.concat(fruits)) -- Concatenate using the default method, Output: bananaorangeapple  
  
print(table.concat(fruits, ",")) -- Specify separator to concatenate, Output: banana,orange,apple  
  
print(table.concat(fruits, ",", 2,3)) -- Specify separator and index to concatenate, Output: orange,apple
```

## table.insert (table, pos, value)

Insert a specified element into the table at the given position.

### Required parameter

- **table:** The table to operate on.
- **value:** The element to insert.

### Optional parameter

- **pos:** The position to insert the element. Default is the length of the array plus one, i.e., inserted to the end of the table.

### Example

```

fruits = {"banana", "orange", "apple"}

table.insert(fruits, "mango") -- Insert at the end
print(fruits[4]) -- Output: mango

table.insert(fruits, 2, "grapes") -- Insert at index 2
print(fruits[2], ",", fruits[3]) -- Output: grapes,orange

```

## table.remove(table, pos)

Remove the element at the specified position from the table and return the removed value.

### Required parameter

- **table:** The table to operate on.

### Optional parameter

- **pos:** The position to insert the element. Default is the length of the array, i.e., inserted to the end of the table.

### Return

The removed value.

### Example

```

fruits = {"banana", "orange", "apple"}

print(table.remove(fruits)) -- Remove the last element, Output: apple

print(table.remove(fruits), 2) -- Remove the second element, Output: orange

```

## table.sort(table, comp)

Sort the elements of the table. You can specify a sorting method.

### Required parameter

- **table:** The table to operate on.

### Optional parameter

- **comp:** Sorting method. This parameter must be a function that meets the following requirements:

- It should accept two elements from the table as parameters.
- Return **true** if the first parameter comes before the second; otherwise, return **false**.

By default, Lua's standard operator < is used for sorting.

### Example

```
fruits = {"banana", "orange", "apple", "grapes"}

print("Before sorting")
for k,v in ipairs(fruits) do
    print(v)          -- Output: banana orange apple grapes
end

-- Sort in ascending order
table.sort(fruits)
print("After sorting")
for k,v in ipairs(fruits) do
    print(v)          -- Output: apple banana grapes orange
end

-- Sort in descending order
table.sort(fruits,function(a,b)
    return a > b
end)
print("After sorting")
for k,v in ipairs(fruits) do
    print(v)          -- Output: orange grapes banana apple
end
```

# General description

## Motion mode

The robot supports the following motion modes:

### Joint motion

The robot plans the motion of each joint based on the difference between the current and target joint angles, ensuring that all joints complete their motion simultaneously. Joint motion does not constrain the TCP (Tool Center Point) trajectory, and the path is usually not a straight line.



Joint motion is not restricted by singularities (refer to the corresponding hardware guide for details). If there are no specific trajectory requirements or the target point is near a singularity, joint motion is recommended.

### Linear motion

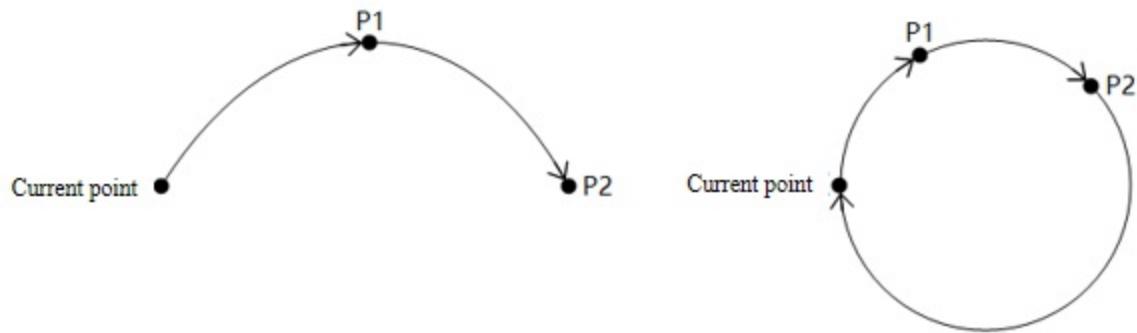
The robot plans the motion trajectory based on the current posture and the target point's posture, ensuring that the TCP moves in a straight line and the end posture changes uniformly during the motion.



When the trajectory passes through a singularity, issuing a linear motion command will result in an error. It is recommended to re-plan the point or use joint motion near the singularity.

### Arc motion

The robot determines an arc or a circle based on three non-collinear points: the current position, P1, and P2. During the motion, the end posture of the robot is interpolated between the current point and the posture at P2, while the posture at P1 is not considered (i.e., when the robot reaches P1 during the motion, its posture may differ from the taught posture).



When the trajectory passes through a singularity, issuing an arc motion command will result in an error. It is recommended to re-plan the point or use joint motion near the singularity.

## Point parameters

All point parameters in this document support three expressions unless otherwise specified.

- Joint variable: describe the target point using the angle of each joint ( $j_1 - j_6$ ) of the robot.

When the joint variable is used as a linear or arc motion parameter, the system will convert it into a posture variable through a positive solution, but the algorithm will ensure that the joint angle of the robot arm when it reaches the target point will be consistent with the set value.

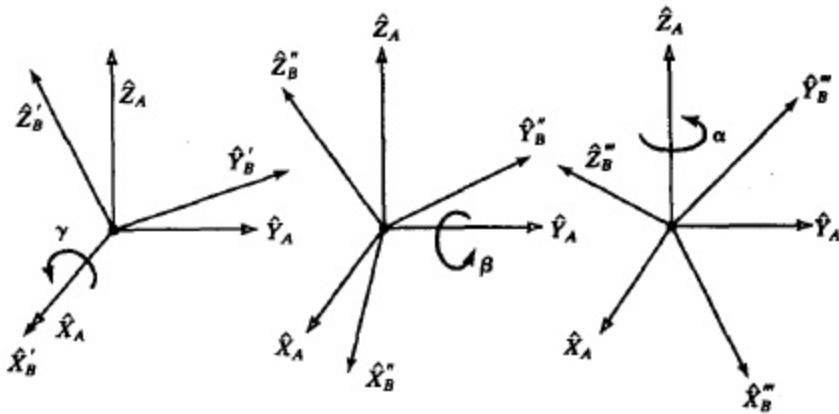
```
{joint = {j1, j2, j3, j4, j5, j6} }
```

- Posture variable: describe the spatial position of the target point in the user coordinate system using Cartesian coordinates ( $x, y, z$ ), and describe the rotation angle of the tool coordinate system relative to the user coordinate system when the TCP (Tool Center Point) reaches a specified point using Euler angles ( $rx, ry, rz$ ).

When the posture variable is used as a point parameter of the joint motion, the system will convert it into a joint variable (the solution closest to the current joint angle of the robot arm) through the inverse solution.

```
{pose = {x, y, z, rx, ry, rz} }
```

The rotation order when calculating the Euler angle of Dobot robot is X->Y->Z, and each axis rotates around a fixed axis (user coordinate system), as shown below ( $rx=\gamma, ry=\beta, rz=\alpha$ ).



Once the rotation order is determined, the rotation matrix (where  $c\alpha$  stands for  $\cos\alpha$ ,  $s\alpha$  stands for  $\sin\alpha$ , and so on)

$$\begin{aligned} {}_B^A R_{XYZ}(\gamma, \beta, \alpha) &= R_z(\alpha) R_y(\beta) R_x(\gamma) \\ &= \begin{bmatrix} c\alpha & -s\alpha & 0 \\ s\alpha & c\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c\beta & 0 & s\beta \\ 0 & 1 & 0 \\ -s\beta & 0 & c\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\gamma & -s\gamma \\ 0 & s\gamma & c\gamma \end{bmatrix} \end{aligned}$$

can be derived as the equation

$${}_B^A R_{XYZ}(\gamma, \beta, \alpha) = \begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma \\ s\alpha c\beta & s\alpha s\beta s\gamma + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma \\ -s\beta & c\beta s\gamma & c\beta c\gamma \end{bmatrix}$$

The posture of the end of robot arm can be calculated through the equation.

- Teaching point: the points obtained through hand-guiding using the software are saved as constants in the following format.

```
-- []
    name: name of teaching point.
    joint: joint coordinates of teaching point.
    tool: tool coordinate system index in hand guiding.
    user: user coordinate system index in hand guiding.
    pose: posture variable value of teaching point
-- ]]
{
    name = "name",
    joint = {j1, j2, j3, j4, j5, j6},
    tool = index,
    user = index,
    pose = {x, y, z, rx, ry, rz}
}
```

## Coordinate system parameters

The "user" and "tool" in the optional parameters are used to specify the user and tool coordinate systems for the target point. Now the coordinate system can be specified only through the index, and the corresponding coordinate system needs to be added in the software first.

Priority of selecting the coordinate system:

1. When a coordinate system is specified in the optional parameters, the specified coordinate system is used. If the point parameter is a taught point, the posture coordinates of the taught point are converted into the values of the specified coordinate system.
2. When no coordinate system is specified in the optional parameters:
  - If the point parameter is a teaching point, the coordinate system index that comes with the teaching point is used.
  - If the target point is a joint variable or a posture variable, the global coordinate system set in the motion parameters is used (See User and Tool commands for details. The default coordinate system is 0 when no command is called).

**i** NOTE

- When a script starts running, the default global coordinate system will be set to 0, regardless of the value set in the Jog panel before running the script.
- If the target point is a joint variable, the coordinate system parameters are invalid.

## Speed parameters

### Relative speed

The optional parameters "a" and "v" are used to specify the acceleration and speed ratio for the robot arm when executing the motion command.

```
Robot actual speed = Maximum speed x Global speed x Command speed  
Robot actual acceleration = Maximum acceleration x Command speed
```

Where, the maximum speed/acceleration is controlled by the **playback parameters** and can be viewed and modified on the "Motion parameters" page of DobotStudio Pro.

The global speed can be adjusted via DobotStudio Pro (speed slider at the top right of the figure above) or the **SpeedFactor** command.

The command rate is carried by the optional parameters of the motion commands. When the acceleration/speed rate is not specified through the optional parameters, the value set in the motion parameters is used by default (see VelJ, AccJ, VelL, AccL commands for details, and the default value is 100 when the command setting is not called).

Example:

```

AccJ(50) -- Set the default acceleration of joint motion to 50%
VelJ(60) -- Set the default speed of joint motion to 60%
AccL(70) -- Set the default acceleration of linear motion to 70%
VelL(80) -- Set the default speed of linear motion to 80%

-- Global speed rate: 20%;

MovJ(P1) -- Move to P1 at the acceleration of (maximum joint acceleration x 50%) and speed of
(maximum joint speed x 20% x 60%) through the joint motion
MovJ(P2,{a = 30, v = 80}) -- Move to P1 at the acceleration of (maximum joint acceleration x 3
0%) and speed of (maximum joint speed x 20% x 80%) through the joint motion

MovL(P1) -- Move to P1 at the acceleration of (maximum Cartesian acceleration x 70%) and speed
of (maximum Cartesian speed x 20% x 80%) in the linear mode
MovL(P1,{a = 40, v = 90}) -- Move to P1 at the acceleration of (maximum Cartesian acceleratio
n x 40%) and speed of (maximum Cartesian speed x 20% x 90%) in the linear mode

```

## Absolute speed

The “speed” in the optional parameter of linear and arc motion commands is used to specify the absolute speed when the robot executes the command.

The absolute speed is not affected by the global speed, but limited by the maximum speed in **Playback settings** (or the maximum speed after reduction if the robot is in reduced mode), i.e. if the target speed set by the speed parameter is greater than the maximum speed in Playback settings, then the maximum speed takes precedence.

Example:

```
MovL(P1,{speed = 1000}) -- Move to P1 in the linear mode at a absolute speed of 1000 mm/s.
```

If the speed set in MovL is 1000 (less than the maximum speed of 2000 in Playback settings), the robot will move at a target speed of 1000 mm/s, which is independent of the global speed at this point. However, if the robot is in reduced mode (with a reduction rate of 10%), the maximum speed becomes 200 mm/s, which is lower than 1000 mm/s, so the robot will move at a target speed of 200 mm/s.

The “speed” and “v” cannot be set at the same time. If both exist, “speed” takes precedence.

## Continuous path parameters

When the robot arm moves through multiple points continuously, it can pass through the intermediate point through a smooth transition so the robot arm will not turn too bluntly. Smooth transitions cannot be applied if the user-specified path points are based on different tool coordinate systems.

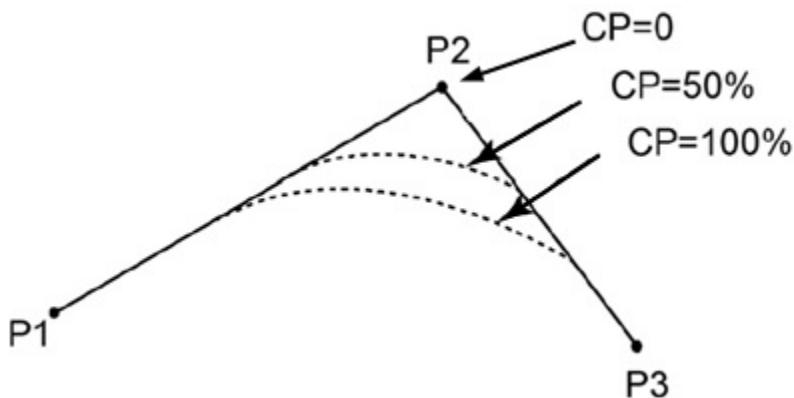
The “cp” or “r” in the optional parameters are used to specify the continuous path ratio (cp) or continuous path radius (r) between the current and the next motion commands. **If r is set, the cp parameter will be ignored.**

### **NOTE**

Joint motion related commands do not support setting the continuous path radius ( $r$ ). See the optional parameters of each command for details.

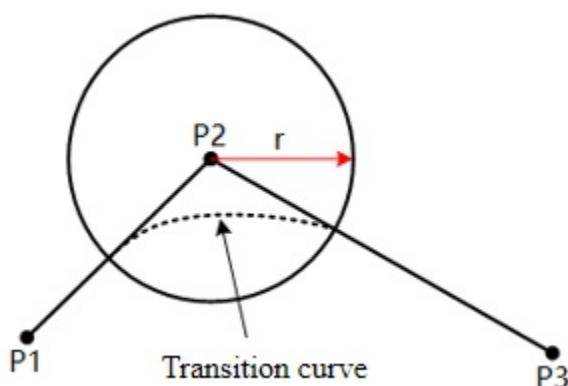
### **CP**

When setting the CP, the system automatically calculates the curvature of the transition curve. The larger the CP value, the smoother the curve, as shown in the figure below. The CP transition curve is affected by speed and acceleration. Even if the path points and CP values are the same, different speeds/accelerations will result in different curvatures.

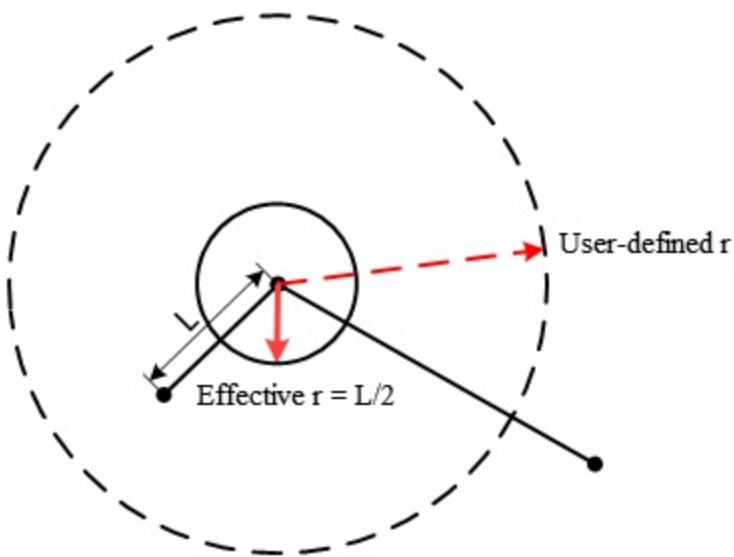


### **R**

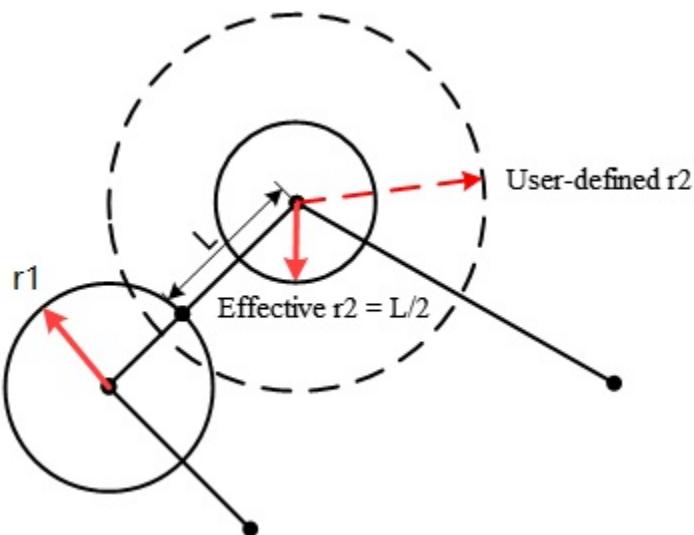
When setting the R, the system calculates the transition curve using the transition point as the center and the specified radius. The R transition curve is not affected by speed or acceleration, but is determined by the path points and the specified radius.



If the continuous path radius is set too large (more than the distance between the start/end point and the intermediate point), the system will automatically calculate the transition curve using half of the shorter distance between the start/end point and the transition point as the continuous path radius.



If two consecutive transition radii ( $r_1$  and  $r_2$  in the figure below) overlap, the system will use the point where the first transition ends as the start point of the second motion, and the second transition radius will be recalculated according to the rules for large transition radii.



### Default values

When the continuous path ratio and radius are not specified in the optional parameters, the continuous path ratio set in the motion parameter is used by default (See CP command for details. The default value is 0 when no command is called).

### NOTE

As CP causes the robot to bypass intermediate points, if CP is set, any IO signal outputs or function settings (e.g., enabling/disabling SafeSkin) between two motion commands will be executed during the transition.

If you want the robot to execute commands precisely when it reaches the intermediate point, set the CP parameters of the previous command to 0.

Due to the different implementations of CP and R, inserting other commands that do not affect motion (such as conditional checks, I/O, or functional settings) between two motion commands requiring smooth transitions will result in differing handling for CP and R.

- If **CP** mode is used, most commands will not affect the transition unless the processing time of the command is too long (e.g., the Wait command).

In the example below, inserting an if statement between two motion commands will not affect the smooth transition in CP mode.

```
-- It can transition normally. The robot will check DI1 when it is about to reach P1. If it is ON, it will transition to the next motion command with a continuous path ratio of 50%.
MovL(P1,{cp=50})
if (DI(1)==ON)
then
    MovL(P2)
end
```

- If **R** mode is used, **only the following whitelisted commands will not affect the smooth transition.** Inserting any other commands will invalidate the smooth transition in R mode.

```
RelPointTool, RelPointUser, DOGroup, DO, AO, ToolDO,
SetUser, SetTool, User, Tool, CP, AccJ, AccL, VelL, VelJ,
SetPayload, SetCollisionLevel, SetBackDistance, EnableSafeSkin, SetSafeSkin
```

In the example below, inserting an if statement between two motion commands will cause the smooth transition settings in R mode to be invalid.

```
-- Invalid CP parameters. The robot will check DI1 after reaching P1.
MovL(P1,{r=5})
if (DI(1)==ON)
then
    MovL(P2)
end
```

## Stop condition

The optional parameter “stopcond” is used to specify the motion stop condition, which is an expression in string form. If the stop condition is specified, the robot will check the stop condition in real time during the execution of the motion command, and will end the current motion and directly execute the next command when the stop condition is satisfied.

The stop condition supports any expression that conforms to Lua syntax. When the expression is “true”, the condition is considered satisfied. Typical applications are as follows:

```
-- Stop running when DI1 is ON
MovJ(P1,{stopcond="DI(1) == ON"})
```

```
-- Stop running when DI1 and DO1 is ON
MovJ(P1,{stopcond="DI(1) == ON and GetDO(1) == ON"})
-- Stop running when the value stored in holding register address 100 is less than 10
MovJ(P1,{stopcond="GetHoldRegs(id, 100, 1)[1] < 10"})
-- Stop running when var1 is not equal to 5
MovJ(P1,{stopcond="var1 ~= 5"})
```

**i NOTE**

- Specifying a stop condition will invalidate the R parameter.
- After a stop condition is set, the CP parameter can still be applied, but the transition segment (the curved part) will not be included in the stop condition range.
- If the previous command before the stop condition includes CP, the stop condition will only be triggered after exiting the transition segment.

## IO signal

The ON and OFF variables are predefined inside the system to indicate whether there is a signal.

- ON = 1: there is a signal
- OFF = 0: there is no signal

ON|OFF in the parameter means that the parameter value is ON or OFF. You can also use 1 or 0 as input.

# Motion

## Command list

The motion commands are used to control the movement of the robot. Please read [General description](#) before using the commands.

Command	Function
MovJ	Joint motion
MovL	Linear motion
Arc	Arc motion
Circle	Circle motion
MovJIO	Move in joint mode and output DO
MovLIO	Move in linear mode and output DO
GetPathStartPose	Get start point of trajectory
StartPath	Play back recorded trajectory
PositiveKin	Forward kinematics (joint angles to posture)
InverseKin	Inverse kinematics (posture to joint angles)

## MovJ

### Command:

```
MovJ(point, {user = 1, tool = 0, a = 20, v = 50, cp = 100, stopcond = "expression"})
```

### Description:

Move from the current position to the target position through joint motion.

### Required parameter:

**point:** Target point.

### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].

- **cp:** The ratio for smooth transition. Range: [0,100].
- **stopcond:** The expression of stop condition. When the conditions are met, the robot will end the current motion and then execute the next command.

See [General description](#) for details.

#### Example:

```
-- The robot arm moves to P1 through joint motion with the default setting.
MovJ(P1)
```

```
-- The robot arm moves to the specified joint angle through joint motion with the default setting.
MovJ({joint={0,0,90,0,90,0}})
```

```
-- The robot arm moves to the specified posture through joint motion, which corresponds to User coordinate system 1 and Tool coordinate system 1, with an acceleration and velocity of 50% and a continuous path rate of 50%.
MovJ({pose={300,200,300,180,0,0} },{user=1,tool=1,a=50,v=50,cp=50})
```

```
-- Define the point and then call it in a motion command. The running effect is the same as last command.
customPoint={pose={300,200,300,180,0,0} }
MovJ(customPoint,{user=1,tool=1,a=50,v=50,cp=50})
```

```
-- The robot arm moves to P1 through joint motion, and end the current movement when DI1 is ON.
```

```
MovJ(P1,{stopcond="DI(1) == ON"})
```

## MovL

#### Command:

```
MovL(point, {user = 1, tool = 0, a = 20, v = 50, speed = 500, cp = 100, r = 5, stopcond = "expression"})
```

#### Description:

Move from the current position to the target position in a linear mode.

#### Required parameter:

**point:** Target point.

## Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the **cp** parameter will be ignored.

- **stopcond:** The expression of stop condition. When the conditions are met, the robot will end the current motion and then execute the next command.

See [General description](#) for details.

## Example:

```
-- The robot arm moves to P1 in a linear mode with the default setting.  
MovL(P1)
```

```
-- The robot arm moves to P1 in a linear mode at an absolute speed of 500m/s.  
MovL(P1,{speed=500})
```

```
-- The robot arm moves to the specified joint angle in a linear mode with the default setting.  
MovL({joint={0,0,90,0,90,0} })
```

```
-- The robot arm moves in a linear mode to the specified posture, which corresponds to User coordinate system 1 and Tool coordinate system 1, with an acceleration and speed of 50% and a continuous path radius of 5 mm.  
MovL({pose={300,200,300,180,0,0} },{user=1,tool=1,a=50,v=50,r=5})
```

```
-- Define the point and then call it in a motion command. The running effect is the same as last command.  
customPoint={pose={300,200,300,180,0,0} }  
MovL(customPoint,{user=1,tool=1,a=50,v=50,r=5})
```

```
-- "speed" takes effect when carried together with "v", and the controller prints the corresponding alarm log at the same time.
-- "r" takes effect when carried together with "cp", and the controller prints the corresponding alarm log at the same time.
MovL(P1,{v=50,speed=500,cp=60,r=5}) -- Only "speed" and "r" of optional parameters take effect during execution.
```

```
-- The robot arm moves to P1 in linear mode, and end the current movement when DI1 is ON.
MovL(P1,{stopcond="DI(1) == ON"})
```

## Arc

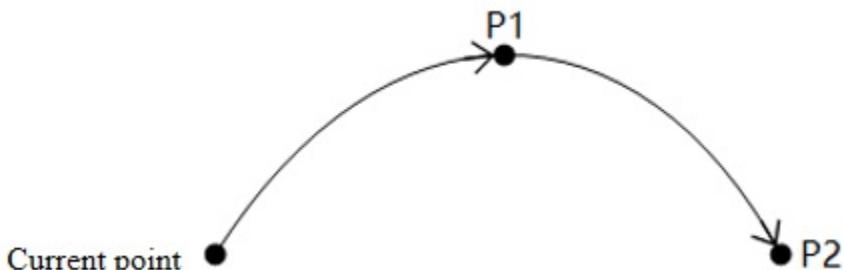
### Command:

```
Arc(P1, P2, {user = 1, tool = 0, a = 20, v = 50, speed = 500, cp = 100, r = 5, stopcond = "expression"})
```

### Description:

Move from the current position to the target position in an arc interpolated mode.

As the arc needs to be determined through the current position, P1 and P2, the current position should not be in a straight line determined by P1 and P2.



During the motion, the end posture of the robot is interpolated between the current point and the posture at P2, while the posture at P1 is not considered (i.e., when the robot reaches P1 during the motion, its posture may differ from the taught posture).

### Required parameter:

- **P1:** Intermediate point of the arc.
- **P2:** Target point.

### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.

- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the **cp** parameter will be ignored.

- **stopcond:** The expression of stop condition. When the conditions are met, the robot will end the current motion and then execute the next command.

See [General description](#) for details.

#### Example:

```
-- The robot arm moves to P1, and then moves to P3 via P2 in an arc interpolated mode.
MovJ(P1)
Arc(P2,P3)
```

```
-- The robot arm moves to P1, and then moves to P3 via {300,200,300,180,0,0}, which corresponds to User coordinate system 1 and Tool coordinate system 1, with an acceleration and speed of 50%.
MovJ(P1)
Arc({pose={300,200,300,180,0,0} },P3,{user=1,tool=1,a=50,v=50})
```

```
-- The robot arm moves to P1, and then moves to P3 via P2 in an arc interpolated mode. The robot arm will end the current movement when DI1 is ON.
MovJ(P1)
Arc(P2,P3,{stopcond="DI(1) == ON"})
```

## Circle

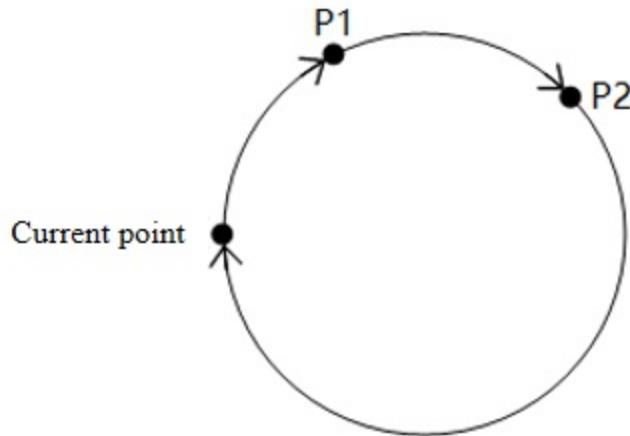
#### Command:

```
Circle(P1, P2, Count, {user = 1, tool = 0, a = 20, v = 50, speed = 500, cp = 100, r = 5, stopcond = "expression"})
```

#### Description:

Move from the current position in a circle interpolated mode, and return to the current position after moving specified circles.

As the circle needs to be determined through the current position, P1 and P2, the current position should not be in a straight line determined by P1 and P2, and the circle determined by the three points cannot exceed the motion range of the robot.



During the motion, the end posture of the robot is interpolated between the current point and the posture at P2, while the posture at P1 is not considered (i.e., when the robot reaches P1 during the motion, its posture may differ from the taught posture).

#### Required parameter:

- **P1:** Point 1 in circle motion.
- **P2:** Point 2 in circle motion.
- **Count:** Number of circles to perform, range: [1 – 999].

#### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the v parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the cp parameter will be ignored.

- **stopcond:** The expression of stop condition. When the conditions are met, the robot will end the current motion and then execute the next command.

See [General description](#) for details.

#### Example:

```
-- The robot arm moves to P1, and then moves a full circle determined by P1, P2 and P3.
MovJ(P1)
Circle(P2,P3,1)
```

```
-- The robot arm moves to P1 and then moves 10 times along the circle determined by P1, {300,2
00,300,180,0,0}, and P3, with both the user and tool coordinate systems at 1 and the acceleration and velocity of 50%.
MovJ(P1)
Circle({pose={300,200,300,180,0,0} },P3,10,{user=1,tool=1,a=50,v=50})
```

```
-- The robot arm moves to P1, and then moves a full circle determined by P1, P2 and P3. The robot arm will end the current movement when DI1 is ON.
MovJ(P1)
Circle(P2,P3,1,{stopcond="DI(1) == ON"})
```

## MovJIO

#### Command:

```
MovJIO(P,{ {Mode,Distance,Index,Status},{Mode,Distance,Index,Status}...}, {user = 1, tool = 0,
a = 20, v = 50, cp = 100})
```

#### Description:

Move from the current position to the target position through joint motion, while simultaneously setting the status of digital output port.

#### Required parameter:

- **P:** The target point.
- **Digital output parameters:** Set the specified DO to be triggered when the robot reaches a specified distance or percentage. Multiple groups can be set, each containing the following parameters:
  - **Mode:** Trigger mode. **0:** percentage trigger. **1:** distance trigger. The system will synthesize the joint angles into an angular vector and calculate the angle difference between the end point and the start point as the total distance of the motion.
  - **Distance:** Specified percentage/angle. It is recommended to use the percentage mode for a more intuitive effect, as angle calculations rely on the synthesized angle vector.
    - If Distance is 0, it indicates that the starting point triggers the action.

- If Distance is positive, it refers to the percentage/angle from the start point.
- If Distance is negative, it refers to the percentage/angle from the target point.
- If Mode is 0, Distance refers to the percentage of total angle. Range: (0,100].
- If Mode is 1, Distance refers to the angle value. Unit: °.
- **Index:** DO index.
- **Status:** DO status. 0/OFF: no signal; 1/ON: an active signal.

#### Optional parameter:

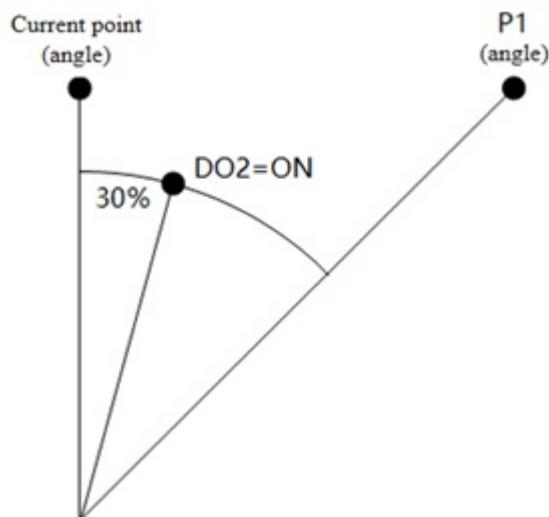
- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **cp:** The ratio for smooth transition. Range: [0,100].

CP will alter the robot's motion trajectory, affecting the timing of DO outputs; please use it with caution.

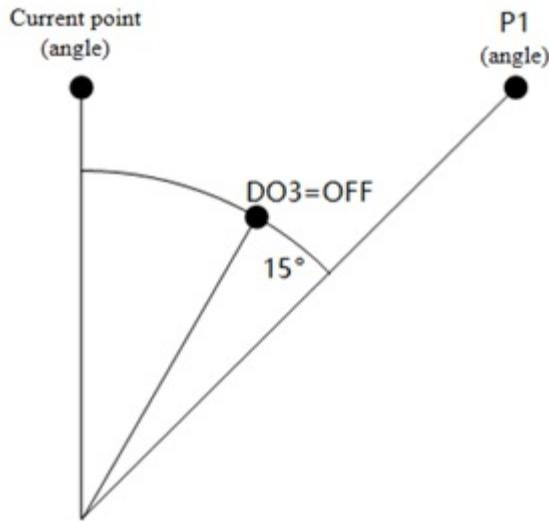
See [General description](#) for details.

#### Example:

```
-- The robot moves to P1 through joint motion with the default settings. When it reaches 30% o
f the distance from the start point, DO2 is set to ON.
MovJIO(P1, { {0, 30, 2, 1} })
```



```
-- The robot moves to P1 through joint motion with the default settings. When it is 15° away f
rom the target point, DO3 is set to OFF.
MovJIO(P1, { {1, -15, 3, 0} })
```



-- The robot moves to P1 through joint motion with the default settings. When it reaches 30% of the distance from the start point, D02 is set to ON. When it is 20% away from the target point, D02 is set to OFF.

```
MovJIO(P1, { {0, 30, 2, 1}, {0, -20, 2, 0} })
```

## MovLIO

### Command:

```
MovLIO(P,{ {Mode,Distance,Index,Status},{Mode,Distance,Index,Status}...},{user = 1, tool = 0, a = 20, v = 50, speed = 500, cp = 100, r = 5})
```

### Description:

Move from the current position to the target position in a linear mode, while simultaneously setting the status of digital output port.

### Required parameter:

- **P:** The target point.
- **Digital output parameters:** Set the specified DO to be triggered when the robot reaches a specified distance or percentage. Multiple groups can be set, each containing the following parameters:
  - **Mode:** Trigger mode. **0:** percentage trigger. **1:** distance trigger.
  - **Distance:** Specified percentage/distance.
    - If Distance is 0, it indicates that the starting point triggers the action.
    - If Distance is positive, it refers to the percentage/distance from the start point.
    - If Distance is negative, it refers to the percentage/distance from the target point.
    - If Mode is 0, Distance refers to the percentage of total distance. Range: (0,100].
    - If Mode is 1, Distance refers to the distance value. Unit: mm.
  - **Index:** DO index.

- **Status:** DO status. 0/OFF: no signal; 1/ON: an active signal.

#### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

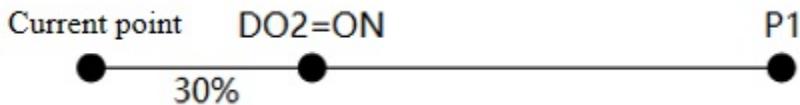
If this parameter is set, the **cp** parameter will be ignored.

CP will alter the robot's motion trajectory, affecting the timing of DO outputs; please use it with caution.

See [General description](#) for details.

#### Example:

```
-- The robot moves to P1 in a linear mode with the default settings. When it reaches 30% of the distance from the start point, DO2 is set to ON.
MovLIO(P1, { {0, 30, 2, 1} })
```



```
-- The robot moves to P1 in a linear mode with the default settings. When it is 15 mm away from the target point, DO3 is set to OFF.
MovLIO(P1, { {1, -15, 3, 0} })
```



```
-- The robot moves to P1 in a linear mode with the default settings. When it reaches 30% of the distance from the start point, DO2 is set to ON. When it is 20% away from the target point, DO2 is set to OFF.
MovLIO(P1, { {0, 30, 2, 1}, {0, -20, 2, 0} })
```



## GetPathStartPose

**Command:**

```
GetPathStartPose(string)
```

**Description:**

This command is typically used in conjunction with the StartPath command to move to the starting point of a trajectory playback. The type of point data (teaching point/joint/posture) may be different according to the trajectory files.

**Required parameter:**

**string:** Trajectory file name (including suffix).

**Example:**

```
-- Get the start point of the "track.csv" file and print it.
local StartPoint = GetPathStartPose("track.csv")
print(StartPoint)
-- StartPoint = {joint={j1,j2,j3,j4,j5,j6}, "name" = "", user = userIndex, tool = toolIndex, pose
= {x,y,z,a,b,c}}
-- StartPoint = {joint={j1,j2,j3,j4,j5,j6}}
-- StartPoint = {pose = {x,y,z,a,b,c}}
```

## StartPath

**Command:**

```
StartPath(string, {multi = 1, isConst = 0, sample = 50, freq = 0.2, user = 0, tool = 0})
```

**Description:**

Play back the recorded trajectory in the specified trajectory file. Before calling this command, you need to manually move the robot to the start point of the trajectory.

**Required parameter:**

**string:** Trajectory file name (including suffix).

### Optional parameter:

- **multi:** Speed multiplier for the playback, valid only when isConst=0. Range: [0.1, 2], 1 by default.
- **isConst:** Whether to play back at a constant speed. 0 by default.
  - 1 means constant speed playback, where the robot will play back the trajectory at a constant speed.
  - 0 means playback at the original recorded speed, with the option to scale the speed proportionally using the "multi" parameter. In this case, the robot's speed is not affected by the global speed setting.
- **sample:** The sampling interval of the trajectory points, i.e. the sampling time differences between two adjacent points when generating the trajectory file. Range: [8, 1000], unit: ms, 50 by default (the sampling interval when the controller records the trajectory file).
- **freq:** Filter coefficient. The smaller the value, the smoother the trajectory curve during playback, but the greater the distortion relative to the original trajectory. Set an appropriate filter coefficient based on the smoothness of the original trajectory. Range: (0,1], 1 refers to turning off filtering, 0.2 by default.
- **user:** Specify the user coordinate system index for the trajectory points. If not specified, the user coordinate system index recorded in the trajectory file is used.
- **tool:** Specify the tool coordinate system index for the trajectory points. If not specified, the tool coordinate system index recorded in the trajectory file is used.

### Example:

```
track.csv is the trajectory file recorded by the user through DobotStudio Pro.
-- After moving to the start point of the trajectory file in MovJ mode, the robot plays back the recorded trajectory at 2x the original speed.
local StartPoint = GetPathStartPose("track.csv")
MovJ(StartPoint)
StartPath("track.csv", {multi = 2, isConst = 0})
```

```
track.csv is the trajectory file recorded by the user through DobotStudio Pro.
-- After moving to the start point of the trajectory file in MovJ mode, the robot plays back the recorded trajectory at a constant speed.
local StartPoint = GetPathStartPose("track.csv")
MovJ(StartPoint)
StartPath("track.csv", {isConst = 1})
```

```
-- customtrack.csv is a trajectory file generated by the user, with a sampling interval of 20ms.
-- After moving to the start point of the trajectory file in MovJ mode, the robot plays back the recorded trajectory at a constant speed, with a filtering coefficient of 1 (fully restoring the recorded trajectory), using both user and tool coordinate systems set to 0.
local StartPoint = GetPathStartPose("customtrack.csv")
MovJ(StartPoint)
StartPath("customtrack.csv", {isConst = 1, sample = 20, freq = 1, user = 0, tool = 0})
```

## PositiveKin

### Command

```
PositiveKin(joint, {user = 1, tool = 0})
```

### Description

Perform forward kinematics. Calculate the coordinates of the end of the robot in the specified Cartesian coordinate system, based on the given angle of each joint.

### Required parameter

**joint:** Joint variables, format: `{joint = {j1, j2, j3, j4, j5, j6} }`.

### Optional parameter

- **user:** User coordinate system index. The global user coordinate system will be used when it is not specified.
- **tool:** Tool coordinate system index. The global tool coordinate system will be used when it is not specified.

### Return

The calculated posture variables through forward kinematics, format: `{pose = {x, y, z, rx, ry, rz} }`.

### Example

```
-- Calculate the coordinates of the end of the robot in the global user/tool coordinate system  
, based on the joint coordinates {0,0,-90,0,90,0}.  
PositiveKin({joint={0,0,-90,0,90,0} })
```

```
-- Calculate the coordinates of the end of the robot in the User coordinate system 1 and Tool  
coordinate system 1, based on the joint coordinates {0,0,-90,0,90,0}.  
PositiveKin({joint={0,0,-90,0,90,0} },{user=1,tool=1})
```

## InverseKin

### Command

```
InverseKin(pose, {useJointNear = true, jointNear = joint, user = 1, tool = 0})
```

### Description

Perform inverse kinematics. Calculate the joint angles of the robot, based on the given coordinates in the specified Cartesian coordinate system.

Since Cartesian coordinates only define the spatial coordinates and tilt angle of the TCP, the robot can reach the same pose with different configurations. This means that one pose can have multiple joint configurations. To get a unique solution, the system requires a specified joint coordinate, and the solution closest to this joint coordinate is selected as the result.

This command allows users to check if a Cartesian point is reachable or not.

### Required parameter

**pose:** Posture variables, format: `{pose = {x, y, z, rx, ry, rz} }`.

### Optional parameter

- **useJointNear:** Boolean value specify whether jointNear is used.
  - **true:** The algorithm selects the joint angles according to JointNear data.
  - **false or null:** JointNear data is ineffective. The algorithm selects the joint angles according to the current angle.
  - If only this parameter is specified without jointNear, it will be ineffective.
- **jointNear:** Joint variables for selecting joint angles, format: `{joint = {j1, j2, j3, j4, j5, j6} }`.
- **user:** User coordinate system index. The global user coordinate system will be used when it is not specified.
- **tool:** Tool coordinate system index. The global tool coordinate system will be used when it is not specified.

### Return

- Error code: 0 indicates the inverse kinematics is successful, -1 indicates failure (no solution).
- The calculated joint variables through inverse kinematics, format: `{joint = {j1, j2, j3, j4, j5, j6} }`. If the inverse kinematics fails, all values for J1 to J6 will be 0.

### Example

```
-- The Cartesian coordinates of the end of the robot in the global user/tool coordinate system  
are {300, 200, 300, 180, 0, 0}. Joint coordinates will be calculated, and the nearest solution  
will be selected.
```

```
local errId, jointPoint = InverseKin({ pose = {300, 200, 300, 180, 0, 0} })
```

```
-- The Cartesian coordinates of the end of the robot in User coordinate system 1 and Tool coordinate  
system 1 are {300, 200, 300, 180, 0, 0}. Joint coordinates will be calculated, and the nearest solution  
will be selected.
```

```
local errId, jointPoint = InverseKin({ pose = {300, 200, 300, 180, 0, 0} }, {user=1, tool=1})
```

```
-- The Cartesian coordinates of the end of the robot in the global user/tool coordinate system  
are {300, 200, 300, 180, 0, 0}. Joint coordinates will be calculated, and the solution closest  
to the joint angles {90, 30, -90, 180, 30, 0} will be selected.
```

```
local errId, jointPoint = InverseKin({ pose = {300, 200, 300, 180, 0, 0} }, {useJointNear = true,  
jointNear = { joint = {90, 30, -90, 180, 30, 0} } })
```

# Relative motion

## Command list

The relative motion commands are used to control the robot for offset movements. Please read [General description](#) before using the commands.

Command	Function
RelPointUser	Offset the point along the user coordinate system
RelPointTool	Offset the point along the tool coordinate system
RelMovJTool	Perform relative joint motion along the tool coordinate system
RelMovLTool	Perform relative linear motion along the tool coordinate system
RelMovJUser	Perform relative joint motion along the user coordinate system
RelMovLUser	Perform relative linear motion along the user coordinate system
RelJointMovJ	Perform joint motion to the specified offset angle
RelJoint	Offset the point by a specified joint angle

## RelPointUser

### Command:

```
RelPointUser(P, {OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz})
```

### Description:

Offset the specified point along the given user coordinate system and return the offset point.

### Required parameter:

- **P:** The point to be offset.
- **{OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz}:** The offset values in the specified user coordinate system. **x, y, z:** Spatial offset values in millimeters (mm). **rx, ry, rz:** Angular offset values in degrees (°).
  - If the specified point is a taught point, the offset is based on the user coordinate system of the taught point.
  - If the specified point is a joint variable or a posture variable, the offset is based on the [global user coordinate system](Motion Params.html#user).

### Return:

- If the specified point is a taught point, return the point constant after offset.
- If the specified point is a joint variable or a posture variable, return the posture variable after offset:  
 $\{\text{pose} = \{\text{x}, \text{y}, \text{z}, \text{rx}, \text{ry}, \text{rz}\}\}$ .

**Example:**

```
-- Offset P1 by a certain distance in the specified user coordinate system, then move to the offset point.
local Offset={OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz}
local p = RelPointUser(P1, Offset)
MovL(p)
```

## RelPointTool

**Command:**

```
RelPointTool(P, {OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz})
```

**Description:**

Offset the specified point along the given tool coordinate system and return the offset point.

**Required parameter:**

- **P:** The point to be offset.
- **{OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz}:** Specify the offset in the tool coordinate system. **x, y, z:** Spatial offset values in millimeters (mm). **rx, ry, rz:** Angular offset values in degrees (°).
  - If the specified point is a taught point, the offset is based on the tool coordinate system of the taught point.
  - If the specified point is a joint variable or a posture variable, the offset is based on the [global tool coordinate system](Motion Params.html#tool).

**Return:**

- If the specified point is a taught point, return the point constant after offset.
- If the specified point is a joint variable or a posture variable, return the posture variable after offset:  
 $\{\text{pose} = \{\text{x}, \text{y}, \text{z}, \text{rx}, \text{ry}, \text{rz}\}\}$ .

**Example:**

```
-- Offset P1 by a certain distance in the specified tool coordinate system, then move to the offset point.
local Offset={OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz}
local p = RelPointTool(P1, Offset)
MovL(p)
```

## RelMovJTool

### **Command:**

```
RelMovJTool({x, y, z, rx, ry, rz}, {user = 1, tool = 0, a = 20, v = 50, cp = 100, stopcond = "expression"})
```

### **Description:**

Perform relative joint motion from the current position along the specified tool coordinate system. The trajectory is not a straight line, and all joints will move simultaneously.

### **Required parameter:**

{**x, y, z, rx, ry, rz**} The offset values relative to the current position in the specified tool coordinate system.  
**x, y, z**: Spatial offset values in millimeters (mm). **rx, ry, rz**: Angular offset values in degrees (°).

### **Optional parameter:**

- **user**: The user coordinate system for the target point.
- **tool**: The tool coordinate system for the target point.
- **a**: The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v**: The velocity ratio for the robot when executing this command. Range: (0,100].
- **cp**: The ratio for smooth transition. Range: [0,100].
- **stopcond**: The expression of stop condition. When the conditions are met, the robot will end the current motion and then execute the next command.

See [General description](#) for details.

### **Example:**

```
-- The robot moves with default settings using joint motion along the global tool coordinate system to the specified offset point.  
RelMovJTool({10, 10, 10, 0, 0, 0})
```

For more examples related to optional parameters, please refer to MovJ.

## **RelMovLTool**

### **Command:**

```
RelMovLTool({x, y, z, rx, ry, rz}, {user = 1, tool = 0, a = 20, v = 50, speed = 500, cp = 100, r = 5, stopcond = "expression"})
```

### **Description:**

Perform relative linear motion from the current position along the specified tool coordinate system.

### **Required parameter:**

**{x, y, z, rx, ry, rz}** The offset values relative to the current position in the specified tool coordinate system.  
**x, y, z**: Spatial offset values in millimeters (mm). **rx, ry, rz**: Angular offset values in degrees (°).

#### Optional parameter:

- **user**: The user coordinate system for the target point.
- **tool**: The tool coordinate system for the target point.
- **a**: The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v**: The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed**: The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp**: The ratio for smooth transition. Range: [0,100].
- **r**: The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the **cp** parameter will be ignored.

- **stopcond**: The expression of stop condition. When the conditions are met, the robot will end the current motion and then execute the next command.

See [General description](#) for details.

#### Example:

```
-- The robot moves with default settings using linear motion along the global tool coordinate
system to the specified offset point.
RelMovLTool({10, 10, 10, 0, 0, 0})
```

For more examples related to optional parameters, please refer to [MovL](#).

## RelMovJUser

#### Command:

```
RelMovJUser({x, y, z, rx, ry, rz}, {user = 1, tool = 0, a = 20, v = 50, cp = 100, stopcond = "expression"})
```

#### Description:

Perform relative joint motion from the current position along the specified user coordinate system. The trajectory is not a straight line, and all joints will move simultaneously.

#### Required parameter:

**{x, y, z, rx, ry, rz}** The offset values relative to the current position in the specified user coordinate system.  
**x, y, z:** Spatial offset values in millimeters (mm). **rx, ry, rz:** Angular offset values in degrees (°).

#### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **cp:** The ratio for smooth transition. Range: [0,100].
- **stopcond:** The expression of stop condition. When the conditions are met, the robot will end the current motion and then execute the next command.

See [General description](#) for details.

#### Example:

```
-- The robot moves with default settings using joint motion along the global user coordinate system to the specified offset point.  
RelMovJUser({10, 10, 10, 0, 0, 0})
```

For more examples related to optional parameters, please refer to MovJ.

## RelMovLUser

#### Command:

```
RelMovLUser({x, y, z, rx, ry, rz}, {user = 1, tool = 0, a = 20, v = 50, speed = 500, cp = 100, r = 5, stopcond = "expression"})
```

#### Description:

Perform relative linear motion from the current position along the specified user coordinate system.

#### Required parameter:

**{x, y, z, rx, ry, rz}** The offset values relative to the current position in the specified user coordinate system.  
**x, y, z:** Spatial offset values in millimeters (mm). **rx, ry, rz:** Angular offset values in degrees (°).

#### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].

- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the **cp** parameter will be ignored.

- **stopcond:** The expression of stop condition. When the conditions are met, the robot will end the current motion and then execute the next command.

See [General description](#) for details.

#### **Example:**

```
-- The robot moves with default settings using linear motion along the global user coordinate
system to the specified offset point.
RelMovLUser({10, 10, 10, 0, 0, 0})
```

For more examples related to optional parameters, please refer to MovL.

## **RelJointMovJ**

#### **Command:**

```
RelJointMovJ({Offset1, Offset2, Offset3, Offset4, Offset5, Offset6}, {a = 20, v = 50, cp = 100
})
```

#### **Description:**

Move from the current position to the specified joint offset angles through joint motion.

#### **Required parameter:**

**{Offset1, Offset2, Offset3, Offset4, Offset5, Offset6}** The offset values for J1 to J6 in the joint coordinate system, unit: °.

#### **Optional parameter:**

- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **cp:** The ratio for smooth transition. Range: [0,100].

See [General description](#) for details.

#### **Example:**

```
-- The robot moves to the offset angle through joint motion with the default settings.  
RelJointMovJ({20,20,10,0,10,0})
```

## RelJoint

### Command:

```
RelJoint(P, {Offset1, Offset2, Offset3, Offset4, offset5, offset6})
```

### Description:

Add offset to J1 to J6 of the specified point in the joint coordinate system, and return the joint variable after offset.

### Required parameter:

- **P:** The point to be offset.
- **Offset1 – Offset6:** The offset values for J1 to J6 in the joint coordinate system, unit: °.

### Return:

Joint variables after offset: {joint = {j1, j2, j3, j4, j5, j6}}.

### Example:

```
-- Offset P1 by a certain angle in J1 to J6, then move to the point after offset.  
local Offset = {Offset1, Offset2, Offset3, Offset4, offset5, offset6}  
local p = RelJoint(P1, Offset)  
MovJ(p)
```

# Motion parameters

## Command list

The motion parameters are used to set or obtain relevant motion parameters of the robot. Please read [General description](#) before using the commands.

Command	Function
CP	Set CP ratio
VelJ	Set the velocity ratio for joint motion
AccJ	Set the acceleration ratio for joint motion
VelL	Set the velocity ratio for linear and arc motion
AccL	Set the acceleration ratio for linear and arc motion
SpeedFactor	Set the global speed for the robot
SetPayload	Set the payload
User	Set the global user coordinate system
SetUser	Modify the specified user coordinate system
CalcUser	Calculate the user coordinate system
Tool	Set the global tool coordinate system
SetTool	Modify the specified tool coordinate system
CalcTool	Calculate the tool coordinate system
GetPose	Get the the robot's real-time posture
GetAngle	Get the real-time joint angle of the robot
GetABZ	Get the current position from the encoder
CheckMovJ	Check the feasibility of the joint motion command
CheckMovL	Check the feasibility of the linear or arc motion command
SetSafeWallEnable	Enable or disable the specified safety wall
SetWorkZoneEnable	Enable or disable the specified safety zone
SetCollisionLevel	Set collision detection level
SetBackDistance	Set collision backoff distance

## CP

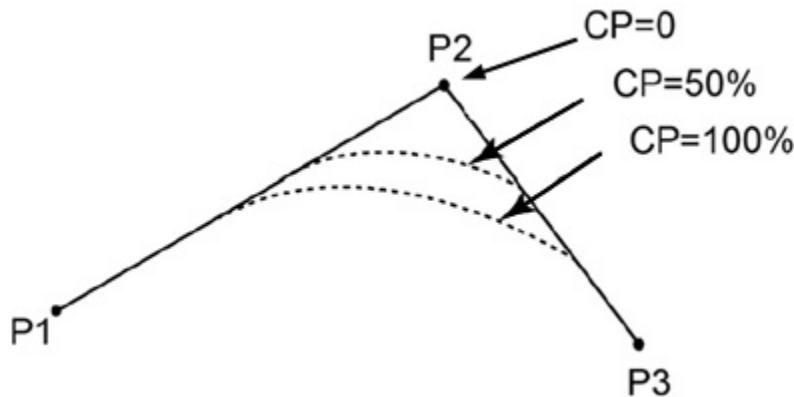
### Command:

```
CP(R)
```

### Description:

Set the CP (continuous path) ratio, which determines whether the robot transitions between multiple points with right angles or curved paths when moving continuously.

The CP ratio set in this command is only valid in the current project, and it is 0 by default when not set.



### Required parameter:

R: CP ratio. Range: [0, 100].

### Example:

```
The robot is currently at P1, smoothly transitioning to P2 with a 50% CP ratio, and ultimately reaching P3.  
CP(50)  
MovL(P2)  
MovL(P3)
```

## VelJ

### Command:

```
VelJ(R)
```

### Description:

Set the velocity ratio for joint motion (MovJ/MovJIO/RelMovJTool/RelMovJUser/RelJointMovJ).

The velocity ratio set in this command is only valid in the current project, and it is 100 by default when not set.

### Required parameter:

**R:** Velocity ratio. Range: [1,100].

**Example:**

```
-- The robot moves to P1 with 20% velocity ratio.  
VelJ(20)  
MovJ(P1)
```

## AccJ

**Command:**

```
AccJ(R)
```

**Description:**

Set the acceleration ratio for joint motion (MovJ/MovJIO/RelMovJTool/RelMovJUser/RelJointMovJ).

The acceleration ratio set in this command is only valid in the current project, and it is 100 by default when not set.

**Required parameter:**

**R:** Acceleration ratio. Range: [1,100].

**Example:**

```
-- The robot moves to P1 with 50% acceleration ratio.  
AccJ(50)  
MovJ(P1)
```

## VelL

**Command:**

```
VelL(R)
```

**Description:**

Set the velocity ratio for linear and arc motion (MovL/Arc/Circle/MovLIO/RelMovLTool/RelMovLUser).

The velocity ratio set in this command is only valid in the current project, and it is 100 by default when not set.

**Required parameter:**

**R:** Velocity ratio. Range: [1,100].

### **Example:**

```
-- The robot moves to P1 with 20% velocity ratio.  
VelL(20)  
MovL(P1)
```

## **AccL**

### **Command:**

```
AccL(R)
```

### **Description:**

Set the acceleration ratio for linear and arc motion  
(MovL/Arc/Circle/MovLIO/RelMovLTool/RelMovLUser).

The acceleration ratio set in this command is only valid in the current project, and it is 100 by default when not set.

### **Required parameter:**

**R:** Acceleration ratio. Range: [1,100].

### **Example:**

```
-- The robot moves to P1 with 50% acceleration ratio.  
AccL(50)  
MovL(P1)
```

## **SpeedFactor**

### **Command:**

```
SpeedFactor(ratio)
```

### **Description:**

Set the global speed for the robot. See [General description](#) for details.

This global speed setting is only effective during the current project. If not set, the system will use the value previously set in the control software (if the script is run via software) or TCP commands (if the script is run through TCP).

### **Required parameter:**

**R:** The global speed ratio. Range: [1,100].

**Example:**

```
-- Set the global speed to 50%.  
SpeedFactor(50)
```

## SetPayload

**Command:**

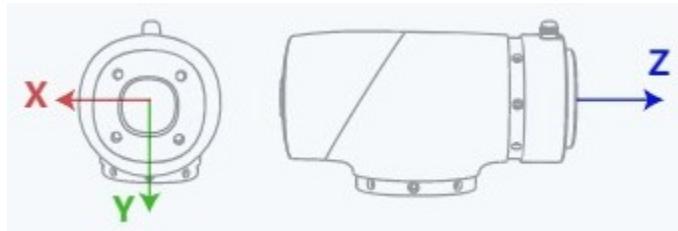
```
SetPayload(payload, {x, y, z})  
SetPayload(name)
```

**Description:**

Set the weight and offset coordinates of the end effector's payload. You can either specify these parameter directly or use preset parameter group. The parameters set by this command are only effective during the current project and will overwrite the previous settings. Once the project stops, the settings will revert to the original values.

**Required parameter 1:**

- **payload:** The weight of the end effector's payload. Unit: kg.
- **{x, y, z}:** The offset coordinates of the payload, see the figure below for axis directions. Unit: mm.

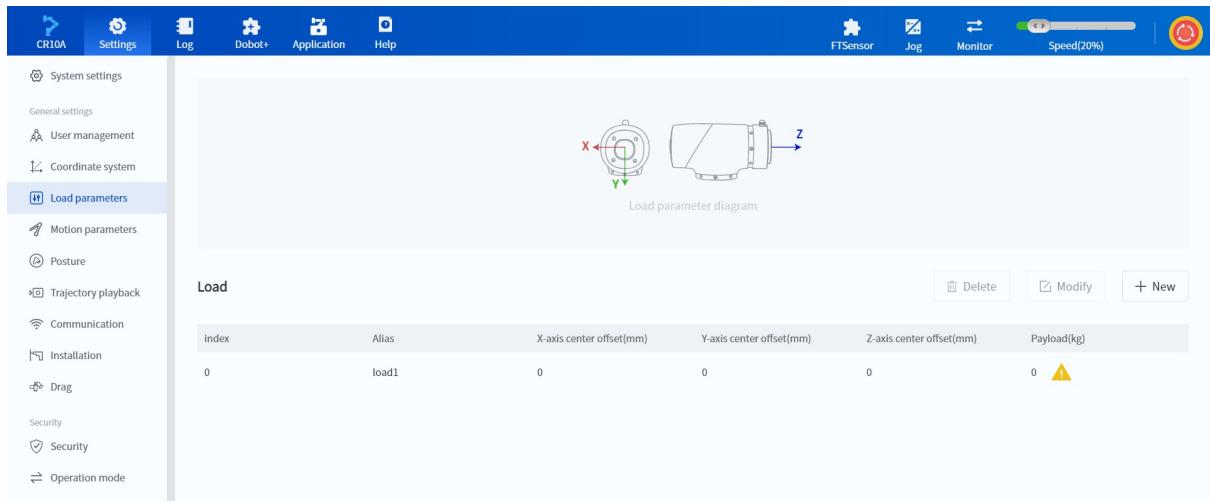


**Example 1:**

```
-- Set the payload to 1.5kg, and offset coordinates to {0, 0, 10}.  
SetPayload(1.5, {0, 0, 10})
```

**Required parameter 2:**

- **name:** The name of the preset load parameter group, which must first be saved in the software.



## Example 2:

```
-- Set the payload using the preset load parameter group named "load1".
SetPayload("load1")
```

# User

### Command:

```
User(user)
```

### Description:

Set the global user coordinate system. If you do not specify the user coordinate system through optional parameters when calling other commands, the global user coordinate system will be used.

The global user coordinate system set by this command is only effective during the current project. If not specified, the default global user coordinate system is User coordinate system 0.

### Required parameter:

**user:** The user coordinate system to switch to, specified by an index, which must first be added in the software. If the specified coordinate system index is null or does not exist, the project will stop running and an error will be reported.

### Example:

```
-- Switch the global coordinate system to User coordinate system 1.
User(1)
-- The following example demonstrates the effective range of the global user coordinate system.

MovL({pose={300,200,300,180,0,0} }) -- Use global user coordinate system (1)
```

```
MovL({pose={300,200,300,180,0,0} },{user=2}) -- Use the user coordinate system (2) specified by the optional parameter  
MovL(P1) -- P1 is the taught point, using its associated user coordinate system.
```



## SetUser

### Command:

```
SetUser(index,table,type)
```

### Description:

Modify the specified user coordinate system.

### Required parameter:

- **index:** The user coordinate system index, which must first be added in the software. If the specified coordinate system index does not exist, the project will stop and return an error.
- **table:** The modified user coordinate system, formatted as {x, y, z, rx, ry, rz}. When dealing with rotation values for rx, ry, rz, it is recommended to use the CalcUser command to obtain them.

### Optional parameter:

- **type:** Whether to save globally. 0 by default.
  - 0: The coordinate system modified by this command will only apply during the current project and will revert to the original values once the project stops.
  - 1: The coordinate system modified by this command will be saved globally and persist even after the project stops. When `type=1` (global save), you need to manually switch to the corresponding coordinate system settings interface and then switch back again to see the updated values, meaning the interface needs to be manually refreshed.

### Example:

```
-- Modify the User coordinate system 1 to the new coordinate system calculated by "CalcUser",  
effective only during project runtime.  
newUser = CalcUser(1,1,{10,10,10,10,10,10})  
SetUser(1,newUser)  
User(1)-- Switch the global coordinate system to 1
```

```
-- Modify the user coordinate system 1 to the new coordinate system calculated by "CalcUser" and save it globally.  
newUser = CalcUser(1,1,{10,10,10,10,10,10})  
SetUser(1,newUser,1)  
MovL(P1,{user=1}) -- Use the optional parameter to specify the reference user coordinate system for this command as 1
```

## CalcUser

### Command:

```
CalcUser(index,matrix_direction,table)
```

### Description:

Calculate the user coordinate system.

### Required parameter:

- **index:** The user coordinate system index, range: [0,50]. The initial value of coordinate system 0 refers to the base coordinate system.
- **matrix\_direction:** Specify the calculation direction.
  - **1:** Left-multiply, meaning the coordinate system specified by the **index** is rotated relative to the base coordinate system using the values provided in **table**.
  - **0:** Right-multiply, meaning the coordinate system specified by the **index** is rotated relative to itself using the values provided in **table**.
- **table:** The offset values for the user coordinate system, formatted as {x, y, z, rx, ry, rz}.

### Return:

The calculated user coordinate system, formatted as {x, y, z, rx, ry, rz}.

### Example:

```
-- The following calculation is equivalent to: A coordinate system with an initial posture the same as User coordinate system 1 is translated by {x=10, y=10, z=10} and rotated by {rx=10, ry=10, rz=10} along the base coordinate system to get the newUser.  
newUser = CalcUser(1,1,{10,10,10,10,10,10})
```

```
-- The following calculation is equivalent to: A coordinate system with an initial posture the same as User coordinate system 1 is translated by {x=10, y=10, z=10} and rotated by {rx=10, ry=10, rz=10} along User coordinate system 1 to get the newUser.  
newUser = CalcUser(1,0,{10,10,10,10,10,10})
```

## Tool

### Command:

```
Tool(tool)
```

### Description:

Switch the global tool coordinate system. If you do not specify the tool coordinate system through optional parameters when calling other commands, the global tool coordinate system will be used.

The global tool coordinate system set by this command is only effective during the current project. If not specified, the default global tool coordinate system is Tool coordinate system 0.

#### Required parameter:

**tool:** The tool coordinate system to switch to, specified by an index, which must first be added in the software. If the specified coordinate system index is null or does not exist, the project will stop running and an error will be reported.

#### Example:

```
-- Switch the global coordinate system to tool coordinate system 1.  
Tool(1)  
-- The following example demonstrates the effective range of the global tool coordinate system.  
  
MovL({pose={300,200,300,180,0,0} }) -- Use global tool coordinate system (1)  
MovL({pose={300,200,300,180,0,0} },{tool=2}) -- Use the tool coordinate system (2) specified by the optional parameter  
MovL(P1) -- P1 is the taught point, using its associated tool coordinate system.
```



## SetTool

#### Command:

```
SetTool(index,table,type)
```

#### Description:

Modify the specified tool coordinate system.

#### Required parameter:

- **index:** The tool coordinate system index, which must first be added in the software. If the specified coordinate system index does not exist, the project will stop and return an error.
- **table:** The modified tool coordinate system, formatted as {x, y, z, rx, ry, rz}. When dealing with rotation values for rx, ry, rz, it is recommended to use the CalcTool command to obtain them.

#### Optional parameter:

- **type:** Whether to save globally. 0 by default.
  - 0: The coordinate system modified by this command will only apply during the current project and will revert to the original values once the project stops.
  - 1: The coordinate system modified by this command will be saved globally and persist even after the project stops. When `type=1` (global save), you need to manually switch to the

corresponding coordinate system settings interface and then switch back again to see the updated values, meaning the interface needs to be manually refreshed.

#### Example:

```
-- Modify the Tool coordinate system 1 to the new coordinate system calculated by "CalcTool",  
effective only during project runtime.  
newTool = CalcTool(1,1,{10,10,10,10,10,10})  
SetTool(1,newTool)  
Tool(1) -- Switch the global tool coordinate system to 1
```

```
-- Modify the tool coordinate system 1 to the new coordinate system calculated by "CalcTool" a  
nd save it globally.  
newTool = CalcTool(1,1,{10,10,10,10,10,10})  
SetTool(1,newTool,1)  
MovL(P1,{tool=1}) -- Use the optional parameter to specify the reference tool coordinate syste  
m for this command as 1
```

## CalcTool

#### Command:

```
CalcTool(index,matrix_direction,table)
```

#### Description:

Calculate the tool coordinate system.

#### Required parameter:

- **index:** The tool coordinate system index, range: [0,50]. The initial value of coordinate system 0 is the flange coordinate system (TCP0).
- **matrix\_direction:** Specify the calculation direction.
  - **1:** Left-multiply, meaning the coordinate system specified by the **index** is rotated relative to the flange coordinate system (TCP0) using the values provided in **table**.
  - **0:** Right-multiply, meaning the coordinate system specified by the **index** is rotated relative to itself using the values provided in **table**.
- **table:** tool coordinate system (format: {x, y, z, rx, ry, rz}).

#### Return:

The calculated tool coordinate system, formatted as {x, y, z, rx, ry, rz}.

#### Example:

```
-- The following calculation is equivalent to: A coordinate system with an initial posture the  
same as Tool coordinate system 1 is translated by {x=10, y=10, z=10} and rotated by {rx=10, r  
y=10, rz=10} along the flange coordinate system (TCP0) to get the newTool.
```

```
newTool = CalcTool(1,1,{10,10,10,10,10,10})
```

```
-- The following calculation is equivalent to: A coordinate system with an initial posture the  
same as Tool coordinate system 1 is translated by {x=10, y=10, z=10} and rotated by {rx=10, r  
y=10, rz=10} along Tool coordinate system 1 to get the newTool.  
newTool = CalcTool(1,0,{10,10,10,10,10,10})
```

## GetPose

### Command:

```
GetPose(user_index, tool_index)
```

### Description:

Get the the robot's real-time posture.

If this command is called between two motion commands, the obtained point may be affected by the continuous path settings.

- If the continuous path is disabled (cp or r is set to 0), the exact target point will be returned.
- If the continuous path is enabled, a point along the transition curve will be returned.

### Optional parameter:

- **user\_index:** The user coordinate system index for the posture. The coordinate system must first be added in the software. The global user coordinate system will be used when the index is not set.
- **tool\_index:** The tool coordinate system index for the posture. The coordinate system must first be added in the software. The global tool coordinate system will be used when the index is not set.

### Return:

Current posture of the robot arm: {pose = {x, y, z, rx, ry, rz}}

### Example:

```
-- The robot moves to P1, then returns to the current posture.  
local currentPose = GetPose()  
MovJ(P1)  
MovJ(currentPose)
```

```
-- The robot moves to P1 to get P1's posture, then moves to P2.  
MovJ(P1,{cp=0})  
local currentPose = GetPose() -- Get P1's posture.  
MovJ(P2)
```

```
-- Get the current posture under User coordinate system 1 and Tool coordinate system 1.
```

```
local currentPose = GetPose(1,1)
```

## GetAngle

### Command:

```
GetAngle()
```

### Description:

Get the real-time joint angle of the robot.

If this command is called between two motion commands, the obtained joint angle may be affected by the continuous path settings.

- If the continuous path function is disabled (cp or r is set to 0), the joint angles corresponding to target point will be accurately returned.
- If the continuous path function is enabled, the joint angles corresponding to a point along the transition curve will be returned.

### Return:

The current joint angle of the robot: joint = {j1, j2, j3, j4, j5, j6} }.

### Example:

```
-- The robot moves to P1, then returns to the current posture.  
local currentAngle = GetAngle()  
MovJ(P1)  
MovJ(currentAngle)
```

```
-- The robot moves to P1 to get the joint angle of P1, then moves to P2.  
MovJ(P1,{cp=0})  
local currentAngle = GetAngle() -- Get the joint angle of P1.  
MovJ(P2)
```

## GetABZ

### Command:

```
GetABZ()
```

### Description:

Get the current position of the encoder.

### **Return:**

The current position of the encoder.

### **Example:**

```
-- Get the current position of the ABZ encoder and assign the value to the variable "abz".  
local abz = GetABZ()
```

## **CheckMovJ**

### **Command:**

```
CheckMovJ(P, {user = 1, tool = 0, a = 20, v = 50, cp = 100})
```

### **Description:**

Check whether a joint motion from the current point to the target point is reachable. The system calculates the entire motion trajectory and checks if any point in the path is unreachable.

### **Required parameter:**

**P:** The target point.

### **Optional parameter:**

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **cp:** The ratio for smooth transition. Range: [0,100].

See [General description](#) for details.

### **Return:**

Check result.

- 0: No error.
- 16: End point closed to shoulder singularity.
- 17: End point inverse kinematics error with no solution.
- 18: Inverse kinematics error with result out of working area.
- 22: Arm orientation error.
- 26: End point closed to wrist singularity.
- 27: End point closed to elbow singularity.
- 29: Speed parameter error.
- 30: Full parameter inverse kinematics error with no solution.
- 32: Shoulder singularity in trajectory.

- 33: Inverse kinematics error with no solution in trajectory.
- 34: Inverse kinematics error with result out of working area in trajectory.
- 35: Wrist singularity in trajectory.
- 36: Elbow singularity in trajectory.
- 37: Joint angle is changed over 180 degrees.

**Example:**

```
-- Check whether the robot can reach P1 through joint motion with the default setting. If it can, move to P1 through joint motion.
local status=CheckMovJ(P1)
if(status==0)
then
    MovJ(P1)
    status=CheckMovJ(P2) -- Perform feasibility check from P1 to P2 after the robot reaches P1
    if(status==0)
    then
        MovJ(P2)
    end
end
```

## CheckMovL

**Command:**

```
CheckMovL(P, {user = 1, tool = 0, a = 20, v = 50, cp = 100, r = 5})
```

**Description:**

Check whether a linear motion from the current point to the target point is reachable. The system calculates the entire motion trajectory and checks if any point in the path is unreachable.

**Required parameter:**

**P:** The target point.

**Optional parameter:**

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the v parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the **cp** parameter will be ignored.

See [General description](#) for details.

#### **Return:**

Check result.

- 0: No error.
- 16: End point closed to shoulder singularity.
- 17: End point inverse kinematics error with no solution.
- 18: Inverse kinematics error with result out of working area.
- 22: Arm orientation error.
- 26: End point closed to wrist singularity.
- 27: End point closed to elbow singularity.
- 29: Speed parameter error.
- 30: Full parameter inverse kinematics error with no solution.
- 32: Shoulder singularity in trajectory.
- 33: Inverse kinematics error with no solution in trajectory.
- 34: Inverse kinematics error with result out of working area in trajectory.
- 35: Wrist singularity in trajectory.
- 36: Elbow singularity in trajectory.
- 37: Joint angle is changed over 180 degrees.

#### **Example:**

```
-- Check whether the robot can reach P1 through linear motion with the default setting. If it
can, move to P1 through linear motion.
local status=CheckMovL(P1)
if(status==0)
then
  MovL(P1)
  status=CheckMovL(P2) -- Perform feasibility check from P1 to P2 after the robot reaches P1
  if(status==0)
  then
    MovL(P2)
  end
end
```

## **SetSafeWallEnable**

#### **Command:**

```
SetSafeWallEnable(index,value)
```

### Description:

Enable or disable the specified safety wall. The safety wall status set by this command is effective only while the current project is running and will revert to the original value once the project stops.

### Required parameter:

- **index:** The safety wall index, which needs to be added in the software in advance. Range: [1,8].
- **value:**
  - true: Enable the safety wall.
  - false: Disable the safety wall.

### Example:

```
-- Enable the safety wall with index 1.  
SetSafeWallEnable(1,true)
```

## SetWorkZoneEnable

### Command:

```
SetWorkZoneEnable(index,value)
```

### Description:

Enable or disable the specified safety zone. The safety zone status set by this command is effective only while the current project is running and will revert to the original value once the project stops.

### Required parameter:

- **index:** The safety zone index, which needs to be added in the software in advance. Range: [1,6].
- **value:**
  - true: Enable the safety zone.
  - false: Disable the safety zone.

### Example:

```
-- Enable the safety zone with index 1.  
SetWorkZoneEnable(1,true)
```

## SetCollisionLevel

### Command:

```
SetCollisionLevel(level)
```

### Description:

Set the collision detection level. For CRA models, this command sets the [safety limits](#) for TCP force and power values. The value set by this command is effective only while the current project is running and will revert to the original value once the project stops.

### Required parameter:

**level:** Collision detection level. **0:** Disable the collision detection. **1 – 5:** The larger the number, the higher the sensitivity.

### Example:

```
-- Set the collision detection level to 3.  
SetCollisionLevel(3)
```

## SetBackDistance

### Command:

```
SetBackDistance(distance)
```

### Description:

Set the distance the robot will retract along its original path after a collision is detected. The value set by this command is effective only while the current project is running and will revert to the original value once the project stops.

### Required parameter:

**distance:** The distance to retract after a collision, range: [0,50], unit: mm.

### Example:

```
-- Set the collision backoff distance to 20mm.  
SetBackDistance(20)
```

# IO

## Command list

The IO commands are used to read and write system IO and set relevant parameters.

Command	Function
DI	Get status of DI port
DIGroup	Get status of multiple DI ports
DO	Set status of DO port
DOGroup	Set status of multiple DO ports
GetDO	Get status of DO port
GetDOGroup	Get status of multiple DO ports
AI	Get value of AI port
AO	Set value of AO port
GetAO	Get value of AO port

## DI

### Command:

```
DI(index)
```

### Description:

Get the status of the digital input (DI) port.

### Required parameter:

**index:** DI index.

### Return:

Status (ON/OFF) of corresponding DI port.

### Example:

```
-- The robot arm moves to P1 through linear motion when DI1 is ON.  
if (DI(1)==ON)  
then  
    MovL(P1)
```

```
end
```

## DIGroup

### Command:

```
DIGroup(index1,...,indexN)
```

### Description:

Get the status of multiple digital input (DI) ports.

### Required parameter:

**index:** DI index. You can input multiple separated by comma.

### Return:

Status (ON/OFF) of corresponding DI port, returned in the format of array.

### Example:

```
-- The robot arm moves to P1 through linear motion when DI1 and DI2 are ON.  
local digroup = DIGroup(1,2)  
if (digroup[1]&digroup[2]==ON)  
then  
    MovL(P1)  
end
```

## DO

### Command:

```
DO(index,ON|OFF,time_ms)
```

### Description:

Set the status of digital output (DO) port.

### Required parameter:

- **index:** DO index.
- **ON|OFF:** The status to set for the DO port.

### Optional parameter:

**time\_ms:** Continuous output time. Unit: ms, range: [25, 60000]. If this parameter is set, the system will automatically invert the DO after the specified time. The inversion is an asynchronous action, which will not block the command queue. After the DO output is executed, the system will execute the next command.

**Example:**

```
-- Set D01 to ON.  
DO(1,ON)
```

```
-- Set D01 to ON, and set it automatically to OFF after 50ms.  
DO(1,ON,50)
```

## DOGroup

**Command:**

```
DOGroup({index1,ON|OFF},...,{indexN,ON|OFF})
```

**Description:**

Set the status of multiple digital output (DO) ports.

**Required parameter:**

- **index:** DO index.
- **ON|OFF:** The status to set for the DO port.

You can set multiple groups, with each group within a brace and different groups separated by commas.

**Example:**

```
-- Set D01 and D02 to ON.  
DOGroup({1,ON},{2,ON})
```

## GetDO

**Command:**

```
GetDO(index)
```

**Description:**

Get the status of digital output (DO) port.

**Required parameter:**

**index:** DO index.

**Return**

Status (ON/OFF) of corresponding DI port.

**Example:**

```
-- Get the status of D01.  
local do1 = GetDO(1)
```

## GetDOGroup

**Command:**

```
GetDOGroup(index1,...,indexN)
```

**Description:**

Get the status of multiple digital output (DO) ports.

**Required parameter:**

**index:** DO index. You can input multiple separated by comma.

**Return:**

Status (ON/OFF) of corresponding DO port, returned in the format of array.

**Example:**

```
-- Get the status of D01 and D02.  
local dogroup = GetDOGroup(1,2)  
local do1 = dogroup[1]  
local do2 = dogroup[2]
```

## AI

**Command:**

```
AI(index)
```

**Description:**

Get the value of analog input (AI) port. The meaning of the value (Voltage/Current) can be viewed and modified in DobotStudio Pro under **Monitor > Controller AI/AO**.

### **Required parameter:**

**index:** AI index.

### **Return:**

Value of corresponding AI port.

### **Example:**

```
-- Get the value of AI1
local ai1 = AI(1)
```

## **AO**

### **Command:**

```
AO(index,value)
```

### **Description:**

Set the value of analog output (AO) port. The meaning of the value (Voltage/Current) can be viewed and modified in DobotStudio Pro under **Monitor > Controller AI/AO**.

### **Required parameter:**

- **index:** AO index.
- **value:** The value to be set. Voltage range: [0, 10], unit: V. Current range: [4, 20], unit: mA.

### **Example:**

```
-- Set the value of A01 to 2.
AO(1,2)
```

## **GetAO**

### **Command:**

```
GetAO(index)
```

### **Description:**

Get the value of analog output (AO) port. The meaning of the value (Voltage/Current) can be viewed and modified in DobotStudio Pro under **Monitor > Controller AI/AO**.

### **Required parameter:**

**index:** AO index.

**Return**

Value of corresponding AO port.

**Example:**

```
-- Get the status of A01.  
local ao1 = GetAO(1)
```

# Tool

## Command list

The tool commands are used to read and write the tool IO and set relevant parameters.

Command	Function
ToolDI	Read status of tool digital input port
ToolDO	Set status of tool digital output port (queue command)
GetToolDO	Get the current status of tool digital output port
ToolAI	Read value of tool analog input port
SetToolMode	Set communication mode of tool multiplex terminal
GetToolMode	Get communication mode of tool multiplex terminal
SetToolPower	Set power status of end tool
SetTool485	Set data type for RS485 interface of end tool

### ToolDI

#### Command:

```
ToolDI(index)
```

#### Description

Read the status of tool digital input port.

#### Required parameter:

**index:** Tool DI index.

#### Return:

Status (ON/OFF) of corresponding DI port.

#### Example:

```
-- The robot moves to P1 in a linear mode when the status of tool DI1 is ON.  
if (ToolDI(1)==ON)  
then  
    MovL(P1)  
end
```

## ToolDO

### Command:

```
ToolDO(index,ON|OFF)
```

### Description

Set the status of tool digital output port.

### Required parameter:

- **index:** Tool DO index.
- **ON|OFF:** The status to set for the DO port.

### Example:

```
-- Set the status of tool D01 to ON.  
ToolDO(1,ON)
```

## GetToolDO

### Command:

```
GetToolDO(index)
```

### Description

Get the current status of tool digital output port.

### Required parameter:

**index:** Tool DO index.

### Return

Status (ON/OFF) of corresponding tool DO port.

### Example:

```
-- Get the current status of tool D01.  
GetToolDO(1)
```

## ToolAI

### Command:

```
ToolAI(index)
```

## Description

Read the value of tool analog input port.

You need to set the port to analog input mode through SetToolMode before using it.

### NOTE

The robot without tool AI interface has no effect when calling this interface.

## Required parameter:

**index:** Tool AI index.

## Return:

Value of corresponding AI port.

## Example:

```
-- Read the value of tool AI1 and assign it to the variable "test".  
test = ToolAI(1)
```

# SetToolMode

## Command:

```
SetToolMode(mode,type,identify)
```

## Description

For robot models with end AI1 and AI2 interfaces that share a multiplex terminal with RS485, you can use this interface to set the mode of the multiplex terminal.

The default mode is **485 mode**.

### NOTE

For robots that do not support tool mode switching, calling this command will have no effect.

## Required parameter:

- **mode:** The mode of the multiplex terminal.
  - 1: RS485 mode.
  - 2: Analog input mode.
- **type:**
  - When mode is 1, this parameter is invalid.
  - When mode is 2, this parameter sets the mode of the analog input.

The units digit indicates the mode of AI1, the tens digit indicates the mode of AI2. If the tens digit is 0, only the unit digit can be specified.

#### **Value:**

- 0: 0–10V voltage input mode.
- 1: Current acquisition mode.
- 2: 0–5V voltage input mode.

#### **Example:**

- 0: Both AI1 and AI2 are in 0–10V voltage input mode.
- 1: AI2 is in 0–10V voltage input mode, AI1 is in current acquisition mode.
- 11: Both AI2 and AI1 are in current acquisition mode.
- 12: AI2 is in current acquisition mode, AI1 is in 0–5V voltage input mode.
- 20: AI2 is in 0–5V voltage input mode, AI1 is in 0–10V voltage input mode.

#### **Optional parameter:**

**identify:** Used to specify the connector when the robot has multiple aviation connectors. 1 by default.

- 1 indicates connector 1.
- 2 indicates connector 2.

#### **Example:**

```
-- Set the tool multiplex terminal to RS485 mode.
SetToolMode(1,0)
```

```
-- Set the multiplex terminal of CR20A aviation connector 2 to RS485 mode.
SetToolMode(1,0,2)
```

```
-- Set the tool multiplex terminal to analog input mode, with 0–10V voltage input mode for both inputs.
SetToolMode(2,0)
```

```
-- Set the tool multiplex terminal to analog input mode, AI1 to 0-10V voltage input mode, AI2 to current mode.  
SetToolMode(2,10)
```

## GetToolMode

### Command:

```
GetToolMode(identify)
```

### Description

Get the current mode of tool multiplex terminal.

### Optional parameter:

**identify:** Used to specify the connector when the robot has multiple aviation connectors. 1 by default.

- 1 indicates connector 1.
- 2 indicates connector 2.

### Return:

- **mode:** The mode of the multiplex terminal.
- **type:** The mode of the analog input.

See the same parameter in SetToolMode.

### Example:

```
-- Get the mode of tool multiplex terminal.  
local mode,type = GetToolMode()
```

```
-- Get the mode of the multiplex terminal for CR20A aviation connector 2.  
local mode,type = GetToolMode(2)
```

## SetToolPower

### Command:

```
SetToolPower(status)
```

### Description

Set the power status of the end tool, generally used for restarting the end power, such as re-powering and re-initializing the gripper.

It is recommended to wait at least **4 ms** between consecutive calls to this interface.

### NOTE

After turning off the end power, the tool DO will also become inactive.

#### **Required parameter:**

**status:** The power status of the end tool.

- 0: Power off.
- 1: Power on.

#### **Example:**

```
-- Restart the power of the end tool.  
SetToolPower(0)  
Wait(5)  
SetToolPower(1)
```

## **SetTool485**

#### **Command:**

```
SetTool485(baud,parity,stopbit,identify)
```

#### **Description**

Set the data type for the RS485 interface of the end tool.

### NOTE

The robot without tool RS485 interface has no effect when calling this interface.

#### **Required parameter:**

**baud:** Baud rate for the RS485 interface.

#### **Optional parameter:**

- **parity:** Whether there is a parity bit.
  - "O": odd.
  - "E": even.
  - "N": no parity bit.

- "N" by default.
- **stopbit:** Stop bit length. Range: 0.5, 1, 1.5, 2. 1 by default.  
If the error is within ±0.1, an approximate value will be automatically taken (0.4001: 0.5; 0.3999: alarm; 1.09999: 1).
- **identify:** Used to specify the connector when the robot has multiple aviation connectors. 1 refers to connector 1, 2 refers to connector 2.

**Example:**

```
-- Set the baud rate for the tool RS485 interface to 115200Hz, parity bit to N, and stop bit length to 1.  
SetTool485(115200,"N",1)
```

# TCP&UDP

## Command list

The TCP/UDP commands are used for TCP/UDP communication.

Command	Function
TCPCreate	Create TCP network object
TCPStart	Establish TCP connection
TCPRead	Receive data from TCP peer
TCPWrite	Send data to TCP peer
TCPDestroy	Disconnect TCP network and destroy socket object
UDPCreate	Create UDP network object
UDPRead	Receive data from UDP peer
UDPWrite	Send data to UDP peer

## TCPCreate

### Command:

```
TCPCreate(isServer, IP, port)
```

### Description:

Create a TCP network object, only one can be created.

### Required parameter:

- **isServer:** Specify whether to create a server.
  - **true:** Create a server.
  - **false:** Create a client.
- **IP:** IP address of the server. It must be in the same subnet as the client's IP, and there should be no conflicts.
  - When creating the server, use the robot's IP address.
  - When creating the client, use the peer's address.
- **port:** Server port.

Avoid using the following ports, as they are occupied by the system, which may cause server creation to fail.

7, 13, 22, 37, 139, 445, 502, 503 (Ports 0 – 1024 are often reserved by the Linux system, which has a high possibility of being occupied, so avoid using them as well),

1501, 1502, 1503, 4840, 8172, 9527,

11740, 22000, 22001, 29999, 30004, 30005, 30006,

60000 – 65504, 65506, 65511 – 65515, 65521, 65522.

**Return:**

- **err:** 0: TCP network has been created successfully. 1: TCP network failed to be created.
- **socket:** The created socket object.

**Example 1:**

```
-- Create a TCP server.  
local ip="192.168.5.1" -- Set the IP address of the robot as that of the server  
local port=6001 -- Server port  
local err=0  
local socket=0  
err, socket = TCPCreate(true, ip, port)
```

**Example 2:**

```
-- Create a TCP client.  
local ip="192.168.5.25" -- Set the IP address of external device (such as a camera) as that of  
the server  
local port=6001 -- Server port  
local err=0  
local socket=0  
err, socket = TCPCreate(false, ip, port)
```

## TCPStart

**Command:**

```
TCPStart(socket, timeout)
```

**Description:**

Establish TCP connection.

- If the robot is acting as a server, it waits for a client to connect.
- If the robot is acting as a client, it actively connects to the server.

**Required parameter:**

- **socket:** The created socket object.
- **timeout:** The waiting timeout, unit: s.
  - If set to 0, it waits until the connection is successfully established.
  - If set to a non-zero value, it will return a connection failure if the time exceeds the specified limit.

**Return:**

Connection result.

- 0: Connection successful.
- 1: Invalid input parameters.
- 2: Socket object does not exist.
- 3: Invalid timeout setting.
- 4: Connection failed.

**Example:**

```
-- Start establishing TCP connection until the connection is successful.
err = TCPStart(socket, 0) -- socket is the object returned by TCPCreate.
```

## TCPRead

**Command:**

```
TCPRead(socket, timeout, type)
```

**Description:**

Receive data from TCP peer.

**Required parameter:**

**socket:** The created socket object.

**Optional parameter:**

- **timeout:** The waiting timeout, unit: s.
  - If not set or set to a value  $\leq 0$ , it will wait until the data is fully read before continuing.
  - If set to a value  $> 0$ , it will proceed without waiting if the time exceeds the limit.
- **type:** The type of the returned value.
  - If not set or set to "table", the RecBuf buffer will be in table format.
  - If set to "string", the RecBuf buffer will be in string format.

**Return:**

- **err:** 0: Data has been received successfully. 1: Data failed to be received.
- **Recbuf:** The data reception buffer.

**Example:**

```
-- Receive TCP data without timeout, saving the data as a table.  
-- socket is the object returned by TCPCreate.  
err, RecBuf = TCPRead(socket) -- The data type of RecBuf is table.
```

```
-- Receive TCP data with a timeout of 5 seconds, saving the data as a table.  
-- socket is the object returned by TCPCreate.  
err, RecBuf = TCPRead(socket,5) -- The data type of RecBuf is table.
```

```
-- Receive TCP data without timeout, saving the data as a string.  
-- socket is the object returned by TCPCreate.  
err, RecBuf = TCPRead(socket,0,"string") -- The data type of RecBuf is string.
```

## TCPWrite

### Command:

```
TCPWrite(socket, buf, timeout)
```

### Description:

Send data to TCP peer.

### Required parameter:

- **socket:** The created socket object.
- **buf:** The data to be sent.

### Optional parameter:

**timeout:** The waiting timeout, unit: s.

- If not set or set to 0, it will wait until the peer has fully received the data before continuing.
- If set to a non-zero value, it will proceed without waiting if the time exceeds the limit.

### Return:

Data sending result.

- 0: Send successful.
- 1: Send failed.

### Example:

```
-- Send TCP data with the content "test", without timeout.  
TCPWrite(socket, "test") -- socket is the object returned by TCPCreate.
```

```
-- Send TCP data with the content "test", with a timeout of 5 seconds.  
TCPWrite(socket, "test", 5) -- socket is the object returned by TCPCreate.
```

## TCPDestroy

### Command:

```
TCPDestroy(socket)
```

### Description:

Disconnect the TCP connection and destroy the socket object.

### Required parameter:

**socket:** The created socket object.

### Return:

Execution result.

- 0: Execution successful.
- 1: Execution failed.

### Example:

```
-- Disconnect from the TCP peer.  
TCPDestroy(socket) -- socket is the object returned by TCPCreate.
```

## UDPCreate

### Command:

```
UDPCreate(isServer, IP, port)
```

### Description:

Create a UDP network object, only one can be created.

### Required parameter:

- **isServer:** Specify whether to create a server.
  - **true:** Create a server.
  - **false:** Create a client.
- **IP:** Fill in the IP address of the peer when creating both the server and client. It must be in the same subnet as the robot's IP, and there should be no conflicts.

- **port:**
  - When creating a server, this port is used by both the local and the peer. Do not use ports already occupied by the system, see the TCPCreate parameter description for details.
  - When creating a client, it refers to the peer's port, while the local side uses a random port for sending data.

**Return:**

- **err:** 0: UDP network has been created successfully. 1: UDP network failed to be created.
- **socket:** The created socket object.

**Example 1:**

```
-- Create a UDP server.
local ip="192.168.5.25" -- Set the IP address of external device (such as a camera) as that of
the peer
local port=6001 -- This port is used by both the local and the peer.
local err=0
local socket=0
err, socket = UDPCreate(true, ip, port)
```

**Example 2:**

```
-- Create a UDP client.
local ip="192.168.5.25" -- Set the IP address of external device (such as a camera) as that of
the peer
local port=6001 -- Peer port
local err=0
local socket=0
err, socket = UDPCreate(false, ip, port)
```

## UDPRead

**Command:**

```
UDPRead(socket, timeout, type)
```

**Description:**

Receive data from UDP peer.

**Required parameter:**

**socket:** The created socket object.

**Optional parameter:**

- **timeout:** The waiting timeout, unit: s.
  - If not set or set to a value  $\leq 0$ , it will wait until the data is fully read before continuing.

- If set to a value > 0, it will proceed without waiting if the time exceeds the limit.
- **type:** The type of the returned value.
  - If not set or set to "table", the RecBuf buffer will be in table format.
  - If set to "string", the RecBuf buffer will be in string format.

**Return:**

- **err:** 0: Data has been received successfully. 1: Data failed to be received.
- **Recbuf:** The data reception buffer.

**Example:**

```
-- Receive UDP data, saving it in both string and table formats.
-- socket is the object returned by UDPCreate.
err, RecBuf = UDPRead(socket,0,"string") -- The data type of RecBuf is string.
err, RecBuf = UDPRead(socket,0) -- The data type of RecBuf is table.
```

## UDPWrite

**Command:**

```
UDPWrite(socket, buf, timeout)
```

**Description:**

Send data to UDP peer.

**Required parameter:**

- **socket:** The created socket object.
- **buf:** The data to be sent.

**Optional parameter:**

**timeout:** The waiting timeout, unit: s.

- If not set or set to 0, it will wait until the peer has fully received the data before continuing.
- If set to a non-zero value, it will proceed without waiting if the time exceeds the limit.

**Return:**

Data sending result.

- 0: Send successful.
- 1: Send failed.

**Example:**

```
-- Send UDP data "test".
UDPWrite(socket, "test") -- socket is the object returned by UDPCreate.
```

# Modbus

## Command list

The Modbus commands are used to establish the communication between Modbus master and slave. For the range and definition of the register address, please refer to the instructions of the corresponding slave.

Command	Function
ModbusCreate	Create Modbus master
ModbusRTUCREATE	Create Modbus master based on RS485
ModbusClose	Disconnect with Modbus slave
GetInBits	Read contact registers
GetInRegs	Read input registers
GetCoils	Read coil registers
SetCoils	Write to coil registers
GetHoldRegs	Read holding registers
SetHoldRegs	Write to holding registers

The Modbus function codes for different types of registers follow the standard Modbus protocol:

Register type	Read register	Write single register	Write multiple registers
Coil register	01	05	0F
Contact (discrete input) register	02	-	-
Input register	04	-	-
Holding register	03	06	10

## ModbusCreate

### Command:

```
ModbusCreate(IP, port, slave_id, isRTU)
```

### Description:

Create Modbus master based on TCP/IP protocol and establish connection with the slave. A maximum of 15 devices can be connected at the same time.

When connecting to the robot's built-in slave, set the IP to the robot's IP (default 192.168.5.1, modifiable) and the port to 502 (map1) or 1502 (map2). See [Appendix A Modbus Register Definition](#) for details.

When connecting to a third-party slave, please refer to the instructions of the corresponding slave for the register address range and definition when reading and writing registers.

#### Required parameter:

- **IP:** Slave IP address.
- **port:** Slave port.

#### Optional parameter:

- **slave\_id:** Slave ID.
- **isRTU:** Boolean.
  - When **isRTU** is false, establish Modbus TCP communication via the controller's Ethernet port.
  - When **isRTU** is true, establish Modbus RTU communication via the robot's end RS485, and only port 60000 can be used.

#### ⚠ NOTICE

This parameter determines the protocol format used to transmit data once the connection has been established, and it does not affect the connection result. Therefore, if you set the parameter incorrectly when creating a master, the master can still be created successfully, but an exception may occur in the subsequent communication.

#### Return:

- **err:**
  - 0: Modbus master has been created successfully.
  - 1: The maximum number of masters has been reached, failed to create a new master.
  - 2: Failed to initialize the master, check if the IP, port, and network are working properly.
  - 3: Failed to connect to the slave, check if the slave is functioning and the network is normal.
- **id:** The returned master index, which is used when other Modbus commands are called. Range: [0, 14].

#### Example:

```
-- Create a ModbusTCP master, and connect with the robot slave.  
-- The IP is the robot's IP, port is 502, and no slave ID is specified  
local ip="192.168.5.1"  
local port=502  
local err=0  
local id=0  
err, id = ModbusCreate(ip, port)
```

```
-- Create a ModbusTCP master, and connect with the specified slave.
```

```
-- Slave IP is 192.168.5.123, port is 503, slave ID is 1.
local ip="192.168.5.123"
local port=503
local err=0
local id=0
err, id = ModbusCreate(ip, port, 1)
```

```
-- Create a ModbusRTU-over-TCP master, and connect with the specified slave.
-- Slave IP is 192.168.5.123, port is 503, slave ID is 1.
local ip="192.168.5.123"
local port=503
local err=0
local id=0
err, id = ModbusCreate(ip, port, 1, true)
```

## ModbusRTUCreate

### Command:

```
ModbusRTUCreate(slave_id, baud, parity, data_bit, stop_bit)
```

### Description:

Create Modbus master based on the controller's RS485 interface and establish connection with the slave. A maximum of 15 devices can be connected at the same time.

### Required parameter:

- **slave\_id:** Slave ID.
- **baud:** Baud rate for the RS485 interface.

### Optional parameter:

- **parity:** Whether there is a parity bit. "O": odd, "E": even, "N": no parity bit. "E" by default.
- **data\_bit:** Data bit length. Range: 8. 8 by default.
- **stopbit:** Stop bit length. Range: 1, 2. 1 by default.

### Return:

- **err:** 0: Modbus master station has been created successfully. 1: Modbus master station failed to be created.
- **id:** The returned master index, which is used when other Modbus commands are called.

### Example:

```
-- Create a Modbus master and establish connection with the slave through RS485 interface, slave ID is 1, baud rate is 115200.
err, id = ModbusRTUCreate(1, 115200)
```

```
-- Create a Modbus master and establish connection with the slave through RS485 interface, slave ID is 1, baud rate is 115200, no parity bit, data bit length is 1, stop bit length is 2.  
err, id = ModbusRTUCreate(1, 115200, "N", 8, 2)
```

## ModbusClose

### Command:

```
ModbusClose(id)
```

### Description:

Disconnect from the Modbus slave and release the master.

### Required parameter:

**id:** The index of the created master.

### Return:

Operation result.

- 0: Disconnection successful.
- 1: Disconnection failed.

### Example:

```
-- Disconnect from the Modbus slave.  
ModbusClose(id)
```

## GetInBits

### Command:

```
GetInBits(id, addr, count)
```

### Description:

Read the contact register value from the Modbus slave. The corresponding Modbus function code is 02.

### Required parameter:

- **id:** The index of the created master.
- **addr:** The starting address of the contact register.
- **count:** The number of contact registers to read. Range: [1, 2000] (ModbusTCP protocol restriction).  
For actual range, please determine according to the number of slave registers or the protocol).

### Return:

The values read from the contact register, stored in a table. The first value in table corresponds to the value of contact register at the starting address.

**Example:**

```
-- Read 5 contact registers starting from address 0.  
inBits = GetInBits(id, 0, 5)
```

## GetInRegs

**Command:**

```
GetInRegs(id, addr, count, type)
```

**Description:**

Read the input register value with the specified data type from the Modbus slave. The corresponding Modbus function code is 04.

**Required parameter:**

- **id:** The index of the created master.
- **addr:** The starting address of the input register.
- **count:** The number of input registers to read. Range: [1, 125] (ModbusTCP protocol restriction. For actual range, please determine according to the number of slave registers or the protocol).

**Optional parameter:**

**type:** Data type.

- Empty: U16 by default.
- U16: 16-bit unsigned integer (two bytes, occupy one register).
- U32: 32-bit unsigned integer (four bytes, occupy two register).
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

**Return:**

The values read from the input register, stored in a table. The first value in table corresponds to the value of input register at the starting address.

**Example:**

```
-- Read a 16-bit unsigned integer starting from address 2048.  
data = GetInRegs(id, 2048, 1)
```

```
-- Read a 32-bit unsigned integer starting from address 2048.  
data = GetInRegs(id, 2048, 2, "U32")
```

```
-- Read two 32-bit single-precision floating-point numbers starting from address 2048.  
data = GetInRegs(id, 2048, 4, "F32")
```

## GetCoils

### Command:

```
GetCoils(id, addr, count)
```

### Description:

Read the coil register value from the Modbus slave. The corresponding Modbus function code is 01.

### Required parameter:

- **id:** The index of the created master.
- **addr:** The starting address of the coil register.
- **count:** The number of coil registers to read. Range: [1, 2000] (ModbusTCP protocol restriction. For actual range, please determine according to the number of slave registers or the protocol).

### Return:

The values read from the coil register, stored in a table. The first value in table corresponds to the value of coil register at the starting address.

### Example:

```
-- Read 5 contact registers starting from address 0.  
Coils = GetCoils(id,0,5)
```

## SetCoils

### Command:

```
SetCoils(id, addr, count, table)
```

### Description:

Write specified values to the coil register at the designated address. The corresponding Modbus function code is 05 (write single) and 0F (write multiple).

### Required parameter:

- **id:** The index of the created master.
- **addr:** The starting address of the coil register.

- **count:** The number of values to write to the coil register. Range: [1, 1968] (ModbusTCP protocol restriction. For actual range, please determine according to the number of slave registers or the protocol).
- **table:** The values to write to the coil register, stored in a table. The first value in table corresponds to the value of coil register at the starting address.

**Return:**

It returns 0 or -1. **0:** The setting is successful, **-1:** The setting fails.

**Example:**

```
-- Starting from address 1024, write 5 values in succession to the coil register.
local coils = {0,1,1,1,0}
SetCoils(id, 1024, #coils, coils)
```

## GetHoldRegs

**Command:**

```
GetHoldRegs(id, addr, count, type)
```

**Description:**

Read the holding register value with the specified data type from the Modbus slave. The corresponding Modbus function code is 03.

**Required parameter:**

- **id:** The index of the created master.
- **addr:** The starting address of the holding register.
- **count:** The number of holding registers to read. Range: [1, 125] (ModbusTCP protocol restriction. For actual range, please determine according to the number of slave registers or the protocol).

**Optional parameter:**

**type:** Data type.

- Empty: U16 by default.
- U16: 16-bit unsigned integer (two bytes, occupy one register).
- U32: 32-bit unsigned integer (four bytes, occupy two register).
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

**Return:**

The values read from the holding register, stored in a table. The first value in table corresponds to the value of holding register at the starting address.

## Example:

```
-- Read a 16-bit unsigned integer starting from address 2048.  
data = GetHoldRegs(id, 2048, 1)
```

```
-- Read a 32-bit unsigned integer starting from address 2048.  
data = GetHoldRegs(id, 2048, 2, "U32")
```

```
-- Read two 32-bit single-precision floating-point numbers starting from address 2048.  
data = GetHoldRegs(id, 2048, 4, "F32")
```

## SetHoldRegs

### Command:

```
SetHoldRegs(id, addr, count, table, type)
```

### Description:

Write specified values to the holding register at the designated address according to the specified data type. The corresponding Modbus function code is 06 (write single) and 10 (write multiple).

### Required parameter:

- **id:** The index of the created master.
- **addr:** The starting address of the holding register.
- **count:** The number of values to write to the holding register. Range: [1, 123] (ModbusTCP protocol restriction. For actual range, please determine according to the number of slave registers or the protocol).
- **table:** The values to write to the holding register, stored in a table. The first value in table corresponds to the value of holding register at the starting address.

### Optional parameter:

**type:** Data type.

- Empty: U16 by default.
- U16: 16-bit unsigned integer (two bytes, occupy one register).
- U32: 32-bit unsigned integer (four bytes, occupy two register).
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

### Return:

It returns 0 or -1. **0:** The setting is successful, **-1:** The setting fails.

**Example:**

```
-- Write a 16-bit unsigned integer starting from address 2048.  
local data = {12}  
SetHoldRegs(id, 2048, 1, data)
```

```
-- Write a 64-bit double-precision floating-point number starting from address 2048.  
local data = {95.32105}  
SetHoldRegs(id, 2048, 4, data, "F64")
```

```
-- Write two 64-bit double-precision floating-point numbers starting from address 2048.  
local data = {95.32105, 104.24411}  
SetHoldRegs(id, 2048, 8, data, "F64")
```

# Bus registers

## Command list

The bus register commands are used to read and write Profinet or Ethernet/IP bus registers.

 **NOTE**

Magician E6 does not support this set of commands.

Command	Function
<a href="#">GetInputBool</a>	Get boolean value from the specified input register address
<a href="#">GetInputInt</a>	Get int value from the specified input register address
<a href="#">GetInputFloat</a>	Get float value from the specified input register address
<a href="#">GetOutputBool</a>	Get boolean value from the specified output register address
<a href="#">GetOutputInt</a>	Get int value from the specified output register address
<a href="#">GetOutputFloat</a>	Get float value from the specified output register address
<a href="#">SetOutputBool</a>	Set boolean value at the specified output register address
<a href="#">SetOutputInt</a>	Set int value at the specified output register address
<a href="#">SetOutputFloat</a>	Set float value at the specified output register address

## GetInputBool

**Command:**

```
GetInputBool(address)
```

**Description:**

Get the boolean value from the specified input register address.

**Required parameter:**

**address:** Register address, range: [0-63].

**Return:**

The value of the specified register address, either 0 or 1.

### **Example:**

```
Execute subsequent operations when the value of input register 0 is 1.  
if(GetInputBool(0)==1)  
then  
    -- Execute subsequent operations  
end
```

## **GetInputInt**

### **Command:**

```
GetInputInt(address)
```

### **Description:**

Get the int value from the specified input register address.

### **Required parameter:**

**address:** Register address, range: [0-23].

### **Return:**

The value of the specified register address, which is an integer (data type: int32).

### **Example:**

```
-- Get the value from input register 1 and assign it to the variable "regInt".  
local regInt = GetInputInt(1)
```

## **GetInputFloat**

### **Command:**

```
GetInputFloat(address)
```

### **Description:**

Get the float value of the specified input register address.

### **Required parameter:**

**address:** Register address, range: [0-23].

### **Return:**

The value of the specified register address, which is a single-precision floating-point number (data type: float).

**Example:**

```
-- Get the value from input register 2 and assign it to the variable "regFloat".
local regFloat = GetInputFloat(2)
```

## GetOutputBool

**Command:**

```
GetOutputBool(address)
```

**Description:**

Get the boolean value from the specified output register address.

**Required parameter:**

**address:** Register address, range: [0-63].

**Return:**

The value of the specified register address, either 0 or 1.

**Example:**

```
Execute subsequent operations when the value of output register 0 is 1.
if(GetOutputBool(0)==1)
then
    -- Execute subsequent operations
end
```

## GetOutputInt

**Command:**

```
GetOutputInt(address)
```

**Description:**

Get the int value from the specified output register address.

**Required parameter:**

**address:** Register address, range: [0-23].

#### **Return:**

The value of the specified register address, which is an integer (data type: int32).

#### **Example:**

```
-- Get the value from output register 1 and assign it to the variable "regInt".  
local regInt = GetOutputInt(1)
```

## **GetOutputFloat**

#### **Command:**

```
GetOutputFloat(address)
```

#### **Description:**

Get the float value from the specified output register address.

#### **Required parameter:**

**address:** Register address, range: [0-23].

#### **Return:**

The value of the specified register address, which is a single-precision floating-point number (data type: float).

#### **Example:**

```
-- Get the value from output register 2 and assign it to the variable "regFloat".  
local regFloat = GetOutputFloat(2)
```

## **SetOutputBool**

#### **Command:**

```
SetOutputBool(address, value)
```

#### **Description:**

Set the boolean value at the specified output register address.

#### **Required parameter:**

- **address:** Register address, range: [0-63].
- **value:** Value to set, supporting boolean or 0/1.

### **Example:**

```
-- Set the value of output register 0 to 0.  
SetOutputBool(0,0)
```

## **SetOutputInt**

### **Command:**

```
SetOutputInt(address, value)
```

### **Description:**

Set the int value at the specified output register address.

### **Required parameter:**

- **address:** Register address, range: [0-23].
- **value:** Value to set, supporting integer (int32).

### **Example:**

```
-- Set the value of output register 1 to 123.  
SetOutputInt(1,123)
```

## **SetOutputFloat**

### **Command:**

```
SetOutputFloat(address, value)
```

### **Description:**

Set the float value at the specified output register address.

### **Required parameter:**

- **address:** Register address, range: [0-23].
- **value:** Value to set, supporting single-precision floating-point number (float).

Limited by the storage mechanism (IEEE754), a single-precision floating-point number can hold about 6 to 7 significant digits (regardless of the decimal point position). Values with more than 6 significant digits may have deviations.

### **Example:**

```
-- Set the value of output register 2 to 12.3.  
SetOutputFloat(2,12.3)
```

# Program control

## Command list

The program control commands are general commands related to program control.

Command	Function
Print	Print debug information to the console
Log	Output custom log
Wait	Wait for a specified time or until a condition is met, then proceed to the next command
Pause	Pause running the script
Halt	Stop running the script
ResetElapsedTime	Start timing
ElapsedTime	Stop timing
Systime	Get current system time
SetGlobalVariable	Set global variable
Popup	Set a popup for script execution

## Print

### Command:

```
Print(value)
```

### Description:

Print debug information to the console (the command can also be written as `print` ).

#### NOTE

The format in which variables are printed may differ slightly from the format described in this document. However, they represent the same data structure, so you can refer to the format described in this document for understanding and usage.

-- For example, if a variable's format is described as `{pose={x,y,z,rx,ry,rz}}`, the printed output might look like:

```



```

### Required parameter:

**value:** The data to be printed.

### Example:

```
-- Print the string "Success" to the console.
Print('Success')
```

## Log

### Command:

```
Log(value)
```

### Description:

Output the custom log message, which can be viewed and exported on the Log page of the software.

The screenshot shows the CR10A software interface with the 'Log' tab selected. The 'History' sub-tab is active. The log table has columns for Date, Type, and Message. The log entries are as follows:

Date	Type	Message
2024-12-16 20:51:15	All	[2024-12-16 20:51:15] [Dobot] Initialize index of tool coordinate system for layer 1.
	Error	[2024-12-16 20:51:15] [Dobot] Error: Failed to initialize tool coordinate system.
	Warning	[2024-12-16 20:51:15] [Dobot] Warning: Tool coordinate system initialized.
	Information	[2024-12-16 20:51:15] [Dobot] Information: Tool coordinate system initialized.
	Custom	[2024-12-16 20:51:15] [Dobot] Custom: Tool coordinate system initialized.
	All	[2024-12-16 20:51:15] [Dobot] All: Tool coordinate system initialized.
	Error	[2024-12-16 20:51:15] [Dobot] Error: Failed to initialize tool coordinate system.
	Warning	[2024-12-16 20:51:15] [Dobot] Warning: Tool coordinate system initialized.
	Information	[2024-12-16 20:51:15] [Dobot] Information: Tool coordinate system initialized.
	Custom	[2024-12-16 20:51:15] [Dobot] Custom: Tool coordinate system initialized.
	All	[2024-12-16 20:51:15] [Dobot] All: Tool coordinate system initialized.

### Required parameter:

**value:** Log message.

**Example:**

```
-- Output a log message with the content "test".
Log('test')
```

## Wait

**Command:**

```
Wait(time_ms)
Wait(check_str)
Wait(check_str, timeout_ms)
```

**Description:**

After the robot completes the previous command, it waits for a specified time or until a condition is met before proceeding to the next command. The maximum wait time is 2,147,483,647 ms. Setting a value exceeding the maximum will invalidate the command.

**Required parameter:**

- **time\_ms:** If the value is an integer, it represents the wait time. If the value is less than or equal to 0, it means no wait. Unit: ms.
- **check\_str:** If the value is a string, it represents a condition. The system will proceed once the condition becomes true.

**Optional parameter:**

**timeout\_ms:** Timeout period, Unit: ms.

- If the condition remains false and the waiting time exceeds this period, the system will proceed to the next command and return false.
- If the value is less than or equal to 0, it means an immediate timeout.
- If this parameter is not set, the system will wait indefinitely until the condition becomes true.

**Return:**

- Return **true** when the condition is met and execution continues.
- Return **false** if the condition is not met and the system continues due to timeout.

**Example:**

```
-- Wait for 300 ms.
Wait(300)
```

```
-- Continue running when DI1 is ON.
Wait("DI(1) == ON")
```

```
-- Continue running when D01 is ON and AI(1) is less than 7.  
Wait("GetDO(1) == ON and AI(1) < 7")
```

```
-- Execute different logics according to DI1 status within 1s.  
flag=Wait("DI(1) == ON", 1000)  
if(flag==true)  
then  
    -- DI1 is ON.  
else  
    -- DI1 is OFF and wait more than 1s.  
end
```

## Pause

### Command:

```
Pause()
```

### Description:

Pause running the script. The script can continue to run only through software control or remote control.

### Example:

```
-- The robot moves to P1 and then pauses running. It can continue moving to P2 only through external control.  
MovJ(P1)  
Pause()  
MovJ(P2)
```

## Halt

### Command:

```
Halt()
```

### Description:

Stop running the script.

### Example:

```
-- When variable "count=100", stop the script.  
if(count==100)
```

```
then
    Halt()
end
```

## ResetElapsedTime

### Command:

```
ResetElapsedTime()
```

### Description:

Starts timing after all commands before this command have been executed. This command should be used with ElapsedTime() to calculate the runtime.

### Example:

Refer to the example of ElapsedTime.

## ElapsedTime

### Command:

```
ElapsedTime()
```

### Description:

Stop timing and return the time difference. This command should be used with ResetElapsedTime().

### Return:

Time difference between the start and the end of timing, unit: ms.

The maximum measurable time is 4,294,967,295 ms (about 49.7 days). After exceeding this time, it will reset and start counting from 0.

### Example:

```
-- Calculate the time taken for the robot to move in a straight line between P1 and P2 back and forth 10 times, and print the result to the console.
MovJ(P2)
ResetElapsedTime()
for i=1,10 do
    MovL(P1)
    MovL(P2)
end
print (ElapsedTime())
```

## Systime

### Command:

```
Systime()
```

### Description:

Get the current system time.

### Return:

The current system time's Unix timestamp in milliseconds, which is the number of milliseconds since 00:00:00 on January 1, 1970, GMT. This command is generally used to calculate time differences.

To get the local time, please convert the obtained GMT according to your local time zone.

### Example:

```
-- Get the current system time.  
local time1 = Systime()  
print(time1) -- > 1686304295963, translated to 2023-06-09 17:51:35 BST (plus 963 milliseconds  
).  
local time2 = Systime()  
print(time2) -- > 1686304421968, translated to 2023-06-09 17:53:41 BST (plus 968 milliseconds  
).  
  
-- Calculate the time taken for the robot to move to P1, unit: ms.  
local time1 = Systime()  
MovL(P1)  
local time2 = Systime()  
print(time2-time1)
```

## SetGlobalVariable

### Command:

```
SetGlobalVariable(key,val)
```

### Description:

Set the global variable. It is recommended to use this function instead of the "==" operator when assigning values to global variables.

### Required parameter:

- **key:** The name of the global variable to set.
- **val:** The value to assign to the global variable. Supported data types include boolean, table, string, and

number.

**Example:**

```
-- Set the value of global variable g1 to 10.  
SetGlobalVariable("g1",10)
```

## Popup

**Command:**

```
Popup(message, title, type, logControl)
```

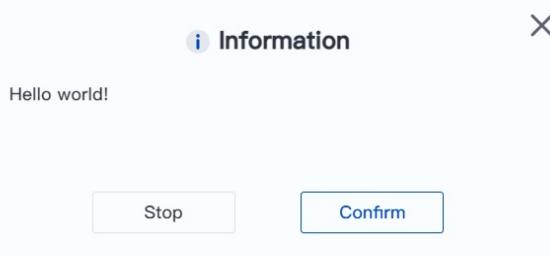
**Description:**

Used to display a user-defined information window during script execution. DobotStudio Pro will only display one popup at a time, showing the most recent popup information.

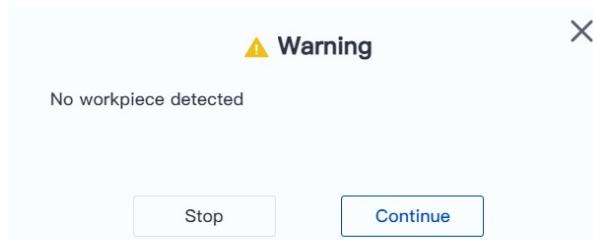
If DobotStudio Pro is not open while the project is running (e.g., when starting the project via IO or Modbus), the popup will not appear. However, warning and error-type popup instructions will still pause the project.

**Required parameter:**

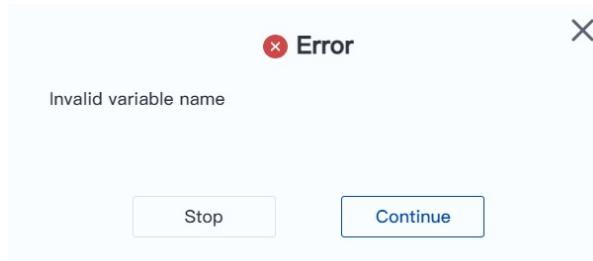
- **message:** The content of the message to display. Can be any [Lua variable](#), including strings. Strings must not exceed 128 characters. Other variable types will be converted to strings, with the resulting string size not exceeding 512 bytes.
- **title:** The title of the popup. Only supports strings, with a maximum length of 128 characters.
- **type:** Message type.
  - **0:** Information. This type of popup does not affect project execution. Click **Stop** to stop the project, or click **Confirm** to close the popup. Once the popup appears, you can also stop the project via IO or Modbus, and the popup disappears automatically after the project is stopped.



- **1:** Warning. This type of popup pauses the project. Click **Stop** to stop the project, or click **Continue** to resume the project. Once the popup appears, you can also stop or resume the project via IO or Modbus, and the popup disappears automatically after the project is stopped or resumed.



- **2:** Error. This type of popup pauses the project. Click **Stop** to stop the project, or click **Continue** to resume the project. Once the popup appears, you can also stop or resume the project via IO or Modbus, and the popup disappears automatically after the project is stopped or resumed.



- **logControl:** Whether to write the popup content to the log.

- **0:** Do not write to the log.
- **1:** Write the popup content to the corresponding type (the "Information" type popups are logged as "Custom" type). The log entry is formatted as "Popup Title: Popup Content".

Current alarm      History

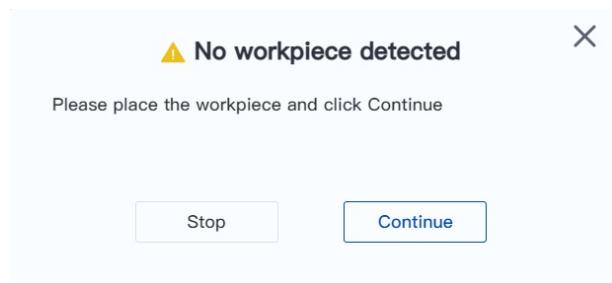
Date:  -  2024-12-11 17:50:00

Key words:

Type:  All  Error  Warning  Information  Custom

### Example 1:

```
-- Warning popup, message as string, writes to log.
Popup("Please place the workpiece and click Continue", "No workpiece detected", 1, 1)
```



### Example 2:

```
-- Information popup, message as variable, do not write to log.
Popup(a, "The value of variable a is", 0, 0)
```

 The value of variable a is

X

nil

Stop

Confirm

# Tray

## Command list

The tray is a carrying device for placing batch materials in a regular arrangement, commonly used in automated loading and unloading. There are usually many grooves distributed in an array in the tray, each of which can place one material. Using tray commands allows you to create a full array of tray points from a few taught points, enabling quick implementation of automated loading and unloading for robots.

Command	Function
CreateTray	Create tray
GetTrayPoint	Get tray point

## CreateTray

### Command:

```
CreateTray(Trayname, {Count}, {P1,P2}) -- 1D tray  
CreateTray(Trayname, {row,col}, {P1,P2,P3,P4}) -- 2D tray  
CreateTray(Trayname, {row,col,layer}, {P1,P2,P3,P4,P5,P6,P7,P8}) -- 3D tray
```

### Description:

Create tray, e.g. 1D, 2D and 3D trays.

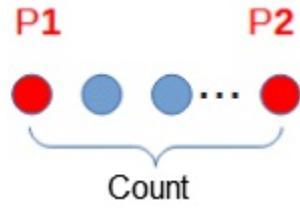
You can create up to 20 trays. Creating a tray with an existing name will overwrite the current tray without increasing the tray count.

### Required parameter:

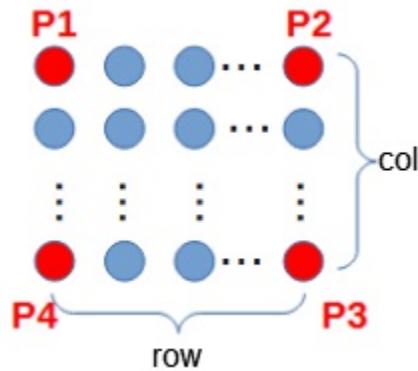
- Trayname: Tray name, a string of up to 32 bytes. Pure numbers or spaces are not allowed.

The last two parameters are table variable. The number of values in the table varies depending on the dimension of the tray to be created, as described below.

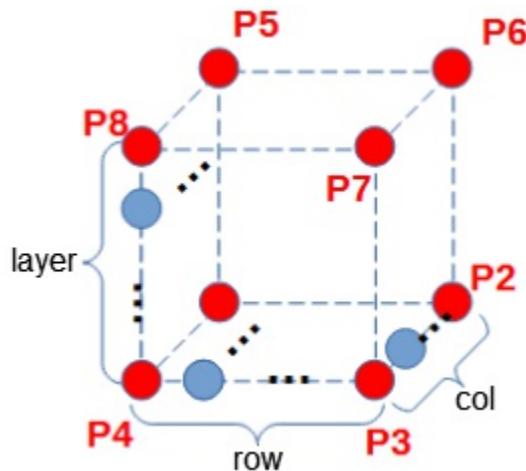
- Create 1D tray: 1D tray is a set of points equidistantly distributed on a straight line.
  - {Count}: Count refers to the number of points, range: [2, 50]. If you enter a non-integer, it will be automatically rounded down.
  - {P1, P2}: P1 and P2 are the two endpoints of the 1D tray respectively, which can be set as taught point and posture variable.



- Create 2D tray: 2D tray is a set of points distributed in an array on a plane.
  - {row,col}: "row" refers to the number of points in the row direction (P1 to P2); "col" refers to the number of points in the column direction (P1 to P4). The value range is the same as the Count of 1D tray.
  - {P1, P2, P3, P4}: P1, P2, P3 and P4 are the four endpoints of the 2D tray respectively, which can be set as taught point and posture variable.



- Create 3D tray: 3D tray is a set of points distributed three-dimensionally in space and can be considered as multiple 2D trays arranged vertically.
  - {row,col,layer}: "row" refers to the number of points in the row direction (P1 to P2); "col" refers to the number of points in the column direction (P2 to P4); "layer" refers to the number of layers (P1 to P5).
  - {P1,P2,P3,P4,P5,P6,P7,P8}: P1 to P8 are the eight endpoints of the 3D tray respectively, which can be set as taught point and posture variable.



### **⚠️NOTICE**

If an end tool is used, please make sure that the tool coordinate system for the end tool is selected when teaching points.

### Example:

```
-- Create a 1D tray of 5 points named t1.  
CreateTray("t1", {5}, {P1,P2})  
-- Create a 4x5 2D tray named t2.  
CreateTray("t2", {4,5}, {P1,P2,P3,P4})  
-- Create a 4x5x6 3D tray named t3.  
CreateTray("t2", {4,5,6}, {P1,P2,P3,P4,P5,P6,P7,P8})
```

## GetTrayPoint

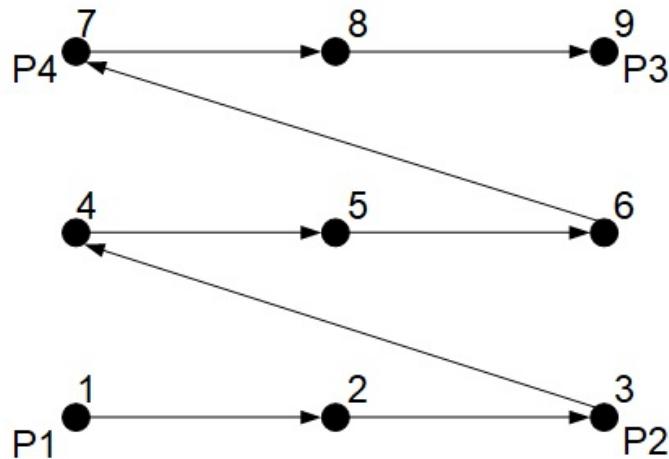
### Command:

```
GetTrayPoint(Trayname, index)
```

### Description:

Get the point of the specified index of the specified tray. The point index is related to the order of points passed in when creating the tray.

- 1D tray: The index of P1 is 1, the index of P2 is the same as the number of points, and so on.
- 2D tray: The following figure takes a 3x3 tray as an example to illustrate the relationship between taught point and point index.



- 3D tray: Similar to 2D tray, the index of the first point on the second layer is the index of the last point on the first layer plus one, and so on.

### Required parameter:

- Trayname: Created tray name, a string of up to 32 bytes.
- index: The index of the point to be obtained.

### Return:

Point coordinates of the corresponding index.

- If the point used to create the tray is a taught point, the point format returned is also a taught point.
- If the point used to create the tray is a posture variable, the point format returned is also a posture variable.

**Example:**

```
-- Get the point numbered 3 of the tray named t1.  
GetTrayPoint("t1",3)
```

# SafeSkin

## Command list

The SafeSkin commands are used to set functions of the SafeSkin.

Command	Function
EnableSafeSkin	Enable or disable the SafeSkin
SetSafeSkin	Set the sensitivity for each part of the SafeSkin

### EnableSafeSkin

#### Command:

```
EnableSafeSkin(ON|OFF)
```

#### Description:

Enable or disable the SafeSkin.

#### Required parameter:

**ON|OFF:** ON represents enabling SafeSkin, and OFF represents disabling SafeSkin.

#### Return:

- 0: Operation failed, SafeSkin is not detected.
- 1: Operation successful.

#### Example:

```
-- Enable SafeSkin.  
EnableSafeSkin(ON)
```

### SetSafeSkin

#### Command:

```
SetSafeSkin(part,status)
```

#### Description:

Set the sensitivity for each part of the SafeSkin.

**Required parameter:**

- **part:** The part to configure, range: [3, 6].
  - 3: Forearm.
  - 4: J4 joint.
  - 5: J5 joint.
  - 6: J6 joint.
- **status:** The sensitivity level to configure, range: [0, 3].
  - 0: OFF.
  - 1: Low sensitivity (sensing distance  $\leq$  5cm).
  - 2: Medium sensitivity (sensing distance  $\leq$  10cm).
  - 3: High sensitivity (sensing distance  $\leq$  15cm).

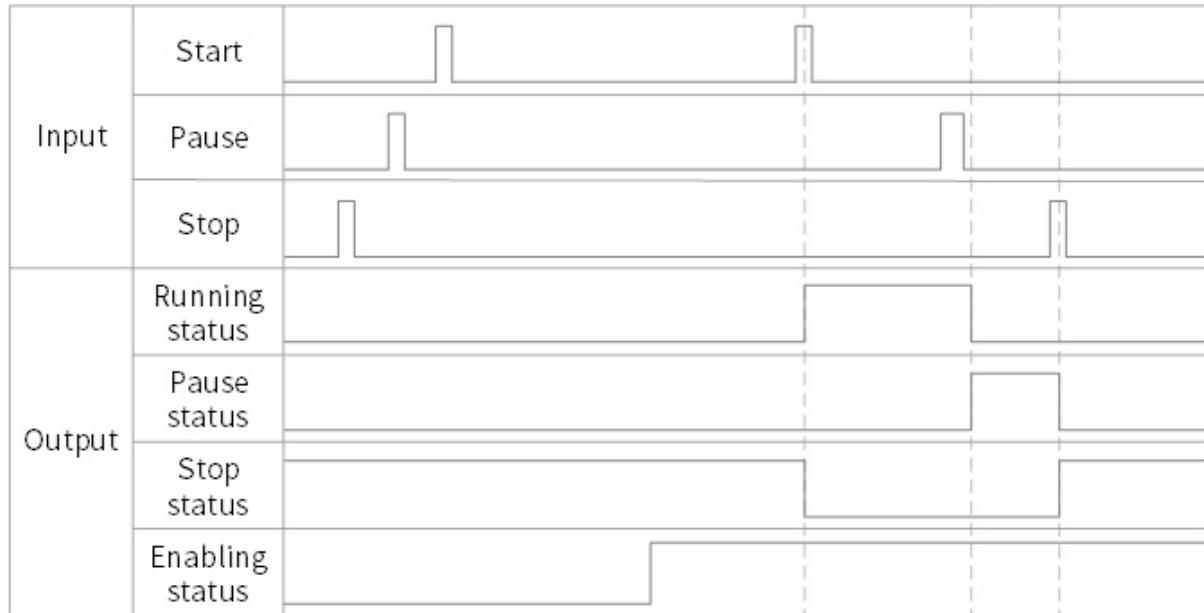
**Example:**

```
Set the SafeSkin sensitivity of the forearm to 0.  
SetSafeSkin(3.0)
```

## Appendix D Remote Control Signal Sequence Diagram

The typical sequence diagram for using remote signals to control the robot's operation is shown below.

When the robot is not enabled, the control signals related to the project will not take effect.



If the protective stop input is configured and the protective stop reset input (safety I/O) is not configured, the typical sequence diagram is shown below.

When the protection stop status takes effect, the control signals related to the project are not effective. If the protection stop status is triggered while the robot is running the project, it will pause the project, and it will resume after being reset.

	Protective stop input	[Diagram: A sequence of four rectangular pulses followed by a long gap, all aligned with vertical dashed grid lines.]			
Input	Start	[Diagram: A single short rectangular pulse aligned with the first vertical dashed line.]	[Diagram: A single short rectangular pulse aligned with the second vertical dashed line.]	[Diagram: A single short rectangular pulse aligned with the third vertical dashed line.]	
	Pause		[Diagram: A single short rectangular pulse aligned with the fourth vertical dashed line.]		
	Stop				[Diagram: A single short rectangular pulse aligned with the fifth vertical dashed line.]
Output	Protective stop status output	[Diagram: A pulse starting at the first vertical dashed line and ending at the second vertical dashed line.]	[Diagram: A pulse starting at the second vertical dashed line and ending at the third vertical dashed line.]	[Diagram: A pulse starting at the third vertical dashed line and ending at the fourth vertical dashed line.]	[Diagram: A pulse starting at the fourth vertical dashed line and ending at the fifth vertical dashed line.]
	Running status	[Diagram: A pulse starting at the first vertical dashed line and ending at the second vertical dashed line.]	[Diagram: A pulse starting at the second vertical dashed line and ending at the third vertical dashed line.]	[Diagram: A pulse starting at the third vertical dashed line and ending at the fourth vertical dashed line.]	[Diagram: A pulse starting at the fourth vertical dashed line and ending at the fifth vertical dashed line.]
	Pause status		[Diagram: A pulse starting at the second vertical dashed line and ending at the third vertical dashed line.]	[Diagram: A pulse starting at the third vertical dashed line and ending at the fourth vertical dashed line.]	[Diagram: A pulse starting at the fourth vertical dashed line and ending at the fifth vertical dashed line.]
	Stop status	[Diagram: A pulse starting at the first vertical dashed line and ending at the second vertical dashed line.]	[Diagram: A pulse starting at the second vertical dashed line and ending at the third vertical dashed line.]		[Diagram: A pulse starting at the fourth vertical dashed line and ending at the fifth vertical dashed line.]

# Appendix E Python Programming Command

- [\*\*E.1 Basic syntax\*\*](#)
- [\*\*E.2 General description\*\*](#)
- [\*\*E.3 Motion command\*\*](#)
- [\*\*E.4 Relative motion command\*\*](#)
- [\*\*E.5 Motion parameter\*\*](#)
- [\*\*E.6 IO\*\*](#)
- [\*\*E.7 Tool\*\*](#)
- [\*\*E.8 TCP&UDP\*\*](#)
- [\*\*E.9 Modbus\*\*](#)
- [\*\*E.10 Program control\*\*](#)

# Basic syntax

## NOTE

This guide assumes readers have a basic understanding of Python programming. It lists only some foundational Python syntax for quick reference.

**Python is case-sensitive, so all identifiers (variable names, function names, etc.) in your program must match the case used in the examples provided in this guide.**

## Variables

Variables are used to store values, pass values as parameters, or return results. In Python, variables do not need to be declared and can be used after assignment.

Variables are assigned by “=”, and the double “=” is used to check if the expressions on the left and right sides are equal.

In addition to the scope rules of Python itself, you can set controller-level global variables via **Monitor > Global Variables** page in DobotStudio Pro, which can be called directly from different projects within the same controller.

Global variable					
NO	Variable Name	Type	Global Hold	Range	Value
1	var_1	number	<input checked="" type="checkbox"/>		50

The following example demonstrates the scope and the global hold function of controller-level global variables.

```

...
Project 1 (main.py file)
Assuming that two global variables, g1 and g2, have been added to the global variable page:
g1 is non-global hold variable, with a value of 10.
g2 is global hold variable, with a value of 20.
...

a = 1
print(a)      --> 1

print(g1)      --> 10
print(g2)      --> 20
g1=11 # Assigning a new value to the non-global hold variable
g2=22 # Assigning a new value to the global hold variable
print(g1)      --> 11
print(g2)      --> 22

...
Project 2 (main.py file)
Project 2 runs after Project 1.
...

print(a)      # Variable does not exist
print(b)      # Variable does not exist
print(g1)      # 10 (The non-global hold variable has reverted to its original value set before Project 1)
print(g2)      # 22 (The global hold variable becomes the value changed in Project 1)

```

Python supports a variety of data types, and the ones commonly used by Dobot in designing its API include Number, Boolean, String, List and Dictionary.

## Number

The number in Python support int (long integer), float (double-precision floating-point number), bool (boolean), and complex (complex number).

```

var1 = 1
var2 = 10

```

## Boolean

The boolean type has only two optional values: True and False. Python treats number 1 as true, and number 0 as false, which can be used for numerical operations. Additionally, any non-zero number or non-empty data types like string, list, and dictionary are considered True, while 0, empty string, empty list, and empty dictionary are considered False.

```

print(2 < 3)  # True
print(2 == 3) # False

a = True

```

```
b = False
print(a and b) # False
print(a or b) # True
print(not a) # False
```

## String

In Python, strings are enclosed in either single quotes ('') or double quotes ("") and can be concatenated using the plus sign (+).

```
str1 = 'Dobot'
str2 = " Robot"
print(str1 + str2) # Dobot Robot
```

Strings can be accessed using index positions, with the index starting at 0 from the beginning and -1 from the end. The syntax for slicing a string is `variable[start_index:end_index]`. Strings in Python are immutable, meaning you cannot modify an individual character by indexing it. However, you can assign a new string to the variable.

```
str1 = 'Dobot'
print(str1[0]) # D
print(str1[-1]) # t
print(str1[0:2]) # Dob
```

## List

A list is a collection of elements written between square brackets ([ ]), separated by commas (,). Like strings, lists can be concatenated using the plus sign (+) and accessed using index positions. However, unlike strings, lists are mutable, meaning you can modify an individual element by indexing it.

```
list1 = [1, 2, 3]
list2 = ["a", "b", "c"]
print(list1 + list2) # [1, 2, 3, "a", "b", "c"]
print(list1[0]) # 1
list1[0] = "a"
print(list1) # ["a", 2, 3]
del list[2] # Delete the third element in the list
```

## Dictionary

A dictionary is a collection of "key: value" written between curly brackets ({}), separated by commas (,). A dictionary is indexed by key, so key must be unique. A dictionary can be added, deleted, or modified using the key.

```
dict1 = {"user": 1, "tool": 0, "a": 20, "v": 50, "cp": 100}
print(dict1["user"]) # 1
dict1["tool"] = 1 # Modify the value of tool in the dictionary to 1
```

```

dict1["r"] = 5          # Add a new key-value pair to the dictionary if the key doesn't exist

del dict1["r"]          # Delete the specified key-value pair

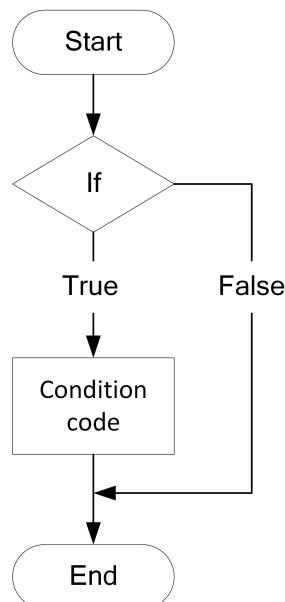
```



## Flow control

### Conditional statement (if)

The `if` statement allows for conditional execution of code based on whether a given condition is true or false.



The standard form of an `if` statement in Python is as follows, where `elif` and `else` are optional:

```

if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3

```

- If "condition\_1" is True, the "statement\_block\_1" will be executed.
- If "condition\_1" is False, the "condition\_2" will be checked.
- If "condition\_2" is True, the "statement\_block\_2" will be executed.
- If "condition\_2" is False, the "statement\_block\_3" will be executed.

The `if` statement can be nested, the following is a typical example.

```

a = 100;
b = 200;

```

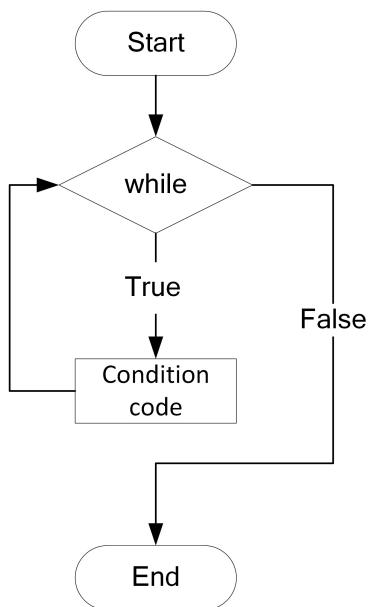
```

# Check conditions
if a == 100:
    # Execute the following code if the condition is True
    if b == 200:
        # Execute the statement block if the condition is True
        print("value of a:", a) # value of a: 100
        print("value of b:", b) # value of b: 200
else:
    # Execute the following statement block if the first condition is True
    print("a≠100")

```

## Loop statement (while)

The `while` statement can execute code blocks repeatedly based on conditions.



The standard form of `while` statement in Python is as follows:

```

while condition:
    statement_block

```

- If "condition" is True, the "statement\_block" will be executed, and then "condition" will be checked again.
- If "condition" is False, the "statement\_block" will be skipped and subsequent statements will be executed directly.

Example:

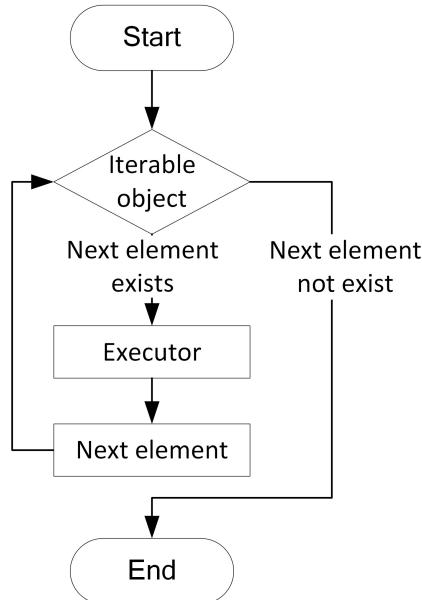
```

a=10
while a < 20
    print("value of a:", a) # Execute 10 times, the output value is 10 to 19
    a = a+1

```

## Loop statement (for)

The `for` statement is used to iterate over a sequence, such as a list or a string, and execute the code block for each element in the sequence.



The standard form of `for` statement in Python is as follows:

```
for variable in sequence:  
    statement_block
```

The sequence is an iterable object, and variable is a variable used to traverse the iterable object.

1. The system assigns the first element in "sequence" to "variable", and then executes the "statement\_block" once.
2. If there is a next element in "sequence", the system assigns the next element to "variable" and executes the "statement\_block" again.
3. Repeat the above steps until all elements in "sequence" have been iterated over.

Example:

```
joint = [j1,j2,j3,j4,j5,j6]  
for angles in joint:  
    print(angles) # Execute 6 times and output the values of j1 to j6
```

# General description

## Motion mode

The robot supports the following motion modes:

### Joint motion

The robot plans the motion of each joint based on the difference between the current and target joint angles, ensuring that all joints complete their motion simultaneously. Joint motion does not constrain the TCP (Tool Center Point) trajectory, and the path is usually not a straight line.



Joint motion is not restricted by singularities (refer to the corresponding hardware guide for details). If there are no specific trajectory requirements or the target point is near a singularity, joint motion is recommended.

### Linear motion

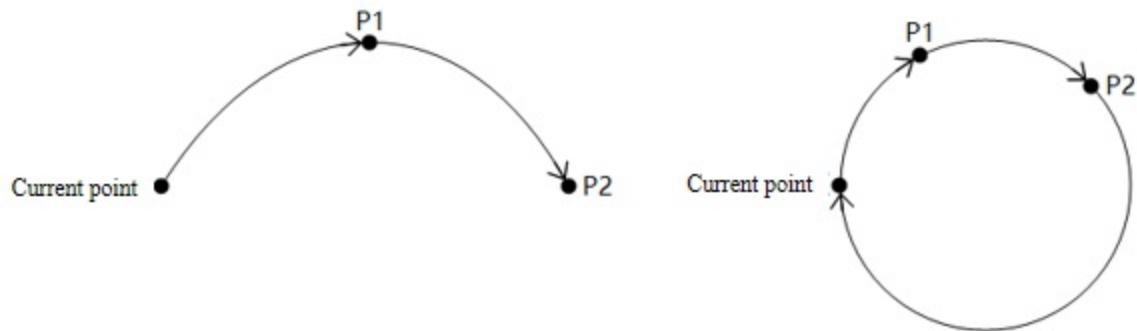
The robot plans the motion trajectory based on the current posture and the target point's posture, ensuring that the TCP moves in a straight line and the end posture changes uniformly during the motion.



When the trajectory passes through a singularity, issuing a linear motion command will result in an error. It is recommended to re-plan the point or use joint motion near the singularity.

### Arc motion

The robot determines an arc or a circle based on three non-collinear points: the current position, P1, and P2. During the motion, the end posture of the robot is interpolated between the current point and the posture at P2, while the posture at P1 is not considered (i.e., when the robot reaches P1 during the motion, its posture may differ from the taught posture).



When the trajectory passes through a singularity, issuing an arc motion command will result in an error. It is recommended to re-plan the point or use joint motion near the singularity.

## Point parameters

All point parameters in this document support three expressions unless otherwise specified.

- Joint variable: describe the target point using the angle of each joint ( $j_1 - j_6$ ) of the robot.

When the joint variable is used as a linear or arc motion parameter, the system will convert it into a posture variable through a positive solution, but the algorithm will ensure that the joint angle of the robot arm when it reaches the target point will be consistent with the set value.

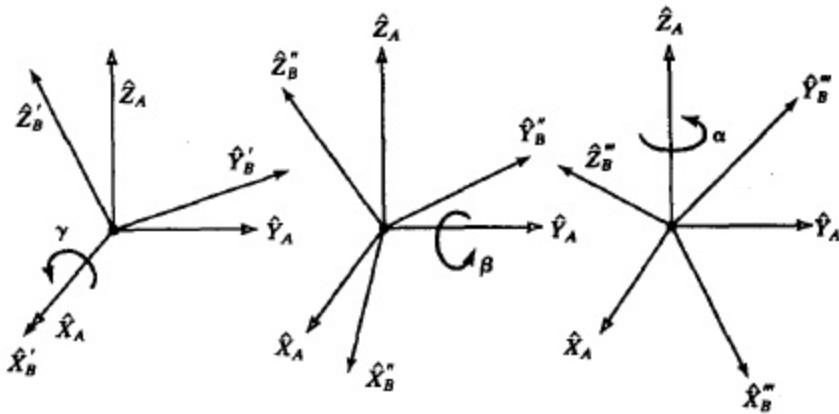
```
{"joint" : [j1, j2, j3, j4, j5, j6]}
```

- Posture variable: describe the spatial position of the target point in the user coordinate system using Cartesian coordinates ( $x, y, z$ ), and describe the rotation angle of the tool coordinate system relative to the user coordinate system when the TCP (Tool Center Point) reaches a specified point using Euler angles ( $rx, ry, rz$ ).

When the posture variable is used as a point parameter of the joint motion, the system will convert it into a joint variable (the solution closest to the current joint angle of the robot arm) through the inverse solution.

```
{"pose" : [x, y, z, rx, ry, rz]}
```

The rotation order when calculating the Euler angle of Dobot robot is X->Y->Z, and each axis rotates around a fixed axis (user coordinate system), as shown below ( $rx=\gamma, ry=\beta, rz=\alpha$ ).



Once the rotation order is determined, the rotation matrix (where  $\cos\alpha$  stands for  $\cos\alpha$ ,  $\sin\alpha$  stands for  $\sin\alpha$ , and so on)

$$\begin{aligned} {}_B^A R_{XYZ}(\gamma, \beta, \alpha) &= R_z(\alpha) R_y(\beta) R_x(\gamma) \\ &= \begin{bmatrix} c\alpha & -s\alpha & 0 \\ s\alpha & c\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c\beta & 0 & s\beta \\ 0 & 1 & 0 \\ -s\beta & 0 & c\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\gamma & -s\gamma \\ 0 & s\gamma & c\gamma \end{bmatrix} \end{aligned}$$

can be derived as the equation

$${}^A_B R_{XYZ}(\gamma, \beta, \alpha) = \begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma \\ s\alpha c\beta & s\alpha s\beta s\gamma + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma \\ -s\beta & c\beta s\gamma & c\beta c\gamma \end{bmatrix}$$

The posture of the end of robot arm can be calculated through the equation.

- Teaching point: the points obtained through hand-guiding using the software are saved as constants in the following format.

```
...
name: name of teaching point.
joint: joint coordinates of teaching point.
tool: tool coordinate system index in hand guiding.
user: user coordinate system index in hand guiding.
pose: posture variable value of teaching point
...
{
  "name" : "name",
  "tool" : index,
  "user" : index,
  "joint" : [j1, j2, j3, j4, j5, j6],
  "pose" : [x, y, z, rx, ry, rz]
}
```

## Coordinate system parameters

The "user" and "tool" in the optional parameters are used to specify the user and tool coordinate systems for the target point. Now the coordinate system can be specified only through the index, and the corresponding coordinate system needs to be added in the software first.

Priority of selecting the coordinate system:

1. When a coordinate system is specified in the optional parameters, the specified coordinate system is used. If the point parameter is a taught point, the posture coordinates of the taught point are converted into the values of the specified coordinate system.
2. If no coordinate system is specified in optional parameters, and the point is a taught point, the system uses the coordinate system index attached to the taught point.
3. If no coordinate system is specified in optional parameters, and the point is a joint variable or posture variable, the global coordinate system set in the motion parameters is used (see **User** and **Tool** commands for details. The default coordinate system is 0 when no command is called).

**i** NOTE

- When a script starts running, the default global coordinate system will be set to 0, regardless of the value set in the Jog panel before running the script.
- When the joint motion command (MovJ/MovJIO/RelMovJTool/RelMovJUser) is called, if the point is a joint variable, the coordinate system parameter is invalid.

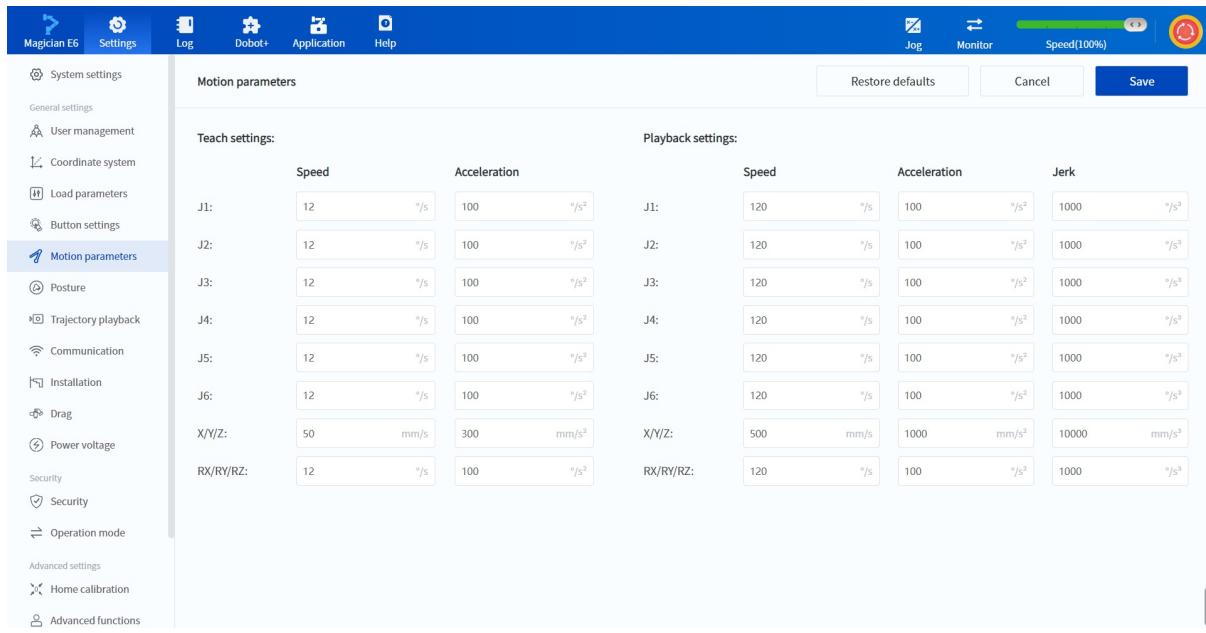
## Speed parameters

### Relative speed

The “a” and “v” in the optional parameters are used to specify the acceleration and speed rate when the robot arm executes motion commands.

```
Robot actual speed = Maximum speed x Global speed x Command speed  
Robot actual acceleration = Maximum acceleration x Command speed
```

The maximum speed/acceleration is controlled by **Playback settings**, which can be viewed and modified on the "Motion parameters" page of DobotStudio Pro.



The global speed can be adjusted via DobotStudio Pro (speed slider at the top right of the figure above) or the **SpeedFactor** command.

The command rate is carried by the optional parameters of the motion commands. When the acceleration/speed rate is not specified through the optional parameters, the value set in the motion parameters is used by default (see VelJ, AccJ, VelL, AccL commands for details, and the default value is 100 when the command setting is not called).

Example:

```

AccJ(50) # Set the default acceleration of joint motion to 50%
VelJ(60) # Set the default speed of joint motion to 60%
AccL(70) # Set the default acceleration of linear motion to 70%
VelL(80) # Set the default speed of linear motion to 80%

# global speed rate: 20;

MovJ(P1) # Move to P1 at the acceleration of (maximum joint acceleration x 50%) and speed of ( maximum joint speed x 20% x 60%) through the joint motion
MovJ(P2,{"a":30, "v":80}) # Move to P1 at the acceleration of (maximum joint acceleration x 30 %) and speed of (maximum joint speed x 20% x 80%) through the joint motion

MovL(P1) # Move to P1 at the acceleration of (maximum Cartesian acceleration x 70%) and speed of (maximum Cartesian speed x 20% x 80%) in the linear mode
MovL(P1,{"a":40, "v":90}) # Move to P1 at the acceleration of (maximum Cartesian acceleration x 40%) and speed of (maximum Cartesian speed x 20% x 90%) in the linear mode

```

## Absolute speed

The “speed” in the optional parameter of linear and arc motion commands is used to specify the absolute speed when the robot executes the command.

The absolute speed is not affected by the global speed, but limited by the maximum speed in **Playback settings** (or the maximum speed after reduction if the robot is in reduced mode), i.e. if the target speed set by the speed parameter is greater than the maximum speed in Playback settings, then the maximum speed takes precedence.

Example:

```
MovL(P1,{"speed":1000}) # Move to P1 in the linear mode at a absolute speed of 1000
```

If the speed set in MovL is 1000 (less than the maximum speed of 2000 in Playback settings), the robot will move at a target speed of 1000 mm/s, which is independent of the global speed at this point. However, if the robot is in reduced mode (with a reduction rate of 10%), the maximum speed becomes 200 mm/s, which is lower than 1000 mm/s, so the robot will move at a target speed of 200 mm/s.

The “speed” and “v” cannot be set at the same time. If both exist, “speed” takes precedence.

## Continuous path parameters

When the robot arm moves through multiple points continuously, it can pass through the intermediate point through a smooth transition so the robot arm will not turn too bluntly. Smooth transitions cannot be applied if the user-specified path points are based on different tool coordinate systems.

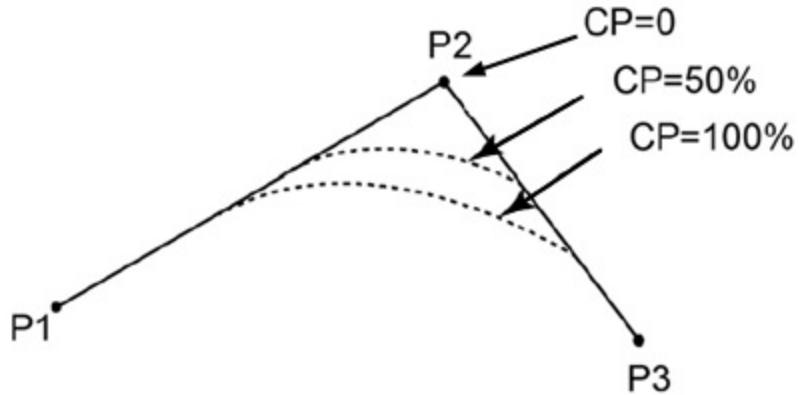
The “cp” or “r” in the optional parameters are used to specify the continuous path rate (cp) or continuous path radius (r) between the current and the next motion commands. The two parameters are mutually exclusive. If both exist, “r” takes precedence.

### NOTE

Joint motion related commands do not support setting the continuous path radius (r). See the optional parameters of each command for details.

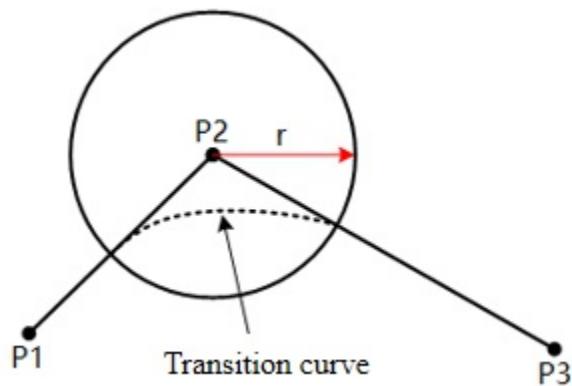
## CP

When setting the CP, the system automatically calculates the curvature of the transition curve. The larger the CP value, the smoother the curve, as shown in the figure below. The CP transition curve is affected by speed and acceleration. Even if the path points and CP values are the same, different speeds/accelerations will result in different curvatures.

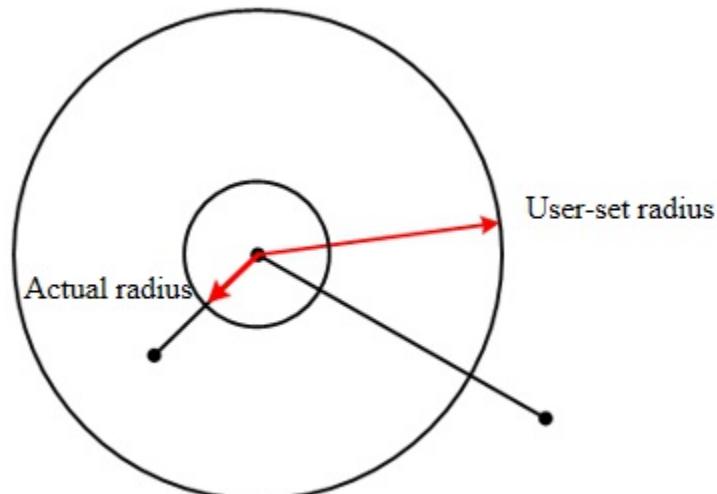


## R

When setting the R, the system calculates the transition curve using the transition point as the center and the specified radius. The R transition curve is not affected by speed or acceleration, but is determined by the path points and the specified radius.



If the continuous path radius is set too large (more than the distance between the start/end point and the intermediate point), the system will automatically calculate the transition curve using half of the shorter distance between the start/end point and the transition point as the continuous path radius.



When the continuous path ratio and radius are not specified in the optional parameters, the continuous path ratio set in the motion parameter is used by default (See CP command for details. The default value is 0 when no command is called).

### NOTE

As CP causes the robot to bypass intermediate points, if CP is set, any IO signal outputs or function settings (e.g., enabling/disabling SafeSkin) between two motion commands will be executed during the transition.

If you want the robot to execute commands precisely when it reaches the intermediate point, set the CP parameters of the previous command to 0.

Due to the different implementations of CP and R, inserting other commands that do not affect motion (such as conditional checks, I/O, or functional settings) between two motion commands requiring smooth transitions will result in differing handling for CP and R.

- If CP mode is used, most commands will not affect the transition unless the processing time of the command is too long (e.g., the Wait command).

In the example below, inserting an if statement between two motion commands will not affect the smooth transition in CP mode.

```
# It can transition normally. The robot will judge DI1 when it is about to reach P1. If it is ON, it will transition to the next motion command with a continuous path rate of 50%.
MovL(P1,{"cp":50})
if DI(1)==ON:
    MovL(P2)
```

- If “r” mode is used, only the following whitelisted commands will not affect the smooth transition. Inserting any other commands will invalidate the smooth transition in R mode.

```
RelPointTool, RelPointUser, DOGroup, DO, AO, ToolDO,
SetUser, SetTool, User, Tool, CP, AccJ, AccL, VelL, VelJ,
SetPayload, SetCollisionLevel, SetBackDistance, EnableSafeSkin, SetSafeSkin
```

In the example below, inserting an if statement between two motion commands will cause the smooth transition settings in R mode to be invalid.

```
# Invalid continuous path parameters. The robot will judge DI1 after reaching P1.
MovL(P1,{"r":5})
if DI(1)==ON:
    MovL(P2)
```

## IO signal

The ON and OFF variables are predefined inside the system to indicate whether there is a signal.

- ON = 1: there is a signal
- OFF = 0: there is no signal

ON|OFF in the parameter means that the parameter value is ON or OFF. You can also use 1 or 0 as input.

# Motion

## Command list

The motion commands are used to control the movement of the robot. Please read [General description](#) before using the commands.

Command	Function
MovJ	Joint motion
MovL	Linear motion
Arc	Arc motion
Circle	Circle motion
MovJIO	Move in joint mode and output DO
MovLIO	Move in linear mode and output DO
StartPath	Play back recorded trajectory
GetPathStartPose	Get start point of trajectory
PositiveKin	Forward kinematics (joint angles to posture)
InverseKin	Inverse kinematics (posture to joint angles)

## MovJ

### Command:

```
MovJ(point, {"user":1, "tool":0, "a":20, "v":50, "cp":100})
```

### Description:

Move from the current position to the target position through joint motion.

### Required parameter:

**point:** Target point.

### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].

- **cp:** The ratio for smooth transition. Range: [0,100].

See [General description](#) for details.

### Example:

```
# The robot moves to P1 through joint motion with the default settings.
MovJ(P1)
```

```
# The robot moves to the specified joint angle through joint motion with the default settings.
MovJ({"joint": [0,0,90,0,90,0]})
```

```
# The robot moves to the specified posture through joint motion, which corresponds to User coordinate system 1 and Tool coordinate system 1, with an acceleration and velocity of 50% and a CP ratio of 50%.
MovJ({"pose": [300,200,300,180,0,0]}, {"user": 1, "tool": 1, "a": 50, "v": 50, "cp": 50})
```

```
# Define the point and then call it in a motion command, achieving the same effect as the previous command.
customPoint={"pose": [300,200,300,180,0,0]}
MovJ(customPoint, {"user": 1, "tool": 1, "a": 50, "v": 50, "cp": 50})
```

## MovL

### Command:

```
MovL(point, {"user": 1, "tool": 0, "a": 20, "v": 50, "speed": 500, "cp": 100, "r": 5})
```

### Description:

Move from the current position to the target position in a linear mode.

### Required parameter:

**point:** Target point.

### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].

- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the **cp** parameter will be ignored.

See [General description](#) for details.

### Example:

```
# The robot moves to P1 in a linear mode with the default settings.
MovL(P1)
```

```
# The robot moves to P1 in a linear mode at an absolute speed of 500m/s.
MovL(P1,{"speed":500})
```

```
# The robot moves to the specified joint angle in a linear mode with the default settings.
MovL({"joint":[0,0,90,0,90,0]})
```

```
# The robot moves in a linear mode to the specified posture, which corresponds to User coordinate system 1 and Tool coordinate system 1, with an acceleration and speed of 50% and a CP ratio of 5 mm.
MovL({"pose":[300,200,300,180,0,0]}, {"user":1, "tool":1, "a":50, "v":50, "r":5})
```

```
# Define the point and then call it in a motion command, achieving the same effect as the previous command.
customPoint={"pose":[300,200,300,180,0,0]}
MovL(customPoint, {"user":1, "tool":1, "a":50, "v":50, "r":5})
```

```
# If both "speed" and "v" are specified, "speed" takes effect, and the controller will log a corresponding warning.
# If both "cp" and "r" are specified, "r" takes effect, and the controller will log a corresponding warning.
MovL(P1, {"v":50, "speed":500, "cp":60, "r":5}) # When executed, only the optional parameters "speed" and "r" take effect.
```

## Arc

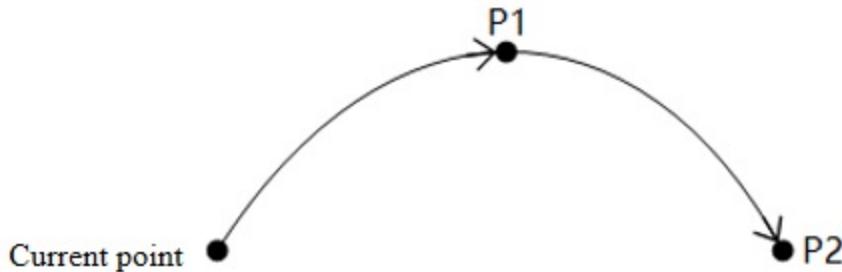
### Command:

```
Arc(P1, P2, {"user":1, "tool":0, "a":20, "v":50, "speed":500, "cp":100, "r":5})
```

### Description:

Move from the current position to the target position in an arc interpolated mode.

As the arc needs to be determined through the current position, P1 and P2, the current position should not be in a straight line determined by P1 and P2.



During the motion, the end posture of the robot is interpolated between the current point and the posture at P2, while the posture at P1 is not considered (i.e., when the robot reaches P1 during the motion, its posture may differ from the taught posture).

### Required parameter:

- **P1:** Intermediate point of the arc.
- **P2:** Target point.

### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the v parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the cp parameter will be ignored.

See [General description](#) for details.

### Example:

```
# The robot moves to P1, and then moves to P3 via P2 in an arc interpolated mode.
MovJ(P1)
Arc(P2,P3)
```

```
# The robot moves to P1, then moves to P3 via [300,200,300,180,0,0], with both User and Tool coordinate systems set to 1, and acceleration and velocity set to 50%.
MovJ(P1)
Arc({pose:[300,200,300,180,0,0]},P3,{"user":1, "tool":1, "a":50, "v":50})
```

## Circle

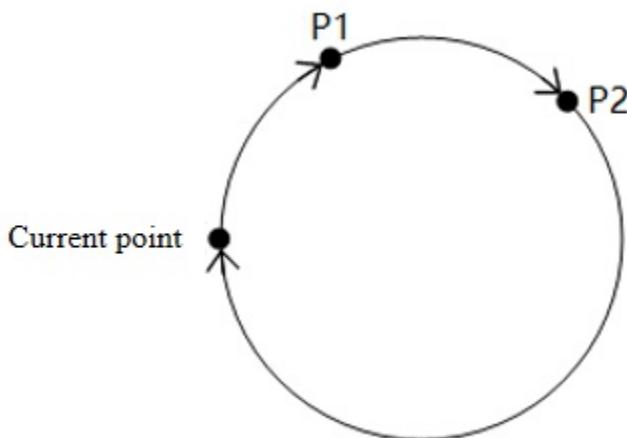
### Command:

```
Circle(P1, P2, Count, {"user":1, "tool":0, "a":20, "v":50, "speed":500, "cp":100, "r":5})
```

### Description:

Move from the current position in a circle interpolated mode, and return to the current position after moving specified circles.

As the circle needs to be determined through the current position, P1 and P2, the current position should not be in a straight line determined by P1 and P2, and the circle determined by the three points cannot exceed the motion range of the robot.



During the motion, the end posture of the robot is interpolated between the current point and the posture at P2, while the posture at P1 is not considered (i.e., when the robot reaches P1 during the motion, its posture may differ from the taught posture).

### Required parameter:

- **P1:** Point 1 in circle motion.
- **P2:** Point 2 in circle motion.
- **Count:** Number of circles to perform, range: [1 – 999].

### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the **cp** parameter will be ignored.

See [General description](#) for details.

### Example:

```
# The robot moves to P1, then moves a full circle defined by P1, P2 and P3.  
MovJ(P1)  
Circle(P2,P3,1)
```

```
# The robot moves to P1, then moves 10 times along the circle defined by P1, [300,200,300,180,  
0,0] and P3, with both User and Tool coordinate systems set to 1, and acceleration and velocit  
y set to 50%.  
MovJ(P1)  
Circle({pose:[300,200,300,180,0,0]},P3,10,{"user":1, "tool":1, "a":50, "v":50})
```

## MovJIO

### Command:

```
MovJIO(P,[[Mode,Distance,Index,Status],[Mode,Distance,Index,Status]...], {"user":1, "tool":0,  
"a":20, "v":50, "cp":100})
```

### Description:

Move from the current position to the target position through joint motion, while simultaneously setting the status of digital output port.

### Required parameter:

- **P:** The target point.
- **Digital output parameters:** Set the specified DO to be triggered when the robot reaches a specified distance or percentage. Multiple groups can be set, each containing the following parameters:
  - **Mode:** Trigger mode. **0:** percentage trigger. **1:** distance trigger. The system will synthesize the joint angles into an angular vector and calculate the angle difference between the end point and the start point as the total distance of the motion.
  - **Distance:** Specified percentage/angle. It is recommended to use the percentage mode for a more intuitive effect, as angle calculations rely on the synthesized angle vector.
    - If Distance is positive, it refers to the percentage/angle from the start point.
    - If Distance is negative, it refers to the percentage/angle from the target point.
    - If Mode is 0, Distance refers to the percentage of total angle. Range: (0,100].
    - If Mode is 1, Distance refers to the angle value. Unit: °.
  - **Index:** DO index.
  - **Status:** DO status. 0/OFF: no signal; 1/ON: an active signal.

#### Optional parameter:

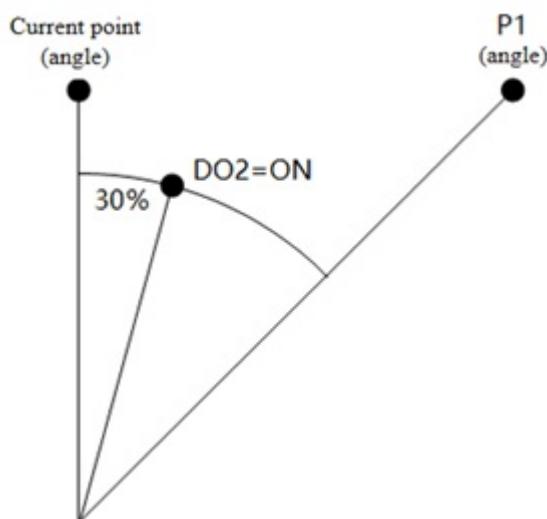
- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **cp:** The ratio for smooth transition. Range: [0,100].

CP will alter the robot's motion trajectory, affecting the timing of DO outputs; please use it with caution.

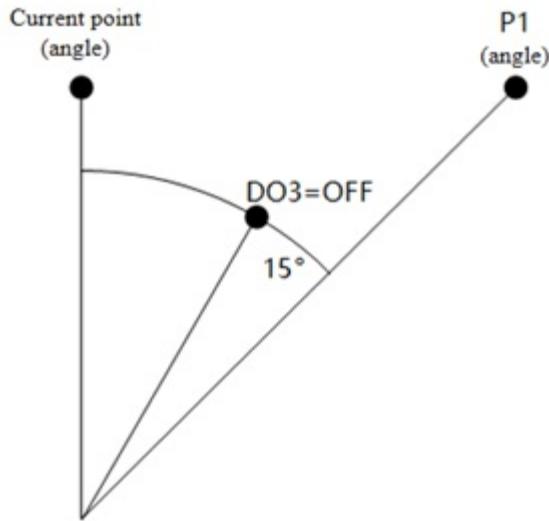
See [General description](#) for details.

#### Example:

```
# The robot moves to P1 through joint motion with the default settings. When it reaches 30% of
# the distance from the start point, DO2 is set to ON.
MovJIO(P1, [[0, 30, 2, 1]])
```



```
# The robot moves to P1 through joint motion with the default settings. When it is 15° away from
# the target point, DO3 is set to OFF.
MovJIO(P1,[[1, -15, 3, 0]])
```



## MovLIO

### Command:

```
MovLIO(P, [[Mode,Distance,Index>Status],[Mode,Distance,Index>Status]...],{"user":1, "tool":0, "a":20, "v":50, "speed":500, "cp":100, "r":5})
```

### Description:

Move from the current position to the target position in a linear mode, while simultaneously setting the status of digital output port.

### Required parameter:

- **P:** The target point.
- **Digital output parameters:** Set the specified DO to be triggered when the robot reaches a specified distance or percentage. Multiple groups can be set, each containing the following parameters:
  - **Mode:** Trigger mode. **0:** percentage trigger. **1:** distance trigger.
  - **Distance:** Specified percentage/distance.
    - If Distance is positive, it refers to the percentage/distance from the start point.
    - If Distance is negative, it refers to the percentage/distance from the target point.
    - If Mode is 0, Distance refers to the percentage of total distance. Range: (0,100].
    - If Mode is 1, Distance refers to the distance value. Unit: mm.
  - **Index:** DO index.
  - **Status:** DO status. 0/OFF: no signal; 1/ON: an active signal.

### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

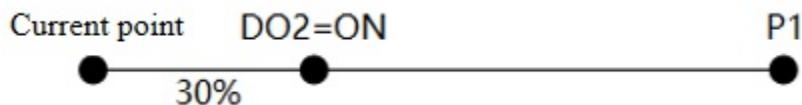
If this parameter is set, the **cp** parameter will be ignored.

CP will alter the robot's motion trajectory, affecting the timing of DO outputs; please use it with caution.

See [General description](#) for details.

### Example:

```
# The robot moves to P1 in a linear mode with the default settings. When it reaches 30% of the
# distance from the start point, DO2 is set to ON.
MovLIO(P1, [[0, 30, 2, 1]])
```



```
# The robot moves to P1 in a linear mode with the default settings. When it is 15 mm away from
# the target point, DO3 is set to OFF.
MovLIO(P1, [[1, -15, 3, 0]])
```



## StartPath

### Command:

```
StartPath(string, {"multi":1, "isConst":0, "sample":50, "freq":0.2, "user":0, "tool":0})
```

### Description:

Play back the recorded trajectory in the specified trajectory file. Before calling this command, you need to manually move the robot to the start point of the trajectory.

**Required parameter:**

**string:** Trajectory file name (including suffix).

**Optional parameter:**

- **multi:** Speed multiplier for the playback, valid only when **isConst=0**. Range: [0.1, 2], 1 by default.
- **isConst:** Whether to play back at a constant speed. 0 by default.
  - 1 means constant speed playback, where the robot will play back the trajectory at a constant speed.
  - 0 means playback at the original recorded speed, with the option to scale the speed proportionally using the "multi" parameter. In this case, the robot's speed is not affected by the global speed setting.
- **sample:** The sampling interval of the trajectory points, i.e. the sampling time differences between two adjacent points when generating the trajectory file. Range: [8, 1000], unit: ms, 50 by default (the sampling interval when the controller records the trajectory file).
- **freq:** Filter coefficient. The smaller the value, the smoother the trajectory curve during playback, but the greater the distortion relative to the original trajectory. Set an appropriate filter coefficient based on the smoothness of the original trajectory. Range: (0,1], 1 refers to turning off filtering, 0.2 by default.
- **user:** Specify the user coordinate system index for the trajectory points. If not specified, the user coordinate system index recorded in the trajectory file is used.
- **tool:** Specify the tool coordinate system index for the trajectory points. If not specified, the tool coordinate system index recorded in the trajectory file is used.

**Example:**

```
# After moving to the start point of the "track.csv" file, the robot plays back the recorded trajectory at 2x the original speed.
StartPoint = GetPathStartPose("track.csv")
MovJ(StartPoint)
StartPath("track.csv", {"multi":2, "isConst":0})
```

```
# After moving to the start point of the "track.csv" file, the robot plays back the recorded trajectory at a constant speed.
StartPoint = GetPathStartPose("track.csv")
MovJ(StartPoint)
StartPath("track.csv", {"isConst":1})
```

## GetPathStartPose

**Command:**

```
GetPathStartPose(string)
```

#### Description:

Get the start point of the trajectory. The type of point data (teaching point/joint/posture) may be different according to the trajectory files.

#### Required parameter:

**string:** Trajectory file name (including suffix).

#### Example:

```
# Get the start point of the "track.csv" file and print it.  
StartPoint = GetPathStartPose("track.csv")  
print(StartPoint)
```

## PositiveKin

#### Command

```
PositiveKin(joint, {"user":1, "tool":0})
```

#### Description

Perform forward kinematics. Calculate the coordinates of the end of the robot in the specified Cartesian coordinate system, based on the given angle of each joint.

#### Required parameter

**joint:** Joint variables, format: `{"joint": [j1, j2, j3, j4, j5, j6]}`.

#### Optional parameter

- **user:** User coordinate system index. The global user coordinate system will be used when it is not specified.
- **tool:** Tool coordinate system index. The global tool coordinate system will be used when it is not specified.

#### Return

The calculated posture variables through forward kinematics, format: `{"pose": [x, y, z, rx, ry, rz]}`.

#### Example

```
PositiveKin({"joint": [0,0,-90,0,90,0]}, {"user":1, "tool":1})
```

Calculate the coordinates of the end of the robot in the User coordinate system 1 and Joint coordinate system 1, based on the joint coordinates `[0,0,-90,0,90,0]` .

## InverseKin

### Command

```
InverseKin(pose, {"useJointNear":True, "jointNear":joint, "user":1, "tool":0})
```

### Description

Perform inverse kinematics. Calculate the joint angles of the robot, based on the given coordinates in the specified Cartesian coordinate system.

Since Cartesian coordinates only define the spatial coordinates and tilt angle of the TCP, the robot can reach the same pose with different configurations. This means that one pose can have multiple joint configurations. To get a unique solution, the system requires a specified joint coordinate, and the solution closest to this joint coordinate is selected as the result.

### Required parameter

**pose:** Posture variables, format: `{"pose":[x, y, z, rx, ry, rz]}` .

### Optional parameter

- **useJointNear:** Boolean value specify whether jointNear is used.
  - **True:** The algorithm selects the joint angles according to JointNear data.
  - **False or null:** JointNear data is ineffective. The algorithm selects the joint angles according to the current angle.
  - If only this parameter is specified without jointNear, it will be ineffective.
- **jointNear:** Joint variables for selecting joint angles, format: `{"joint":[j1, j2, j3, j4, j5, j6]}` .
- **user:** User coordinate system index. The global user coordinate system will be used when it is not specified.
- **tool:** Tool coordinate system index. The global tool coordinate system will be used when it is not specified.

### Return

- Error code: 0 indicates the inverse kinematics is successful, -1 indicates failure (no solution).
- The calculated joint variables through inverse kinematics, format: `{"joint":[j1, j2, j3, j4, j5, j6]}` . If the inverse kinematics fails, all values for J1 to J6 will be 0.

### Example

```
errId, jointPoint = InverseKin({"pose":[300, 200, 300, 180, 0, 0]}, {"useJointNear" :True, "jointNear":{"joint":[90, 30, -90, 180, 30, 0]} })
```

The Cartesian coordinates of the end of the robot in the global user coordinate system and global joint coordinate system are [300, 200, 300, 180, 0, 0] . Joint coordinates will be calculated, and the solution closest to the joint angles [90, 30, -90, 180, 30, 0] will be selected.

# Relative motion

## Command list

The relative motion commands are used to control the robot for offset movements. Please read [General description](#) before using the commands.

Command	Function
<a href="#">RelPointUser</a>	Offset the point along the user coordinate system
<a href="#">RelPointTool</a>	Offset the point along the tool coordinate system
<a href="#">RelMovJTool</a>	Perform relative joint motion along the tool coordinate system
<a href="#">RelMovLTool</a>	Perform relative linear motion along the tool coordinate system
<a href="#">RelMovJUser</a>	Perform relative joint motion along the user coordinate system
<a href="#">RelMovLUser</a>	Perform relative linear motion along the user coordinate system
<a href="#">RelJointMovJ</a>	Perform joint motion to the specified offset angle
<a href="#">RelJoint</a>	Offset the point by a specified joint angle

## RelPointUser

### Command:

```
RelPointUser(P, [OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz])
```

### Description:

Offset the specified point along the given user coordinate system and return the offset point.

### Required parameter:

- **P:** The point to be offset.
- **[OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz]:** The offset values in the specified user coordinate system. **x, y, z:** Spatial offset values in millimeters (mm). **rx, ry, rz:** Angular offset values in degrees (°).
  - If the specified point is a taught point, the offset is based on the user coordinate system of the taught point.
  - If the specified point is a joint variable or a posture variable, the offset is based on the [global user coordinate system](#).

### Return:

- If the specified point is a taught point, return the point constant after offset.
- If the specified point is a joint variable or a posture variable, return the posture variable after offset:  
`{pose = {x, y, z, rx, ry, rz}} .`

**Example:**

```
# Offset P1 by a certain distance in the specified user coordinate system, then move to the offset point.
Offset=[OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz]
p = RelPointUser(P1, Offset)
MovL(p)
```

## RelPointTool

**Command:**

```
RelPointTool(P, [OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz])
```

**Description:**

Offset the specified point along the given tool coordinate system and return the offset point.

**Required parameter:**

- **P:** The point to be offset.
- **[OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz]:** The offset values in the specified tool coordinate system. **x, y, z:** Spatial offset values in millimeters (mm). **rx, ry, rz:** Angular offset values in degrees (°).
  - If the specified point is a taught point, the offset is based on the tool coordinate system of the taught point.
  - If the specified point is a joint variable or a posture variable, the offset is based on the [global tool coordinate system](Motion Params.html#tool).

**Return:**

- If the specified point is a taught point, return the point constant after offset.
- If the specified point is a joint variable or a posture variable, return the posture variable after offset:  
`{pose = {x, y, z, rx, ry, rz}} .`

**Example:**

```
# Offset P1 by a certain distance in the specified tool coordinate system, then move to the offset point.
Offset=[OffsetX, OffsetY, OffsetZ, OffsetRx, OffsetRy, OffsetRz]
p = RelPointTool(P1, Offset)
MovL(p)
```

## RelMovJTool

### Command:

```
RelMovJTool([x, y, z, rx, ry, rz], {"user":1, "tool":0, "a":20, "v":50, "cp":100})
```

### Description:

Perform relative joint motion from the current position along the specified tool coordinate system. The trajectory is not a straight line, and all joints will move simultaneously.

### Required parameter:

[**x, y, z, rx, ry, rz**]: The offset values relative to the current position in the specified tool coordinate system. **x, y, z**: Spatial offset values in millimeters (mm). **rx, ry, rz**: Angular offset values in degrees (°).

### Optional parameter:

- **user**: The user coordinate system for the target point.
- **tool**: The tool coordinate system for the target point.
- **a**: The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v**: The velocity ratio for the robot when executing this command. Range: (0,100].
- **cp**: The ratio for smooth transition. Range: [0,100].

See [General description](#) for details.

### Example:

```
# The robot moves with default settings using joint motion along the global tool coordinate system to the specified offset point.  
RelMovJTool([10, 10, 10, 0, 0, 0])
```

## RelMovLTool

### Command:

```
RelMovLTool([x, y, z, rx, ry, rz], {"user":1, "tool":0, "a":20, "v":50, "speed":500, "cp":100, "r":5})
```



### Description:

Perform relative linear motion from the current position along the specified tool coordinate system.

### Required parameter:

[**x, y, z, rx, ry, rz**]: The offset values relative to the current position in the specified tool coordinate system. **x, y, z**: Spatial offset values in millimeters (mm). **rx, ry, rz**: Angular offset values in degrees (°).

### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the **cp** parameter will be ignored.

See [General description](#) for details.

#### Example:

```
# The robot moves with default settings using linear motion along the global tool coordinate system to the specified offset point.
RelMovLTool([10, 10, 10, 0, 0, 0])
```

## RelMovJUser

#### Command:

```
RelMovJUser([x, y, z, rx, ry, rz], {"user":1, "tool":0, "a":20, "v":50, "cp":100})
```

#### Description:

Perform relative joint motion from the current position along the specified user coordinate system. The trajectory is not a straight line, and all joints will move simultaneously.

#### Required parameter:

**[x, y, z, rx, ry, rz]:** The offset values relative to the current position in the specified user coordinate system. **x, y, z:** Spatial offset values in millimeters (mm). **rx, ry, rz:** Angular offset values in degrees (°).

#### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].

- **cp:** The ratio for smooth transition, incompatible with “r”. Range: [0,100].

See [General description](#) for details.

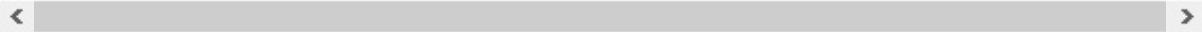
#### Example:

```
# The robot moves with default settings using joint motion along the global user coordinate system to the specified offset point.
RelMovJUser([10, 10, 10, 0, 0, 0])
```

## RelMovLUser

#### Command:

```
RelMovLUser([x, y, z, rx, ry, rz], {"user":1, "tool":0, "a":20, "v":50, "speed":500, "cp":100, "r":5})
```



#### Description:

Perform relative linear motion from the current position along the specified user coordinate system.

#### Required parameter:

**[x, y, z, rx, ry, rz]:** The offset values relative to the current position in the specified user coordinate system. **x, y, z:** Spatial offset values in millimeters (mm). **rx, ry, rz:** Angular offset values in degrees (°).

#### Optional parameter:

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the v parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the cp parameter will be ignored.

See [General description](#) for details.

#### Example:

```
# The robot moves with default settings using linear motion along the global user coordinate system to the specified offset point.  
RelMovLUser([10, 10, 10, 0, 0, 0])
```

## RelJointMovJ

### Command:

```
RelJointMovJ([Offset1, Offset2, Offset3, Offset4, Offset5, Offset6], {"a":20, "v":50, "cp":100})
```

### Description:

Move from the current position to the specified joint offset angles through joint motion.

### Required parameter:

**[Offset1, Offset2, Offset3, Offset4, Offset5, Offset6]:** The offset values for J1 to J6 in the joint coordinate system, unit: °.

### Optional parameter:

- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **cp:** The ratio for smooth transition. Range: [0,100].

See [General description](#) for details.

### Example:

```
# The robot moves to the offset angle through joint motion with the default settings.  
RelJointMovJ([20,20,10,0,10,0])
```

## RelJoint

### Command:

```
RelJoint(P, [Offset1, Offset2, Offset3, Offset4, offset5, offset6])
```

### Description:

Add offset to J1 to J6 of the specified point in the joint coordinate system, and return the joint variable after offset.

### Required parameter:

- **P:** The point to be offset.

- **Offset1 – Offset6:** The offset values for J1 to J6 in the joint coordinate system, unit: °.

**Return:**

Joint variable after offset: {"joint": [j1, j2, j3, j4, j5, j6]} .

**Example:**

```
# Offset P1 by a certain angle in J1 to J6, then move to the point after offset.  
Offset = [Offset1, Offset2, Offset3, Offset4, offset5, offset6]  
p = RelJoint(P1, Offset)  
MovJ(p)
```

# Motion parameters

## Command list

The motion parameters are used to set or obtain relevant motion parameters of the robot. Please read [General description](#) before using the commands.

Command	Function
CP	Set CP ratio
VelJ	Set the velocity ratio for joint motion
AccJ	Set the acceleration ratio for joint motion
VelL	Set the velocity ratio for linear and arc motion
AccL	Set the acceleration ratio for linear and arc motion
SpeedFactor	Set the global speed for the robot
SetPayload	Set the payload
User	Set the global user coordinate system
SetUser	Modify the specified user coordinate system
CalcUser	Calculate the user coordinate system
Tool	Set the global tool coordinate system
SetTool	Modify the specified tool coordinate system
CalcTool	Calculate the tool coordinate system
GetPose	Get the the robot's real-time posture
GetAngle	Get the real-time joint angle of the robot
GetABZ	Get the current position from the encoder
CheckMovJ	Check the feasibility of the joint motion command
CheckMovL	Check the feasibility of the linear or arc motion command
SetSafeWallEnable	Enable or disable the specified safety wall
SetWorkZoneEnable	Enable or disable the specified safety zone
SetCollisionLevel	Set collision detection level
SetBackDistance	Set collision backoff distance

## CP

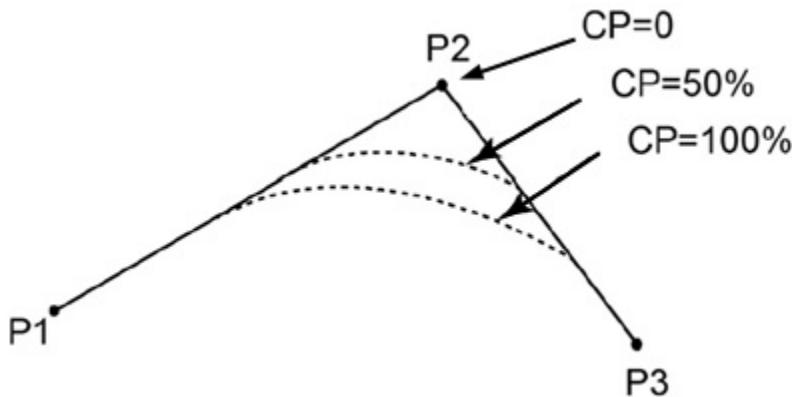
### Command:

```
CP(R)
```

### Description:

Set the CP (continuous path) ratio, which determines whether the robot transitions between multiple points with right angles or curved paths when moving continuously.

The CP ratio set in this command is only valid in the current project, and it is 0 by default when not set.



### Required parameter:

R: CP ratio. Range: [0, 100].

### Example:

```
# The robot moves from P1 to P3, passing through P2 with 50% CP ratio.  
CP(50)  
MovL(P1)  
MovL(P2)  
MovL(P3)
```

## VelJ

### Command:

```
VelJ(R)
```

### Description:

Set the velocity ratio for joint motion (MovJ/MovJIO/RelMovJTool/RelMovJUser/RelJointMovJ).

The velocity ratio set in this command is only valid in the current project, and it is 100 by default when not set.

### Required parameter:

**R:** Velocity ratio. Range: [1,100].

**Example:**

```
# The robot moves to P1 with 20% velocity ratio.  
VelJ(20)  
MovJ(P1)
```

## AccJ

**Command:**

```
AccJ(R)
```

**Description:**

Set the acceleration ratio for joint motion (MovJ/MovJIO/RelMovJTool/RelMovJUser/RelJointMovJ).

The acceleration ratio set in this command is only valid in the current project, and it is 100 by default when not set.

**Required parameter:**

**R:** Acceleration ratio. Range: [1,100].

**Example:**

```
# The robot moves to P1 with 50% acceleration ratio.  
AccJ(50)  
MovJ(P1)
```

## VelL

**Command:**

```
VelL(R)
```

**Description:**

Set the velocity ratio for linear and arc motion (MovL/Arc/Circle/MovLIO/RelMovLTool/RelMovLUser).

The velocity ratio set in this command is only valid in the current project, and it is 100 by default when not set.

**Required parameter:**

**R:** Velocity ratio. Range: [1,100].

### **Example:**

```
# The robot moves to P1 with 20% velocity ratio.  
VelL(20)  
MovL(P1)
```

## **AccL**

### **Command:**

```
AccL(R)
```

### **Description:**

Set the acceleration ratio for linear and arc motion  
(MovL/Arc/Circle/MovLIO/RelMovLTool/RelMovLUser).

The acceleration ratio set in this command is only valid in the current project, and it is 100 by default when not set.

### **Required parameter:**

**R:** Acceleration ratio. Range: [1,100].

### **Example:**

```
# The robot moves to P1 with 50% acceleration ratio.  
AccL(50)  
MovL(P1)
```

## **SpeedFactor**

### **Command:**

```
SpeedFactor(ratio)
```

### **Description:**

Set the global speed for the robot. See [General description](#) for details.

This global speed setting is only effective during the current project. If not set, the system will use the value previously set in the control software (if the script is run via software) or TCP commands (if the script is run through TCP).

### **Required parameter:**

**R:** The global speed ratio. Range: [1,100].

**Example:**

```
# Set the global speed to 50%.  
SpeedFactor(50)
```

## SetPayload

**Command:**

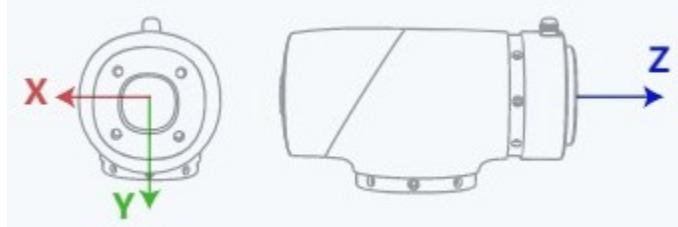
```
SetPayload(payload, [x, y, z])  
SetPayload(name)
```

**Description:**

Set the weight and offset coordinates of the end effector's payload. You can either specify these parameter directly or use preset parameter group. The parameters set by this command are only effective during the current project and will overwrite the previous settings. Once the project stops, the settings will revert to the original values.

**Required parameter 1:**

- **payload:** The weight of the end effector's payload. Unit: kg.
- **{x, y, z}:** The offset coordinates of the payload, see the figure below for axis directions. Unit: mm.

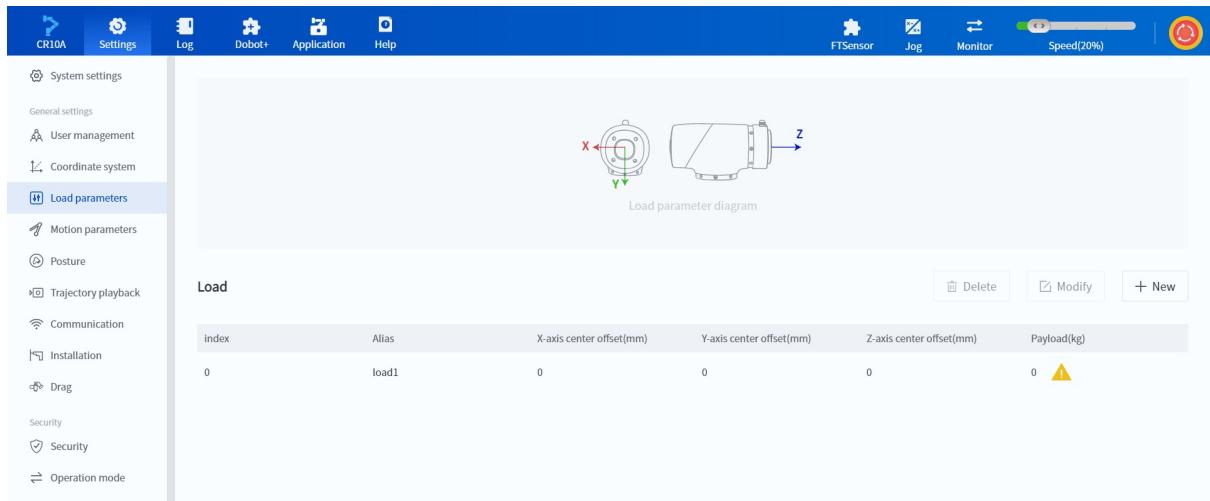


**Example 1:**

```
# Set the payload to 1.5kg, and offset coordinates to [0, 0, 10].  
SetPayload(1.5, [0, 0, 10])
```

**Required parameter 2:**

- **name:** The name of the preset load parameter group, which must first be saved in the software.



## Example 2:

```
# Set the payload using the preset load parameter group named "load1".
SetPayload("load1")
```

# User

## Command:

```
User(user)
```

## Description:

Set the global user coordinate system. If you do not specify the user coordinate system through optional parameters when calling other commands, the global user coordinate system will be used.

The global user coordinate system set by this command is only effective during the current project. If not specified, the default global user coordinate system is User coordinate system 0.

## Required parameter:

**user:** The user coordinate system to switch to, specified by an index, which must first be added in the software. If the specified coordinate system index does not exist, the project will stop and return an error.

## Example:

```
# Switch the global coordinate system to User coordinate system 1.
User(1)
```

# SetUser

## Command:

```
SetUser(index,table,type)
```

### Description:

Modify the specified user coordinate system.

#### Required parameter:

- **index:** The user coordinate system index, which must first be added in the software. If the specified coordinate system index does not exist, the project will stop and return an error.
- **table:** The user coordinate system to modify to, formatted as `[x, y, z, rx, ry, rz]`. It's recommended to use the CalcUser command to obtain this value.

#### Optional parameter:

- **type:** Whether to save globally.
  - 0: The coordinate system modified by this command will only apply during the current project and will revert to the original values once the project stops.
  - 1: The coordinate system modified by this command will be saved globally and persist even after the project stops. When `type=1` (global save), you need to manually switch to the corresponding coordinate system settings interface and then switch back again to see the updated values, meaning the interface needs to be manually refreshed.

### Example:

```
# Modify the User coordinate system 1 to the new coordinate system calculated by "CalcUser", effective only during project runtime.  
newUser = CalcUser(1,[10,10,10,10,10,10])  
SetUser(1,newUser,0)
```

## CalcUser

### Command:

```
CalcUser(index,matrix_direction,table)
```

### Description:

Calculate the user coordinate system.

#### Required parameter:

- **index:** The user coordinate system index, range: [0,50]. The initial value of coordinate system 0 refers to the base coordinate system.
- **matrix\_direction:** Specify the calculation direction.
  - 1: Left-multiply, meaning the coordinate system specified by the **index** is rotated relative to the base coordinate system using the values provided in **table**.

- **0:** Right-multiply, meaning the coordinate system specified by the **index** is rotated relative to itself using the values provided in **table**.
- **table:** The offset values for the user coordinate system, formatted as `[x, y, z, rx, ry, rz]` .

#### Return:

The calculated user coordinate system, formatted as `[x, y, z, rx, ry, rz]` .

#### Example:

```
# The following calculation is equivalent to: A coordinate system with an initial posture the same as User coordinate system 1 is translated by [x=10, y=10, z=10] and rotated by [rx=10, ry =10, rz=10] along the base coordinate system to get the newUser.
newUser = CalcUser(1,1,[10,10,10,10,10,10])
```

```
# The following calculation is equivalent to: A coordinate system with an initial posture the same as User coordinate system 1 is translated by [x=10, y=10, z=10] and rotated by [rx=10, ry =10, rz=10] along User coordinate system 1 to get the newUser.
newUser = CalcUser(1,0,[10,10,10,10,10,10])
```

## Tool

#### Command:

```
Tool(tool)
```

#### Description:

Switch the global tool coordinate system. If you do not specify the tool coordinate system through optional parameters when calling other commands, the global tool coordinate system will be used.

The global tool coordinate system set by this command is only effective during the current project. If not specified, the default global tool coordinate system is Tool coordinate system 0.

#### Required parameter:

**tool:** The tool coordinate system to switch to, specified by an index, which must first be added in the software. If the specified coordinate system index does not exist, the project will stop and return an error.

#### Example:

```
# Switch the global coordinate system to Tool coordinate system 1.
Tool(1)
```

## SetTool

#### Command:

```
SetTool(index,table,type)
```

### Description:

Modify the specified tool coordinate system.

#### Required parameter:

- **index:** The tool coordinate system index, which must first be added in the software. If the specified coordinate system index does not exist, the project will stop and return an error.
- **table:** The tool coordinate system to modify to, formatted as `[x, y, z, rx, ry, rz]`. It's recommended to use the CalcTool command to obtain this value.

#### Optional parameter:

- **type:** Whether to save globally.
  - 0: The coordinate system modified by this command will only apply during the current project and will revert to the original values once the project stops.
  - 1: The coordinate system modified by this command will be saved globally and persist even after the project stops. When `type=1` (global save), you need to manually switch to the corresponding coordinate system settings interface and then switch back again to see the updated values, meaning the interface needs to be manually refreshed.

### Example:

```
# Modify the Tool coordinate system 1 to the new coordinate system calculated by "CalcTool", effective only during project runtime.  
newTool = CalcTool(1,1,[10,10,10,10,10,10])  
SetTool(1,newTool,0)
```

## CalcTool

### Command:

```
CalcTool(index,matrix_direction,table)
```

### Description:

Calculate the tool coordinate system.

#### Required parameter:

- **index:** The tool coordinate system index, range: [0,50]. The initial value of coordinate system 0 is the flange coordinate system (TCP0).
- **matrix\_direction:** Specify the calculation direction.
  - 1: Left-multiply, meaning the coordinate system specified by the **index** is rotated relative to the flange coordinate system (TCP0) using the values provided in **table**.

- **0:** Right-multiply, meaning the coordinate system specified by the **index** is rotated relative to itself using the values provided in **table**.
- **table:** Tool coordinate system, formatted as `[x, y, z, rx, ry, rz]` .

**Return:**

The calculated tool coordinate system, formatted as `[x, y, z, rx, ry, rz]` .

**Example:**

```
# The following calculation is equivalent to: A coordinate system with an initial posture the same as Tool coordinate system 1 is translated by [x=10, y=10, z=10] and rotated by [rx=10, ry =10, rz=10] along the flange coordinate system (TCP0) to get the newTool.
newTool = CalcTool(1,1,[10,10,10,10,10,10])
```

```
# The following calculation is equivalent to: A coordinate system with an initial posture the same as Tool coordinate system 1 is translated by [x=10, y=10, z=10] and rotated by [rx=10, ry =10, rz=10] along Tool coordinate system to get the newTool.
newTool = CalcTool(1,0,[10,10,10,10,10,10])
```

## GetPose

**Command:**

```
GetPose(user_index, tool_index)
```

**Description:**

Get the the robot's real-time posture.

If this command is called between two motion commands, the obtained point may be affected by the continuous path settings.

- If the continuous path is disabled (cp or r is set to 0), the exact target point will be returned.
- If the continuous path is enabled, a point along the transition curve will be returned.

**Optional parameter:**

- **user\_index:** The user coordinate system index for the posture. The coordinate system must first be added in the software. The global user coordinate system will be used when the index is not set.
- **tool\_index:** The tool coordinate system index for the posture. The coordinate system must first be added in the software. The global tool coordinate system will be used when the index is not set.

**Return:**

The robot's current posture: `{"pose": [x, y, z, rx, ry, rz]}` .

**Example:**

```
# The robot moves to P1, and then returns to the current posture.  
currentPose = GetPose()  
MovJ(P1)  
MovJ(currentPose)
```

```
# The robot moves to P1 to get P1's posture, then moves to P2.  
MovJ(P1,{"cp":0})  
currentPose = GetPose() # Get P1's posture.  
MovJ(P2)
```

## GetAngle

### Command:

```
GetAngle()
```

### Description:

Get the real-time joint angle of the robot.

If this command is called between two motion commands, the obtained joint angle may be affected by the continuous path settings.

- If the continuous path function is disabled (cp or r is set to 0), the joint angles corresponding to target point will be accurately returned.
- If the continuous path function is enabled, the joint angles corresponding to a point along the transition curve will be returned.

### Return:

The current joint angle of the robot: {"joint":[j1, j2, j3, j4, j5, j6]} .

### Example:

```
# The robot moves to P1, and then returns to the current posture.  
currentAngle = GetAngle()  
MovJ(P1)  
MovJ(currentAngle)
```

```
# The robot moves to P1 to get the joint angle of P1, then moves to P2.  
MovJ(P1,{"cp":0})  
currentAngle = GetAngle() # Get the joint angle of P1.  
MovJ(P2)
```

## GetABZ

**Command:**

```
GetABZ()
```

**Description:**

Get the current position of the encoder.

**Return:**

The current position of the encoder.

**Example:**

```
# Get the current position of the ABZ encoder and assign the value to the variable "abz".  
abz = GetABZ()
```

## CheckMovJ

**Command:**

```
CheckMovJ(P, {"user":1, "tool":0, "a":20, "v":50, "cp":100})
```

**Description:**

Check whether a joint motion from the current point to the target point is reachable. The system calculates the entire motion trajectory and checks if any point in the path is unreachable.

**Required parameter:**

**P:** The target point.

**Optional parameter:**

- **user:** The user coordinate system for the target point.
- **tool:** The tool coordinate system for the target point.
- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **cp:** The ratio for smooth transition. Range: [0,100].

See [General description](#) for details.

**Return:**

Check result.

- 0: No error.
- 16: End point closed to shoulder singularity.

- 17: End point inverse kinematics error with no solution.
- 18: Inverse kinematics error with result out of working area.
- 22: Arm orientation error.
- 26: End point closed to wrist singularity.
- 27: End point closed to elbow singularity.
- 29: Speed parameter error.
- 30: Full parameter inverse kinematics error with no solution.
- 32: Shoulder singularity in trajectory.
- 33: Inverse kinematics error with no solution in trajectory.
- 34: Inverse kinematics error with result out of working area in trajectory.
- 35: Wrist singularity in trajectory.
- 36: Elbow singularity in trajectory.
- 37: Joint angle is changed over 180 degrees.

**Example:**

```
# Check whether the robot can reach P1 through joint motion with the default setting. If it can, move to P1 through joint motion.
status=CheckMovJ(P1)
if status==0:
    MovJ(P1)
    status=CheckMovJ(P2) # Perform feasibility check from P1 to P2 after the robot reaches P1
    if(status==0)
        MovJ(P2)
```

## CheckMovL

**Command:**

```
CheckMovL(P, {"user":1, "tool":0, "a":20, "v":50, "speed":500, "cp":100, "r":5})
```

**Description:**

Check whether a linear motion from the current point to the target point is reachable. The system calculates the entire motion trajectory and checks if any point in the path is unreachable.

**Required parameter:**

**P:** The target point.

**Optional parameter:**

- **user:** The user coordinate system for the target point. If the target point is given as joint variables, this parameter is ignored.
- **tool:** The tool coordinate system for the target point. If the target point is given as joint variables, this parameter is ignored.

- **a:** The acceleration ratio for the robot when executing this command. Range: (0,100].
- **v:** The velocity ratio for the robot when executing this command. Range: (0,100].
- **speed:** The target speed for the robot when executing this command. Range: [1, maximum speed], Unit: mm/s.

If this parameter is set, the **v** parameter will be ignored.

- **cp:** The ratio for smooth transition. Range: [0,100].
- **r:** The radius for smooth transition. Range: [0,100], Unit: mm.

If this parameter is set, the **cp** parameter will be ignored.

See [General description](#) for details.

#### **Return:**

Check result.

- 0: No error.
- 16: End point closed to shoulder singularity.
- 17: End point inverse kinematics error with no solution.
- 18: Inverse kinematics error with result out of working area.
- 22: Arm orientation error.
- 26: End point closed to wrist singularity.
- 27: End point closed to elbow singularity.
- 29: Speed parameter error.
- 30: Full parameter inverse kinematics error with no solution.
- 32: Shoulder singularity in trajectory.
- 33: Inverse kinematics error with no solution in trajectory.
- 34: Inverse kinematics error with result out of working area in trajectory.
- 35: Wrist singularity in trajectory.
- 36: Elbow singularity in trajectory.
- 37: Joint angle is changed over 180 degrees.

#### **Example:**

```
# Check whether the robot can reach P1 through linear motion with the default setting. If it can, move to P1 through linear motion.
status=CheckMovL(P1)
if(status==0)
    MovL(P1)
    status=CheckMovL(P2) # Perform reachability check from P1 to P2 after the robot reaches P1
    if(status==0)
        MovL(P2)
```

## **SetSafeWallEnable**

### **Command:**

```
SetSafeWallEnable(index,value)
```

### **Description:**

Enable or disable the specified safety wall. The safety wall status set by this command is effective only while the current project is running and will revert to the original value once the project stops.

### **Required parameter:**

- **index:** The safety wall index, which needs to be added in the software in advance. Range: [1,8].
- **value:**
  - True: Enable the safety wall.
  - False: Disable the safety wall.

### **Example:**

```
# Enable the safety wall with index 1.  
SetSafeWallEnable(1,True)
```

## **SetWorkZoneEnable**

### **Command:**

```
SetWorkZoneEnable(index,value)
```

### **Description:**

Enable or disable the specified safety zone. The safety zone status set by this command is effective only while the current project is running and will revert to the original value once the project stops.

### **Required parameter:**

- **index:** The safety zone index, which needs to be added in the software in advance. Range: [1,6].
- **value:**
  - True: Enable the safety zone.
  - False: Disable the safety zone.

### **Example:**

```
# Enable the safety zone with index 1.  
SetWorkZoneEnable(1,True)
```

## **SetCollisionLevel**

### **Command:**

```
SetCollisionLevel(level)
```

### **Description:**

Set the collision detection level. The value set by this command is effective only while the current project is running and will revert to the original value once the project stops.

### **Required parameter:**

**level:** Collision detection level. **0:** Disable the collision detection. **1 – 5:** The larger the number, the higher the sensitivity.

### **Example:**

```
# Set the collision detection level to 3.  
SetCollisionLevel(3)
```

## **SetBackDistance**

### **Command:**

```
SetBackDistance(distance)
```

### **Description:**

Set the distance the robot will retract along its original path after a collision is detected. The value set by this command is effective only while the current project is running and will revert to the original value once the project stops.

### **Required parameter:**

**distance:** The distance to retract after a collision, range: [0,50], unit: mm.

### **Example:**

```
# Set the collision backoff distance to 20mm.  
SetBackDistance(20)
```

# IO

## Command list

The IO commands are used to read and write system IO and set relevant parameters.

Command	Function
DI	Get status of DI port
DIGroup	Get status of multiple DI ports
DO	Set status of DO port
DOGroup	Set status of multiple DO ports
GetDO	Get status of DO port
GetDOGroup	Get status of multiple DO ports

## DI

### Command:

```
DI(index)
```

### Description:

Get the status of the digital input (DI) port.

### Required parameter:

**index:** DI index.

### Return:

Status (ON/OFF) of corresponding DI port.

### Example:

```
# The robot moves to P1 through linear motion when DI1 is ON.  
if DI(1)==ON:  
    MovL(P1)
```

## DIGroup

### Command:

```
DIGroup(index1,...,indexn)
```

### Description:

Get the status of multiple digital input (DI) ports.

### Required parameter:

**index:** DI index. You can input multiple separated by comma.

### Return:

Status (ON/OFF) of corresponding DI port, returned in the format of array.

### Example:

```
# The robot arm moves to P1 through linear motion when DI1 and DI2 are ON.  
digroup = DIGroup(1,2)  
if digroup[1]&digroup[2]==ON:  
    MovL(P1)
```

## DO

### Command:

```
DO(index,ON|OFF,time_ms)
```

### Description:

Set the status of digital output (DO) port.

### Required parameter:

- **index:** DO index.
- **ON|OFF:** The status to set for the DO port.

### Optional parameter:

**time\_ms:** Continuous output time. Unit: ms, range: [25,60000]. If this parameter is set, the system will automatically invert the DO after the specified time. The inversion is an asynchronous action, which will not block the command queue. After the DO output is executed, the system will execute the next command.

### Example:

```
# Set D01 to ON.  
DO(1,ON)
```

```
# Set D01 to ON, and set it automatically to OFF after 50ms.  
DO(1,ON,50)
```

## DOGroup

### Command:

```
DOGroup([index1,ON|OFF],...,[indexN,ON|OFF])
```

### Description:

Set the status of multiple digital output (DO) ports.

### Required parameter:

- **index:** DO index.
- **ON|OFF:** The status to set for the DO port.

You can set multiple groups, with each group within a brace and different groups separated by commas.

### Example:

```
# Set D01 and D02 to ON.  
DOGroup([1,ON],[2,ON])
```

## GetDO

### Command:

```
GetDO(index)
```

### Description:

Get the status of digital output (DO) port.

### Required parameter:

**index:** DO index.

### Return

Status (ON/OFF) of corresponding DI port.

### Example:

```
# Get the status of D01.  
GetDO(1)
```

## GetDOGroup

### Command:

```
GetDOGroup(index1,...,indexN)
```

### Description:

Get the status of multiple digital output (DO) ports.

### Required parameter:

**index:** DO index. You can input multiple separated by comma.

### Return:

Status (ON/OFF) of corresponding DO port, returned in the format of array.

### Example:

```
# Get the status of D01 and D02.  
GetDOGroup(1,2)
```

# Tool

## Command list

The tool commands are used to read and write the tool IO and set relevant parameters.

Command	Function
ToolDI	Read status of tool digital input port
ToolDO	Set status of tool digital output port (queue command)
GetToolDO	Get the current status of tool digital output port
SetToolPower	Set power status of end tool

## ToolDI

### Command:

```
ToolDI(index)
```

### Description:

Read the status of tool digital input port.

### Required parameter:

**index:** Tool DI index.

### Return:

Status (ON/OFF) of corresponding DI port.

### Example:

```
# The robot moves to P1 in a linear mode when the status of tool DI1 is ON.  
if ToolDI(1)==ON:  
    MovL(P1)
```

## ToolDO

### Command:

```
ToolDO(index,ON|OFF)
```

### Description:

Set the status of tool digital output port.

### Required parameter:

- **index:** Tool DO index.
- **ON|OFF:** The status to set for the DO port.

### Example:

```
# Set the status of tool D01 to ON.  
ToolDO(1,ON)
```

## GetToolDO

### Command:

```
GetToolDO(index)
```

### Description:

Get the current status of tool digital output port.

### Required parameter:

**index:** Tool DO index.

### Return

Status (ON/OFF) of corresponding tool DO port.

### Example:

```
# Get the current status of tool D01.  
GetToolDO(1)
```

## SetToolPower

### Command:

```
SetToolPower(status)
```

### Description:

Set the power status of the end tool, generally used for restarting the end power, such as re-powering and re-initializing the gripper. It is recommended to wait at least **4 ms** between consecutive calls to this interface.

**Required parameter:**

**status:** The power status of the end tool.

- 0: Power off.
- 1: Power on.

**Example:**

```
# Restart the end tool.  
SetToolPower(0)  
Wait(5)  
SetToolPower(1)
```

# TCP&UDP

## Command list

The TCP/UDP commands are used for TCP/UDP communication.

Command	Function
TCPCreate	Create TCP network object
TCPStart	Establish TCP connection
TCPRead	Receive data from TCP peer
TCPWrite	Send data to TCP peer
TCPDestroy	Disconnect TCP network and destroy socket object
UDPCreate	Create UDP network object
UDPRead	Receive data from UDP peer
UDPWrite	Send data to UDP peer

## TCPCreate

### Command:

```
TCPCreate(isServer, IP, port)
```

### Description:

Create a TCP network object, only one can be created.

### Required parameter:

- **isServer:** Specify whether to create a server.
  - **True:** Create a server.
  - **False:** Create a client.
- **IP:** IP address of the server. It must be in the same subnet as the client's IP, and there should be no conflicts.
  - When creating the server, use the robot's IP address.
  - When creating the client, use the peer's address.
- **port:** Server port.

Avoid using the following ports, as they are occupied by the system, which may cause server creation to fail.

7, 13, 22, 37, 139, 445, 502, 503 (Ports 0 – 1024 are often reserved by the Linux system, which has a high possibility of being occupied, so avoid using them as well),

1501, 1502, 1503, 4840, 8172, 9527,

11740, 22000, 22001, 29999, 30004, 30005, 30006,

60000 – 65504, 65506, 65511 – 65515, 65521, 65522.

**Return:**

- **err:** 0: TCP network has been created successfully. 1: TCP network failed to be created.
- **socket:** The created socket object.

**Example 1:**

```
# Create a TCP server.  
ip="192.168.5.1" # Set the robot's IP address as the server's IP.  
port=6001 # Server port.  
err=0  
socket=0  
err, socket = TCPCreate(True, ip, port)
```

**Example 2:**

```
# Create a TCP client.  
ip="192.168.5.25" # Set the IP address of external device (such as a camera) as the server's I  
P.  
port=6001 # Server port.  
err=0  
socket=0  
err, socket = TCPCreate(False, ip, port)
```

## TCPStart

**Command:**

```
TCPStart(socket, timeout)
```

**Description:**

Establish TCP connection.

- If the robot is acting as a server, it waits for a client to connect.
- If the robot is acting as a client, it actively connects to the server.

**Required parameter:**

- **socket:** The created socket object.
- **timeout:** The waiting timeout, unit: s.
  - If set to 0, it waits until the connection is successfully established.
  - If set to a non-zero value, it will return a connection failure if the time exceeds the specified limit.

**Return:**

Connection result.

- 0: Connection successful.
- 1: Invalid input parameters.
- 2: Socket object does not exist.
- 3: Invalid timeout setting.
- 4: Connection failed.

**Example:**

```
# Start establishing TCP connection until the connection is successful.
err = TCPStart(socket, 0) # socket is the object returned by TCPCreate.
```

## TCPRead

**Command:**

```
TCPRead(socket, timeout, type)
```

**Description:**

Receive data from TCP peer.

**Required parameter:**

**socket:** The created socket object.

**Optional parameter:**

- **timeout:** The waiting timeout, unit: s.
  - If not set or set to a value  $\leq 0$ , it will wait until the data is fully read before continuing.
  - If set to a value  $> 0$ , it will proceed without waiting if the time exceeds the limit.
- **type:** The type of the returned value.
  - If not set or set to "table", the RecBuf buffer will be in table format.
  - If set to "string", the RecBuf buffer will be in string format.

**Return:**

- **err:** 0: Data has been received successfully. 1: Data failed to be received.
- **Recbuf:** The data reception buffer.

**Example:**

```
# Receive TCP data, saving it in both string and table formats.  
# socket is the object returned by TCPCreate.  
err, RecBuf = TCPRead(socket,0,"string") # The data type of RecBuf is string.  
err, RecBuf = TCPRead(socket,0) # The data type of RecBuf is table.
```

## TCPWrite

### Command:

```
TCPWrite(socket, buf, timeout)
```

### Description:

Send data to TCP peer.

### Required parameter:

- **socket:** The created socket object.
- **buf:** The data to be sent.

### Optional parameter:

**timeout:** The waiting timeout, unit: s.

- If not set or set to 0, it will wait until the peer has fully received the data before continuing.
- If set to a non-zero value, it will proceed without waiting if the time exceeds the limit.

### Return:

Data sending result.

- 0: Send successful.
- 1: Send failed.

### Example:

```
# Send TCP data "test".  
TCPWrite(socket, "test") # socket is the object returned by TCPCreate.
```

## TCPDestroy

### Command:

```
TCPDestroy(socket)
```

### Description:

Disconnect the TCP connection and destroy the socket object.

**Required parameter:**

**socket:** The created socket object.

**Return:**

Execution result.

- 0: Execution successful.
- 1: Execution failed.

**Example:**

```
# Disconnect from the TCP peer.  
TCPDestroy(socket) # socket is the object returned by TCPCreate.
```

## UDPCreate

**Command:**

```
UDPCreate(isServer, IP, port)
```

**Description:**

Create a UDP network object, only one can be created.

**Required parameter:**

- **isServer:** Specify whether to create a server.
  - **True:** Create a server.
  - **False:** Create a client.
- **IP:** Fill in the IP address of the peer when creating both the server and client. It must be in the same subnet as the robot's IP, and there should be no conflicts.
- **port:**
  - When creating a server, this port is used by both the local and the peer. Do not use ports already occupied by the system, see the TCPCreate parameter description for details.
  - When creating a client, it refers to the peer's port, while the local side uses a random port for sending data.

**Return:**

- **err:** 0: UDP network has been created successfully. 1: UDP network failed to be created.
- **socket:** The created socket object.

**Example 1:**

```
# Create a UDP server.
```

```

local ip="192.168.5.25" # Set the IP address of external device (such as a camera) as the peer
's IP.
local port=6001 # This port is used by both the local and the peer.
local err=0
local socket=0
err, socket = UDPCreate(True, ip, port)

```

### Example 2:

```

# Create a UDP client.
local ip="192.168.5.25" # Set the IP address of external device (such as a camera) as the peer
's IP.
local port=6001 # Peer port
local err=0
local socket=0
err, socket = UDPCreate(False, ip, port)

```

## UDPRead

### Command:

```
UDPRead(socket, timeout, type)
```

### Description:

Receive data from UDP peer.

### Required parameter:

**socket:** The created socket object.

### Optional parameter:

- **timeout:** The waiting timeout, unit: s.
  - If not set or set to a value  $\leq 0$ , it will wait until the data is fully read before continuing.
  - If set to a value  $> 0$ , it will proceed without waiting if the time exceeds the limit.
- **type:** The type of the returned value.
  - If not set or set to "table", the RecBuf buffer will be in table format.
  - If set to "string", the RecBuf buffer will be in string format.

### Return:

- **err:** 0: Data has been received successfully. 1: Data failed to be received.
- **Recbuf:** The data reception buffer.

### Example:

```

# Receive UDP data, saving it in both string and table formats.
# socket is the object returned by UDPCreate.

```

```
err, RecBuf = UDPRead(socket,0,"string") # The data type of RecBuf is string.  
err, RecBuf = UDPRead(socket, 0) # The data type of RecBuf is table.
```

## UDPWrite

### Command:

```
UDPWrite(socket, buf, timeout)
```

### Description:

Send data to UDP peer.

### Required parameter:

- **socket:** The created socket object.
- **buf:** The data to be sent.

### Optional parameter:

**timeout:** The waiting timeout, unit: s.

- If not set or set to 0, it will wait until the peer has fully received the data before continuing.
- If set to a non-zero value, it will proceed without waiting if the time exceeds the limit.

### Return:

Data sending result.

- 0: Send successful.
- 1: Send failed.

### Example:

```
# Send UDP data "test".  
UDPWrite(socket, "test") # socket is the object returned by UDPCreate.
```

# Modbus

## Command list

The Modbus commands are used to establish the communication between Modbus master and slave. For the range and definition of the register address, please refer to the instructions of the corresponding slave.

Command	Function
ModbusCreate	Create Modbus master
ModbusRTUCREATE	Create Modbus master based on RS485
ModbusClose	Disconnect with Modbus slave
GetInBits	Read contact registers
GetInRegs	Read input registers
GetCoils	Read coil registers
SetCoils	Write to coil registers
GetHoldRegs	Read holding registers
SetHoldRegs	Write to holding registers

The Modbus function codes for different types of registers follow the standard Modbus protocol:

Register type	Read register	Write single register	Write multiple registers
Coil register	01	05	0F
Contact (discrete input) register	02	-	-
Input register	04	-	-
Holding register	03	06	10

## ModbusCreate

### Command:

```
ModbusCreate(IP, port, slave_id, isRTU)
```

### Description:

Create Modbus master, and establish connection with the slave. A maximum of 15 devices can be connected at the same time.

When connecting to the robot's built-in slave, set the IP to the robot's IP (default 192.168.5.1, modifiable) and the port to 502 (map1) or 1502 (map2). See [Appendix A Modbus Register Definition](#) for details.

When connecting to a third-party slave, please refer to the instructions of the corresponding slave for the register address range and definition when reading and writing registers.

**Required parameter:**

- **IP:** Slave IP address.
- **port:** Slave port.

**Optional parameter:**

- **slave\_id:** Slave ID.
- **isRTU:** Boolean.
  - When **isRTU** is false, establish Modbus TCP communication via the controller's Ethernet port.
  - When **isRTU** is true, establish Modbus RTU communication via the robot's end RS485, and only port 60000 can be used.

**⚠️NOTICE**

This parameter determines the protocol format used to transmit data once the connection has been established, and it does not affect the connection result. Therefore, if you set the parameter incorrectly when creating a master, the master can still be created successfully, but an exception may occur in the subsequent communication.

**Return:**

- **err:**
  - 0: Modbus master has been created successfully.
  - 1: The maximum number of masters has been reached, failed to create a new master.
  - 2: Failed to initialize the master, check if the IP, port, and network are working properly.
  - 3: Failed to connect to the slave, check if the slave is functioning and the network is normal.
- **id:** The returned master index, which is used when other Modbus commands are called. Range: [0, 14].

**Example:**

```
# Create the Modbus master, and connect with the specified slave.
ip="192.168.5.123"
port=503
err=0
id=0
err, id = ModbusCreate(ip, port, 1)
```

```
# Create the Modbus master, and connect with the robot slave.
```

```
ip="192.168.5.1"
port=502
err=0
id=0
err, id = ModbusCreate(ip, port)
```

## ModbusRTUCREATE

### Command:

```
ModbusRTUCREATE(slave_id, baud, parity, data_bit, stop_bit)
```

### Description:

Create Modbus master based on the controller's RS485 interface and establish connection with the slave. A maximum of 15 devices can be connected at the same time.

### Required parameter:

- **slave\_id:** Slave ID.
- **baud:** Baud rate for the RS485 interface.

### Optional parameter:

- **parity:** Whether there is a parity bit.
  - "O": odd.
  - "E": even.
  - "N": no parity bit.
  - "E" by default.
- **data\_bit:** Data bit length. Range: 8. 8 by default.
- **stopbit:** Stop bit length. Range: 1, 2. 1 by default.

### Return:

- **err:** 0: Modbus master station has been created successfully. 1: Modbus master station failed to be created.
- **id:** The returned master index, which is used when other Modbus commands are called.

### Example:

```
# Create Modbus master and establish connection with the slave through RS485 interface, slave
ID is 1, baud rate is 115200.
err, id = ModbusRTUCREATE(1, 115200)
```

## ModbusClose

### Command:

```
ModbusClose(id)
```

#### Description:

Disconnect from the Modbus slave and release the master.

#### Required parameter:

**id:** The index of the created master.

#### Return:

Operation result.

- 0: Disconnection successful.
- 1: Disconnection failed.

#### Example:

```
# Disconnect from the Modbus slave.  
ModbusClose(id)
```

## GetInBits

#### Command:

```
GetInBits(id, addr, count)
```

#### Description:

Read the contact register value from the Modbus slave. The corresponding Modbus function code is 02.

#### Required parameter:

- **id:** The index of the created master.
- **addr:** The starting address of the contact register.
- **count:** The number of contact registers to read. Range: 1 – 2000 (ModbusTCP protocol restriction).  
For actual range, please determine according to the number of slave registers or the protocol).

#### Return:

The values read from the contact registers, stored in a table. The first value in table corresponds to the value of contact register at the starting address.

#### Example:

```
# Read 5 contact registers starting from address 0.  
inBits = GetInBits(id,0,5)
```

## GetInRegs

### Command:

```
GetInRegs(id, addr, count, type)
```

### Description:

Read the input register value with the specified data type from the Modbus slave. The corresponding Modbus function code is 04.

### Required parameter:

- **id:** The index of the created master.
- **addr:** The starting address of the input register.
- **count:** The number of input registers to read. Range: [1, 125] (ModbusTCP protocol restriction. For actual range, please determine according to the number of slave registers or the protocol).

### Optional parameter:

**type:** Data type.

- Empty: U16 by default.
- U16: 16-bit unsigned integer (two bytes, occupy one register).
- U32: 32-bit unsigned integer (four bytes, occupy two register).
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

### Return:

The values read from the input registers, stored in a table. The first value in table corresponds to the value of input register at the starting address.

### Example:

```
# Read a 32-bit unsigned integer starting from address 2048.  
data = GetInRegs(id, 2048, 2, "U32")
```

## GetCoils

### Command:

```
GetCoils(id, addr, count)
```

### Description:

Read the coil register value from the Modbus slave. The corresponding Modbus function code is 01.

### **Required parameter:**

- **id:** The index of the created master.
- **addr:** The starting address of the coil register.
- **count:** The number of coil registers to read. Range: [1, 2000] (ModbusTCP protocol restriction. For actual range, please determine according to the number of slave registers or the protocol).

### **Return:**

The values read from the coil registers, stored in a table. The first value in table corresponds to the value of coil register at the starting address.

### **Example:**

```
# Read 5 contact registers starting from address 0.  
Coils = GetCoils(id,0,5)
```

## **SetCoils**

### **Command:**

```
SetCoils(id, addr, count, table)
```

### **Description:**

Write specified values to the coil register at the designated address. The corresponding Modbus function code is 05 (write single) and 0F (write multiple).

### **Required parameter:**

- **id:** The index of the created master.
- **addr:** The starting address of the coil register.
- **count:** The number of values to write to the coil register. Range: [1, 1968] (ModbusTCP protocol restriction. For actual range, please determine according to the number of slave registers or the protocol).
- **table:** The values to write to the coil register, stored in a table. The first value in table corresponds to the value of coil register at the starting address.

### **Example:**

```
# Starting from address 1024, write 5 values in succession to the coil register.  
local Coils = {0,1,1,1,0}  
SetCoils(id, 1024, #coils, Coils)
```

## **GetHoldRegs**

### **Command:**

```
GetHoldRegs(id, addr, count, type)
```

### Description:

Read the holding register value with the specified data type from the Modbus slave. The corresponding Modbus function code is 03.

### Required parameter:

- **id:** The index of the created master.
- **addr:** The starting address of the holding register.
- **count:** The number of holding registers to read. Range: [1, 125] (ModbusTCP protocol restriction).  
For actual range, please determine according to the number of slave registers or the protocol).

### Optional parameter:

**type:** Data type.

- Empty: U16 by default.
- U16: 16-bit unsigned integer (two bytes, occupy one register).
- U32: 32-bit unsigned integer (four bytes, occupy two register).
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

### Return:

The values read from the holding registers, stored in a table. The first value in table corresponds to the value of holding register at the starting address.

### Example:

```
# Read a 32-bit unsigned integer starting from address 2048.  
data = GetHoldRegs(id, 2048, 2, "U32")
```

## SetHoldRegs

### Command:

```
SetHoldRegs(id, addr, count, table, type)
```

### Description:

Write specified values to the holding register at the designated address according to the specified data type. The corresponding Modbus function code is 06 (write single) and 10 (write multiple).

### Required parameter:

- **id:** The index of the created master.

- **addr:** The starting address of the holding register.
- **count:** The number of values to write to the holding register. Range: [1, 123] (ModbusTCP protocol restriction. For actual range, please determine according to the number of slave registers or the protocol).
- **table:** The values to be written to the holding registers, stored in a table. The first value in table corresponds to the value of holding register at the starting address.

**Optional parameter:**

**type:** Data type.

- Empty: U16 by default.
- U16: 16-bit unsigned integer (two bytes, occupy one register).
- U32: 32-bit unsigned integer (four bytes, occupy two register).
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers).
- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers).

**Example:**

```
# Write a 64-bit double-precision floating-point number starting from address 2048.
local data = {95.32105}
SetHoldRegs(id, 2048, 4, data, "F64")
```

# Program control

## Command list

The program control commands are general commands related to program control.

Command	Function
Print	Print debug information to the console
Wait	Wait for a specified time or until a condition is met, then proceed to the next command
Pause	Pause running the script
ResetElapsedTime	Start timing
ElapsedTime	Stop timing
Systime	Get current system time

## Print

### Command:

```
Print(value)
```

### Description:

Print debug information to the console (the command can also be written as `print` ).

### Required parameter:

**value:** The data to be printed.

### Example:

```
Print the string "Success" to the console.  
Print('Success')
```

## Wait

### Command:

```
Wait(time_ms)  
Wait(check_str)  
Wait(check_str, timeout_ms)
```

---

## Description:

After the robot completes the previous command, it waits for a specified time or until a condition is met before proceeding to the next command.

## Required parameter:

- **time\_ms:** If the value is an integer, it represents the wait time. If the value is less than or equal to 0, it means no wait. Unit: ms.
- **check\_str:** If the value is a string, it represents a condition. The system will proceed once the condition becomes true.

## Optional parameter:

**timeout\_ms:** Timeout period. Unit: ms.

- If the condition remains false and the waiting time exceeds this period, the system will proceed to the next command and return false.
- If the value is less than or equal to 0, it means an immediate timeout.
- If this parameter is not set, the system will wait indefinitely until the condition becomes true.

## Return:

- Return **true** when the condition is met and execution continues.
- Return **false** if the condition is not met and the system continues due to timeout.

## Example:

```
# Wait for 300 ms.  
Wait(300)
```

```
# Continue running when DI1 is ON.  
Wait("DI(1) == ON")
```

```
# Continue running when DO1 is ON and AI(1) is less than 7.  
Wait("GetDO(1) == ON and AI(1) < 7")
```

```
# Execute different logics according to DI1 status within 1s.  
if Wait("DI(1) == ON", 1000):  
    # DI1 is ON.  
else:  
    # DI1 is OFF and wait more than 1s.
```

## Pause

## Command:

```
Pause()
```

### Description:

Pause running the script. The script can continue to run only through software control or remote control.

### Example:

```
# The robot moves to P1 and then pauses running. It will continue to move to P2 only through external control.  
MovJ(P1)  
Pause()  
MovJ(P2)
```

## ResetElapsedTime

### Command:

```
ResetElapsedTime()
```

### Description:

Starts timing after all commands before this command have been executed. This command should be used with ElapsedTime() to calculate the runtime.

### Example:

Refer to the example of ElapsedTime.

## ElapsedTime

### Command:

```
ElapsedTime()
```

### Description:

Stop timing and return the time difference. This command should be used with ResetElapsedTime().

### Return:

Time difference between the start and the end of timing, unit: ms. The maximum measurable time is 4,294,967,295 ms (about 49.7 days). After exceeding this time, it will reset and start counting from 0.

### Example:

```
# Calculate the time taken for the robot to move from its current position to P2 via P1, and p
```

```
rint the result to the console.  
ResetElapsedTime()  
MovL(P1)  
MovL(P2)  
print(ElapsedTime())
```

## Systime

### Command:

```
Systime()
```

### Description:

Get the current system time.

### Return:

The current system time's Unix timestamp in milliseconds, which is the number of milliseconds since 00:00:00 on January 1, 1970, GMT. This command is generally used to calculate time differences. To get the local time, please convert the obtained GMT according to your local time zone.

### Example:

```
# Get the current system time.  
time1 = Systime()  
print(time1) # 1686304295963, translated to 2023-06-09 17:51:35 BST (plus 963 milliseconds).  
time2 = Systime()  
print(time2) # 1686304421968, translated to 2023-06-09 17:53:41 BST (plus 968 milliseconds).  
  
# Calculate the time taken for the robot to move to P1, unit: ms.  
time1 = Systime()  
MovL(P1)  
time2 = Systime()  
print(time2-time1)
```