DobotStudio Pro User Guide

(MG400 & M1 Pro)

Shenzhen Yuejiang Technology Co.,Ltd.|China

Table of Contents

Preface
1 Getting Started
2 Connecting to Robot
3 Main Interface
  3.1 Overview
  3.2 Top toolbar
  3.3 Control panel
4 Settings
  4.1 Basic settings
  4.2 Communication settings
  4.3 Coordinate System
    4.3.1 User coordinate system
    4.3.2 Tool coordinate system
  4.4 Load parameters
  4.5 Motion parameters
  4.6 Security setting
  4.7 Remote control
  4.8 Hand calibration (M1 Pro)
  4.9 Firmware update
  4.10 Home calibration
5 I/O Monitoring
6 Modbus
7 Global Variable
8 Programming
  8.1 DobotBlockly
  8.2 Script
9 Best Practice
Appendix A Modbus Register Definition
Appendix B Blockly Commands
  B.1 Quick start
    B.1.1 Control robot movement

B.1.2 Read and write Modbus register data
B.1.3 Transmit data by TCP communication
B.1.4 Palletize
B.2 Block description
B.2.1 Event
B.2.2 Control
B.2.3 Operator
B.2.4 String
B.2.5 Custom
B.2.6 IO
B.2.7 Motion
B.2.8 Motion advanced configuration
B.2.9 Posture
B.2.10 Modbus
B.2.11 TCP
Appendix C Script Commands
C.1 Lua basic grammar
C.1.1 Variable and data type
C.1.2 Operator
C.1.3 Process control
C.2 Command description
C.2.1 Motion
C.2.2 Motion parameter
C.2.3 Relative motion
C.2.4 IO
C.2.5 TCP/UDP
C.2.6 Modbus
C.2.7 Program control
C.2.8 Vision

Preface

Purpose

This document describes the functions and operations of DobotStudio Pro for controlling four-axis robots (MG400 and M1Pro), which is convenient for users to fully understand and use the software.

Intended Audience

This document is intended for：

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

Change History

Date	Change Description
2023/05/16	Update Modbus register definition and refine some descriptions
2023/04/21	Update blockly programming demos
2023/01/12	Update to V2.6.0
2022/11/30	Add description on DobotBlockly commands and Script commands in V2.5.0
2022/10/31	Adjust the catalogue and update the content based on the latest software (V2.4.0) Add an appendix about Modbus register definition Divide the content about six-axis robots and four-axis robots into two separate documents
2022/03/25	Rename the software as DobotStudio Pro Update MG400 description according to the latest software interface, add alarm description, motion parameter settings, WiFi Settings, etc. Add description on CR robots (Chapter 3) Delete description on M1
2020/05/20	The first release

Symbol Conventions

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

Symbol	Description
⚠ DANGER	Indicates a hazard with a high level of risk which, if not avoided, could result in death or serious injury

⚠ WARNING	Indicates a hazard with a medium level or low level of risk which, if not avoided, could result in minor or moderate injury, robotic arm damage
⚠ NOTICE	Indicates a potentially hazardous situation which, if not avoided, can result in robotic arm damage, data loss or unanticipated result
NOTE	Provides additional information to emphasize or supplement important points in the main text

1 Getting Started

DobotStudio Pro is a multi-functional control software for robot arms independently developed by Dobot. With simple interface, easy-to-use functions and strong practicability, it can help you quickly master the use of various robot arms.

This document mainly introduces how to use DobotStudio Pro to control four-axis robot arm (MG400 and M1Pro). As the control modes of M1 Pro and MG400 are similar, this document takes MG400 as an example to introduce the usage of DobotStudio Pro.

DobotStudio Pro supports the following operation systems:

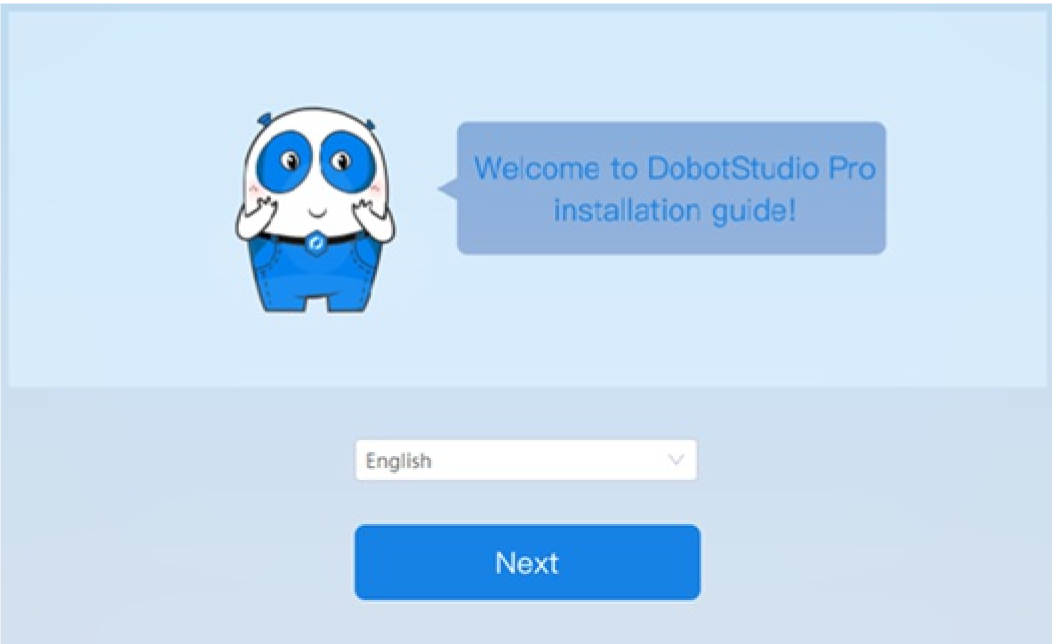
- Win7
- Win10
- Win11

DobotStudio Pro Installation

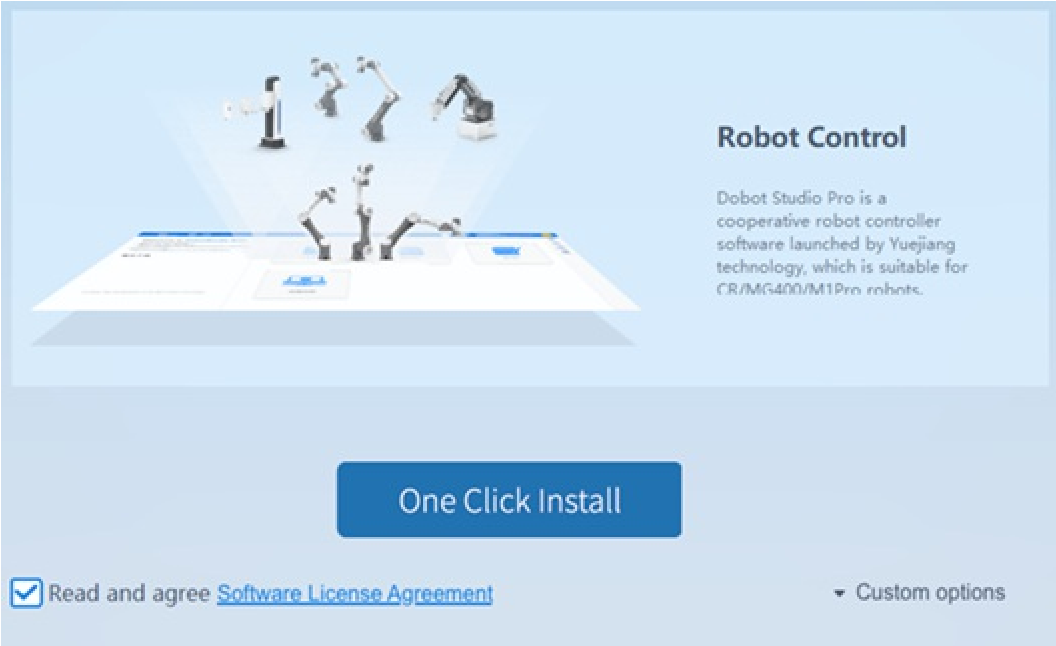
Please visit Dobot website to download the latest DobotStudio Pro installation package.

Procedure

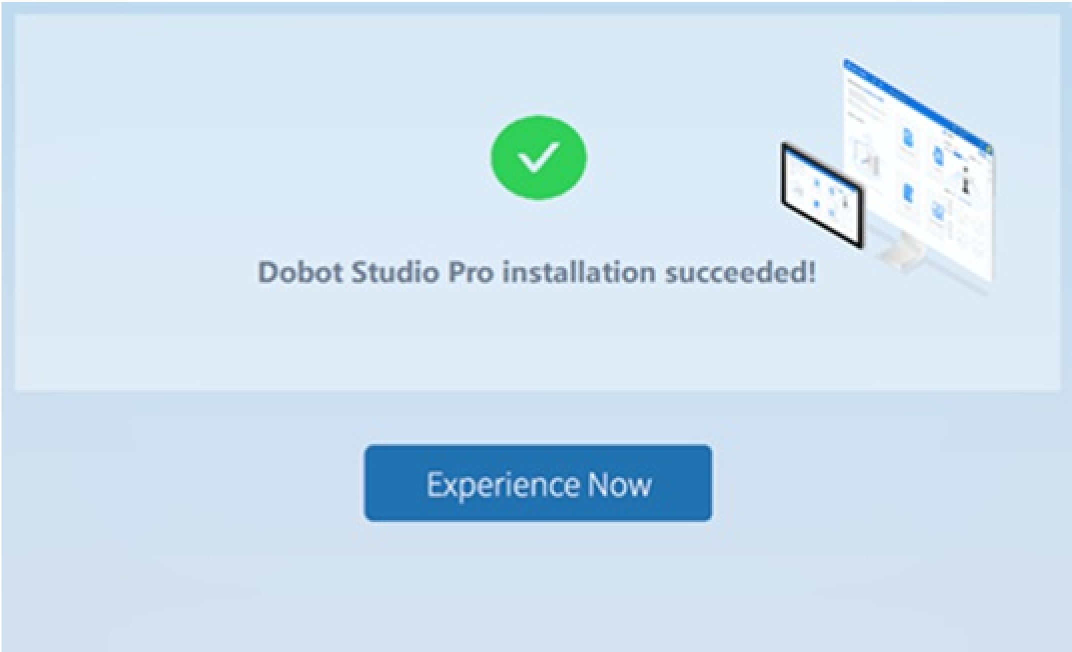
Step 1: Double-click DobotStudio Pro installation package. Select a language for installation. Click **Next**.



Step 2: Click **One Click Install**, or start installation after setting the installation path in Custom options.



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



Guidance

If you are using DobotStudio Pro for the first time, it is recommended to read this Guide in the following order.

1. Connecting to Robot: Connect DobotStudio Pro to the robot arm.
2. Main Interface: Know about the main interface of DobotStudio Pro and roughly understand the functions of DobotStudio Pro.
3. Settings: Configure the robot arm based on actual requirements.
4. I/O Monitoring: Know about the monitoring function provided by DobotStudio Pro.

5. Programming: Know about the programming and process module of DobotStudio Pro and try creating your own project.
6. Remote Control: After developing a project, try running the project through remote control.

2 Connecting to Robot

DobotStudio Pro supports wired (LAN) and wireless (WiFi) connection to the robot.

> NOTE
>
> If the robot controller version is lower than 1.5.6.0, you need to open SMB1 protocol before connecting to the robot. See (Optional) Open SMB protocol in this chapter for details.

Wired connection

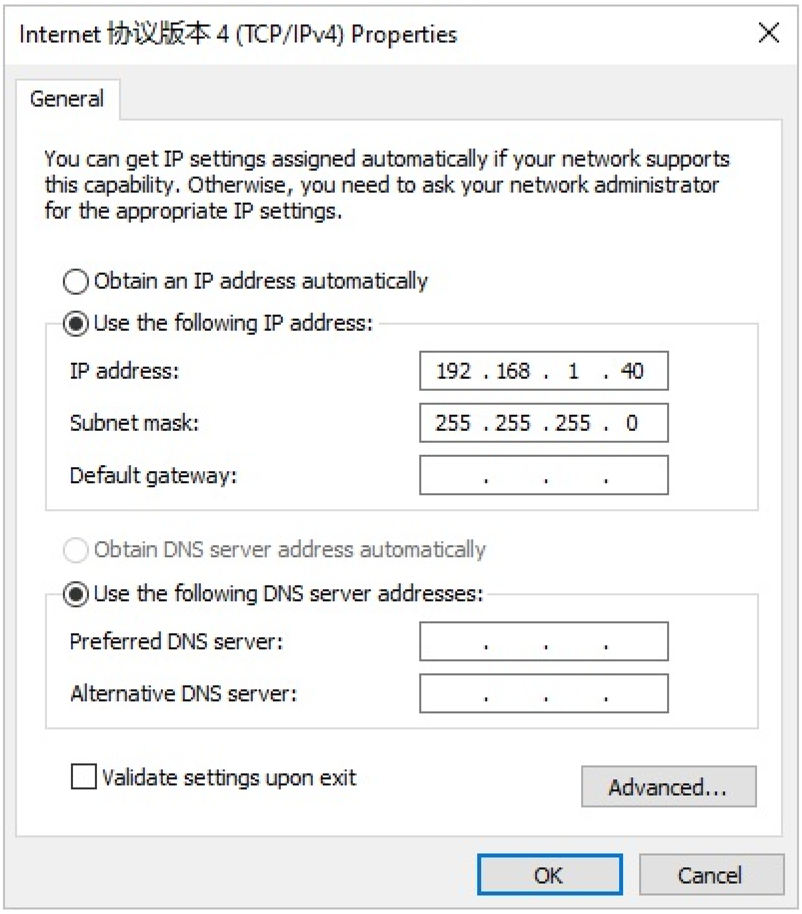
Connect one end of the network cable to the LAN interface on the controller and the other end to the PC. Change the IP address of the PC to make it in the same network segment as that of the controller. The default IP address of the controller LAN1 is 192.168.1.6, and the controller LAN1 is 192.168.2.6, which can be modified in Communication settings.

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

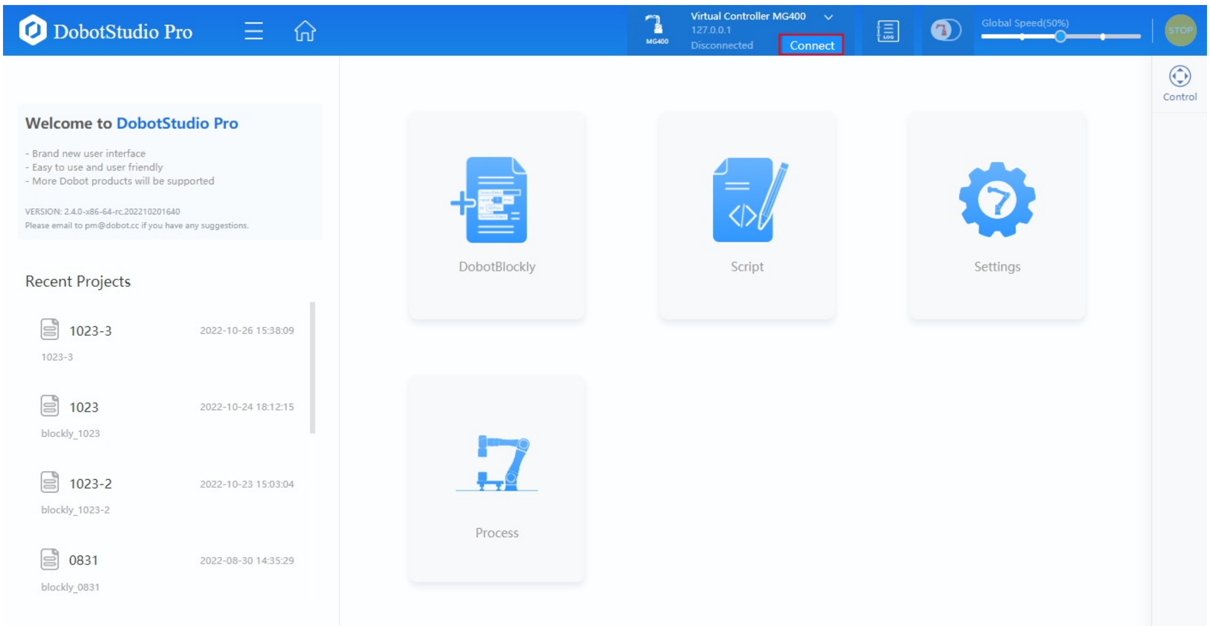
Step 1: Search **View network connections**, and click **Open**.

Step 2: Right-click **Properties** on the currently-connected network. Then double-click **Internet Protocol Version 4(TCP/IPv4)** in the pop-up window.

Step 3: Select **Use the following IP address** in "Internet Protocol Version 4(TCP/IPv4)" page, and change the IP address, subnet mask and gateway of the PC. You can change the IP address of the PC to make it in the same network segment as that of the controller without conflict. The subnet mask and gateway of the PC must be the same as that of controller. For example, set the IP address to 192.168.1.40, and subnet mask to 255.255.255.0.



Step 4: Start DobotStudio Pro. Select a device and click **Connect**.



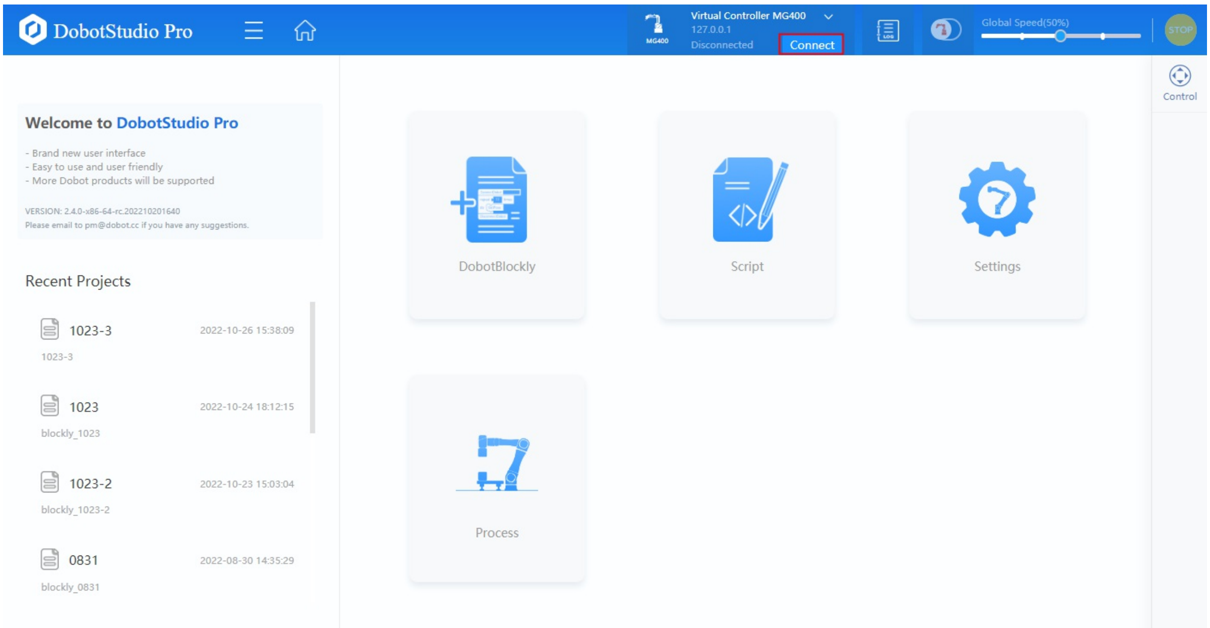
Step 5: After connection, you will see the pop-up prompt information in DobotStudio Pro interface, and the connection status turns to **Connected**. If you want to disconnect the robot, click **Connected**.

Wireless Connection

Before connecting to the robot, ensure that a WiFi module has been installed in the controller.

Step 1: Search Dobot controller WiFi name and connect it. The WiFi SSID is MagicianPro, and WiFi password is 1234567890 by default. You can modify the WIFI SSID and password in Communication settings.

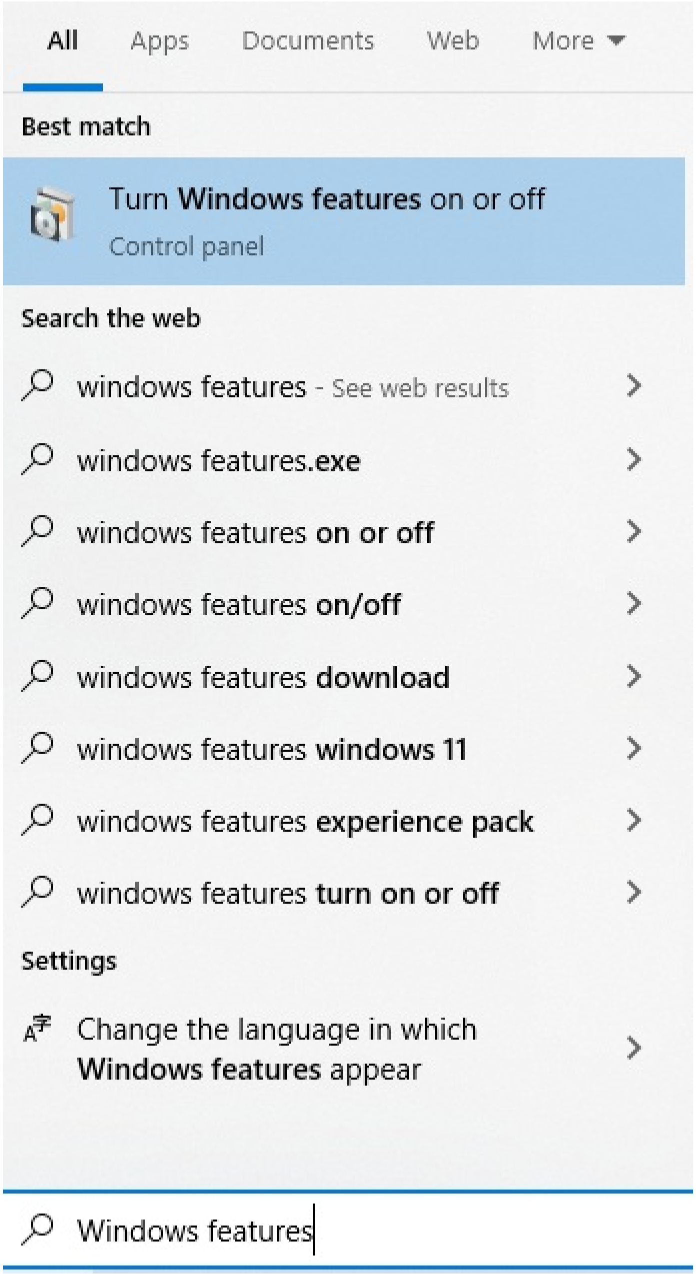
Step 2: Select a robot in the top of DobotStudio Pro interface and click **Connect**.



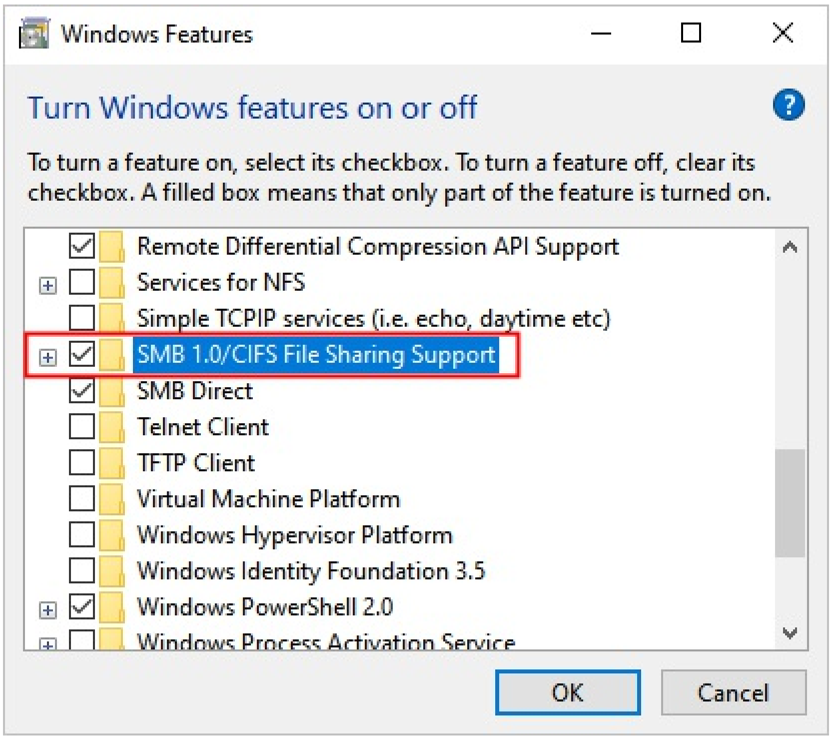
Step 3: After connection, you will see the pop-up prompt information in DobotStudio Pro interface, and the connection status turns to **Connected**. If you want to disconnect the robot, click **Connected**.

(Optional) Open SMB protocol

1. Taking Windows 10 as an example, search "Windows features" in the taskbar, and click **Turn Windows features on or off**.



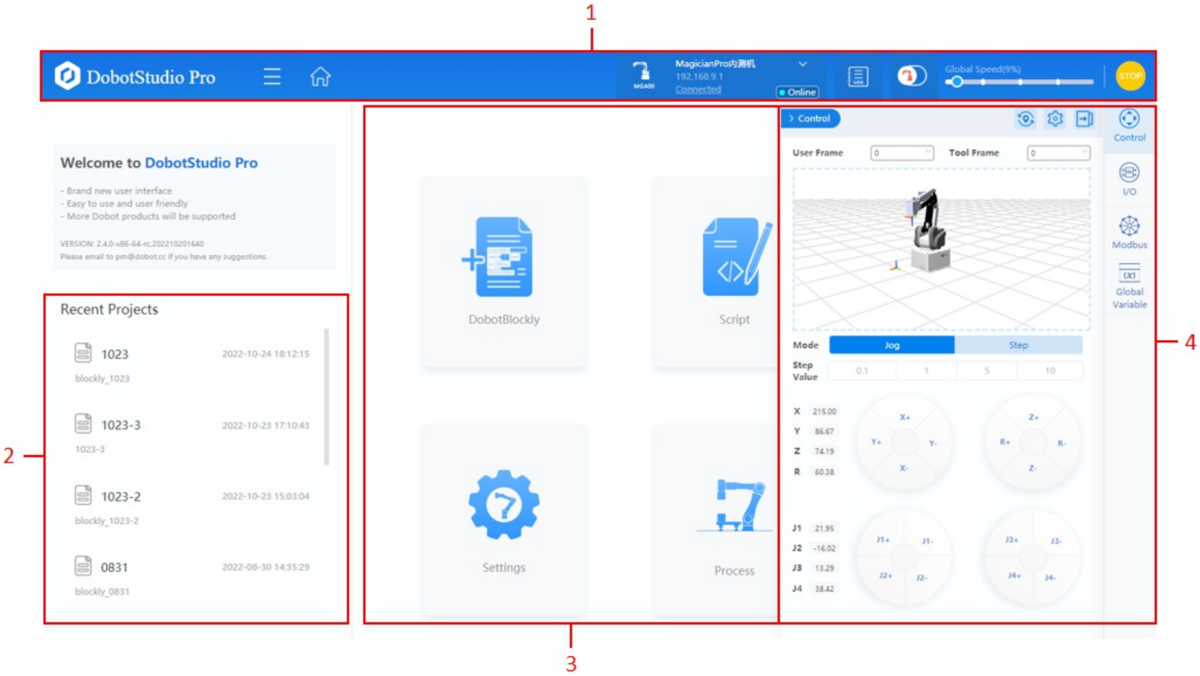
2. Select **SMB 1.0/CIFS File Sharing Support**, and click **OK**.


Windows Features
Turn Windows features on or off
To turn a feature on, select its checkbox. To turn a feature off, clear its checkbox. A filled box means that only part of the feature is turned on.
Remote Differential Compression API Support
Services for NFS
Simple TCPIP services (i.e. echo, daytime etc)
SMB 1.0/CIFS File Sharing Support
SMB Direct
Telnet Client
TFTP Client
Virtual Machine Platform
Windows Hypervisor Platform
Windows Identity Foundation 3.5
Windows PowerShell 2.0
Windows Process Activation Service
OK
Cancel

3 Main Interface

- 3.1 Overview
- 3.2 Top toolbar
- 3.3 Control panel

3.1 Overview



No.	Description
1	Top toolbar
2	Display the recent projects, which you can click to open quickly.
3	Major functions, including DobotBlockly, Script, Settings and Process. For instructions on various processes, refer to the corresponding manual of each process.
4	Click the icon on the right toolbar to display or hide the corresponding panels, including Control, I/O, Modbus and Global Variable. The control panel is displayed by default after DobotStudio Pro is connected to the robot successfully.

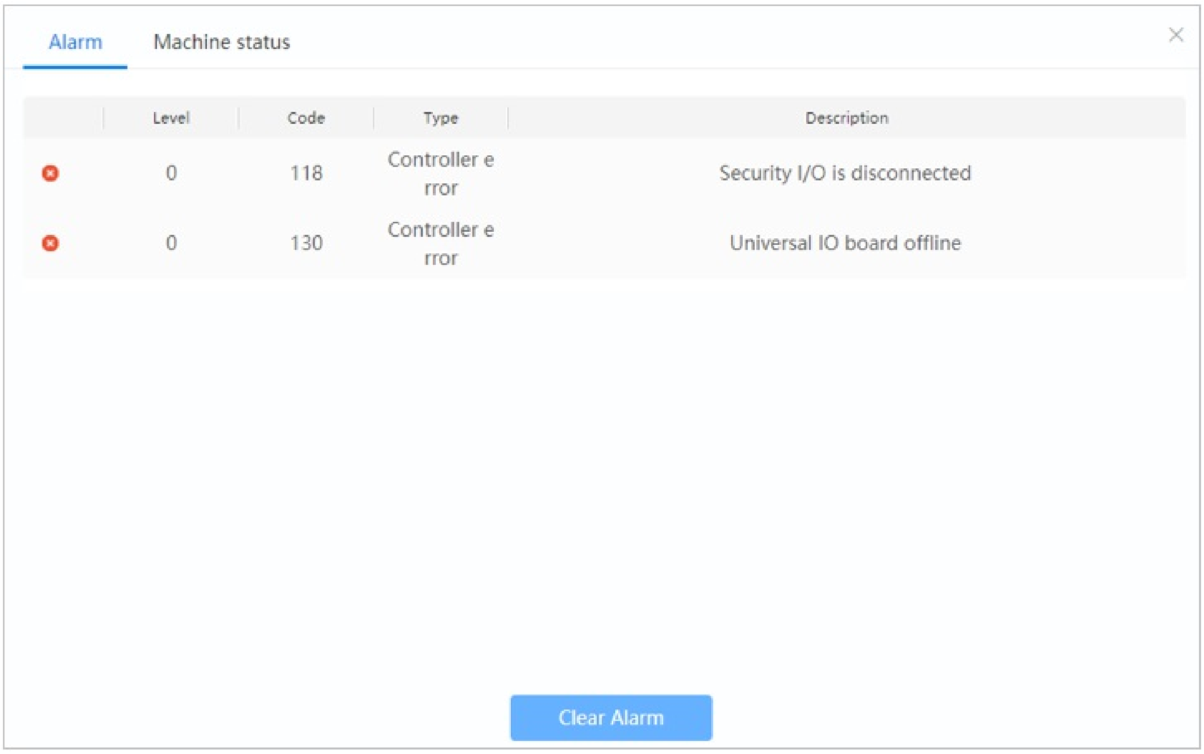
3.2 Top toolbar

No.	Description
1	Click the icon, and the following items will pop up: • Settings: Click to open Settings page • Language: Select a language • Help: Access help functions, such as help documents, debugging tools, etc. • Check updates: View the version information of the software • About: View the components of the software
2	Click to return to the main interface.
3	Connection panel. See Connecting to Robot for details
4	Alarm log button. See Alarm log below.
5	See Enabling status for details.
6	Drag the blue slider or click the speed bar to adjust the global speed ratio. The global speed ratio is the calculation factor of the actual running speed of the robot arm. For the calculation method, see Jog setting
7	Emergency stop button. Press the button in an emergency, and robot arm will stop running and be powered off. See Emergency stop button for details.

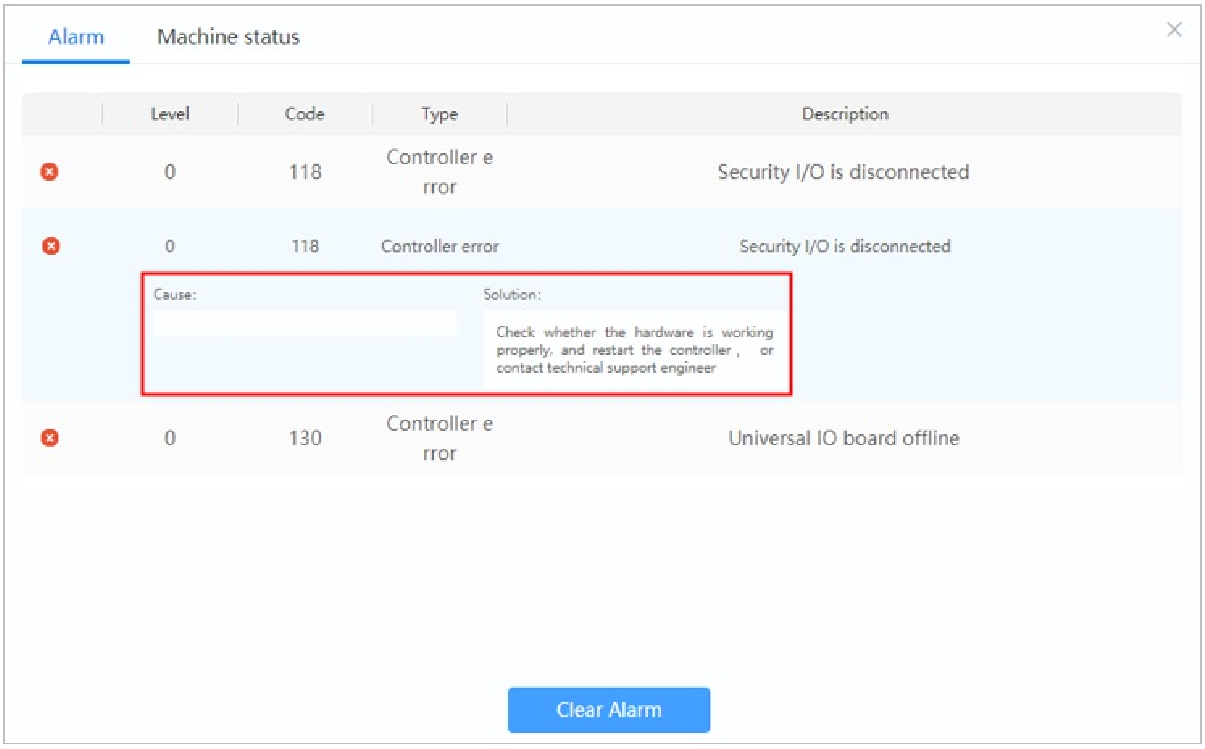
Alarm log

If a point is saved incorrectly, for example, a robot moves to where a point is at a limited position or a singular point, an alarm will be triggered.

If an alarm is triggered when a robot is running, the alarm icon turns into . You can check the alarm information on the Alarm page.

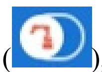


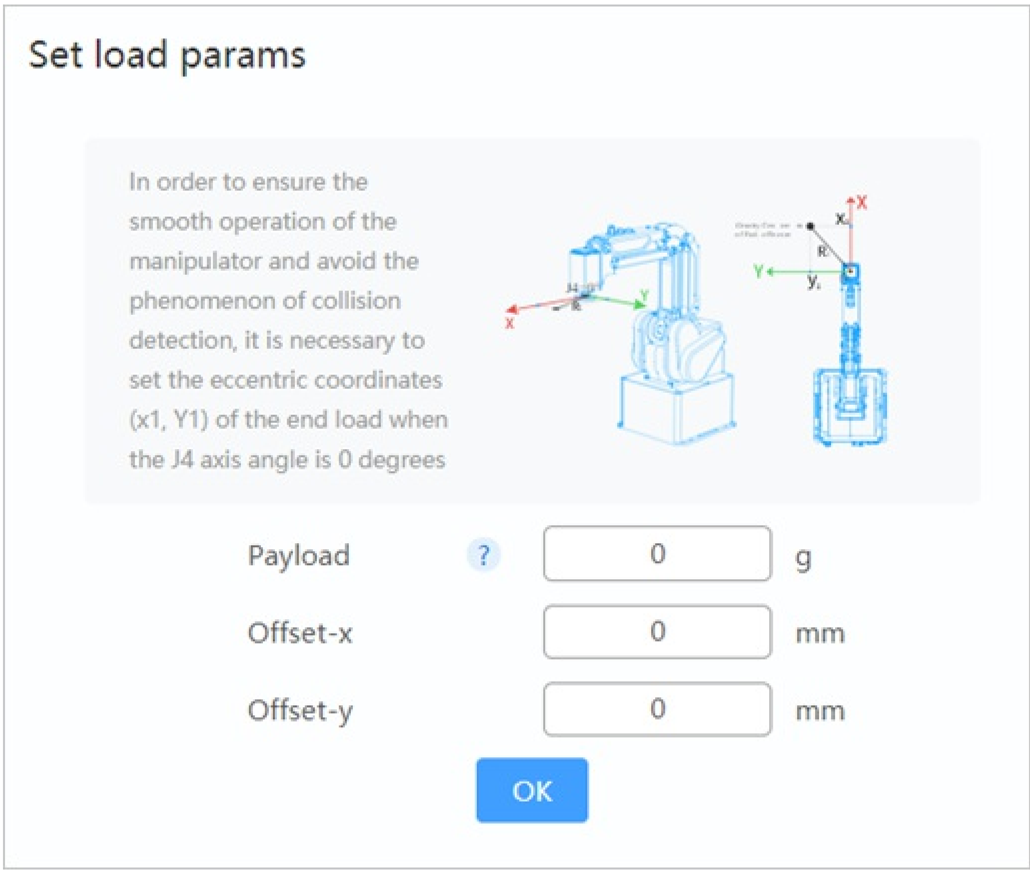
In this case, you can double click the alarm information to view the cause and solution, and click **Clear Alarm**.



Enabling status

The robot arm can work only in the enabled state.

- When the Enabling button is red (), the robot arm is in the disabled status. Click the button, and the "Set load params" window will pop up (the eccentric coordinate of the end load should be set when the J4 axis is 0°, and the load value should not exceed the maximum allowable load weight of the robot). After setting the parameters, click **OK** to enable the robot. Then the Enabling button moves to the right side, and the icon turns green ()



- When the Enabling button is green, the robot arm is in the enabled status. Click the button, and a confirmation box will be displayed. Click **OK**, and the robot arm starts to be disabled. After the robot is disabled, the Enabling button turns blue.
- When the Enabling button flashes blue, the robot is in the drag mode. In this case you cannot disable the robot or control the robot motion (run projects, jog, Run To specified postures, etc.) through the software.

Emergency stop button

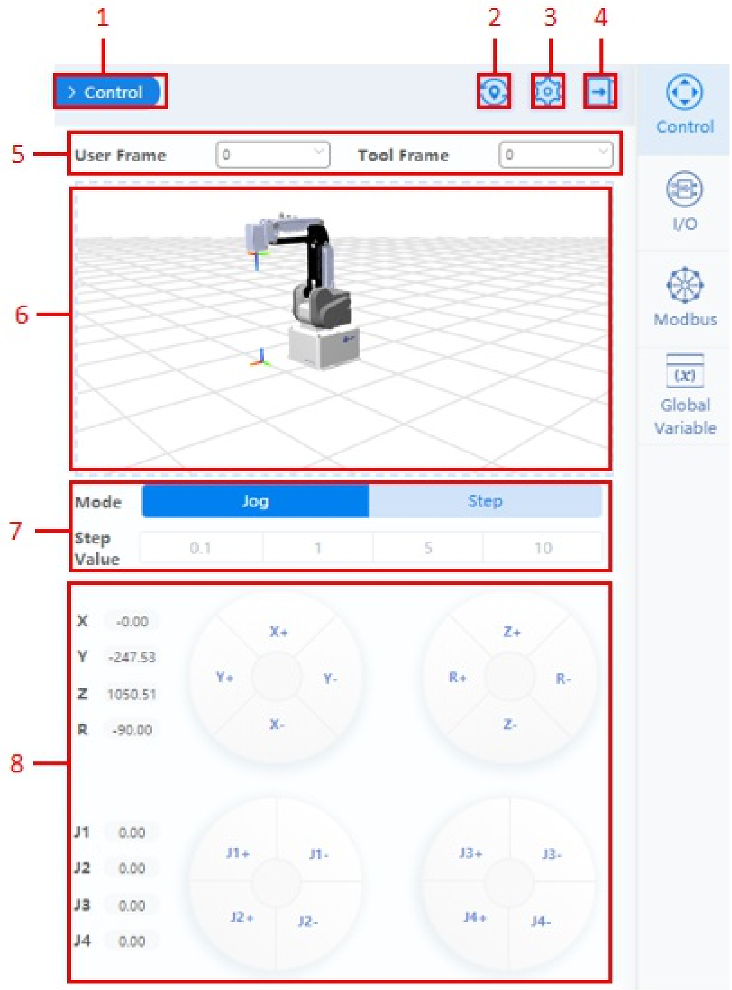
Once the emergency stop button is pressed, the robot arm will stop running and be powered off, and the emergency stop icon will turn red.

If you need to enable the robot arm again, please reset the emergency stop button, power on the robot and then enable it.

> NOTE
>
> If the physical emergency stop button is pressed, the icon of the emergency stop button on the software will not change. Before clearing the alarm, you need to reset the physical emergency stop

> button first (generally by rotating the button clockwise).

3.3 Control panel



No.	Description
1	Click to hide the panel. You can click **Control** in the right toolbar to display the panel.
2	Long-press the button to move the robot to its initial posture, which can be set in Basic.
3	Click to open Settings page.
4	Click to fold the control panel, and click it again to unfold the panel.
5	Click the drop-down list on the right of User coordinate system or Tool coordinate system to select an index of the coordinate system that you need to use.
6	Dispaly the movement of the robot arm in real time when you are jogging or running the robot arm.
7	Select the motion mode of the robot. • Jog: The robot keeps moving when you press and hold the jog button, and stops moving when the jog button is released. • Step: The specific value (such as 0.1) indicates that the robot moves this distance when you press the jog button. Long pressing the jog button can make the robot

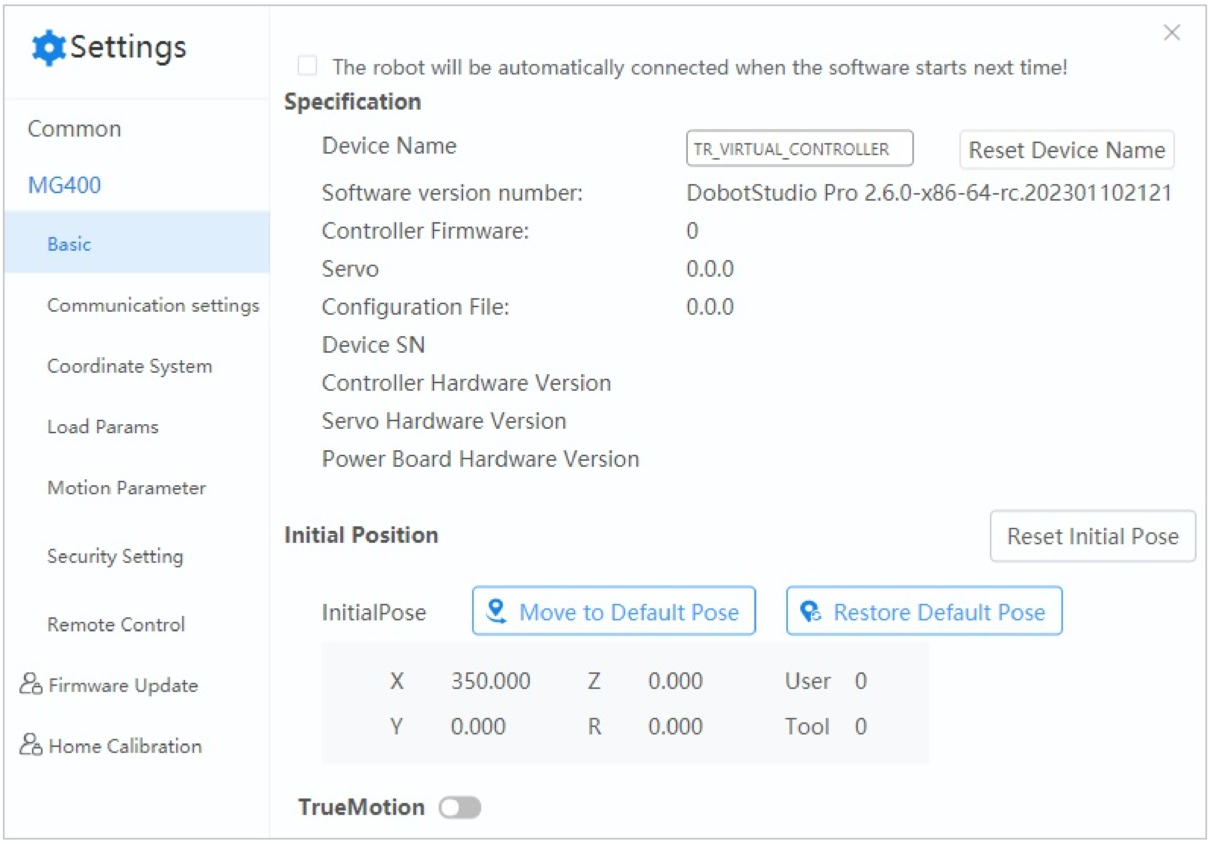
	moving continuously. In the Cartesian coordinate system, the unit of this value is mm, and 0.1 represents a displacement of 0.1mm for each step. In the joint coordinate system, the unit of this value is °, and 0.1 represents a displacement of 0.1° for each step.
8	Jog the operation panel. The upper part is jog buttons for Cartesian coordinate system, and the lower part is jog buttons for Joint coordinate system. • Take **X+**, **X-** as an example under Cartesian coordinate system: Click **X+**, **X-**: The robot arm moves along X-axis in the positive or negative direction. • Take **J1**+, **J1** as an example under Joint coordinate system: Click **J1**+, **J1**: The base motor of robot arm rotates in the positive or negative direction.

4 Settings

- 4.1 Basic settings
- 4.2 Communication settings
- 4.3 Coordinate System
  - 4.3.1 User coordinate system
  - 4.3.2 Tool coordinate system
- 4.4 Load parameters
- 4.5 Motion parameters
- 4.6 Security setting
- 4.7 Remote control
- 4.8 Hand calibration (M1 Pro)
- 4.9 Firmware update
- 4.10 Home calibration

4.1 Basic settings

The Basic Settings page is used to view the device specifications and set the robot posture.

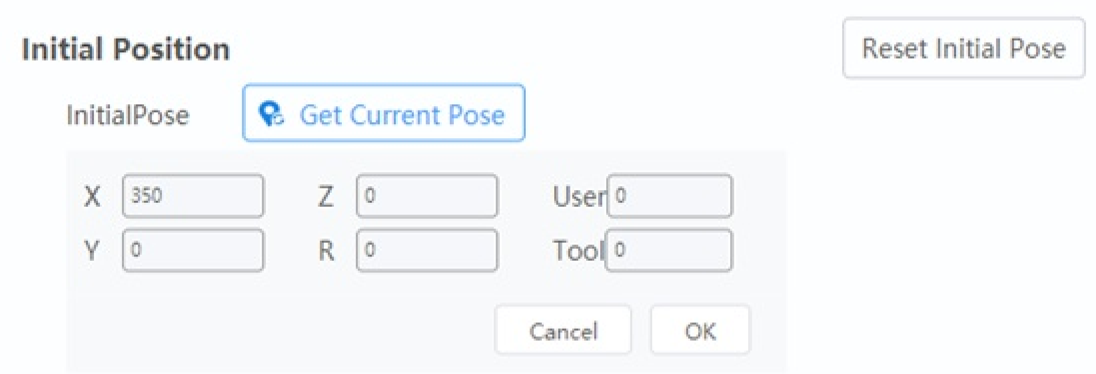


- Select **The robot will be automatically connected when the software starts next time**, and the software will try connecting to the current robot automatically when the software starts next time.
- You can click **Reset Device Name** to modify the device name.

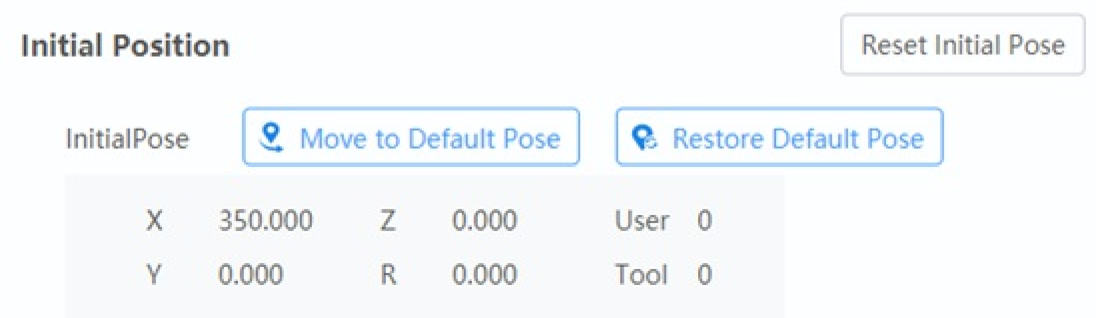
Initial posture

The initial posture is a self-defined posture, which is the home posture by default, namely, all joint angles are 0.

You can click **Reset Initial Pose** to modify the initial posture.



You can enter the angles of all joints, or move the robot to a specified posture and click **Get Current Pose** to obtain the current angles of all joints. After confirming all joint angles, click **OK** to update the initial posture.



You can long press **Move to Default Pose** to move the robot to the initial point. Click **Restore Default Pose** to recover the initial posture to the default posture.

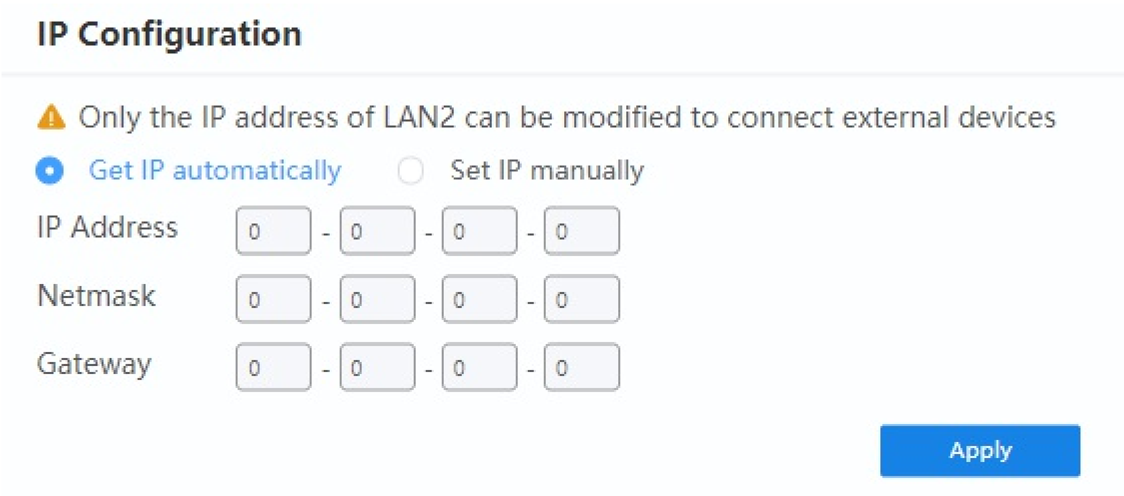
TrueMotion

The TrueMotion function can play back the trajectory more accurately and make the movement speed more stable. In scenarios with high requirements for trajectory precision and speed stability (such as gluing), you can enable this function.

4.2 Communication settings

IP Setting

The robot can communicate with external equipment thourgh the LAN2 interface which supports TCP, UDP and Modbus protocols. You can modify the 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. The default IP address is 192.168.2.6.



WiFi Setting

The robot system can communicate with external equipment through WiFi. You can modify the WiFi name and password and then restart the controller to make it effective. The default password is 1234567890.

4.3 Coordinate system

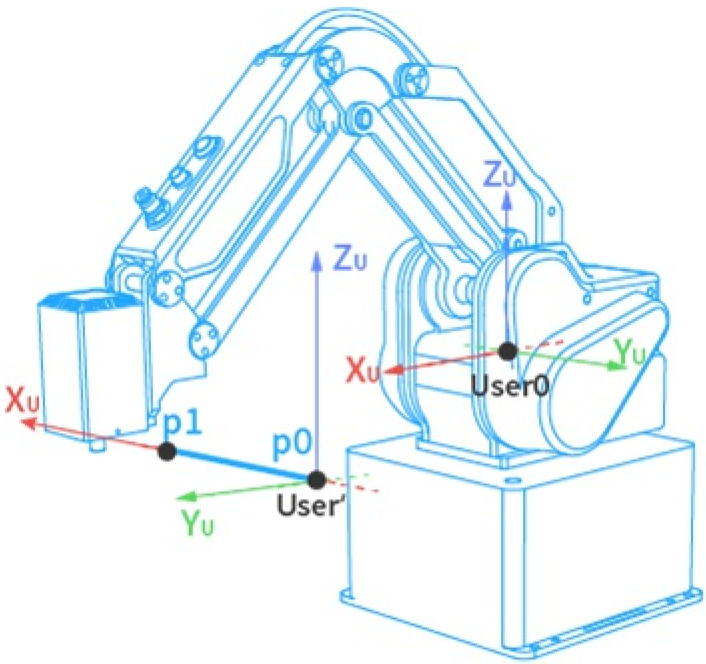
4.3.1 User coordinate system

When the position of workpiece is changed or a robot program needs to be reused in multiple processing systems of the same type, you can create a coordinate system on the workpiece so that all paths synchronously update with the user coordinates, which greatly simplifies teaching and programming.

DobotStudio Pro supports 10 user coordinate systems, of which the User coordinate system 0 is defined as the base coordinate system by default and cannot be changed.

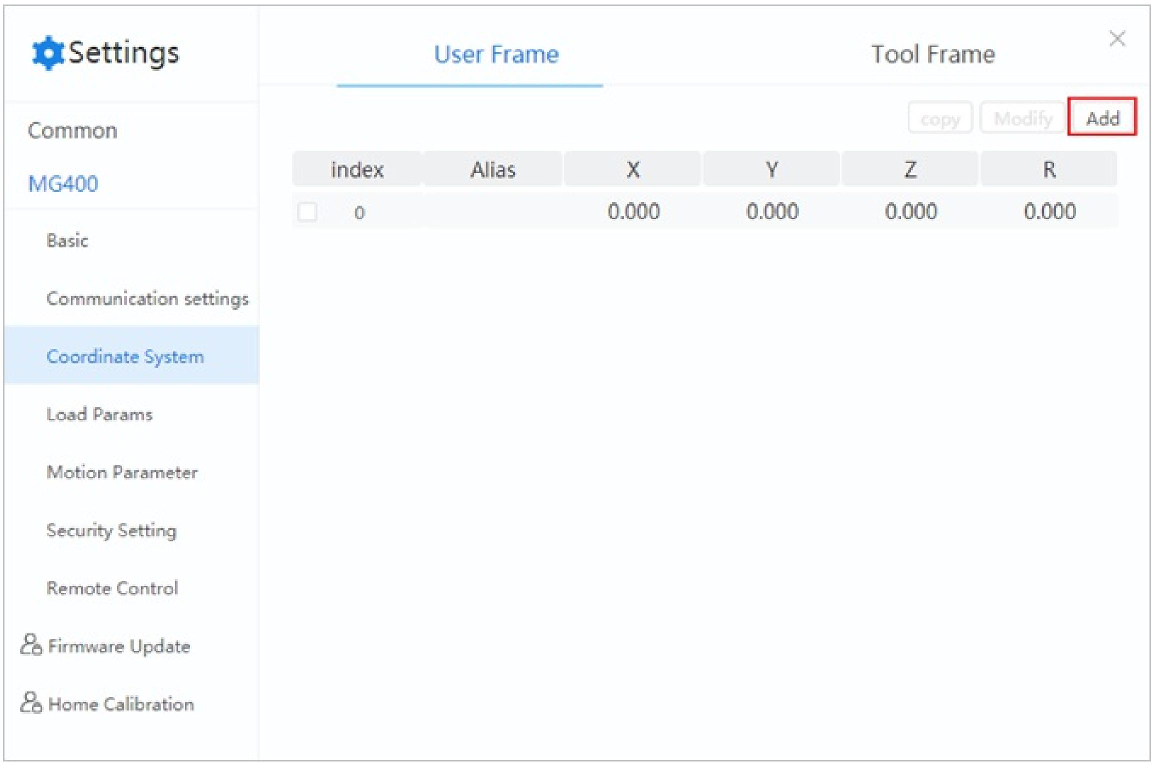
> NOTE
>
> When creating a user coordinate system, make sure that the reference coordinate system is the base coordinate system.

The four-axis user coordinate system is created by two-point calibration method. Move the robot to two random points: P0(x0, y0, z0) and P1(x1, y1, z1). Point P0 is defined as the origin and the line from point P0 to point P1 is defined as the positive direction of x-axis. Then the y-axis and z-axis can be defined based on the right-hand rule, as shown below.

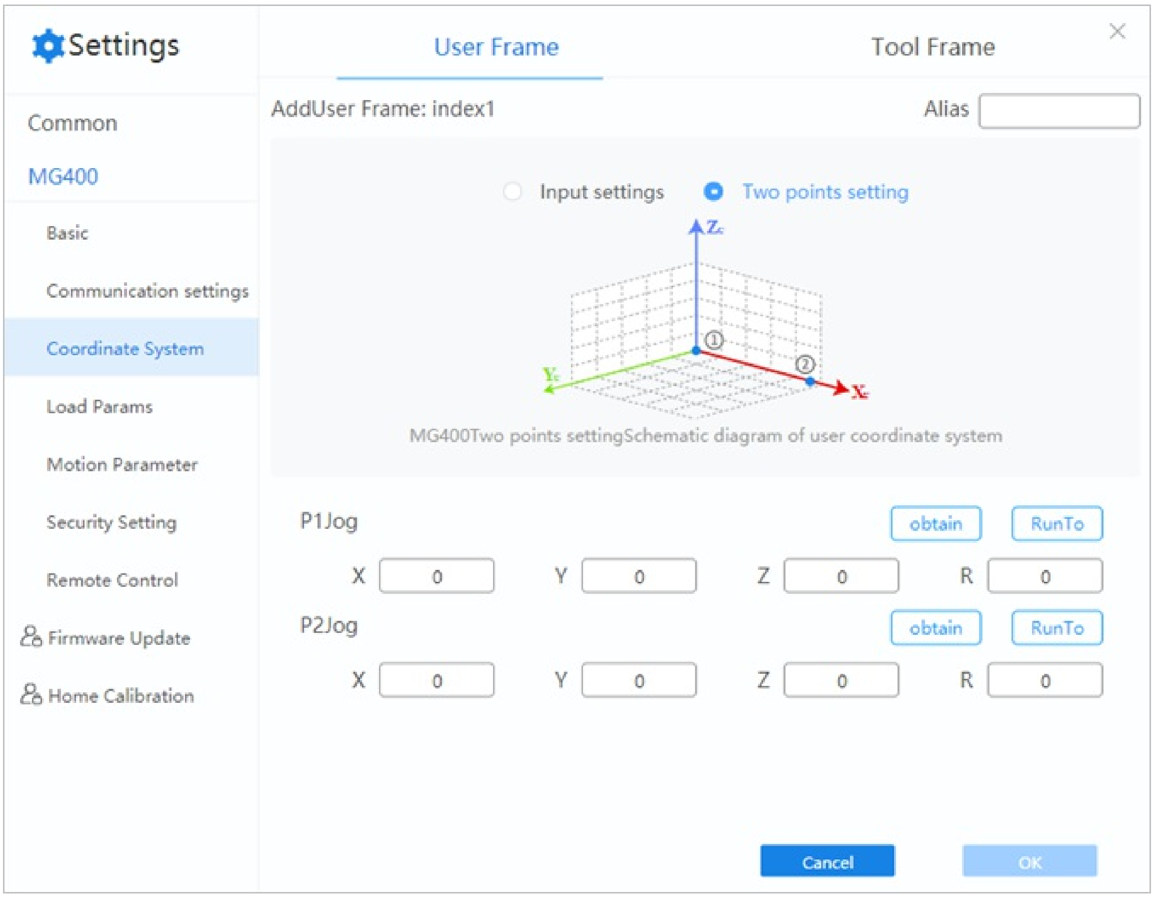


Creating user coordinate system

1. Click **Add**.



2. Select **Two points setting** in "Add User Frame: index1" page.



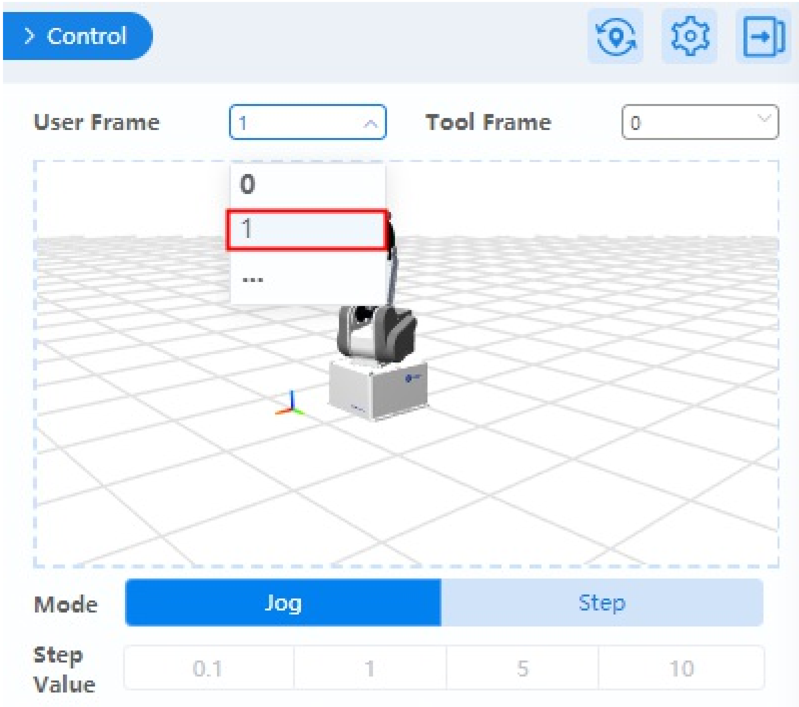
NOTE

- When creating a user coordinate system, make sure that the reference coordinate system is

> the base coordinate system, that is, the user coordinate system is 0 when you jog the robot.
> - Long pressing **Run To** can move the robot to the set points.

1. Jog the robot to the point P1 and click **obtain** on the P1 panel.
2. Jog the robot to the point P2 and click **obtain** on the P2 panel.
3. Click **OK**. The user coordinate system is created successfully.

Now you can select a user coordinate system in the control panel and jog the robot arm.



> NOTE
> When creating or modifying a user coordinate system, you can also select **Input settings** and directly enter X, Y, Z, R values, then click **OK**.

Other operations

- Modify a coordinate system: Select a coordinate system and click **Modify**. The procedure to modify a coordinate system is the same as to add a coordinate system.
- Copy a coordinate system: Select a coordinate system and click **copy** , and you will create a new coordinate system the same as the selected one.

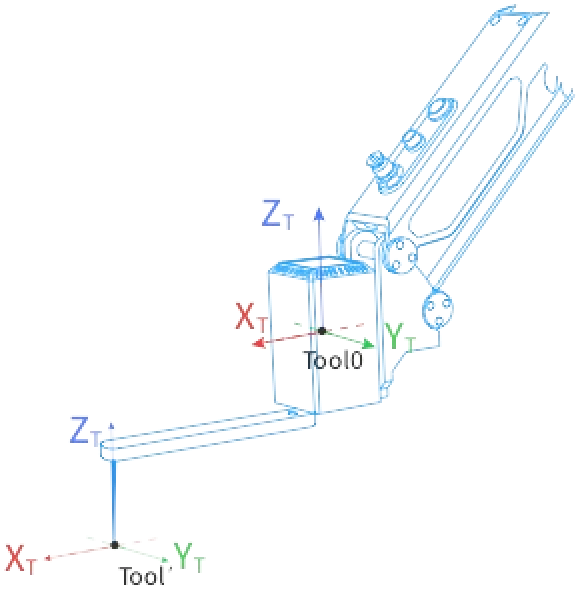
4.3.2 Tool coordinate system

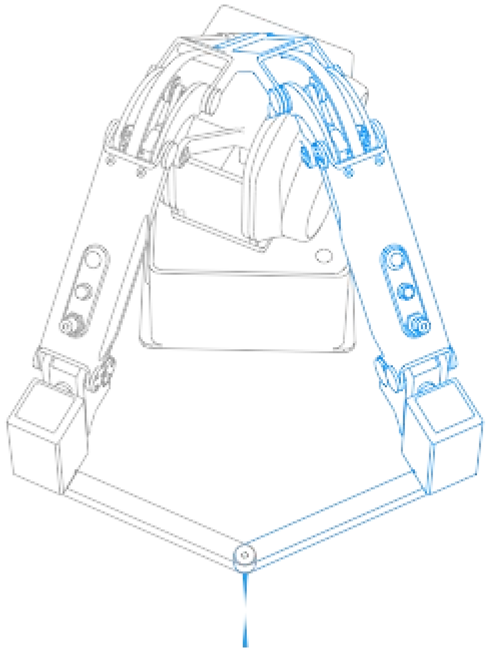
When an end effector such as welding gun or gripper is mounted on the robot, the tool coordinate system is required for programming and operating a robot. For example, when using multiple grippers to carry multiple workpieces simultaneously, you can set a tool coordinate system for each gripper to improve the efficiency.

DobotStudio Pro supports 10 tool coordinate systems. Tool coordinate system 0 is the base coordinate system which is located at the robot flange and cannot be changed.

> NOTE
> When creating a tool coordinate system, make sure that the reference coordinate system is the base coordinate system.

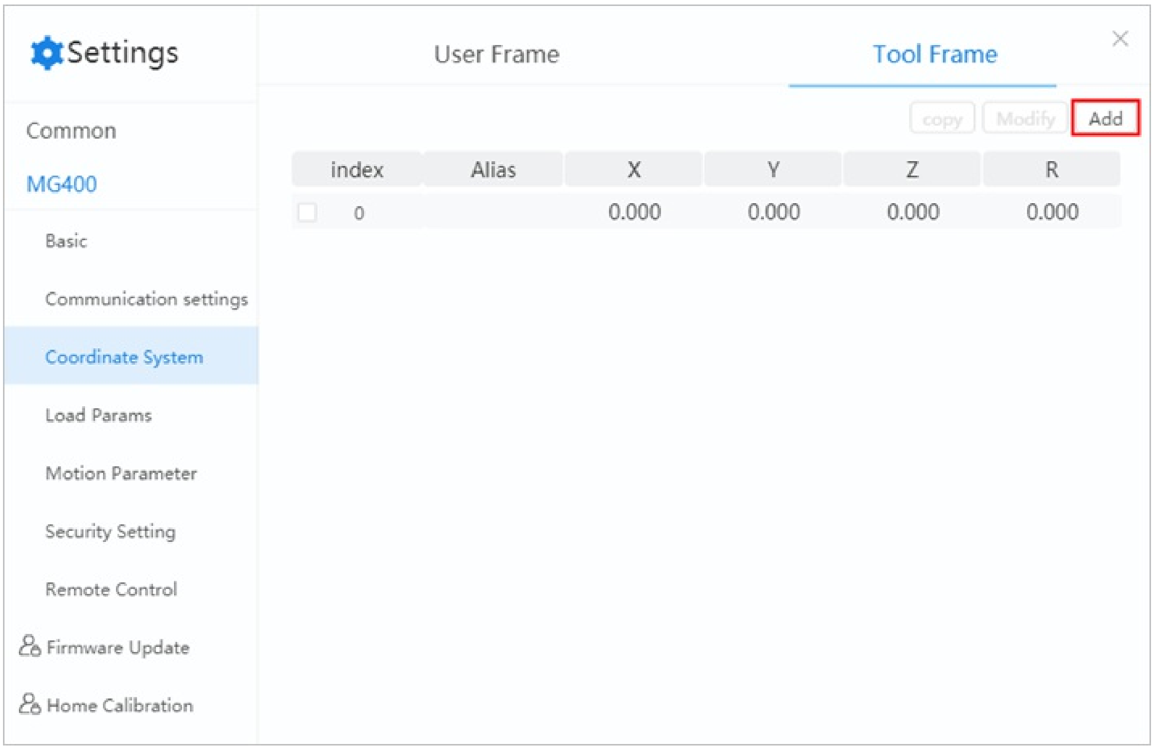
The four-axis tool coordinate system is created by two-point calibration method: After an end effector is mounted, adjust the direction of this end effector to make the TCP (Tool Center Point) align with the same point (reference point) in two different directions, for obtaining the position offset to generate a tool coordinate system, as shown below.



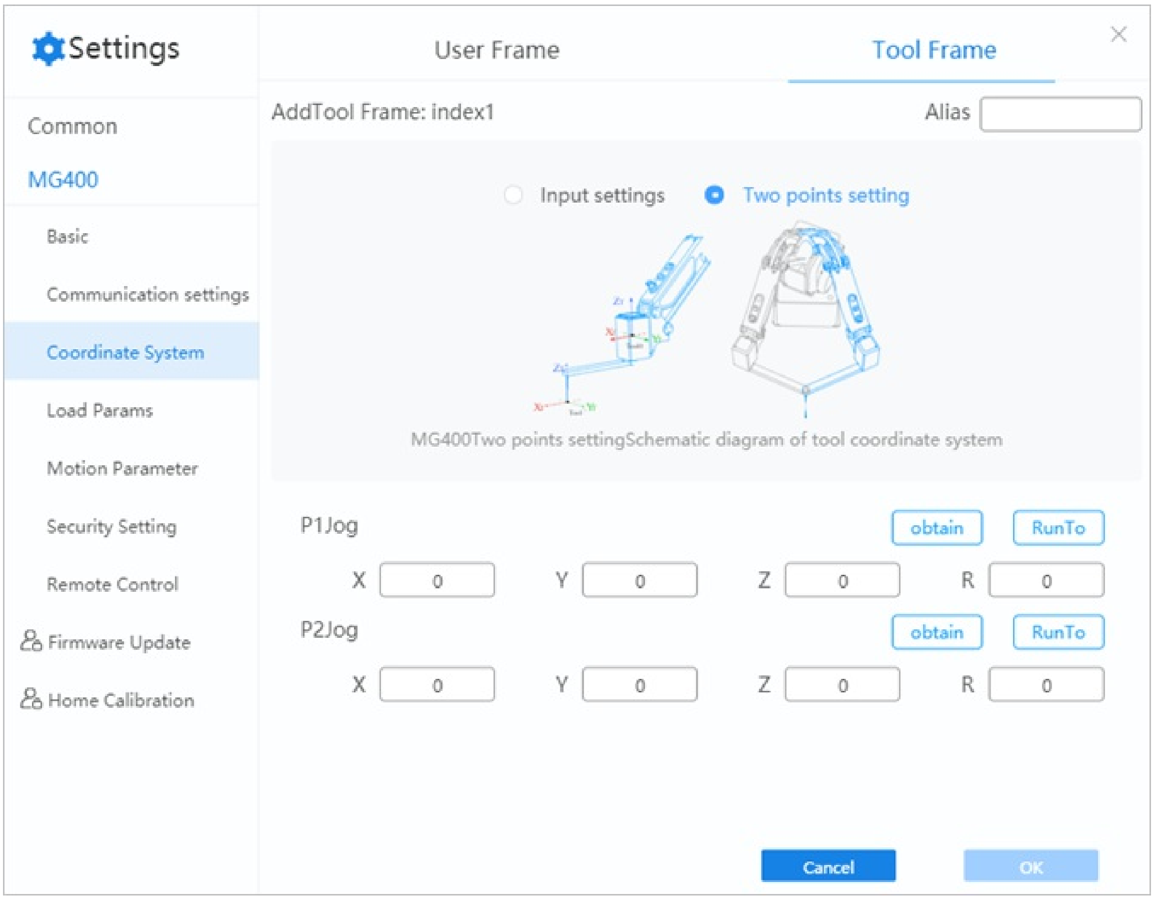


Creating tool coordinate system

1. Mount an end effector on the robot.
2. Click **Add**.



3. Select **Two points setting** in "Add Tool Frame: index1" page.



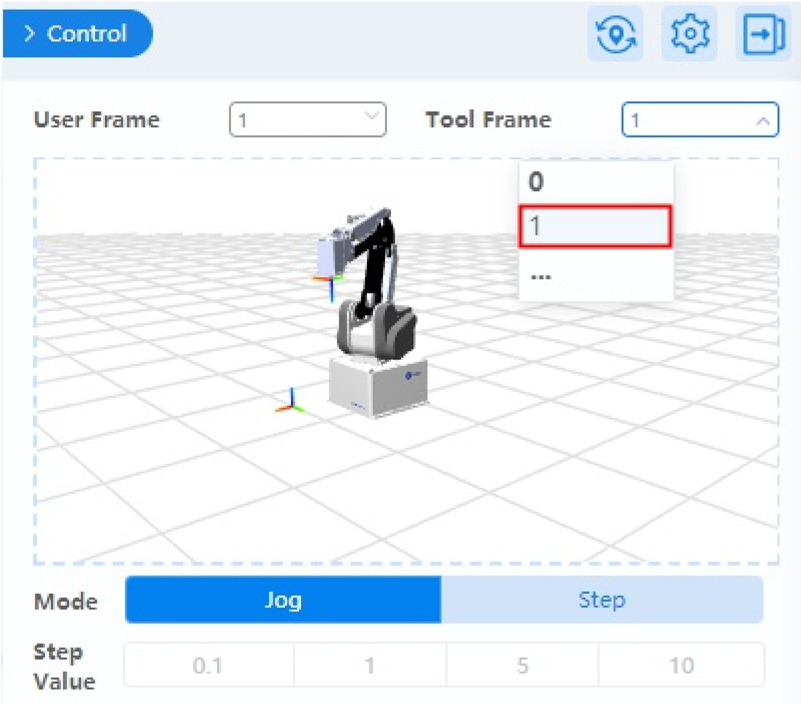
NOTE

- When creating a tool coordinate system, make sure that the reference coordinate system is

> the base coordinate system, that is, the tool coordinate system is 0 when you jog the robot.
> - Long pressing **Run To** can move the robot to the set points.

1. Jog the robot to the reference point in the first direction, then click **obtain** on the P1 panel.
2. Jog the robot to the reference point in the second direction, then click **obtain** on the P2 panel.
3. Click **OK**. The tool coordinate system is created successfully.

After adding or modifying a tool coordinate system, you can select a tool coordinate system in the control panel and jog the robot arm.



> NOTE
> When creating or modifying a Tool coordinate system, you can also select **Input settings**, modify X, Y, Z and R values and click **OK**.

Other operations

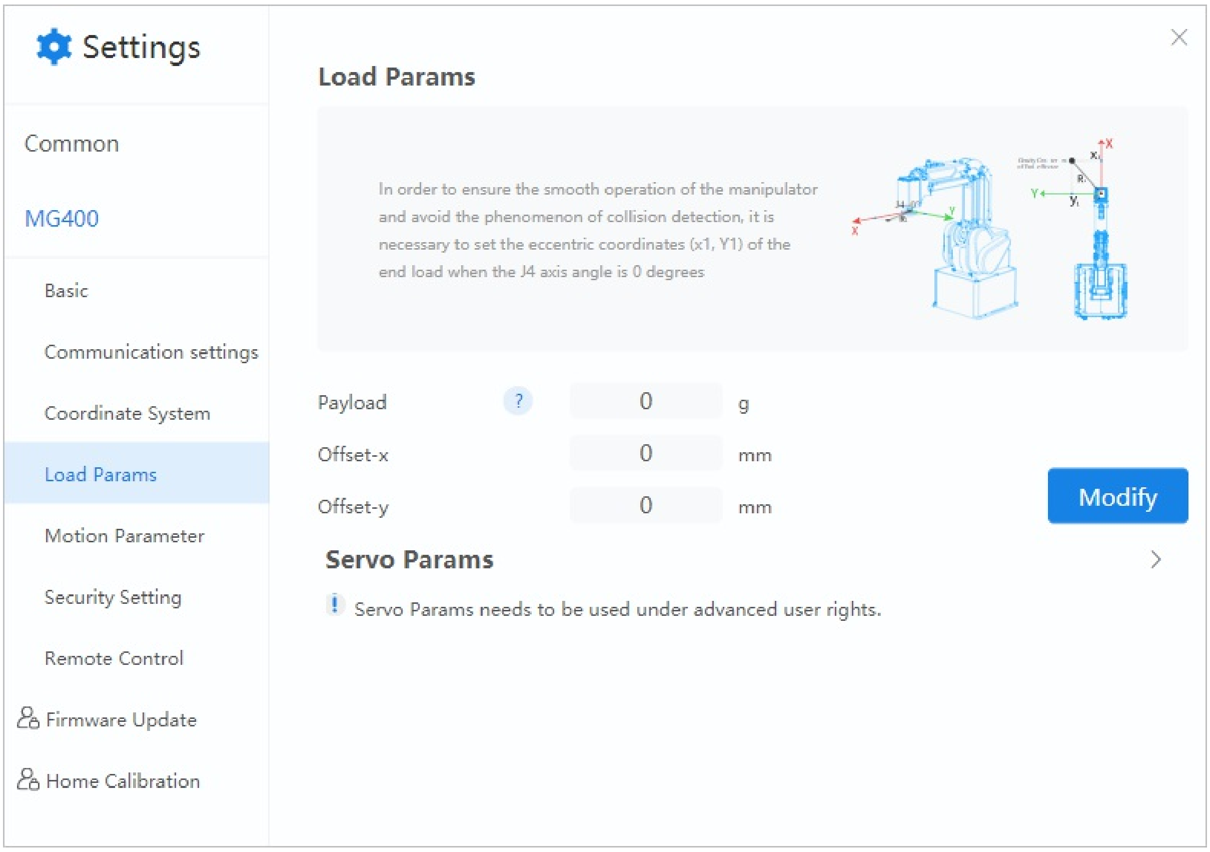
- Modify a coordinate system: Select a coordinate system and click **Modify**. The procedure to modify a coordinate system is the same as to add a coordinate system.
- Copy a coordinate system: Select a coordinate system and click **copy** , and you will create a new coordinate system the same as the selected one.

4.4 Load parameters

To ensure optimum robot performance, it is important to make sure the load and eccentric coordinates of the end effector are within the maximum range for the robot, and that Joint 4 does not become eccentric. Setting load and eccentric coordinates improves the motion of robot, reduces vibration and shortens the operating time.

NOTE

- Every time you enable the robot, a "Set load params" window will pop up which requires you to set the load parameters. The parameters you set will be synchronized to the "Load Params" page.
- The servo parameter is an advanced function. Please use it under the guidance of technical support.



Click **Modify** to modify the load parameters.

- You need to set the eccentric coordinate of the load when J4 axis is 0°.
- The load weight includes the weight of the end effector and workpiece, which should not exceed the maximum load of the robot arm.

NOTICE

> Incorrect load weight may lead to collision detection anomaly alarm or cause the robot uncontrolled when being dragged.

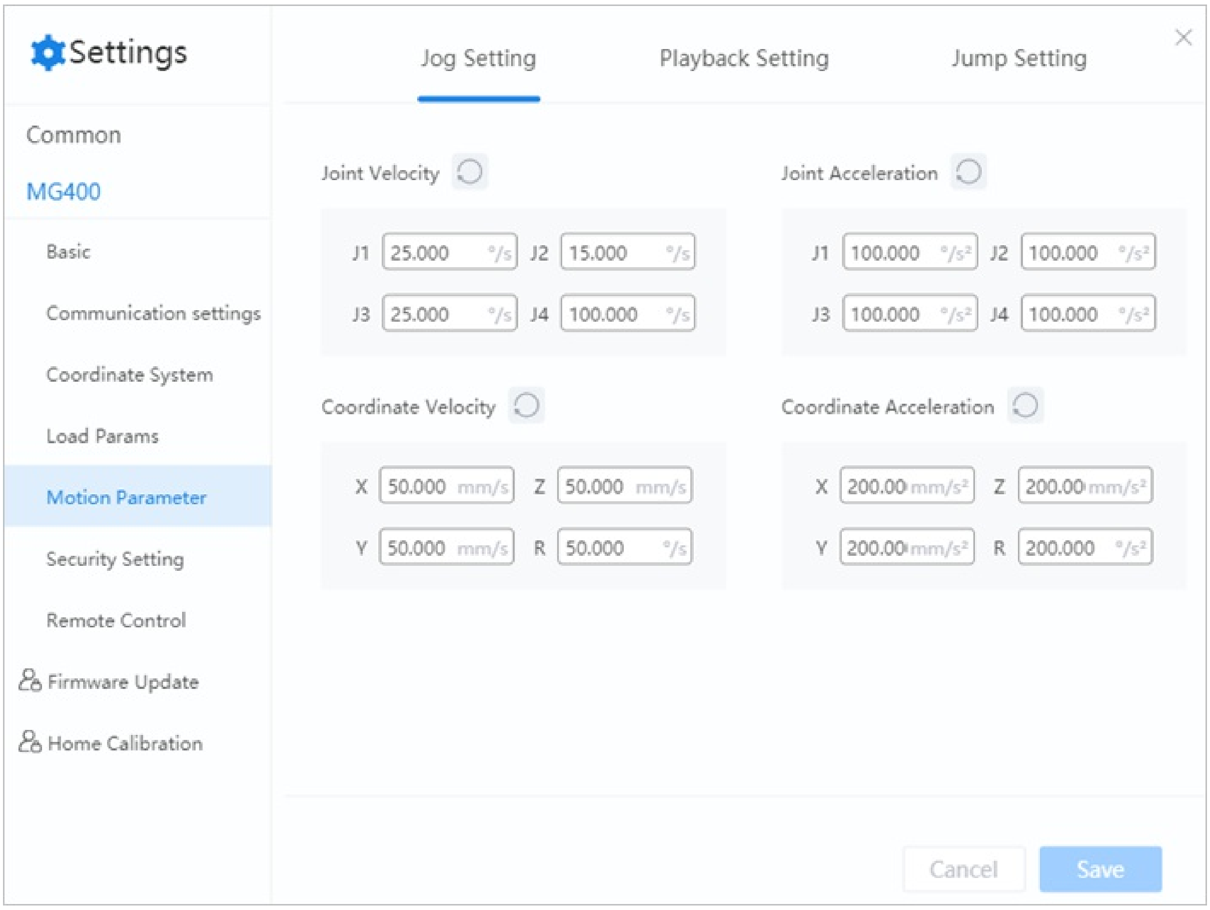
After setting the parameters, click **OK**.

Please set load and eccentric coordinates properly. Otherwise, it may cause errors or excessive shock, and shorten the life cycle of parts.

4.5 Motion Parameters

Jog Setting

You can set the maximum speed and acceleration in the Joint coordinate system and Cartesian coordinate system. Click **Save** after setting the parameters.

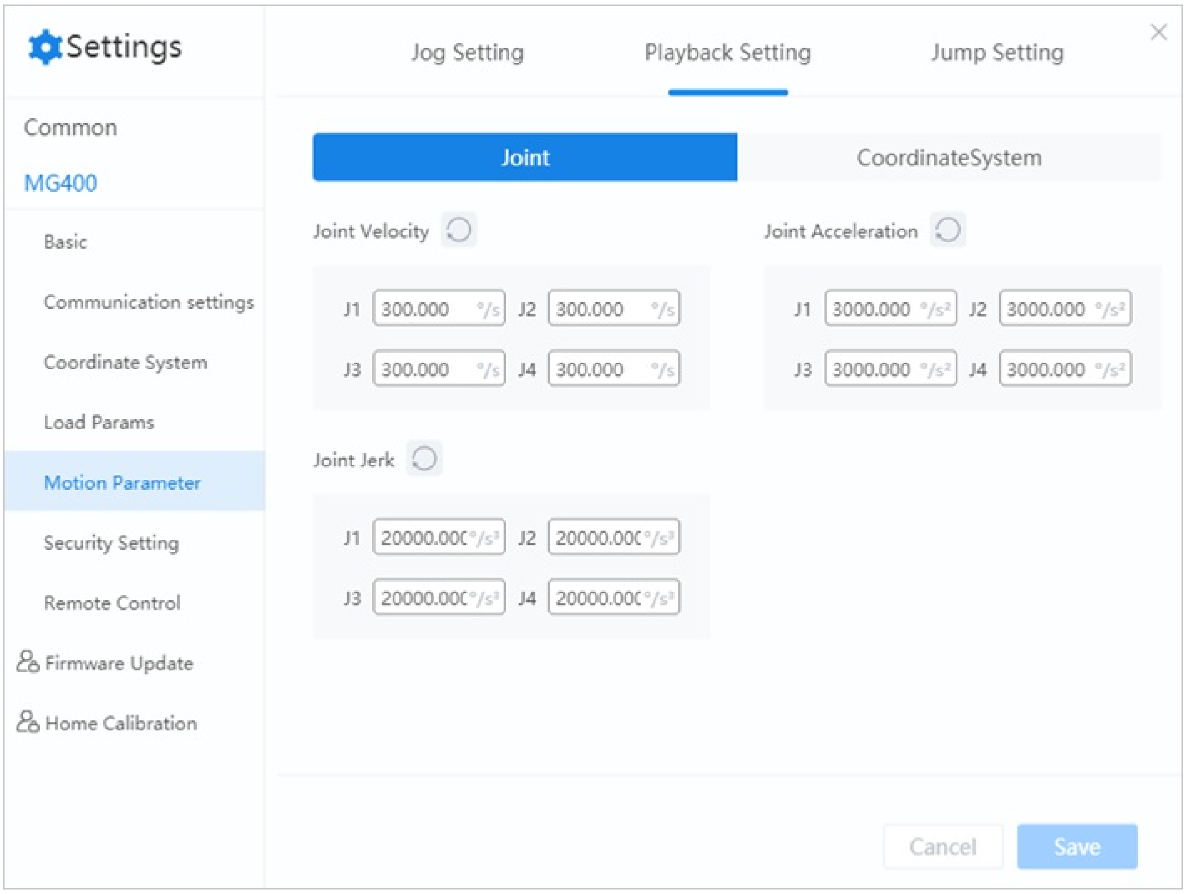


Actual robot speed/acceleration = set speed/acceleration × global speed ratio.

Clicking  will restore all the values in the corresponding module to the default values.

Playback Setting

You can set the velocity, acceleration and jerk in the Joint coordinate system and Cartesian coordinate system. Click **Save** after setting the parameters.



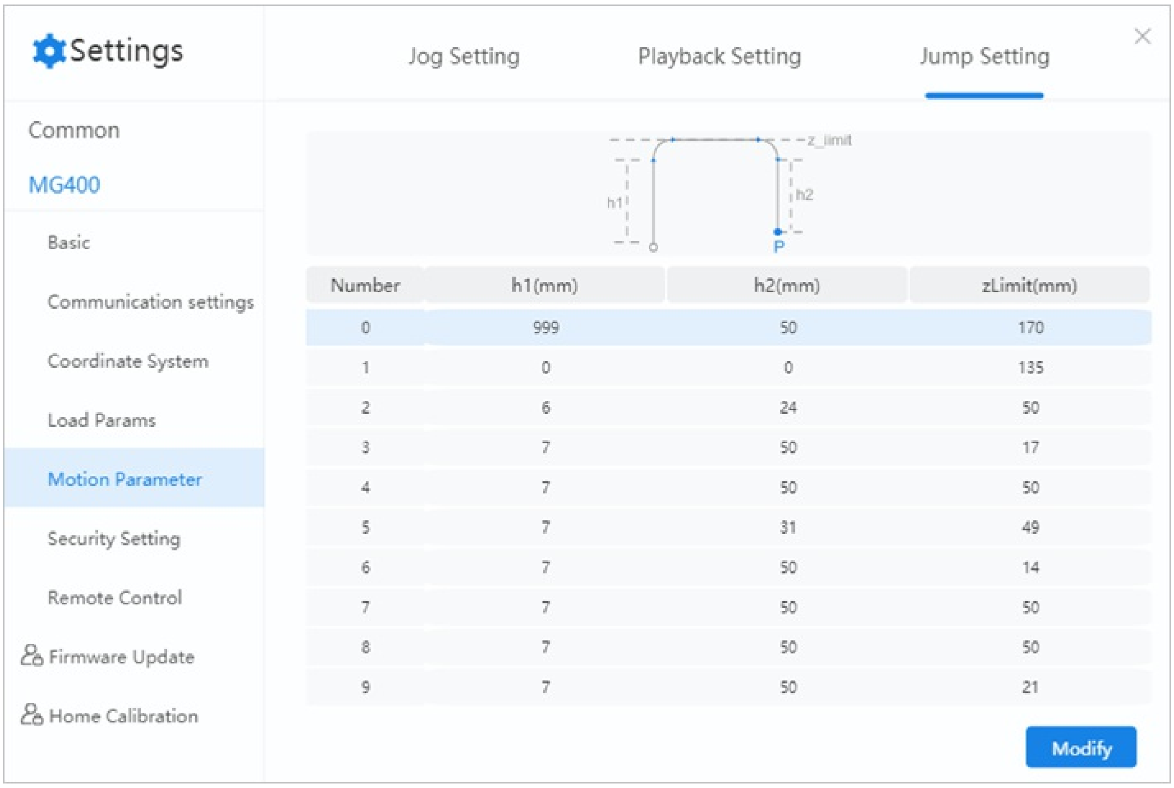
Actual robot speed/acceleration = set speed/acceleration × global speed ratio × set percentage in speed commands when programming.

Clicking  will restore all the values in the corresponding module to the default values.

Jump Setting

If the motion mode is Jump during playback, you need to set Start height **(h1)**, End height **(h2)** and **zLimit**.

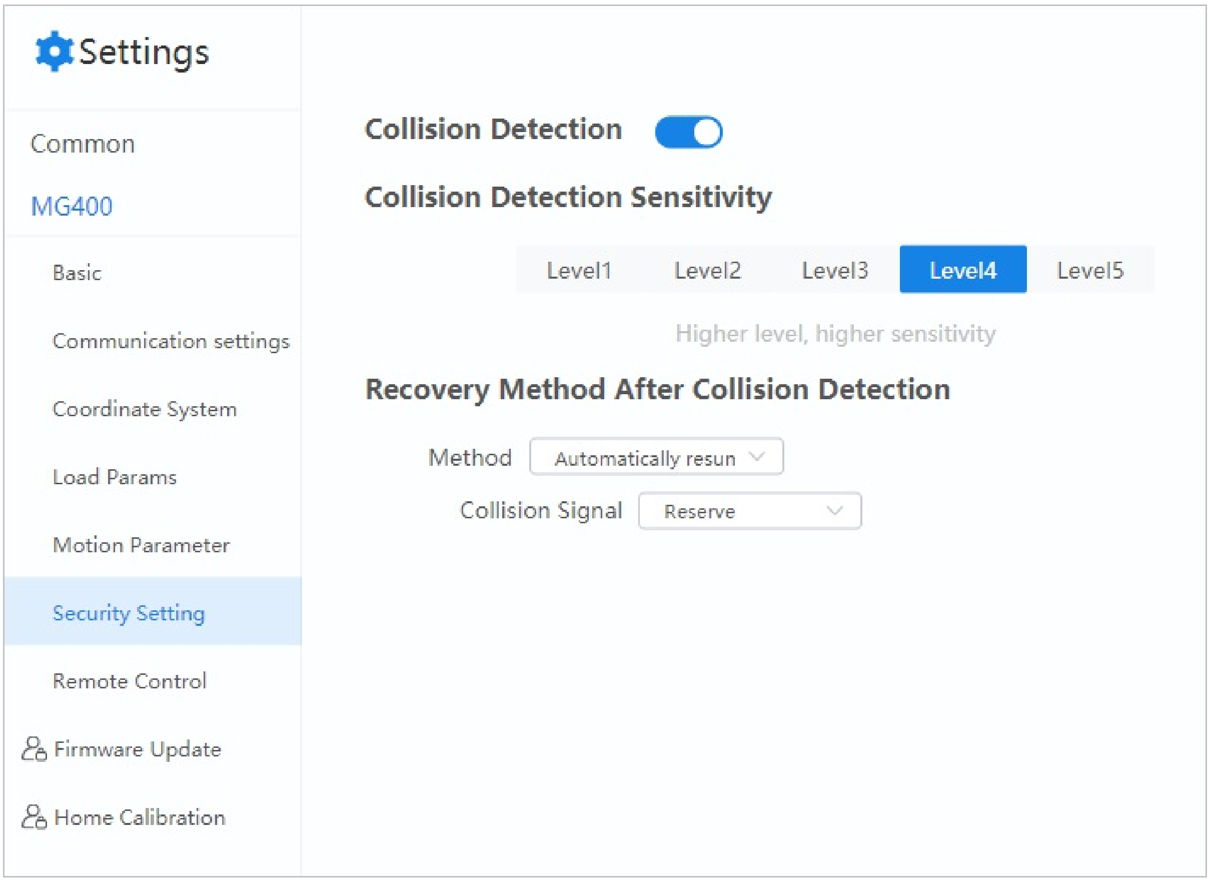
You can set 10 sets of Jump parameters. Selecting a set of parameters and clicking **Modify** (or double clicking a set of parameters) can modify Jump parameters.



You can select one or more sets of the parameters to call them during programming (use Arch to set the index when calling Jump parameters).

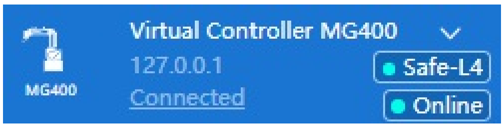
4.6 Security setting

Collision detection is mainly used for reducing the impact on the robot to avoid damage to the robot or external equipment. If collision detection is activated, the robot arm will suspend running automatically when hitting an obstacle.



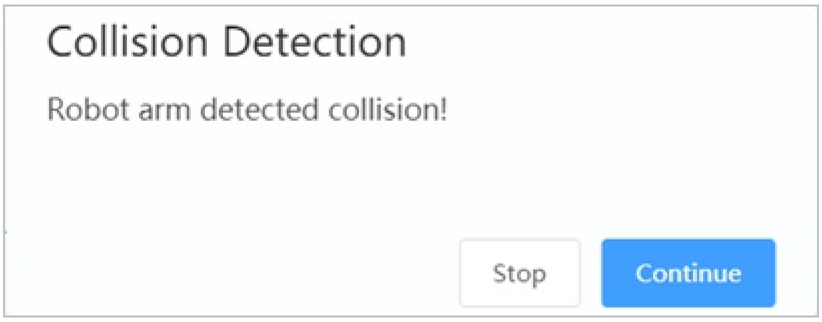
If you configures **Collision Signal**, the robot arm will trigger the corresponding DO port after it stops caused by collision.

After you enable **Collision Detection**, the safety level will be displayed in the connection panel in the top toolbar.

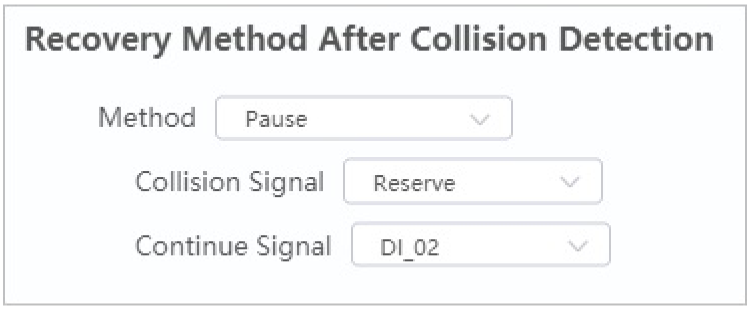


DobotStudio Pro supports three handling modes after the robot stops caused by a collision during playback.

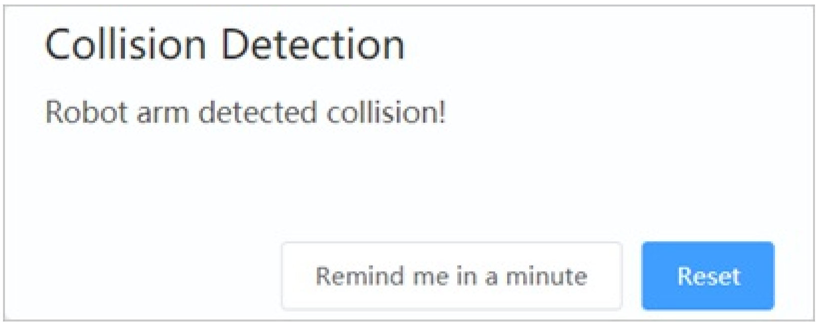
- Automatically resume after 5s: The robot resumes running automatically after 5 seconds.
- Pause: When a collision is detected, a prompt window will pop up, and the robot arm pauses running. You need to resume the operation or stop running the project through the software interface or remote I/O.



When you select **Pause**, you can configure the continue signal, which is the same as the continue signal in Remote Control by default. After modification, it will be automatically synchronized to the remote IO configuration, and vice versa.



- Stop: When a collision is detected, a prompt window will pop up, and the robot arm stops running. In this case, you need to resolve the cause of the collision and click **Reset**. If you need to operate the software to resolve the collision cause, click **Remind me in a minute** to temporarily close the pop-up window (a pop-up message will be displayed again in one minute).



4.7 Remote control

External equipment can send commands to a robot (control and run a taught program file) in different remote control modes, such as remote I/O mode and remote Modbus mode.

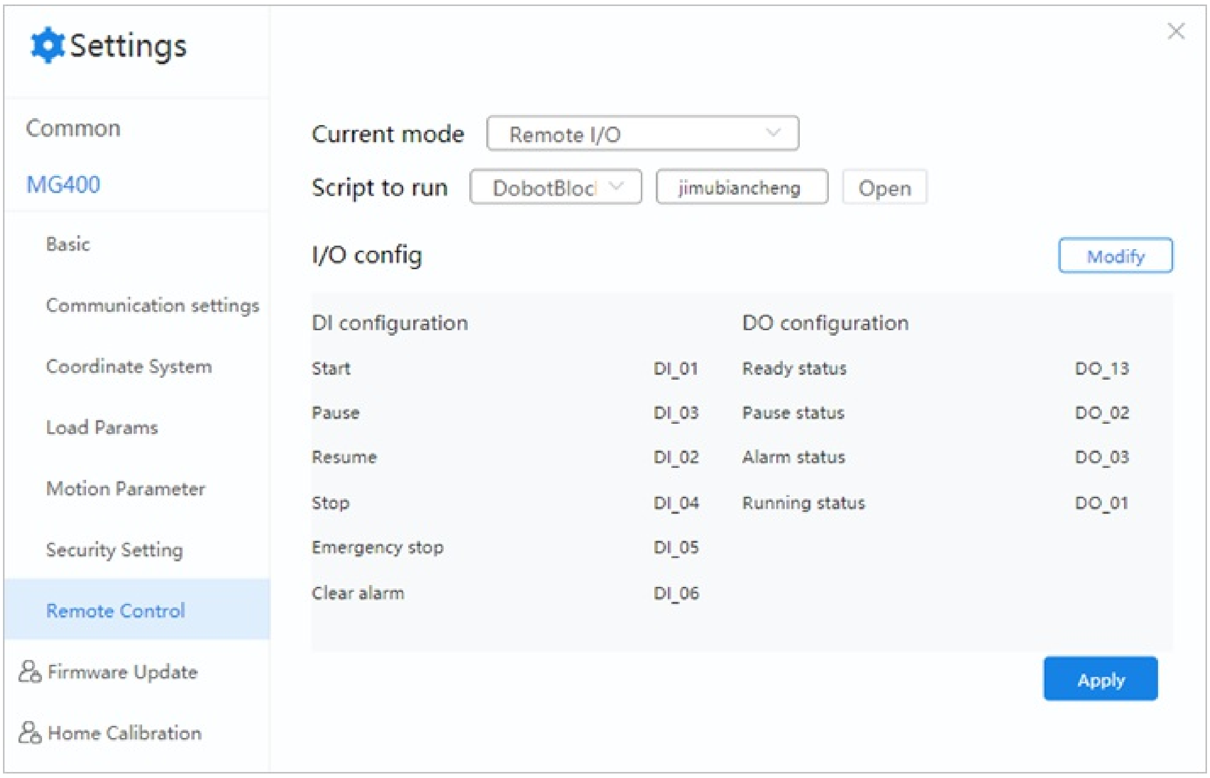
> NOTE
>
> - You do not need to restart the robot control system when switching remote control mode.
> - No matter what mode the robot control system is in, the emergency stop switch is always effective.
> - If the robot is running in the remote control mode, the project will stop running automatically when you switch to other working modes.

Online mode

It is the default control mode. You can control the robot arm through DobotStudio Pro.

Remote I/O

External equipment can control the robot arm in the remote I/O mode.



The specific I/O interface definition of the control system is shown in the figure above. You can click **Modify** to edit it.

The procedure of running the project in the remote I/O mode is shown below.

Prerequisite

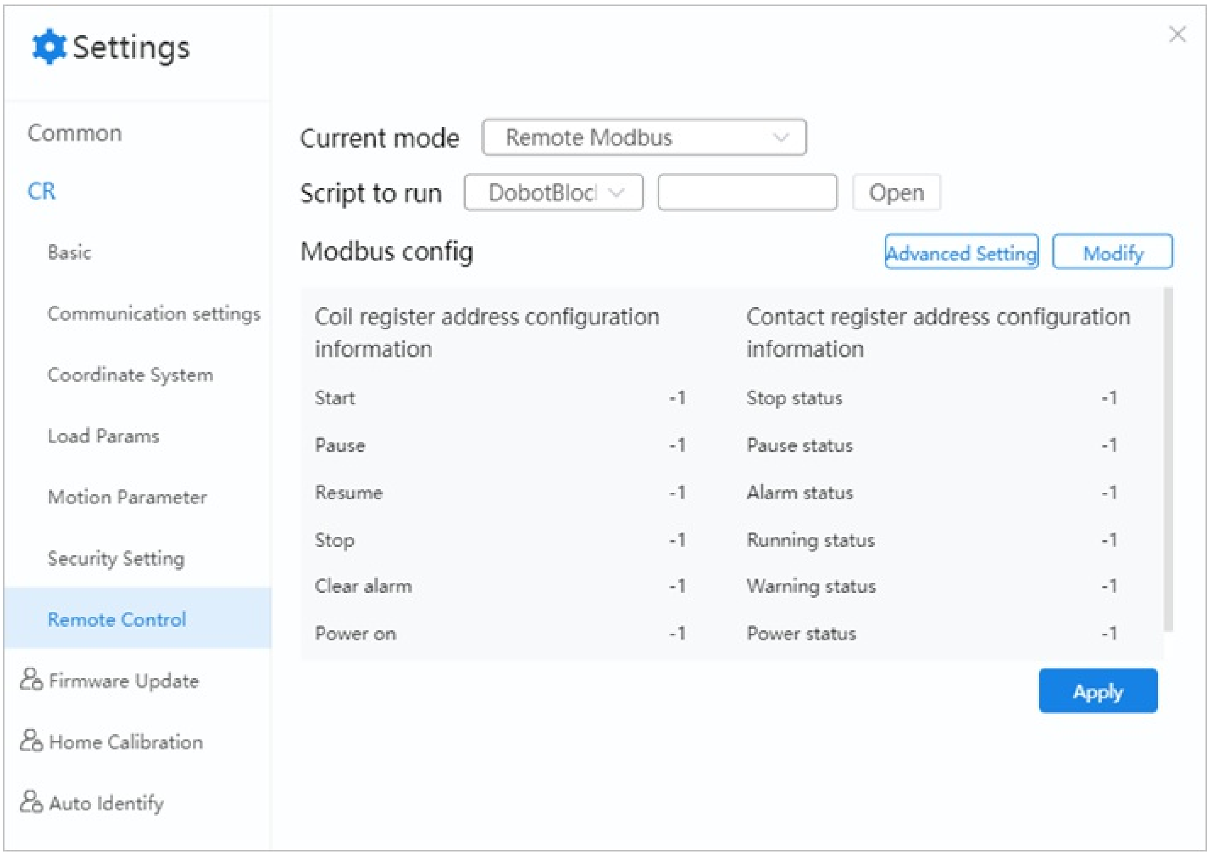
- The project to be running in the remote mode has been prepared.
- The external equipment has been connected to the robot arm by I/O interface.
- The robot arm has been powered on.

Procedure

1. Set **Current mode** to **Remote I/O**, and select an offline project (block program or script) for running.
2. Click **Apply**. Now the robot arm enters remote IO mode. Only the emergency stop command is available.
3. Trigger the starting signal on the external equipment. The robot will move according to the selected project file.
4. If the stop signal is triggered, the robot arm will stop moving and be disabled.

Remote Modbus

External equipment can control the robot arm in the remote Modbus mode.



The specific functions of Modbus registers are shown above. You can click **Modify** to edit it.

The procedure of running the project in the remote Modbus mode is described below.

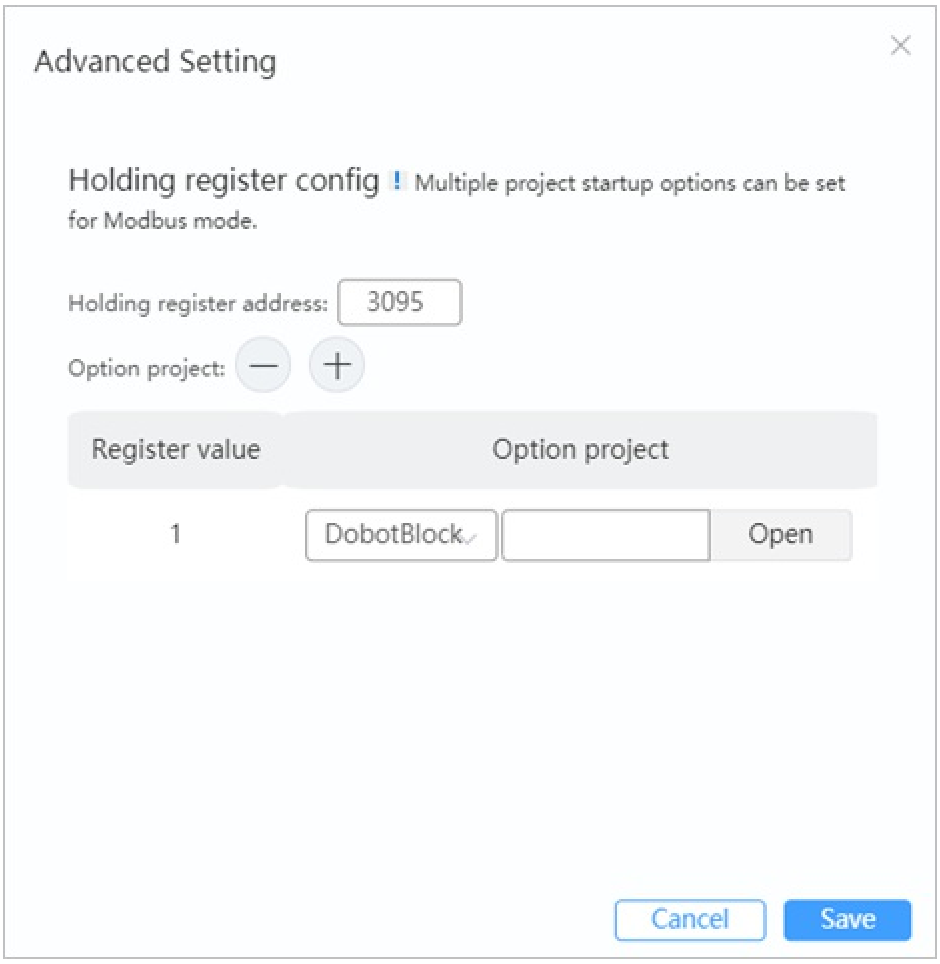
Prerequisites

- The project to be running in the remote mode has been prepared.
- The robot has been connected to the external equipment through the LAN2 interface. You can connect them directly or through a router. The IP address of the robot and the external equipment must be within the same network segment without conflict. The default IP address is 192.168.2.6. You can configure the IP address in Communication settings.
- The robot arm has been powered on.

Procedure

1. Set **Current mode** to **Remote-Modbus**, and select an offline project (block program or script) for running.

2. If you need to start multiple different projects through Modbus, click **Advanced Setting**. In Advanced Setting, you can set **Hold register address** of the option project and configure the list of option projects, as shown in the following figure.



3. Click **Apply**. Now the robot arm enters remote Modbus mode. Only the emergency stop command is available.
4. Trigger the starting signal on the external equipment. The robot will move according to the selected

project file.

5. If the stop signal is triggered, the robot arm will stop moving and be disabled.

TCP/IP secondary development

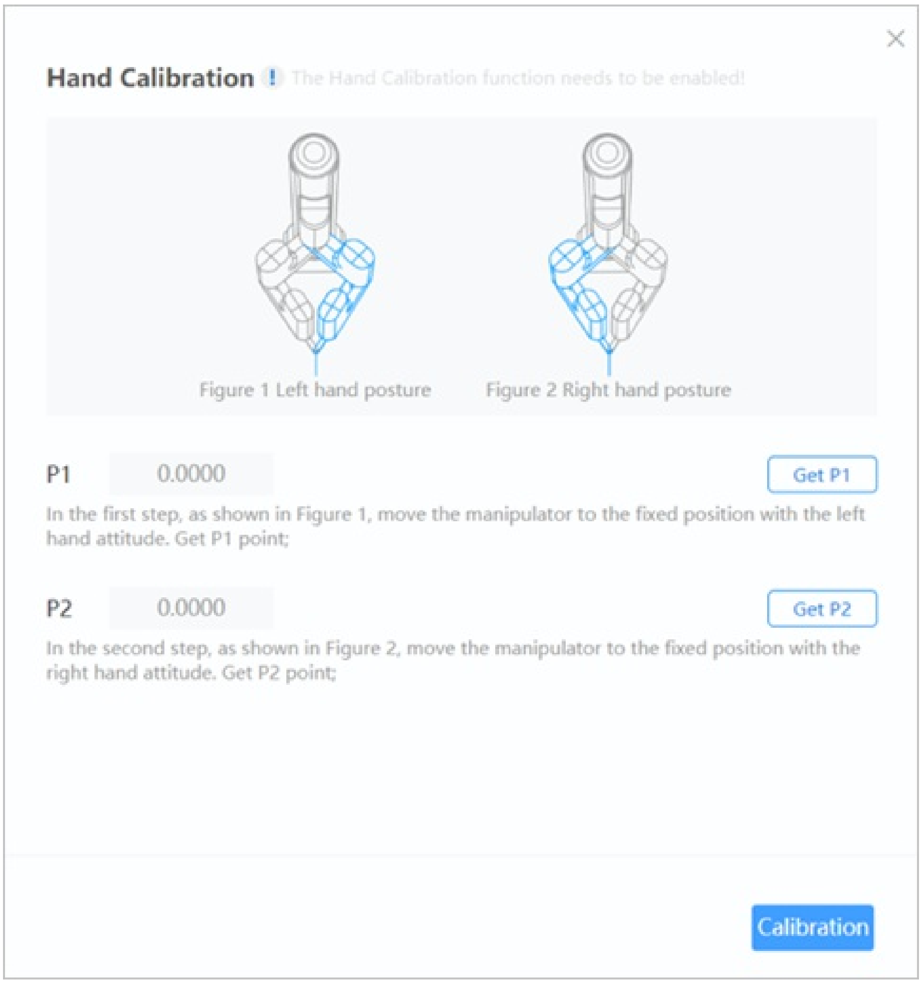
This mode is for users to develop control software based on TCP. If you need to develop the software, refer to Dobot TCP/IP Protocol (placed in Github)

4.8 Hand Calibration (M1 Pro)

When using M1 Pro, you need to perform hand calibration if higher absolute precision is required.

In hand calibration, you need to move the robot to the same point with different arm orientations. In this process the J2 coordinates should be axisymmetric. If not, the absolute precision will be decreased. So it is necessary to make the J2 coordinates axisymmetric by compensating the joint angel of J2 to improve the absolute precision.

> NOTE
> The home calibration function can be used after you enter the manager password (default password: 888888).



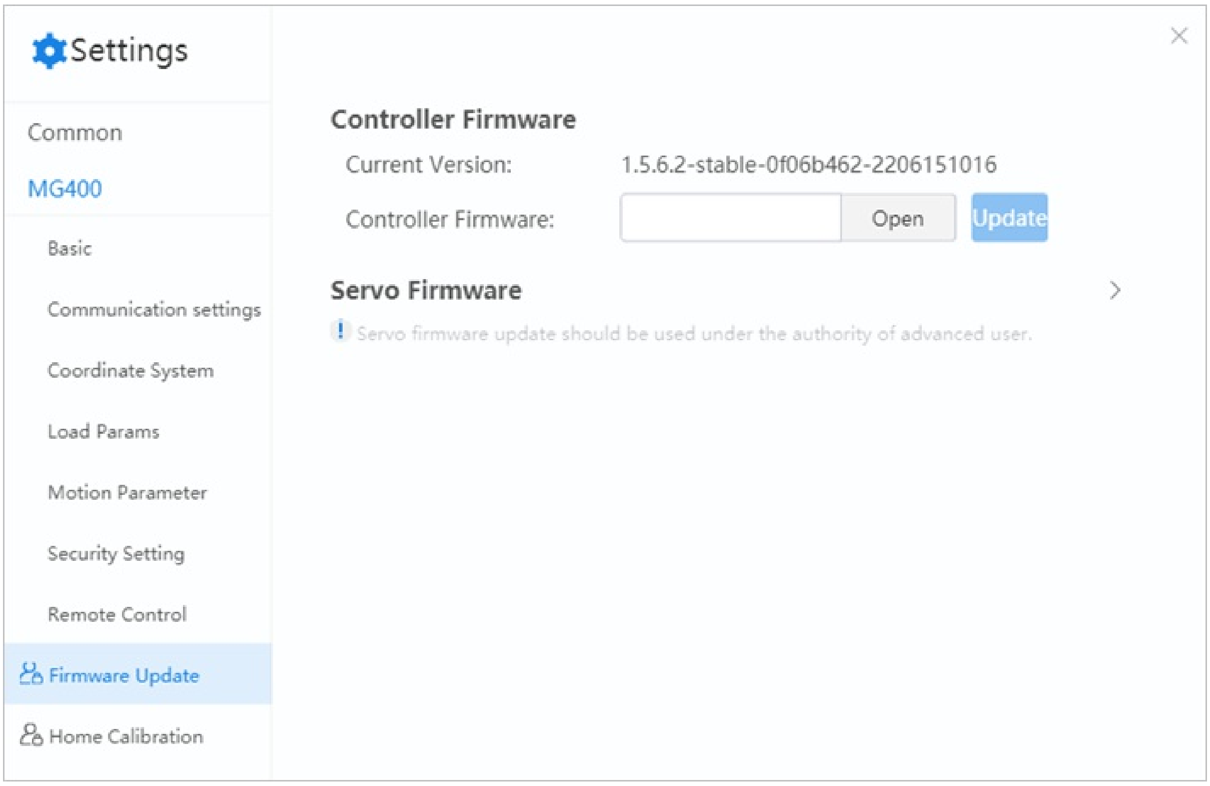
1. Jog or drag the robot to a point in left-hand direction, then click **Get P1**.
2. Jog or drag the robot to the same point as Step 1 in right-hand direction, then click **Get P2**.
3. Click **Calibration**.

4.9 Firmware update

When the controller firmware needs to be updated, you can import the latest firmware on the **Firmware Update** page.

⚠ DANGER

During the updating, DO NOT perform any other operations on the robot arm or power it off to avoid it in an abnormal state. Otherwise, it may cause damage to devices or personal injury.



Click **Open** to import the latest controller firmware from local and click **Update**. The controller firmware will be updated automatically. Reboot the controller and reconnect it according to the pop-up window.

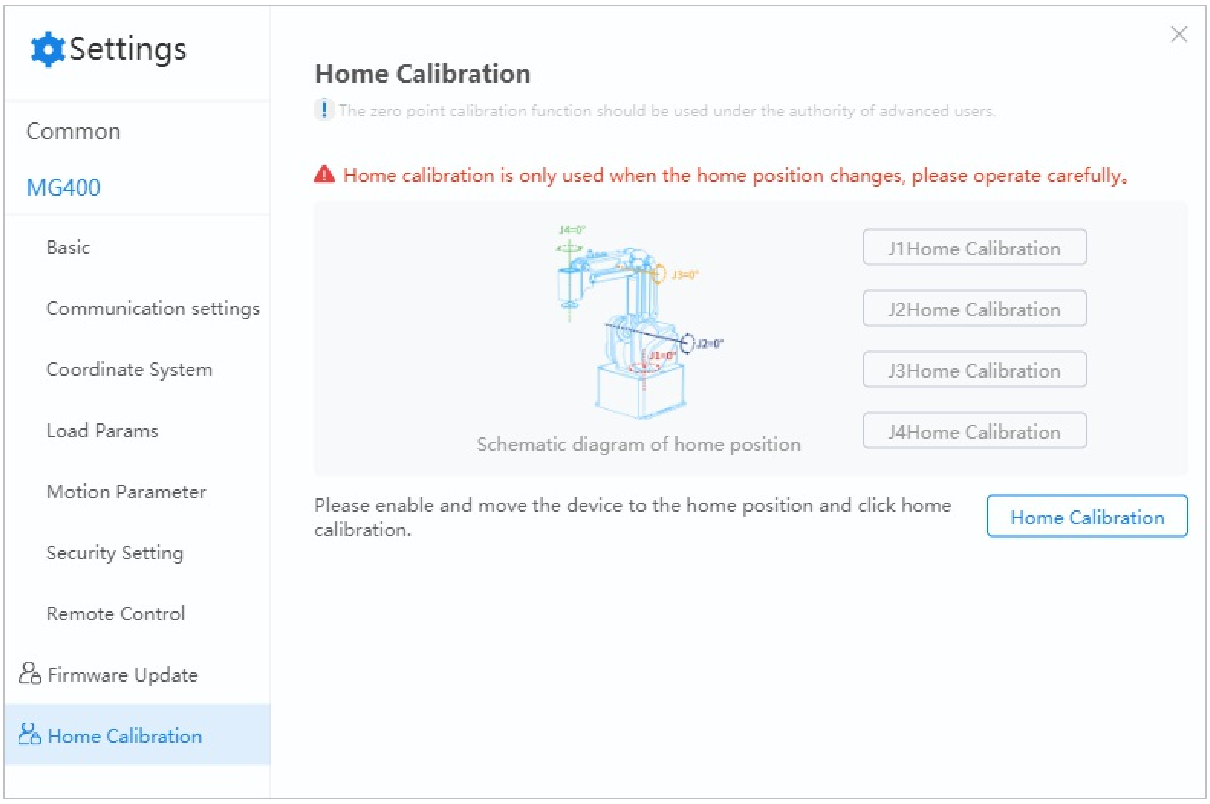
DobotStudio Pro supports servo firmware updating. Please use this function under the guidance of technical support.

4.10 Home calibration

After some parts (motors, reduction gear units) of the robot arm have been replaced or the robot has been hit, the home point of the robot will be changed. In this case you need to reset the home point.

NOTE

- Home calibration is used only when the home position changes. Please operate cautiously.
- The home calibration function can be used after you enter the manager password (default password: 888888)

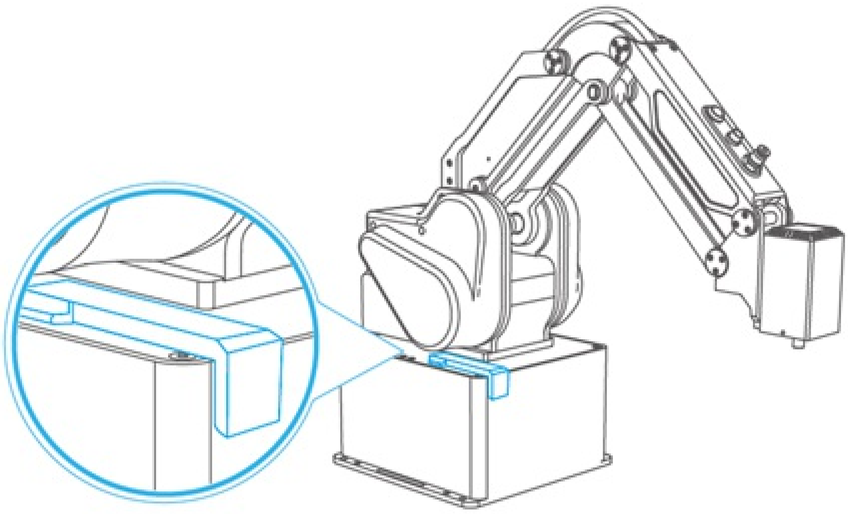


You can calibrate each axis separately, or calibrate the whole robot arm through **Home Calibration** .

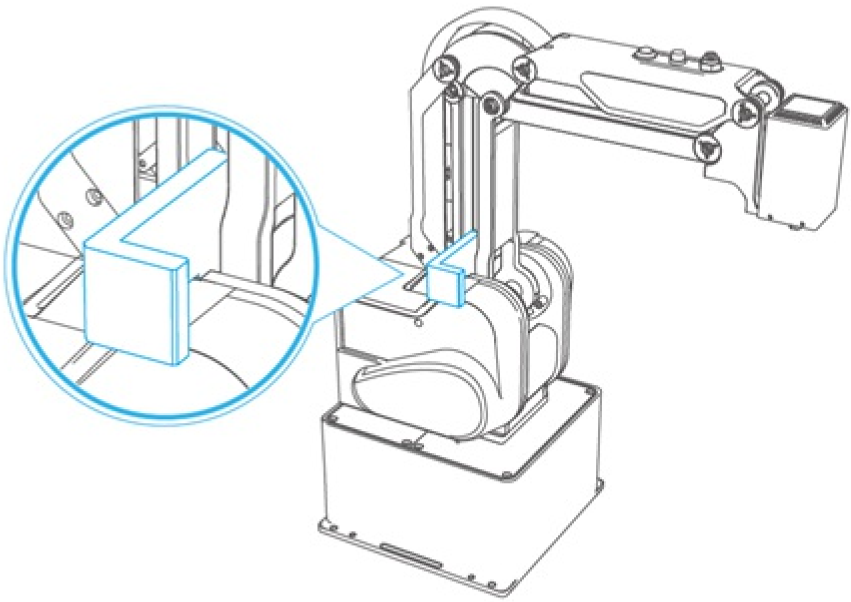
For home calibration, you can use the calibration block as shown below.

This section takes the whole-arm calibration as an example to describe the procedure for home calibration.

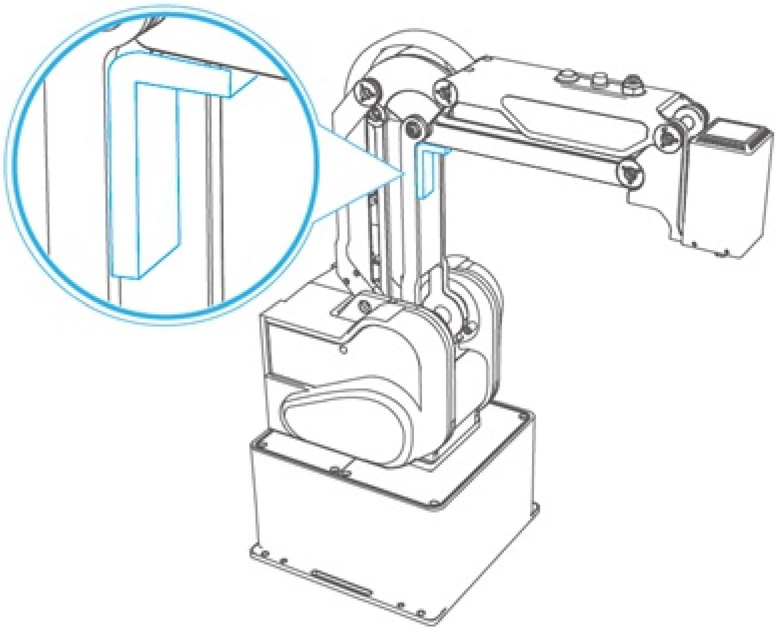
1. Put the calibration block in the position shown below and close to the rotating plate. Rotate J1 axis to make the rotating plate parallel and close to the calibration block.



2. Clamp the convex groove at the bottom of the calibration block in the gap shown in the figure below, and make the short side of the calibration block face the upper arm. Press the hand-teaching button, drag J2 axis and J3 axis to make the upper arm parallel and close to the calibration block, and make the angle between the upper arm and the forearm greater than 90°.



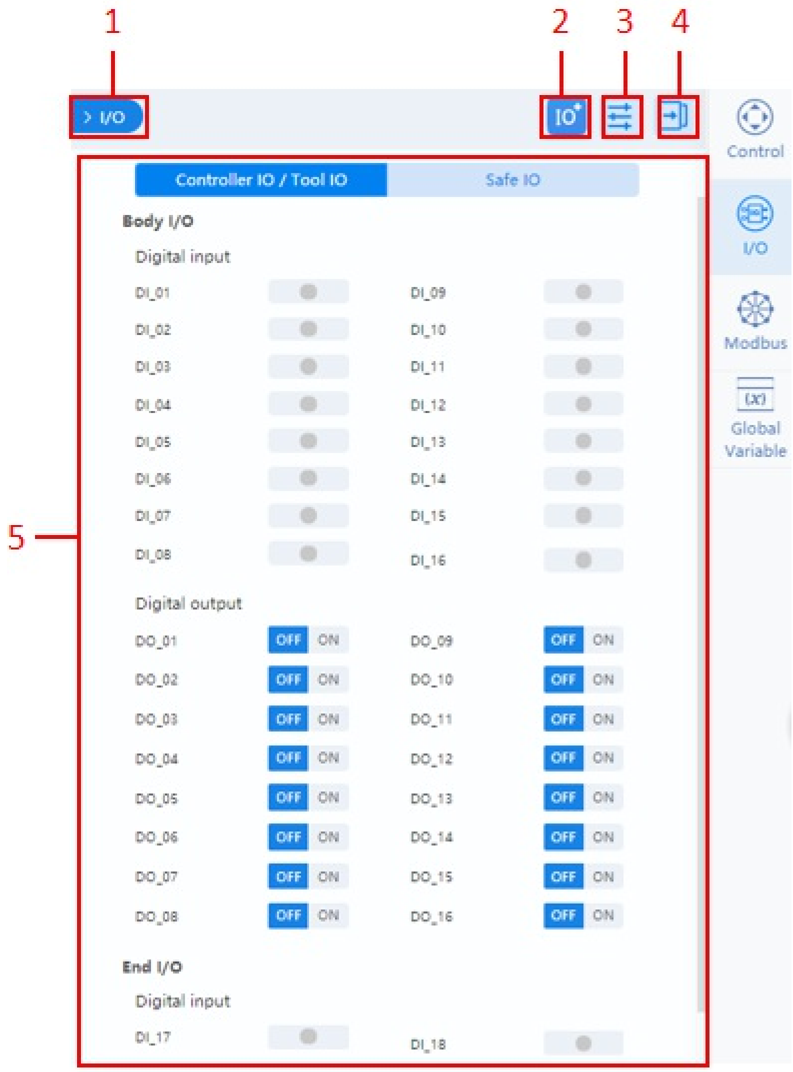
3. Put the calibration block in the position shown below, namely the angle between the upper arm and the forearm, to make the long side of the calibration block parallel and close to the upper arm. Jog J3 axis on the jog board to make the forearm parallel and close to the short side of the calibration block.



4. Click **Home calibration** to confirm. After home calibration, all joint angles on the jog board should be zero.

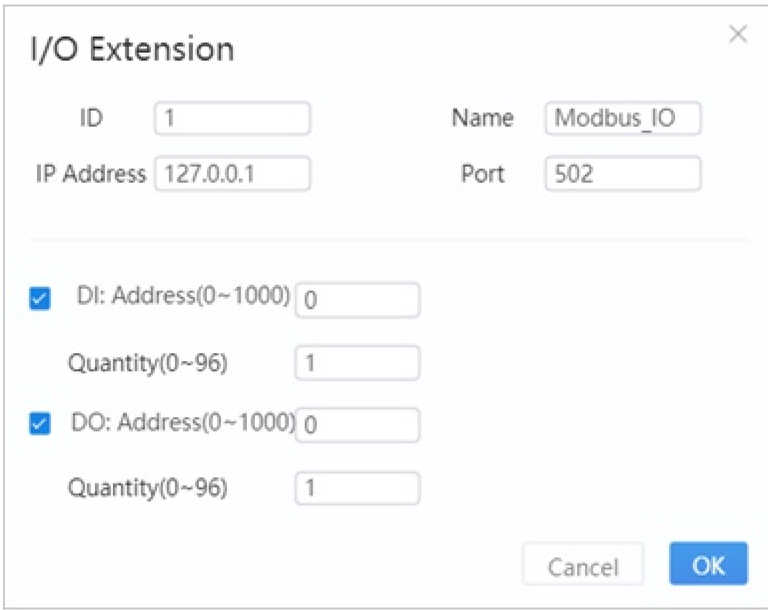
5 I/O Monitoring

You can monitor and set the I/O status of the controller and the end tool in the I/O page. For the I/O definition, refer to the IO description in the corresponding robot hardware guide. As different controllers vary in the number of I/O, the screenshots in this document are for reference only.



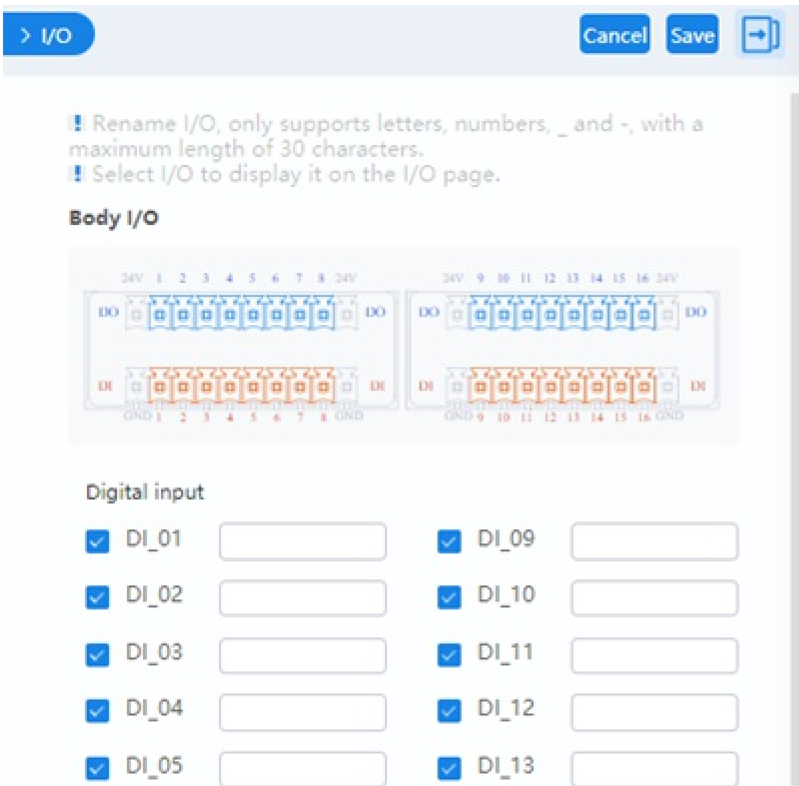
No.	Description
1	Click to hide the panel, and click **I/O** in the right toolbar to display the panel.
2	Click to add extended I/O, which can be used for monitoring Modbus communication. See I/O extension for details.
3	Click to set I/O alias or whether to display. See I/O configuration for details.
4	Click to fold the control panel, and click again to unfold the panel.
5	IO monitoring area. See Monitoring for details.

I/O extension



- ID: Slave device ID. - Name: Name of the slave device. - IP address: Enter the address of the Modbus device. - Port: Port number of Modbus communication. - DI/DO: Configure the register address and number of DI/DO after selecting the function. After clicking **OK**, a new I/O will appear in the bottom of I/O panel. The monitoring function takes effect only after you restart the controller. Clicking **x** on the right of the tab can delete the tab.

I/O configuration



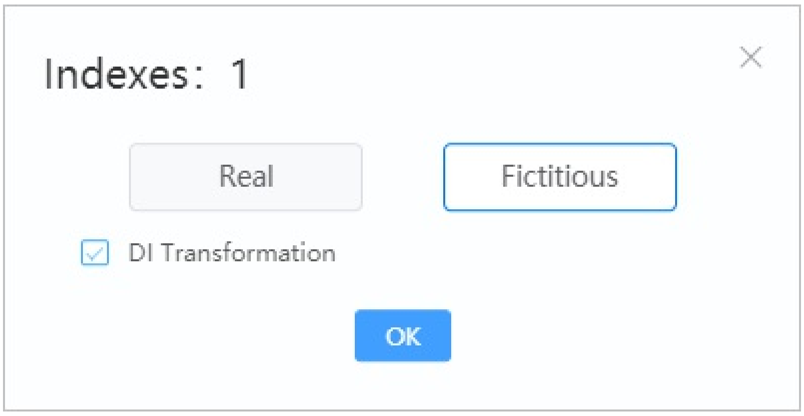
- Select IO to display it in the monitoring page.
- Enter the alias of the IO on the right side, and the alias will be displayed on the monitoring page. At

the same time, you can also call the corresponding IO through the alias in block programming and script programming.

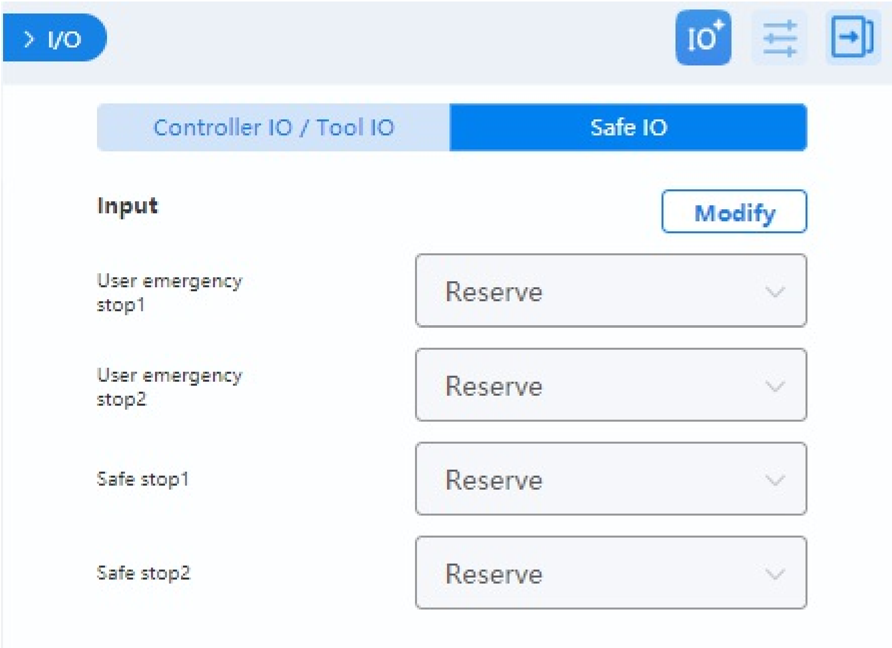
Monitoring

Controller I/O and Tool I/O page supports the following functions.

- Output: Set the digital output. You can click the corresponding switch on the right side to switch its status.
- Monitor: Check the real status of the input and output. The dot on the right of the digital input indicates the status of the corresponding DI. Grey means DI is not triggered, and green means DI is triggered.
- Simulation: Simulate the status of digital input to facilitate debugging and running programs. Click the status display area of the corresponding DI, and a setting window will pop up. Click **Fictitious** and select **DI Transformation**, and the DI will turn to virtual trigger status (green dot), which is regarded as ON logically. If you do not select **DI Transformation**, the DI will maintain its real status.



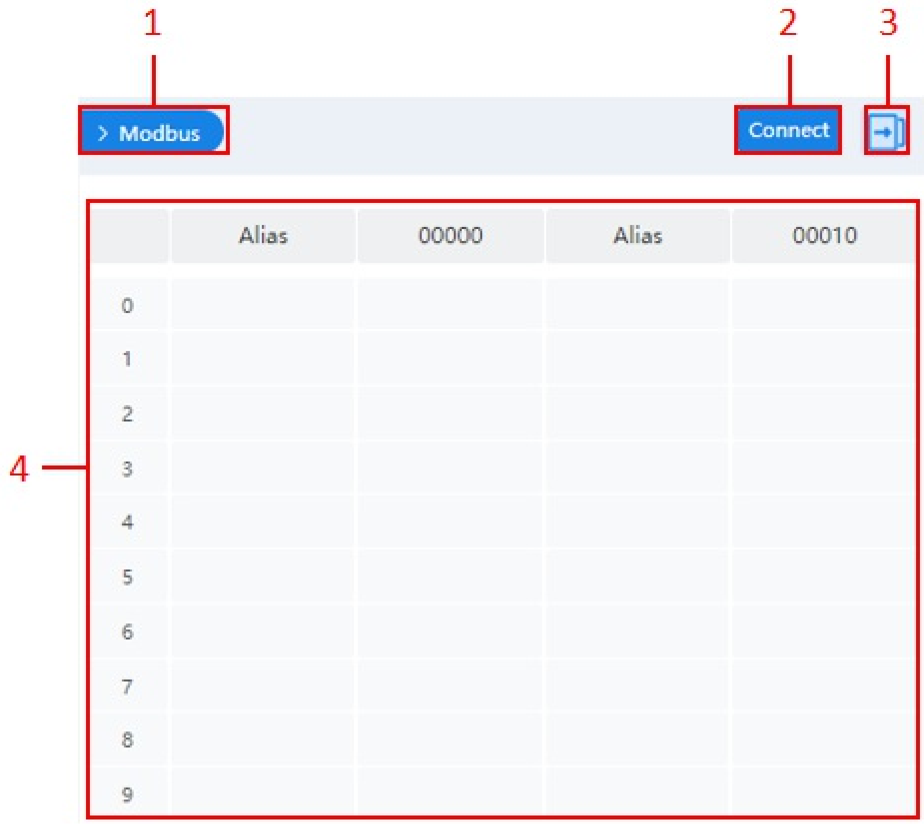
You can set the DI corresponding to the safety functions in the Safe IO page. Click **Modify** to modify the function, and click **Save** after setting. Please configure the safe I/O according to your actual requirement.



> I/O
IO
Controller IO / Tool IO
Safe IO
Input
Modify
User emergency stop1
Reserve
User emergency stop2
Reserve
Safe stop1
Reserve
Safe stop2
Reserve

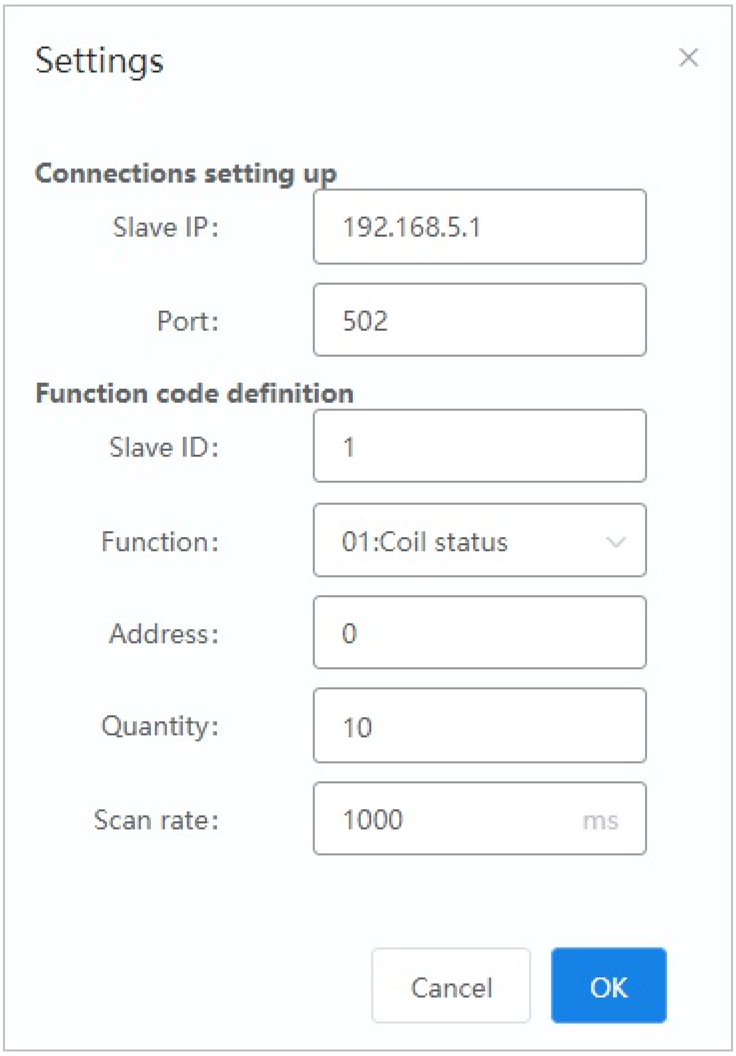
6 Modbus

Modbus module, serving as Modbus master, is used to connect Modbus slave.



No.	Description
1	Click to hide the panel, and click **Modbus** on the right toolbar to display the panel
2	Click to connect Modbus slave. See Connecting Modbus slave for details
3	Click to fold the control panel, and click again to unfold the panel
4	Display register information of connected slaves

Connecting Modbus slave

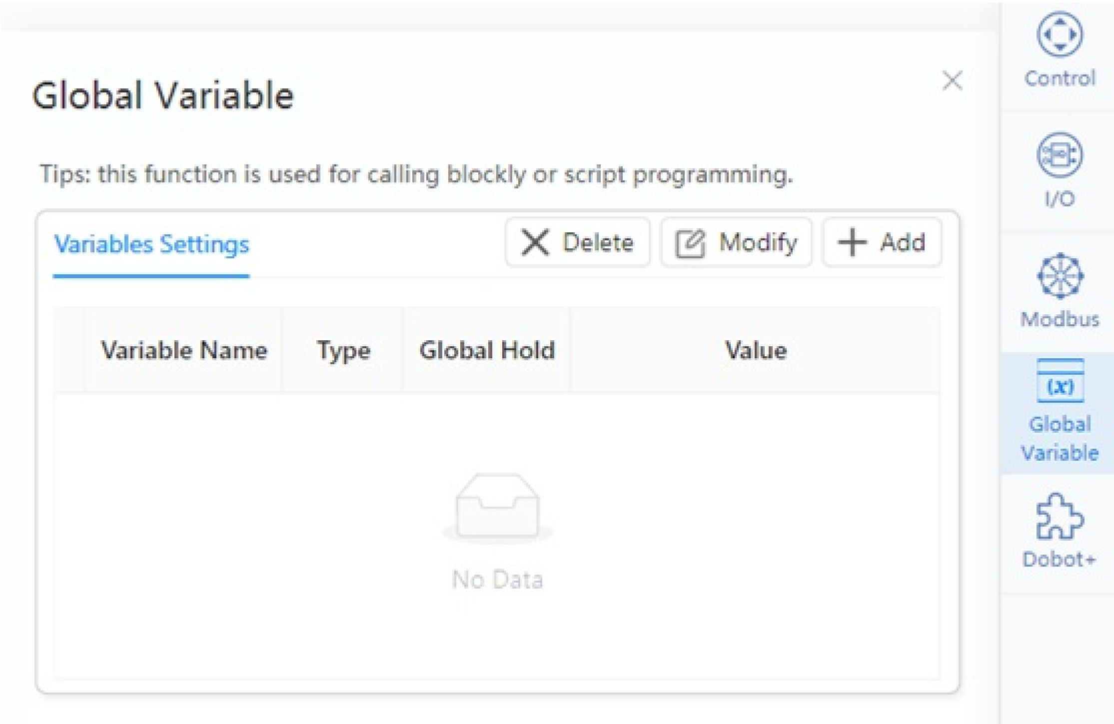


- Slave IP: address of Modbus device.
- Port: port number of Modbus communication.
- Slave ID: slave device ID.
- Function: select the function type of the slave device.
- Address/Quantity: address and number of registers.
- Scanning rate: time interval of scanning the slave station by the robot arm.

7 Global Variable

The module is used to configure and check the global variables.

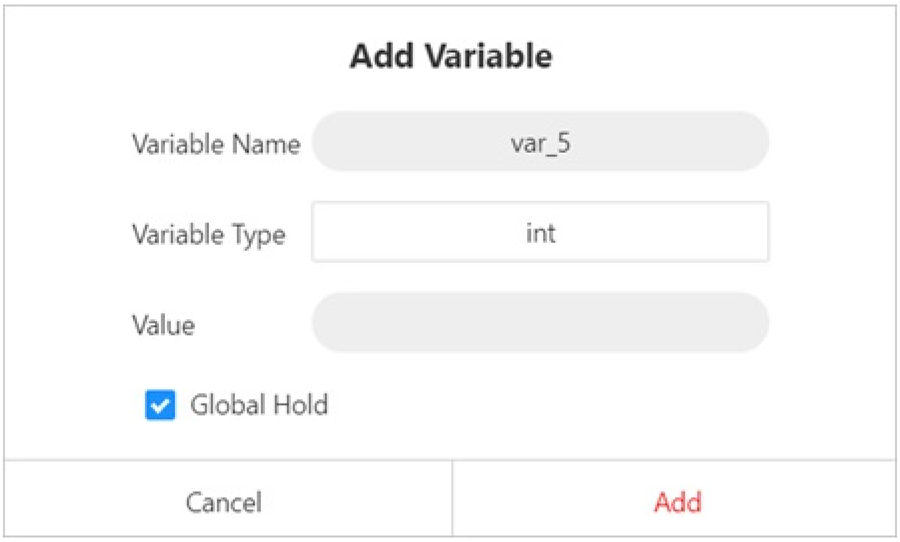
After setting the global variable, you can call the variable through relevant blocks in block programming, or call the variable through the variable name in script programming.



DobotStudio Pro supports the following types of global variables:

- bool: Boolean value
- String: String
- int: Integer
- float: Double precision floating number
- point: The point of the robot can be obtained by moving the robot to the specified position, as shown in the figure below.

When the variable is set as **Global Hold**, the global variable can keep its value after the robot is powered off.



Add Variable
Variable Name
var_5
Variable Type
int
Value
Global Hold
Cancel
Add

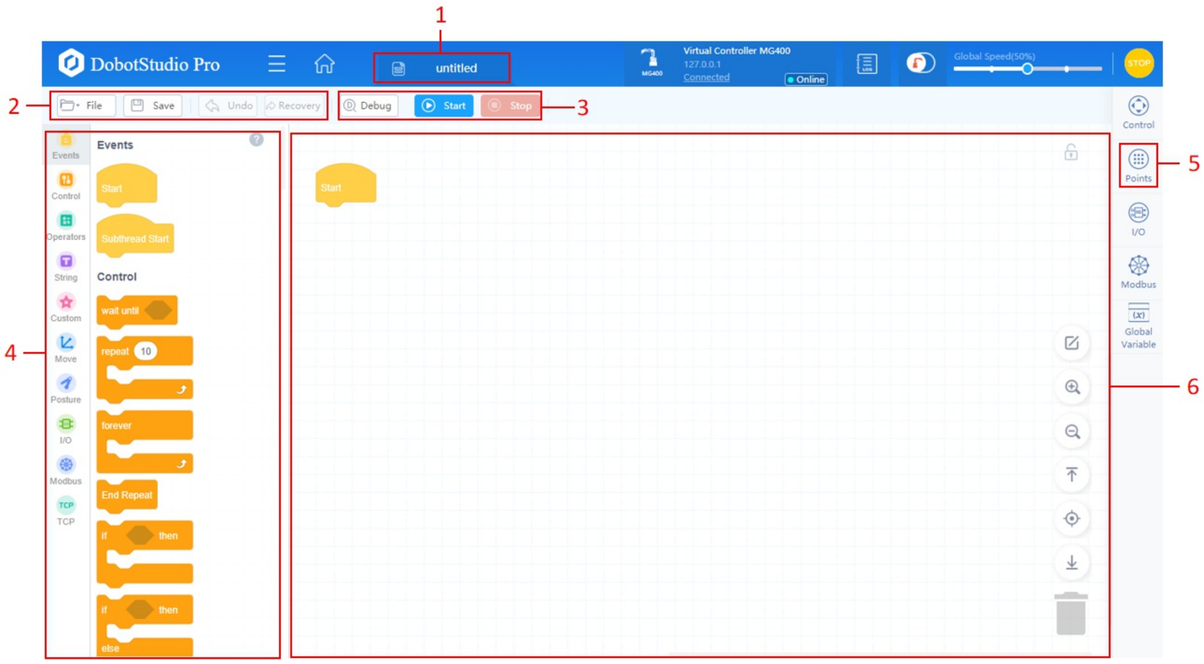
8 Programming

- 8.1 DobotBlockly
- 8.2 Script

8.1 DobotBlockly

DobotStudio Pro provides blockly programming. You can program through dragging the blocks to control the robot.

> **NOTE**
>
> This document only introduces the use of blockly programming. For specific description on blocks , see *Dobot Blockly User Guide (CR)*.



No.	Description
1	Display the current project name
2	It is used to manage project files and undo or restore programming operations. In the **File** drop-down list, you can convert a blockly program to script. After successful conversion, you can open the converted project in Script module
3	Control the running of the project. See Running project for details.
4	Provide blocks used in programming, which are divided into different colors and categories. Click ? on the right top of the module to view the relevant description on the blocks.
5	Program editing area. You can drag the blocks to the area to edit a program. Right-click the block in the programming area to open the menu, which supports copying blocks, deleting blocks, and turning a group of blocks into sub-routines. If a block is modified but not saved, you will see the edit icon on the left side of the block, which prompts that the block has been modified.

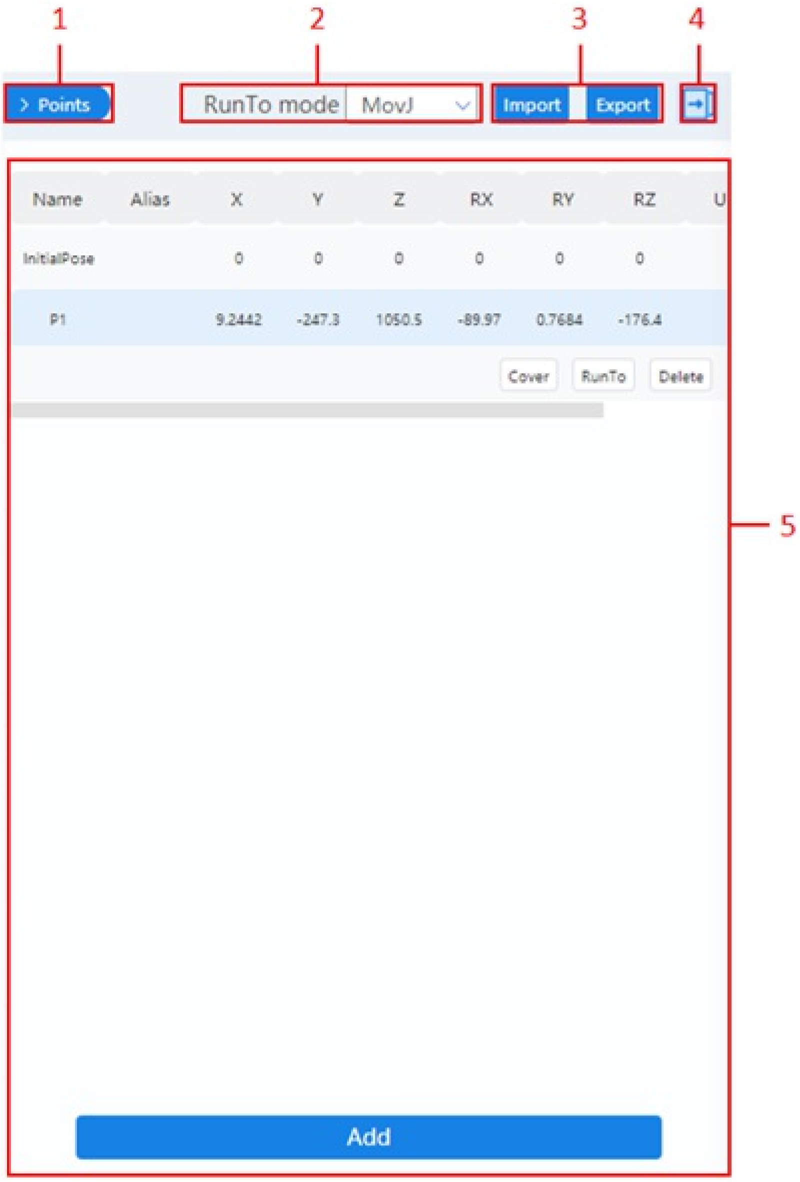
6	Click to open **Points** panel. If there are unsaved changes, you will see a red dot in the lower right of the icon. See Points for details.

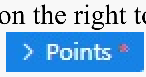
The icons on the right side of the programming area is described below.

Icon	Description
	Enter editing mode. In editing mode, you can select multiple or all blocks to copy or delete. Click **Cancel Checking** or do other operations in the programming area to exit the editing mode. 
	Lock/Unlock the programming area.
	Zoom in/Zoom out/Restore the programming area.
	Back to the top of blocks/Center blocks/Back to the bottom of blocks.
	Drag the block to this icon to delete it, or long press the block and select **Delete Block** to delete it.

Points

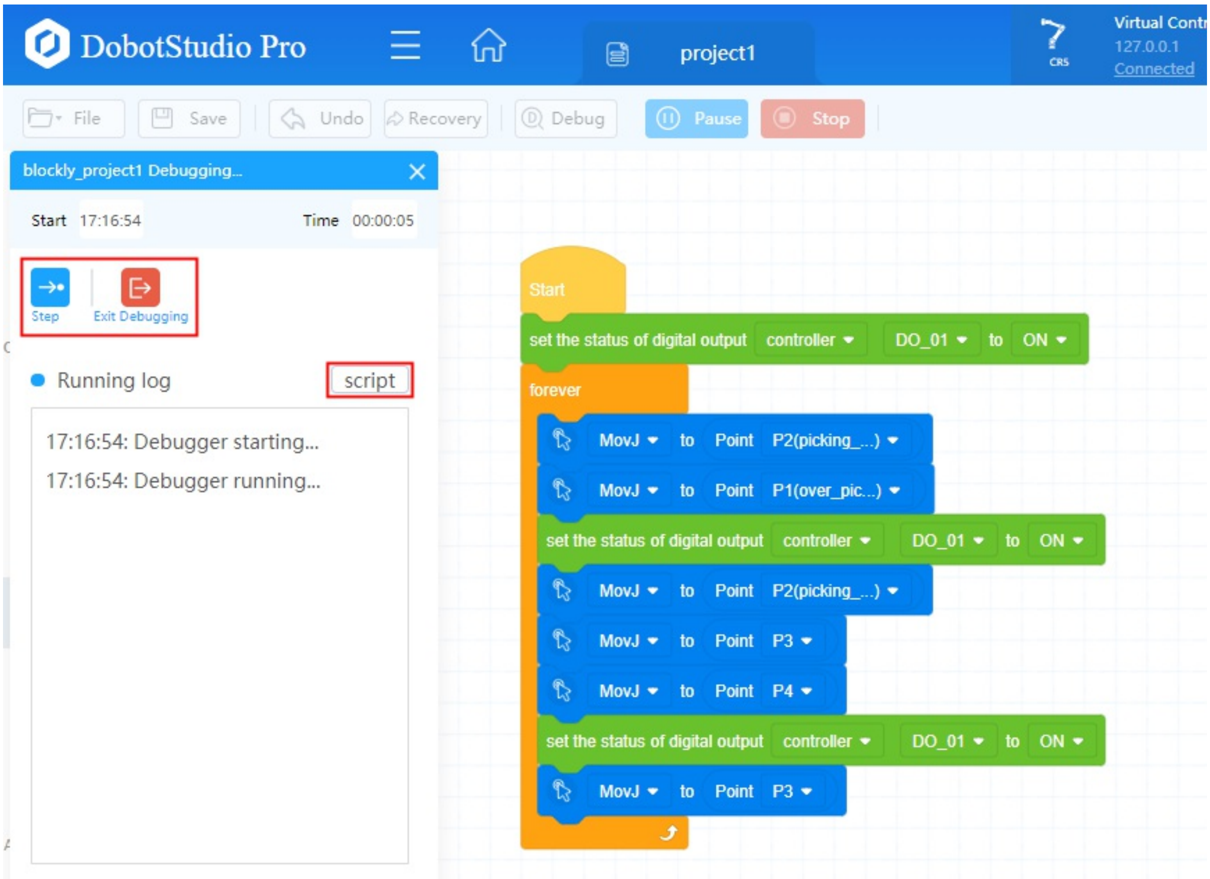
The Point interface is used to manage the points in programming, as shown below.



No.	Description
1	Click to hide **Points** panel. You can click **Points** on the right toolbar to restore its display. If there are unsaved changes, the icon will turn to  .
2	Set the motion mode of **Run To**
3	Import a point list file or export the current point list to a file.
4	Click to fold the control panel, and click it again to unfold the panel.
5	Point management area. • After moving the robot arm to a specified point, click **Add** to save the current point of the robot arm as a new teaching point. • After selecting a teaching point, double-click any value except **Name** of the teaching point to directly modify the value. • After selecting a teaching point, click **Cover** to overwrite it with the current point. • After selecting a teaching point, long-press **RunTo** to move the robot arm to the point. • After selecting a teaching point, click **Delete** to delete the teaching point.

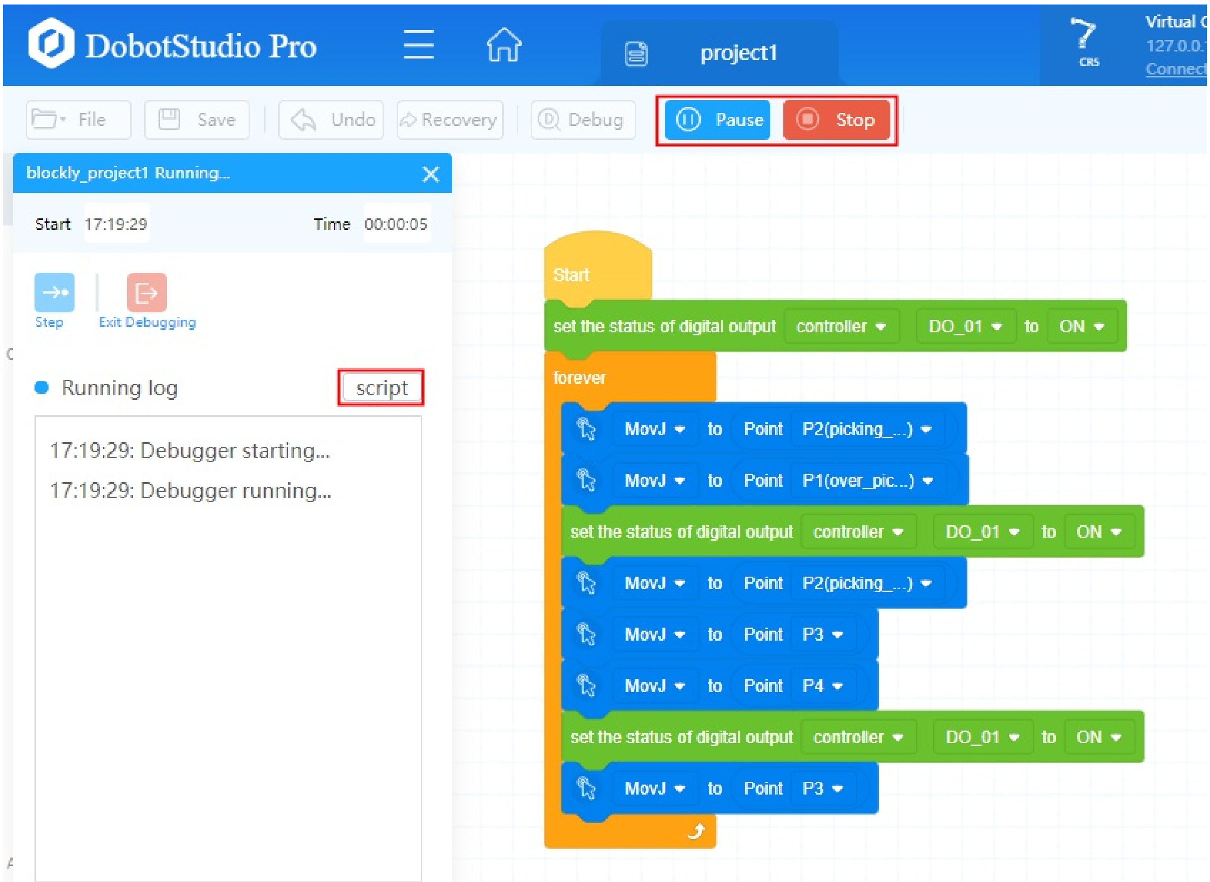
Debugging and running project

Click **Debug** after saving the project, and the project will start running step by step. You can view view the operation log in this process. - Click **Step** to run the project step by step. - Click **Exit Debugging** to exit the debug mode.. - Click **script** to display the running script corresponding to the project.



Click **Start** after saving the project, and the project will start running. The log of the running process will be displayed.

- Click **Pause** to pause running the project, and the button changes to **Continue**. Click **Resume** to continue running the project.
- Click **Stop** to stop running the project.
- Click **script** to display the running script corresponding to the project.



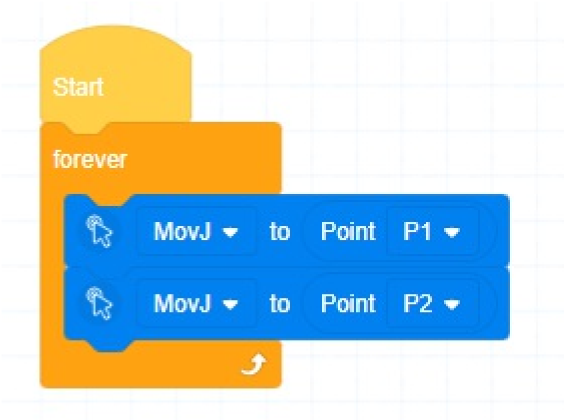
Operation procedure

The following example describes the procedure of editing a block program to control the robot to move between two points repeatedly.

1. Open the **Points** panel. Move the robot arm to a point (P1), and click **Add** to save the point P1.
2. Move the robot arm to a point (P2), and click **Add** to save the point P2.
3. Drag the **forever** block from the block area and place it under the **Start** block.
4. Drag 

inside the **forever** block, and select P1 for the target point.
5. Drag 

under the previous block, and select P2 for the target point.

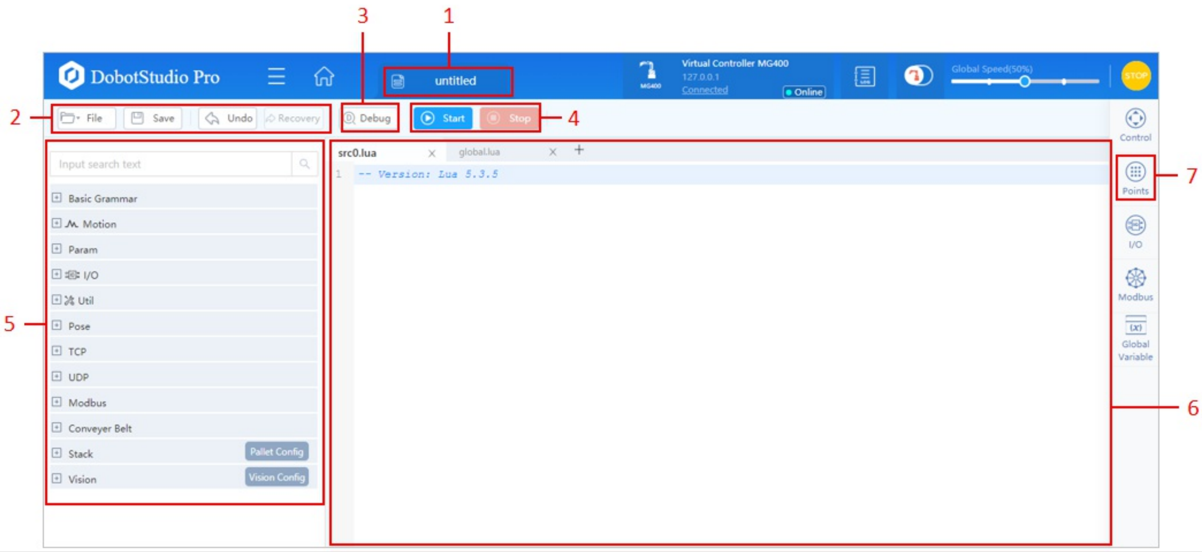


6. Click **Save**, enter the project name and click **OK**.
7. Click **Start**, and the robot starts to move.

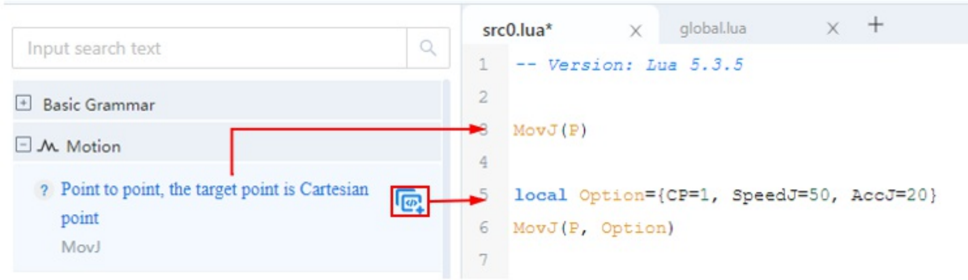
8.2 Script

Dobot robots provides various APIs, such as motion commands, TCP/UDP commands etc., which uses Lua language for secondary development. DobotStudio Pro provides a programming environment for Lua scripts. You can write your own Lua scripts to control the operation of robots.

> NOTE
>
> This section mainly introduces the use of script programming. For specific description on commands, see *DOBOT Lua Syntax Guide(CR)*.

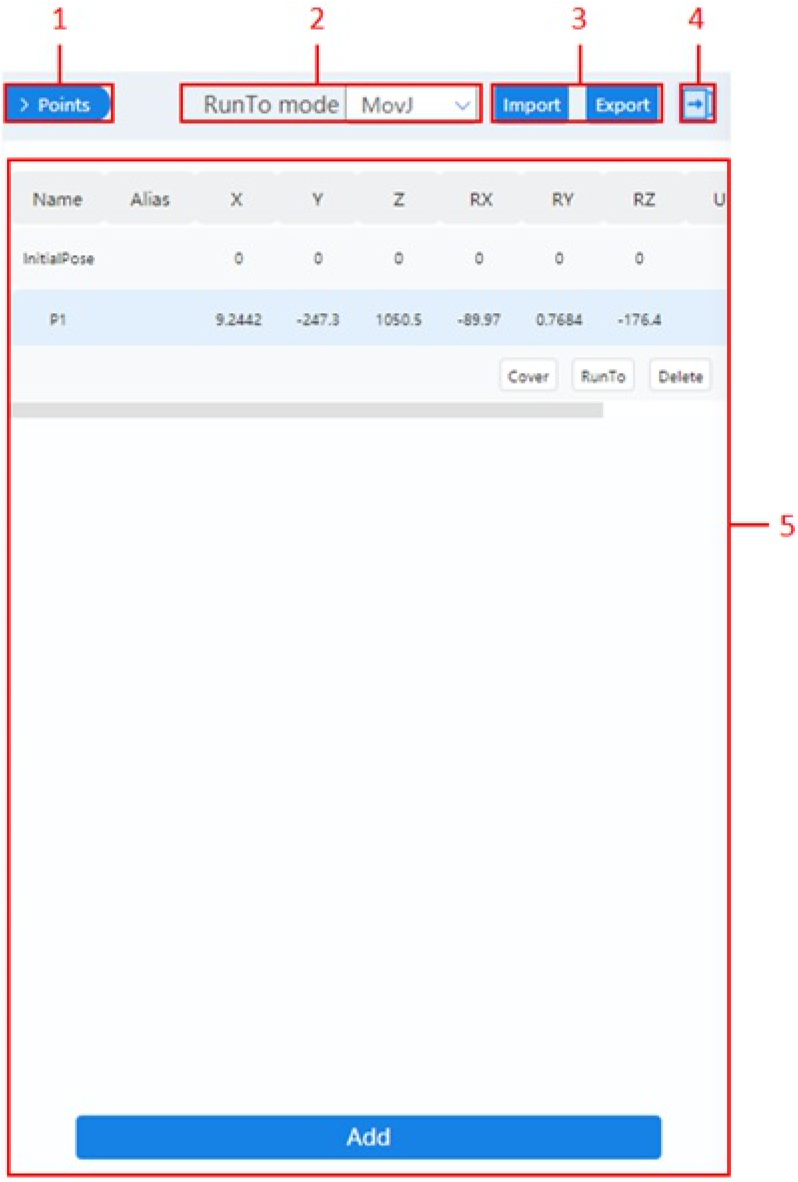


No.	Description
1	Display the current project name
2	It is used to manage project files and undo or restore programming operations.
3	Open the debug page. See Debugging project for details.
4	Control the running of the project. See Running project for details.
5	Command list. • Click ? on the left side of the command to view the command description. • Double-click the command to quickly add Lua command to the programming area on the right • If there is a blue icon on the right side of the command, double-click the blue button to quickly add Lua command with detailed parameters to the programming area on the right.

	
6	Program editing area. • The "src0.lua" file is the main thread and can call any commands. • The "global.lua" file is only used to define variables and subfunctions. • Click + to add subthreads. Subthreads are parallel programs that run with the main program. You can set I/O, variables, etc. in subthreads, but cannot call motion commands.
7	Click to open **Points** panel. If there are unsaved changes, you will see a red dot in the lower right of the icon. See Points for details.

Points

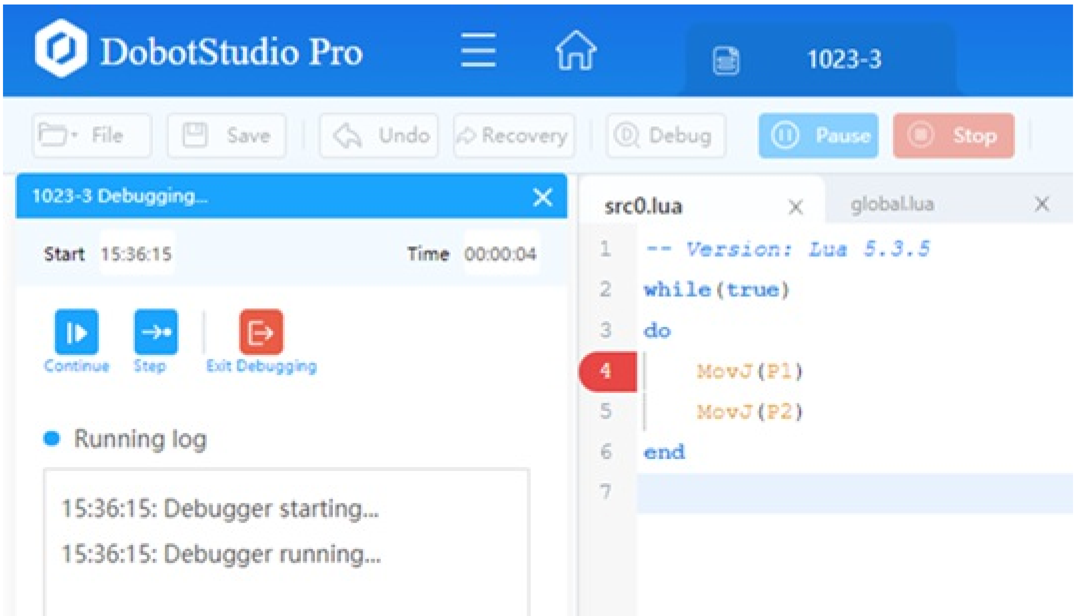
The Point interface is used to manage the points in programming, as shown below.



No.	Description
1	Click to hide **Points** panel. You can click **Points** on the right toolbar to restore it. If there are unsaved points, the icon will turn to  .
2	Set the motion mode of **Run To**.
3	Import a point list file or export the current point list to a file.
4	Click to fold the control panel, and click it again to unfold the panel.
5	Point management area. • After moving the robot arm to a specified point, click **Add** to save the current point of the robot arm as a new teaching point. • After selecting a teaching point, click **Cover** to overwrite it with the current point. • After selecting a teaching point, long-press **RunTo** to move the robot arm to the point. • After selecting a teaching point, click **Delete** to delete the teaching point.

Debugging project

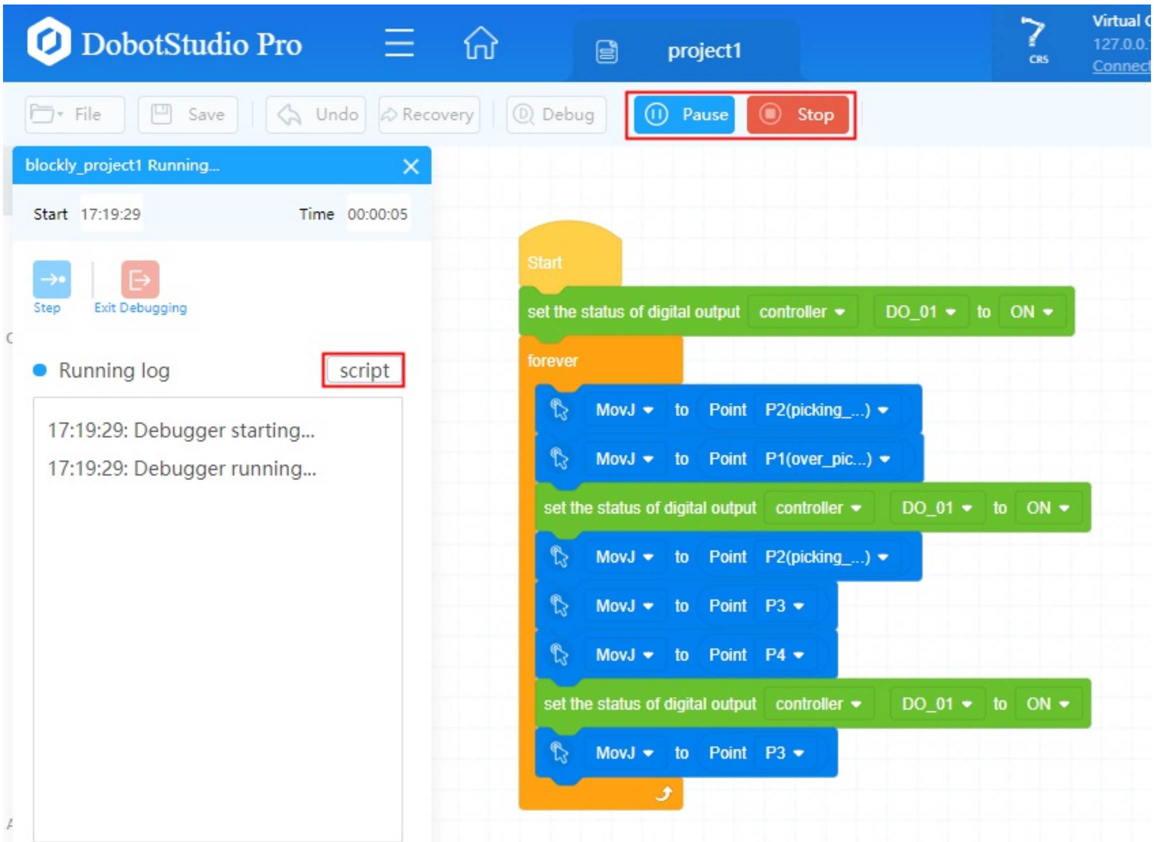
Click **Debug** after saving the project, and the project will enter debug mode.
- Clicking the line number on the left side of the code can set a breakpoint. The program will automatically pause when it runs to the breakpoint in debug mode.
- After the program is paused at the breakpoint, you can click **Continue** to keep the program running; or click **Step** to run the program step by step.
- Click **Exit Debugging** to exit debug mode.



Running project

Click **Start** after saving the project, and the project will start running. The log of the running process will be displayed.

- Click **Pause** to pause running the project, and the button will change to **Resume**. Clicking **Resume** will continue running the project.
- Click **Stop** to stop running the project.



Operation procedure

The following example describes the procedure of editing a script program to control the robot to move between two points repeatedly.

1. Open the **Points** panel. Move the robot arm to a point (P1), and click **Add** to save the point P1.
2. Move the robot arm to another point (P2), and click **Add** to save the point P2.
3. Add loop commands in the programming area.
4. Add a motion command under the loop command, and set P1 as the target point.
5. Add another motion command, and set P2 as the target point.

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

6. Click **Save**, enter the project name and click **OK**.
7. Click **Start**, and the robot starts to move.

9 Best Practice

This chapter describes the complete process of controlling a robot arm through remote I/O to help you understand how the various functions of DobotStudio Pro are used in a coordinated manner.

Now assume the following scene: after pressing the start button, the running indicator light is on. The robot arm grasps the material from the picking point through the end gripper, moves to the target point to release the material, and then returns to the picking point again to grasp the material… The process is executed repeatedly.

In order to achieve the scene above, you need to install a gripper at the end of the robot arm (assume that the installed gripper is controlled by the end DI1, which opens when the end DI1 is ON and closes when the end DI1 is OFF), and connect the buttons and indicators to the controller I/O interface (assuming the start button is connected to DI11 and the stop button is connected to DI12; the running indicator is connected to DO11, and the alarm indicator is connected to DO12. For the wiring, refer to the corresponding hardware guide of the robot).

Overall process

After installing the hardware and the powering on the robot arm, perform the software operations as follows:

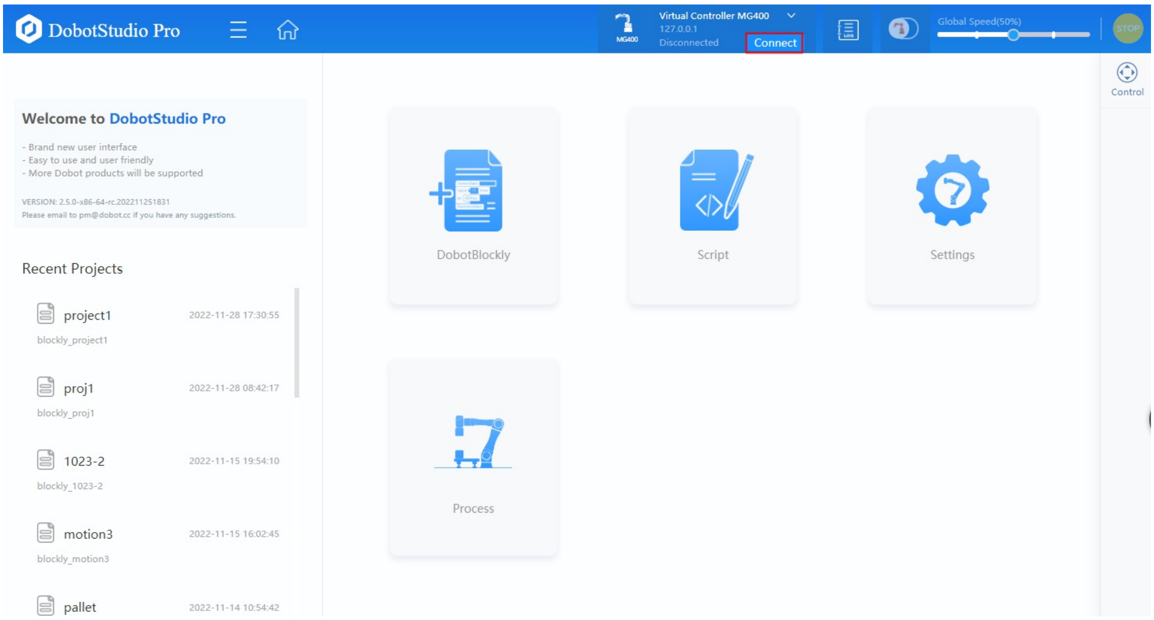
1. Connect the robot
2. Set and select a tool coordinate system
3. Edit the project file
4. Configure and enter remote I/O mode

Procedure

Connecting and enabling robot

For details about connecting to the robot, refer to Connecting to Robot.

1. Search Dobot controller WiFi name and connect it. The WiFi SSID is MagicianPro, and WiFi password is 1234567890 by default.
2. Select a robot on the top of DobotStudio Pro interface and click **Connect**.

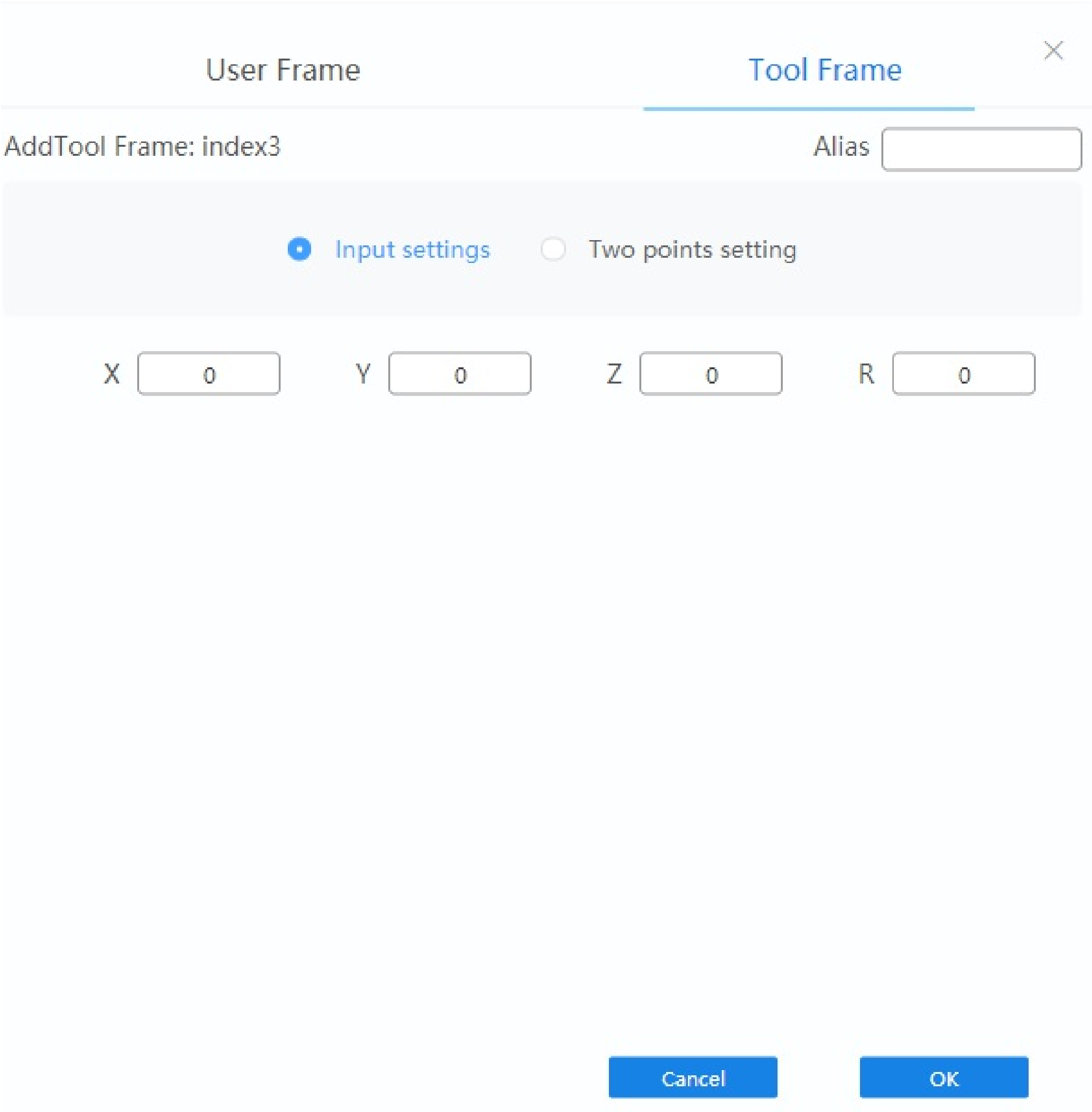


3. Click the enabling button and set the load parameters to enable the robot.

Setting and selecting tool coordinate system

For details about tool coordinate system, refer to Tool coordinate system. Here takes input settings as an example.

1. Open **Settings** > **Coordinate System** > **Tool Coordinate System** page.
2. Add or modify a coordinate system. Enter the offset of the tool center point relative to the flange center point, and click **OK**.



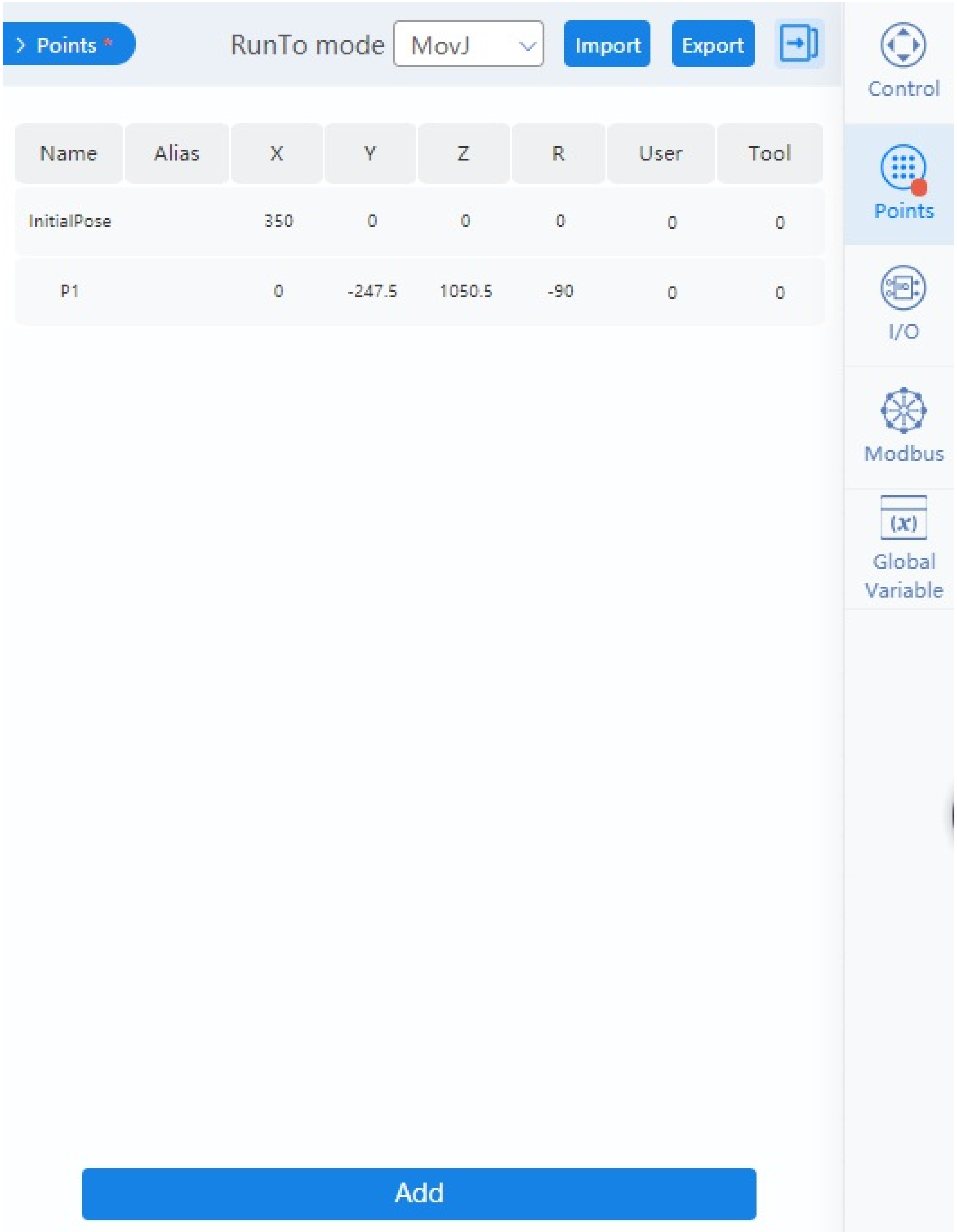
3. Select the tool coordinate system that you set in the last step in the control panel.

Editting project

For details on programming, refer to DobotBlockly and Script. Here takes DobotBlockly as an example.

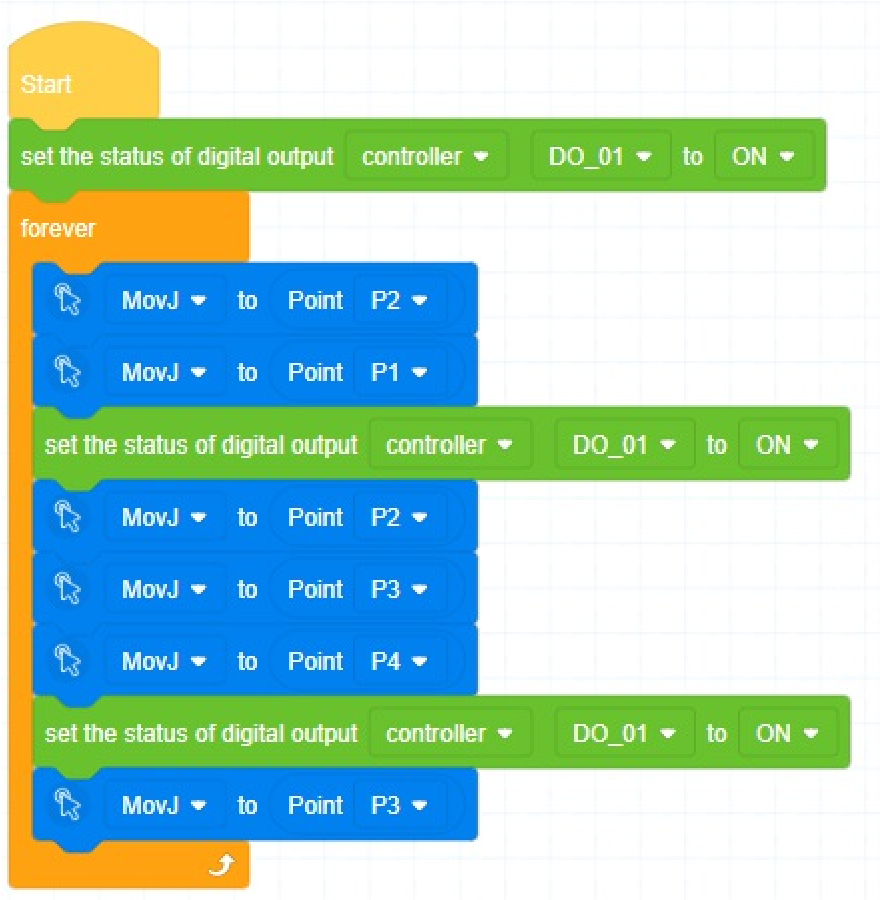
To achieve the scene described at the beginning of this chapter, you need to teach four points, namely the picking point P1, the transition point P2 (above the picking point), the transition point P3 (above the uploading point), and the uploading point P4.

1. Open the Points page, move the robot arm to P1, and click **Add**.



2. Add P2, P3 and P4 in the same way.
3. Drag the blocks to the programming area to realize picking and unloading the material. The figure

below shows a simple program for your reference.

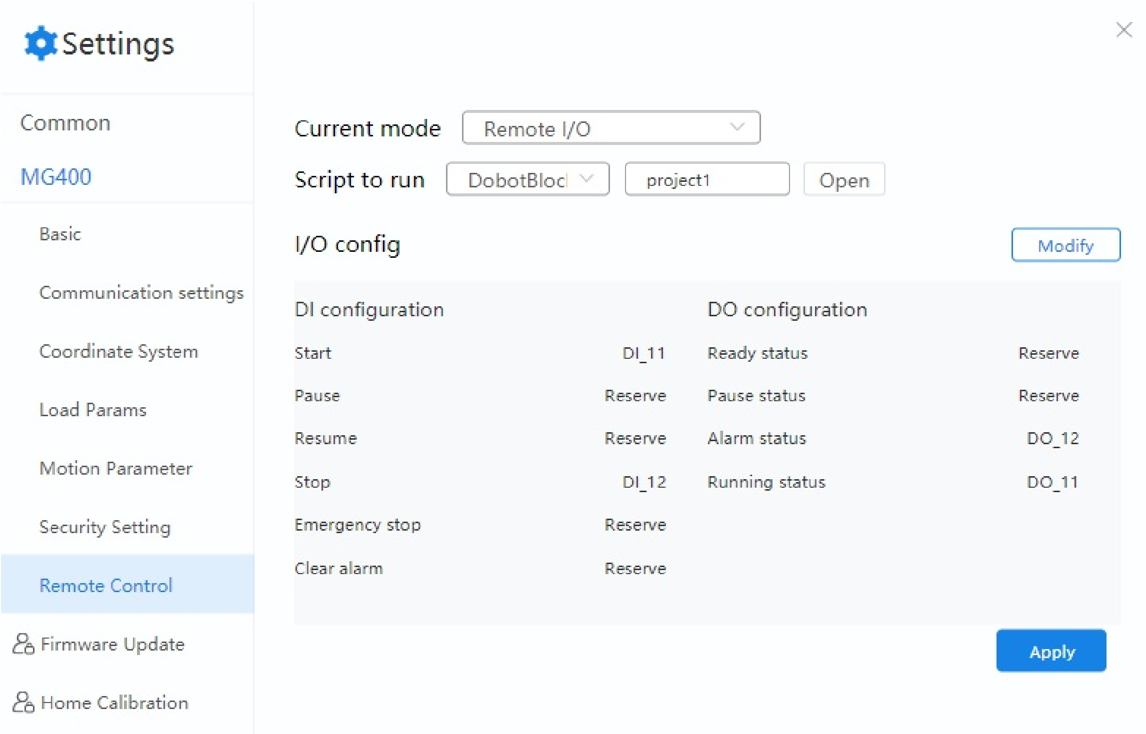


4. Save the project.

Configuring and entering remote I/O mode

For details about remote control, refer to Remote control. Here only describes the steps to configure and enter remote I/O mode based on the example scene.

1. Open **Settings** > **Remote Control** page.
2. Set **Current mode** to **Remote I/O**.
3. Click **Open** and select the DobotBlockly project that you have saved before.
4. Click **Modify** to modify the I/O configuration according to the scene described at the beginning of this chapter.
5. Click **Apply** to enter remote IO mode.



After entering the remote I/O mode, press the start button connected to the robot arm controller, and the robot arm will start running the project.

Appendix A Modbus Register Definition

Modbus data mainly includes four types: coil status, discrete input, input register and holding registers. Based on the robot memory space, four types of registers are defined: coil, contact (discrete input), input and holding registers, for data interaction between the external equipment and robot system. Each register has 4096 addresses. For details, see the description below. **The definition of the coil and contact registers can be modified. Please refer to the actual value on the remote control interface.**

1 Coil register (control robot)

PLC address	Script address (Get/SetCoils)	Register type	Function
00001	0	Bit	Start
00002	1	Bit	Pause
00003	2	Bit	Continue
00004	3	Bit	Stop
00005	4	Bit	Emergency stop
00006	5	Bit	Clear alarm
00007	6	Bit	Reset
00051~0066	50~65	Bit	Base IO: DO1~DO16
00067~0070	66~69	Bit	Tool IO: DO17~DO20
03096~04096	3095~4095	Bit	User-defined

2 Discrete input (robot status)

PLC address	Script address (GetInBits)	Register type	Function
10002	1	Bit	Stop status
10003	2	Bit	Pause status
10004	3	Bit	Running status
10005	4	Bit	Alarm status
10006	5	Bit	Collision status
10007	6	Bit	Manual/Automatic mode
10008	7	Bit	Reserved
10051~10066	50~65	Bit	Base IO: DI1~DI16

10051~10066	50~65	Bit	Base IO: DI1~DI16
10067~10070	66~69	Bit	Tool IO: DI17~DI20

3 Input register

PLC address	Script address(GetInRegs)	Data type	Function
30203	202	F32	Robot running position (joint angle 1)
30205	204	F32	Robot running position (joint angle 2)
30207	206	F32	Robot running position (joint angle 3)
30209	208	F32	Robot running position (joint angle 4)
30211	210	F32	Robot running position (joint angle 5)
30213	212	F32	Robot running position (joint angle 6)
30243	242	F32	Robot running position (x)
30245	244	F32	Robot running position (y)
30247	246	F32	Robot running position (z)
30249	248	F32	Robot running position (a)
30251	250	F32	Robot running position (b)
30253	252	F32	Robot running position (c)

4 Holding register (interaction between robot and PLC)

PLC address	Script address (Get/SetHoldRegs)	Data type	Function
40001~41281	0~1280	U16	Palletizing
41301	1300	U16	Switch to HMI jog mode
41302	1301	U16	Ready to switch HMI jog mode
41303	1302	U16	Jog or step mode: joint/Cartesian

41304	1303	U16	Jog/Step selection
41305	1304	U16	Global speed: percentage
41306	1305	F32	Step distance: mm
41308	1307	F32	Step angle: °
41310	1309	U16	Tool coordinate system selection: index
41311	1310	U16	User coordinate system selection: index
41312	1311	U16	Hand coordinate system
41313	1312	U16	Notification for modifying parameters
41314	1313	U16	Start jogging
41315	1314	U16	J1+/X+
41316	1315	U16	J1-/X-
41317	1316	U16	J2+/Y+
41318	1317	U16	J2-/Y-
41319	1318	U16	J3+/Z+
41320	1319	U16	J3-/Z-
41321	1320	U16	J4+/A+
41322	1321	U16	J4-/A-
41323	1322	U16	J5+/B+
41324	1323	U16	J5-/B-
41325	1324	U16	J6+/C+
41326	1325	U16	J6-/C-
41327	1326	F32	P1(X)
41329	1328	F32	P1(Y)
41331	1330	F32	P1(Z)
41333	1332	F32	P1(R/A)
41335	1334	F32	P1(B)
41337	1336	F32	P1(C)
41339	1338	U16	P1(ARM)
41340	1339	U16	P1(User)
41341	1340	U16	P1(Tool)

41342~41551	1341~1550	F32&U16	P2~P15
41552	1551	F32	P16(X)
41554	1553	F32	P16(Y)
41556	1555	F32	P16(Z)
41558	1557	F32	P16(R/A)
41560	1559	F32	P16(B)
41562	1561	F32	P16(C)
41564	1563	U16	P16(ARM)
41565	1564	U16	P16(User)
41566	1565	U16	P16(Tool)
41567	1566	U16	Save points
41568	1567	U16	RUNTO: go/move
41569	1568	U16	RUNTO: point index
41570	1569	U16	RUNTO: start
41571	1570	U16	Clear alarms
42010	2009	F32	Multi-PC1 (master) x
42012	2011	F32	Multi-PC1 (master) y
42014	2013	F32	Multi-PC1 (master) r
42016	2015	F32	Multi-PC1 (master) encCount
42018	2017	U16	Multi-PC1 (master) type
42019	2018	U16	Multi-PC1 (master) available
42020~42029	2019~2028	U16	Reserved
42030	2029	F32	Multi-PC2 (slave) x
42032	2031	F32	Multi-PC2 (slave) y
42034	2033	F32	Multi-PC2 (slave) r
42036	2035	F32	Multi-PC2 (slave) encCount
42038	2037	U16	Multi-PC2 (slave) type
42039	2038	U16	Multi-PC2 (slave) available
42040~42049	2039~2048	U16	Reserved
42050	2049	F32	Multi-PC3 (slave) x
42052	2051	F32	Multi-PC3 (slave) y
42054	2053	F32	Multi-PC3 (slave) r

42056	2055	F32	Multi-PC3 (slave) encCount
42058	2057	U16	Multi-PC3 (slave) type
42059	2058	U16	Multi-PC3 (slave) available
43095~44095	3095~4095	U16	User-defined

Appendix B Blockly Commands

- B.1 Quick start
  - B.1.1 Control robot movement
  - B.1.2 Read and write Modbus register data
  - B.1.3 Transmit data by TCP communication
  - B.1.4 Palletize
- B.2 Block description
  - B.2.1 Event
  - B.2.2 Control
  - B.2.3 Operator
  - B.2.4 String
  - B.2.5 Custom
  - B.2.6 IO
  - B.2.7 Motion
  - B.2.8 Motion advanced configuration
  - B.2.9 Posture
  - B.210 Modbus
  - B.2.11 TCP

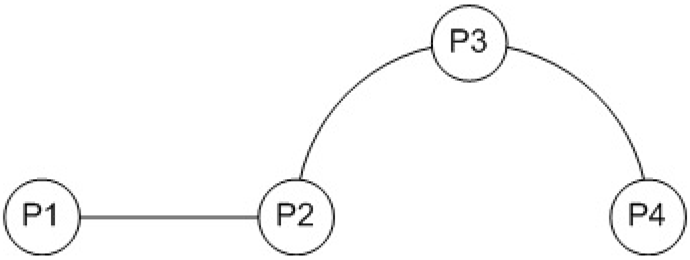
Quick start

Control robot movement

Scene description

In order to experience how to control the movement of the robot arm through blockly programming, you can assume the following scene:

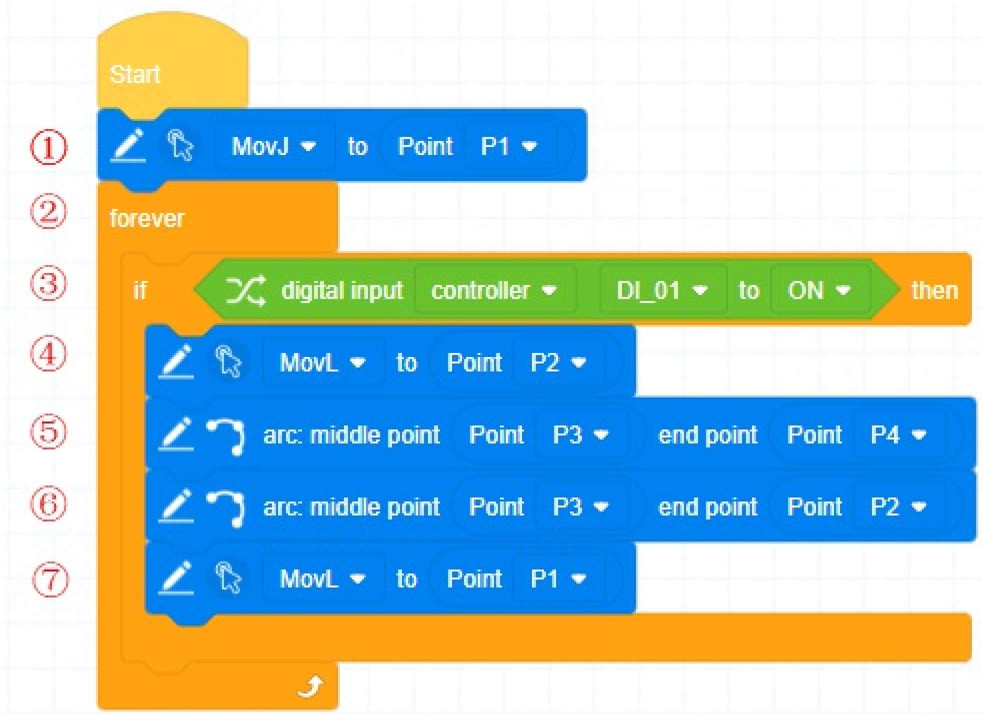
When the controller DI1 is ON, the robot moves from P1 to P2 in a linear mode, moves to P4 via P3 in a arc mode, and then returns along the same way. When the controller DI1 is OFF, the robot arm does not move.



Please teach P1~P4 first according to the figure above.

Steps for programming

To achieve this scene, you need to edit the program as shown in the figure below.



1. The robot arm moves to the starting point through joint motion (P1).

2. Set an unconditional loop to make subsequent commands cycle while the program is running.
3. Judge whether the controller DI1 is ON. The subsequent program wil be executed only when the controller DI1 is ON. Otherwise, it will directly enter the next loop and reacquire the status of DI1.
4. The robot arm moves to P2 in the linear mode.
5. The robot arm moves to P4 via P3 through the arc motion.
6. The robot arm moves to P2 via P3 through the arc motion (return along the same way).
7. The robot arm moves to P1 in the linear mode, and then enters the next loop (return to Step 3).

Run program

Run the program after teaching the points and programming. You can set the status of DI1 through virtual DI in the IO panel.

Read and write Modbus register data

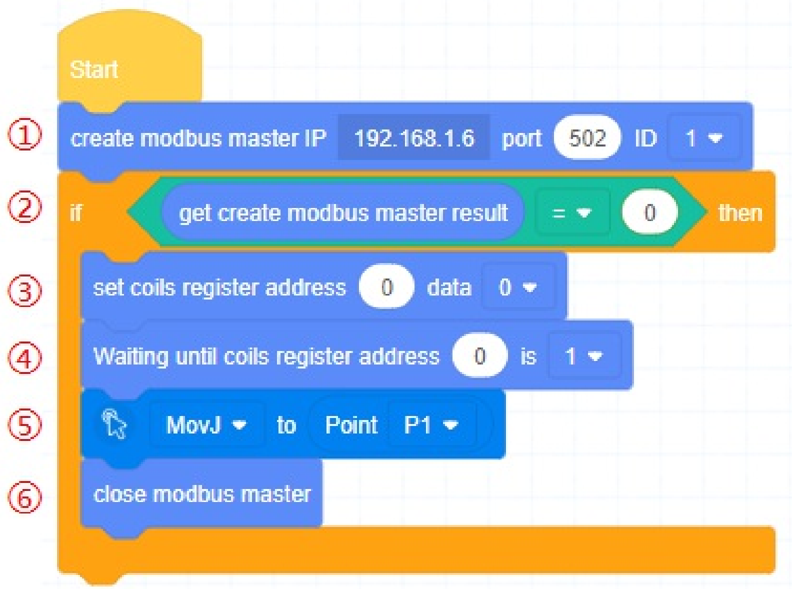
Scene description

To experience how to read and write Modbus data through blockly programming, you can assume the following scene:

Create a Modbus master for the robot. Connect to the external slave and read the address from the specified coil register. If the value is 1, the robot moves to P1.

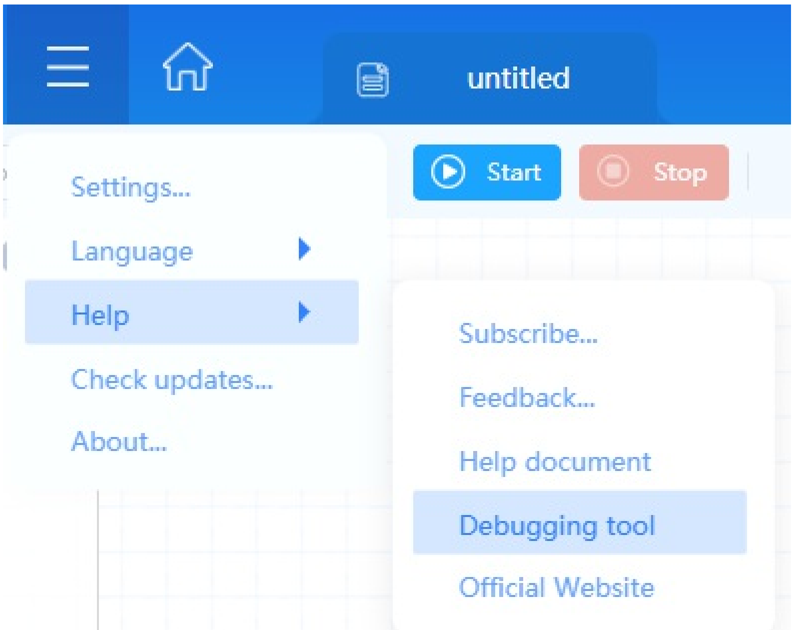
Steps for programming

To achieve this scene, you need to edit the program as shown in the figure below.



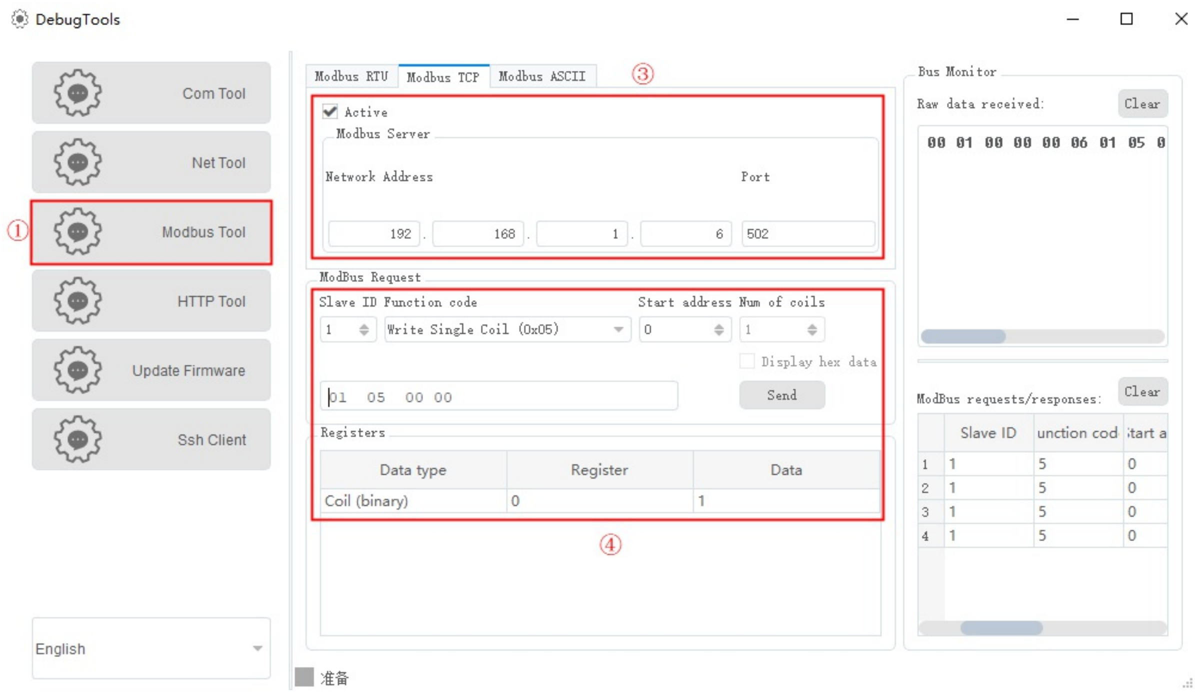
1. Create the master station. Set the IP address to the slave address, and the port and ID to the default values. In this demo the IP is set to robot address, as the robot slave is used here for quick verification.
2. Determine whether the master station is created successfully. The subsequent steps will be executed only if the creation is successful, otherwise, the program will end directly.
3. If the value of coil register 0 of the robot has been modified, it may affect the subsequent program. So you need to set the value of coil register 0 to 0 first.
4. Wait for the value of coil register 0 to change to 1.
5. Control the robot to move to P1, which is a user-defined point.
6. Close the master station.

Run program
If you need to run the program quickly, you can use the debug tool of DobotStudio Pro to modify the value of coil register.



1. Open the debug tool and enter "Net Tool > Modbus TCP" page.
2. Move the robot to a point other than P1 (for observing whether the robot executes the motion command). Then save and run the program.
3. After you see "Create Modbus Master Success" in the running log, select **Active** in the debug tool, and modify **Network Address** and **Port**.
4. Modify **Slave ID Function Code** to **Write Single coil**, and modify **Data** of **Resister 0** to **1**. Then click **Send**.
5. Observe whether the robot moves to P1.

The figure below shows the interface of the debug tool. The marked numbers correspond to the steps above.



Transmit data by TCP communication

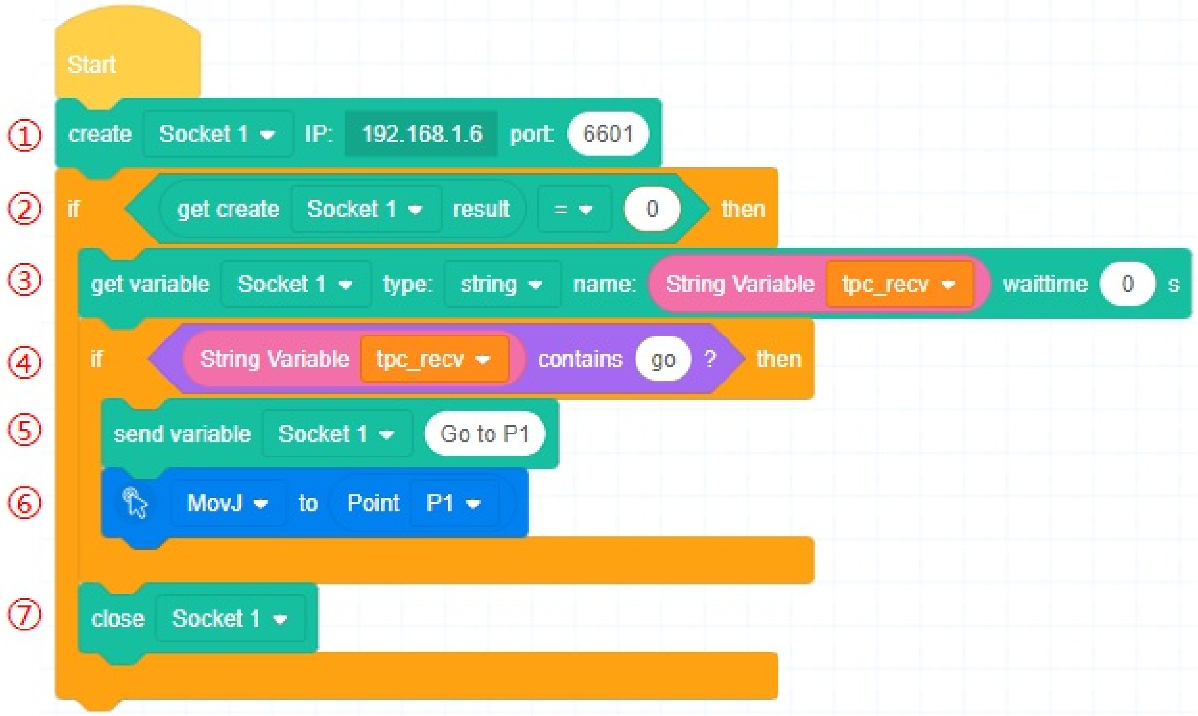
Scene description

To experience how to perform TCP communication through blockly programming, you can assume the following scene:

Create a TCP server for the robot. Wait for the client to connect to the server and send "go" command. Then the server returns "Go to P1" message and the robot starts to move to P1.

Steps for programming

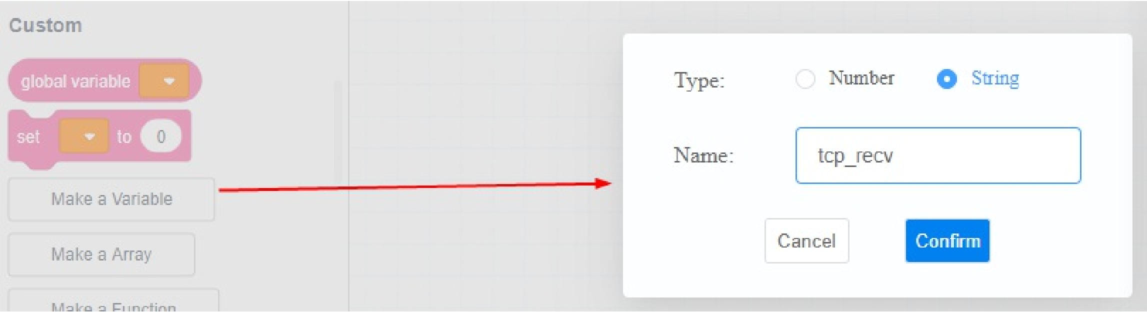
To achieve this scene, you need to edit the program as shown in the figure below.



1. Create the TCP server (Socket 1). Set the IP (robot IP) and port (custom) .

2. Determine whether the TCP server is created successfully. The subsequent steps will be executed only if the creation is successful, otherwise, the program will end directly.

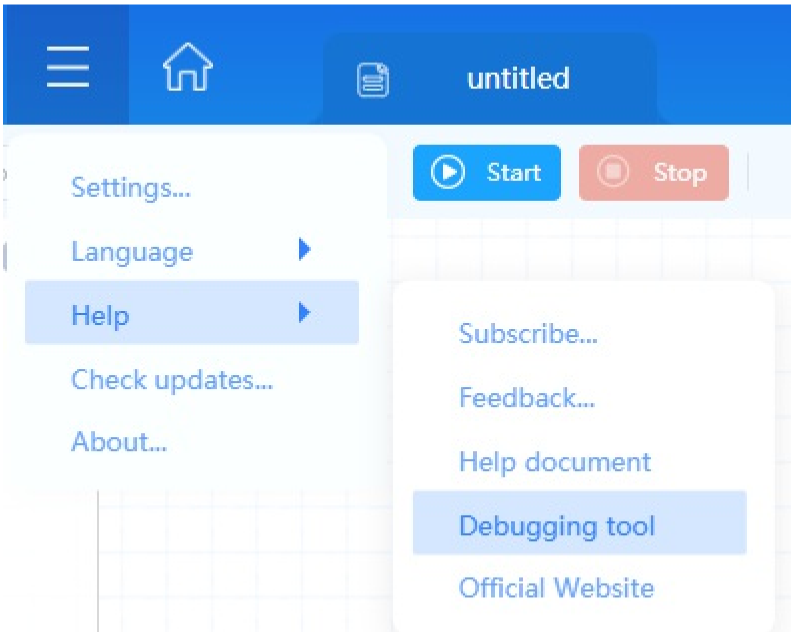
3. Wait for the client to connect and send the string. Save the received string to the string variable "tcp_recv". You need to create the string variable in advance.



4. Determine whether the received string includes "go". if it does, execute step 5 and 6. Otherwise, execute step 7 directly.
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.

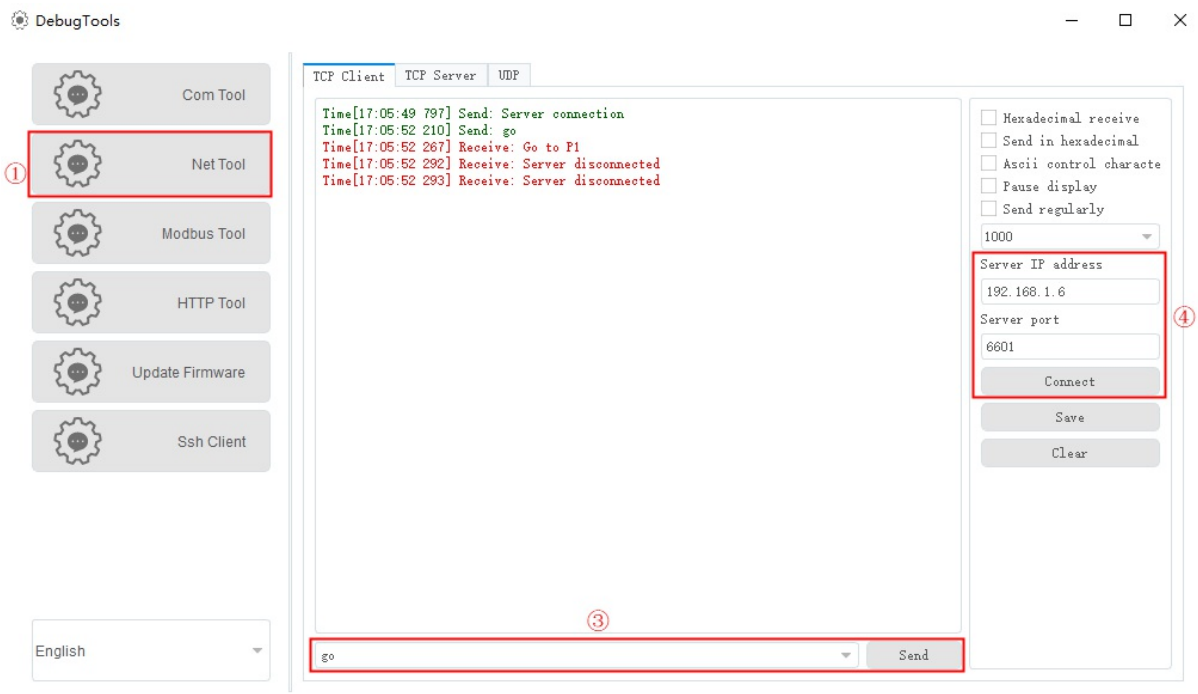
Run program

If you need to run the program quickly, you can use the debug tool of DobotStudio Pro as the TCP client.



1. Open the debug tool and enter "Net Tool" > "TCP Client" page.
2. Move the robot to a point other than P1 (for observing whether the robot executes the motion command). Then save and run the program.
3. After you see "Create TCP Server Success" in the running log, modify the IP address and port of the server in DebugTools page, and click **Connect**.
4. After the connection is successful, enter "go" at the bottom of DebugTools page and click **Send**.
5. Observe whether the debug tool receives the "Go to P1" message and whether the robot moves to P1.

The figure below shows the interface of the debug tool. The marked numbers correspond to the steps above.

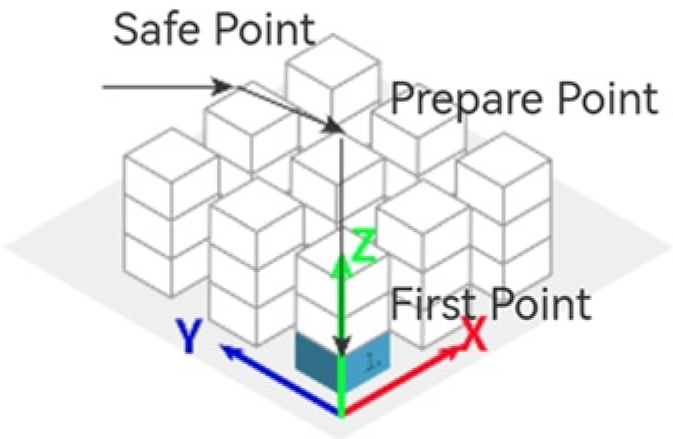


Palletize

Scene description

In a case in which the materials to be carried are arranged regularly and evenly spaced, teaching the position of each material one by one may lead to large errors and low efficiency. Palletizing process can effectively solve such problems.

Assume that the material needs to be stacked into a cube. You need to manually palletize a target stack type, and then teach the relevant points:

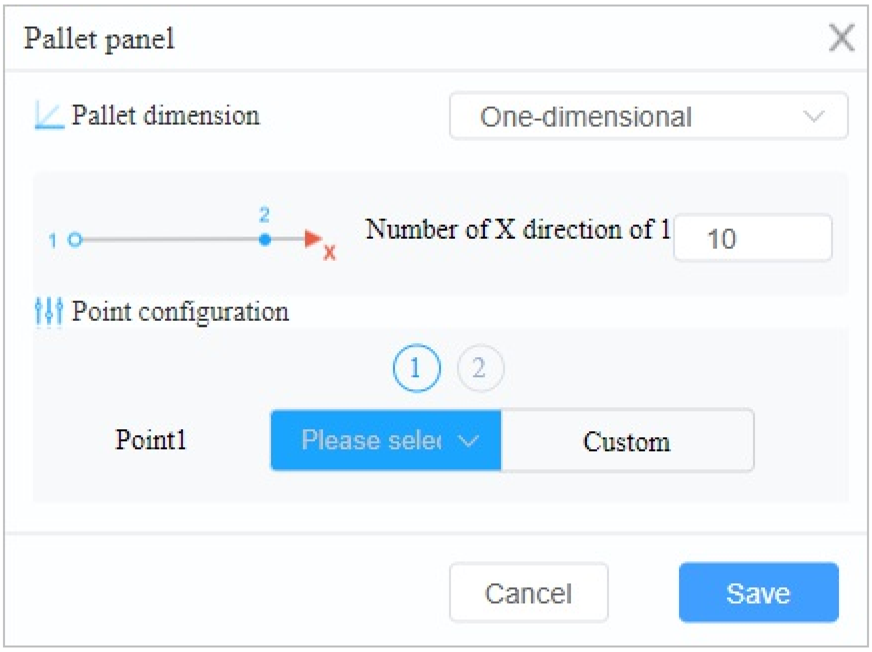


- Safe point (P1): A point the robot must move to when assembling or dismantling stacks for safe transition. It can be set to a point over the picking point.
- Picking point (P2).
- Preparation point and target point do not need to be taught one by one. Please refer to Configuring stack type.

Then assume that a gripper or suction cup has been installed at the end of the robot arm, which is controlled by controller DO1 to grip or release materials.

Configuring stack type

Drag the pallet block to the programming area, and click the block to open the pallet panel.



Pallet dimension

- One-dimensional: The materials are arranged in a row, and the total number of materials is equal to the number in the X direction.
- Two-dimensional: The materials are arranged in a square, and the total number of materials is equal to the product of the number in the X direction and the Y direction.
- Three-dimensional: The materials are stacked into a cube, and the total number of materials is equal to the product of the numbers in three directions.

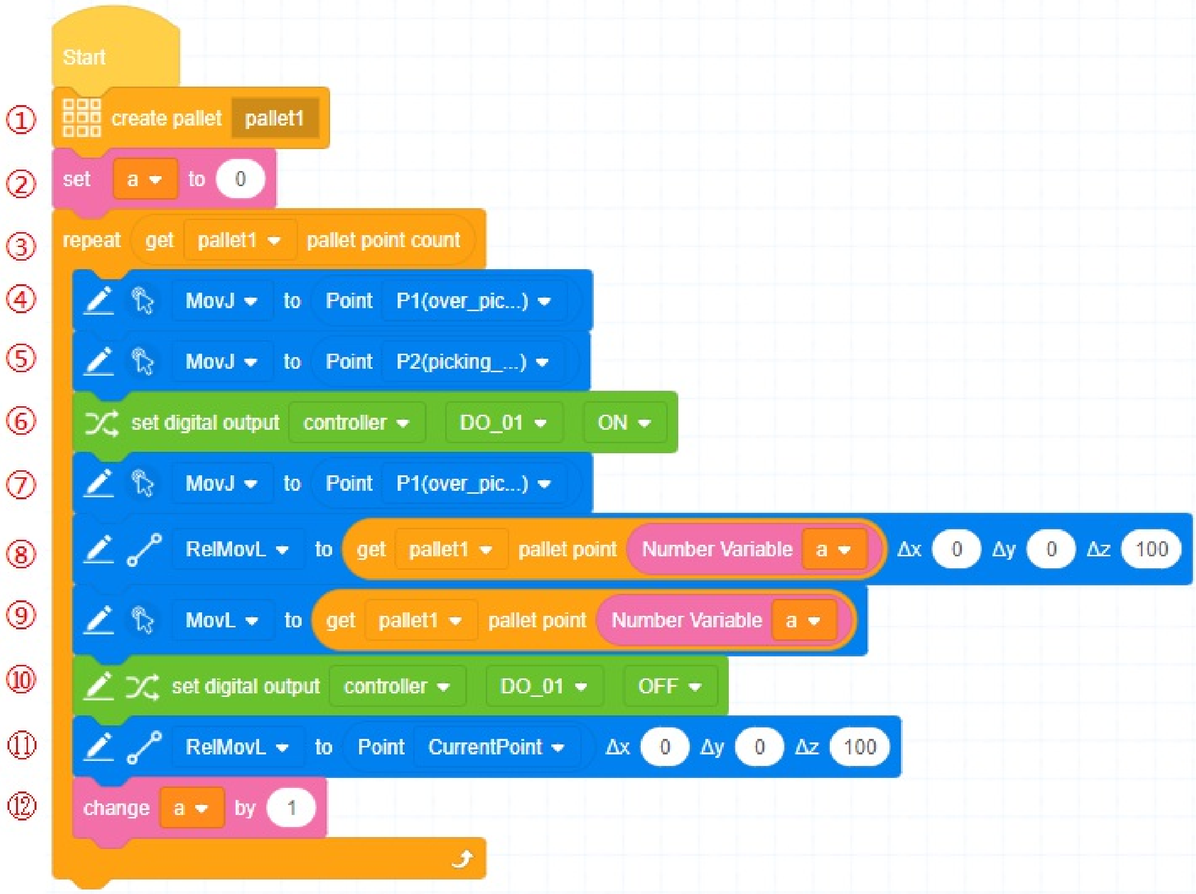
This section takes the three-dimensional stacking as an example. Here the number of materials in each direction is set to 10, so this demo contains 1000 materials.

Point configuration Taking the three-dimensional stack as an example, you need to configure eight points, which correspond to the material positions on the eight corners of the cube. The control system will automatically calculate the target point of each material through the eight points and the number of materials, and then perform palletizing in the order of X -> Y -> Z coordinate axes.

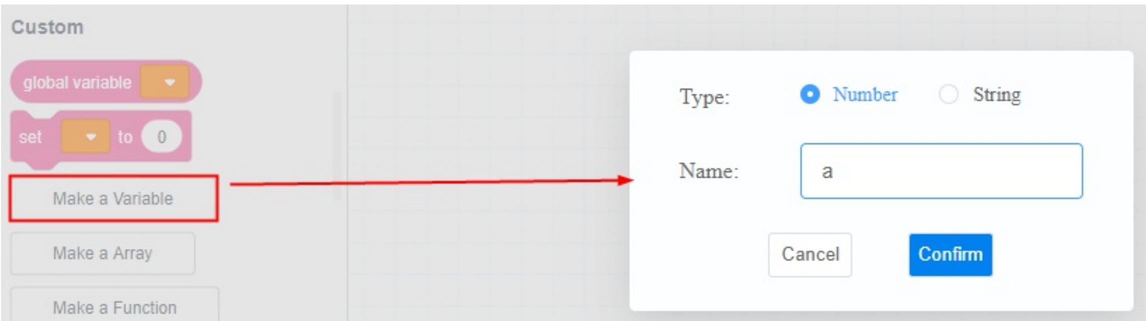
When configuring points, you can select the points that have been taught in the project, or you can click **Custom** to obtain the current point of the robot arm. The configured point icon will turn green.

Steps for programming

To achieve this scene, you need to edit the program as shown in the figure below.



1. Create pallet1.
2. Create a custom number variable and set it to 1, which is used to record the repeat times.



3. Execute the subsequent commands cyclically, and set the number of times to the total number of points corresponding to the pallet.
4. The robot moves over the picking point (P1).
5. The robot moves to the picking point (P2).
6. Set DO1 to ON to control the gripper to pick up the material.
7. The robot returns over the picking point (P1).
8. The robot moves to 100mm over the current pallet point.
9. The robot moves to the current pallet point.
10. Set DO1 to OFF to control the gripper to release the material.
11. The robot returns to 100mm over the current pallet point.

12. The repeat times is incremented by 1. Return to Step 4.

The program in this section is only a simple example. You can add more IO control and judgment commands according to the actual condition, such as not performing subsequent actions if the material is not picked up.

Run program

Run the program after teaching the points, configuring the stack type and programming. You can check the status of DO1 in the IO panel.

Block description

Event

The event commands are used as a mark to start running a program.

Start command

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 place other non-event blocks under the **Start** block to program.

Limitation: A project can only has one **Start** block.

Sub-thread start command

Description: It is the mark of the sub-thread of a program. The sub-thread will run synchronously with the main thread, but the sub-thread cannot call robot control commands. It can only perform variable operation or I/O control. Please determine whether to use the sub-thread according to the logic requirement.

Limitation: A project can only has five sub-threads.

Control

The control blocks are used to control the running path of the program.

Wait until…

Description: The program pauses running, and it continues to run if the parameter is true .

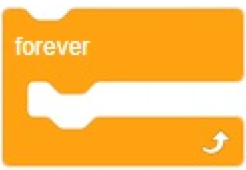
Parameter: Use other hexagonal blocks as the parameter.

Repeat n times

Description: Embed other blocks inside the block, and the embedded block command will be executed repeatedly for the specified times.

Parameter: number of times the execution is repeated.

Repeat continuously

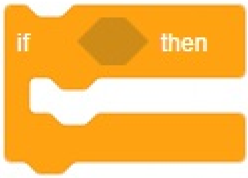


Description: When other blocks are embedded inside this block, the embedded commands will be executed repeatedly until meeting  .

End repetition

Description: It is used to be embedded inside the blocks for repeating execution. When the program runs to this block, it will directly end the repetition and execute the blocks after the block for repeating execution.

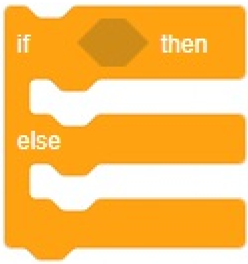
if…then…



Description: If the parameter is true, execute the embedded block. If the parameter is false, jump directly 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, execute the embedded blocks before "else". If the parameter is false, execute the embedded blocks after "else".

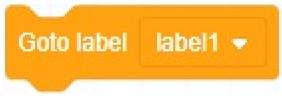
Parameter: Use other hexagonal blocks which return a Boolean value (true or false) as the parameter.

Repeat until…

Description: Repeatedly execute the embedded block until the parameter is true.

Parameter: Use other hexagonal blocks which return a Boolean value (true or false) as the parameter.

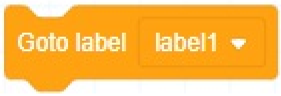
Set label

Description: Set a label, then you can jump to the label through 

.

Parameter: Label name, which must starts with a letter, and special characters such as spaces cannot be used.

Goto label



Description: When the program runs to the block, it will jump to the specified label directly and execute the blocks after the label.

Parameter: label name

Fold commands

Description: Fold the embedded blocks. It has no control effect but to make the program more readable.

Parameter: A name to describe the folded blocks

Pause

Description: The program pauses automatically after running to the block. It can continue to run only through control software or remote control operations.

Set collision detection

Description: Set collision detection. The collision detection level set through this block is valid only when the project is running, and will restore the previous value after the project stops.

Parameter: Select the sensitivity of the collision detection. You can turn it off or select from level 1 to level 5. The higher the level is, the more sensitive the collision detection is.

Modify user coordinate system

Description: Modify the specified user coordinate system. The modification is valid only when the project is running, and the coordinate system will restore the previous value after the project stops.

Parameter:

- Specify the index of user coordinate system
- Specify the parameters of modified user coordinate system

Modify tool coordinate system

Description: Modify the specified tool coordinate system. The modification is valid only when the project is running, and the coordinate system will restore the previous value after the project stops.

Parameter:

- Specify the index of tool coordinate system
- Specify the parameters of modified tool coordinate system

Create pallet

Description: Create the stack type of a pallet. See Palletizing for details.

Parameter: : pallet name

Obtain pallet point count

Description: Obtain the number of target points of the specified pallet

Parameter: : pallet name

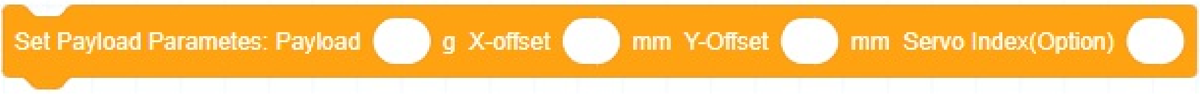
Obtain pallet point coordinates

Description: Obtain the specified point coordinates of the specified pallet

Parameter: :

- pallet name
- point index, starting from 1

Set load parameters

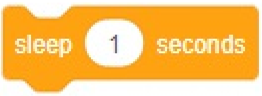


Description: Set the load parameters of the robot arm.

Parameter:

- load weight, which cannot exceed the maximum load weight of the robot arm. unit: g.
- If an eccentric tool is installed at the end, you need to set the corresponding eccentric coordinates. When no eccentric tool is installed, set it to 0. unit: mm.
- The servo index is an advanced function, which can be empty. If you need to use it, please set it under the guidance of the engineers.

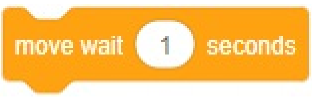
Delay execution



Description: When the program runs to the block, it will pause for a specified time before it continues to run.

Parameter: pause time of the program

Motion waiting



Description: It is used before or after a motion block to delay the delivery of motion commands or delay the delivery of the next command after the former motion is completed.

Parameter: delay time to deliver the command

Get system time

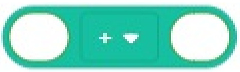
Description: Get the current time of the system.

Return: Unix timestamp of the current system time.

Operator

The operator commands are used for calculating 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 numeric values, or directly enter the value in the blanks.
- Select an operator.

Return: Value after operation

Comparison command

Description: Compare the parameters.

Parameter:

- Fill in both blanks with variables or constants. You can use oval blocks that return numeric values, or directly enter values in the blanks.
- Select a comparison operator.

Return: It returns **true** if the comparison result is true, and **false** if the result is false.

A and B Command

Description: Perform **and** operation to the parameters.

Parameter: Fill in both blanks with variables (using hexagonal blocks).

Return: It returns **true** if the two parameters are true, and **false** if any one of them is false.

A or B Command

Description: Perform **or** operation to the parameters.

Parameter: Fill in both blanks (using hexagonal blocks).

Return: It returns **true** if any one of the parameters is true, and **false** if both of them are false.

Not A Command

Description: Perform **not** operation to the parameters.

Parameter: Fill in the blank with a variable (using hexagonal blocks).

Return: It returns **false** if the parameter is true, and **true** if the parameter is false.

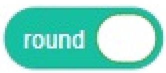
Get Remainder

Description: Get the remainder of parameters.

Parameter: Fill in both blanks with variables or constants. You can use oval blocks that return numeric values, or directly fill the value in the blanks.

Return: Value after operation

Round-off Operation

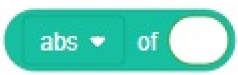


Description: Perform round-off operation to parameters.

Parameter: Fill in the blank with a variable or constant. You can use oval blocks that return numeric values, or directly fill the value in the blank.

Return: Value after operation

Monadic operation



Description: Perform various Monadic operations to parameters.

Parameter:

- Select an operator.
    - abs
    - floor
    - ceiling
    - sqrt
    - sin
    - cos
    - tan
    - asin
    - acos
    - atan
    - ln
    - loh
    - e^
    - 10^
- Fill in the blank with a variable or constant. You can use oval blocks that return numeric values, or directly fill the value in the blank.

Return: Value after operation

Print command

Description: Output the parameters to the console, which is mainly used for debugging.

Parameter:

- Select **Sync** or **Async**. For **Sync**, it will print information after all the commands that have been delivered are executed. For **Async**, it will print information immediately when the program runs to the block.
- Variables or constants to be output. You can use oval blocks, or directly fill in the blank.

String

The string commands include general functions of string and array.

Get character in a certain position of string

Description: Get the character in the specified position of the string.

Parameter:

- 1st parameter: specify the position of character to be returned in the string
- 2nd parameter: string, you can use other oval blocks or fill in directly.

Return: character in the specified position of the string

Determine whether String A contains String B

Description: Determine whether the first string contains the second string.

Parameter: Two strings. You can use oval blocks which return string, or fill in directly.

Return: If the first string contains the second string, it returns **true**, otherwise it returns **false**.

Connect two strings

Description: Connect two strings into one string. The second string will follow the first string.

Parameter: Two strings to be connected. You can use oval blocks which return string, or fill in directly.

Return: Jointed string.

Get length of string or array

Description: Get the length of the specified string or array. The length of a string refers to how many characters the string has, and the length of an array refers to how many elements the array has.

Parameter: A string or array. You can use oval blocks that return string or array.

Return: length of string or array

Compare two strings

Description: Compare the sizes of two strings according to ACSII codes.

Parameter: Two strings to be compared. You can use oval blocks which return string, or fill in directly.

Return: It returns 0 when string 1 and string 2 are equal, -1 when string 1 is less than string 2, and 1 when string 1 is greater than string 2.

Convert array to string

Description: Convert the specified array to a string, and the different array elements in the string are separated by the specified delimiter. For example, if the array is {1,2,3} and the delimiter is |, then the converted string is "1|2|3".

Parameter:

- An array to be converted to string. You can use oval blocks which return string
- Delimiter used in conversion

Return: Converted string.

Convert string to array

Description: Convert the specified string to an array, using the specified delimiter to separate strings. For example, if the array is “1|2|3” and the delimiter is |, then the converted array is {[1]=1,[2]=2,[3]=3}.

Parameter:

- A string to be converted to array. You can use oval blocks which return string or fill in directly
- Delimiter used in conversion

Return: Converted array.

Get element in a certain position of array

Description: Get the element at the specified subscript position in the specified array. The subscript represents the position of the element in the array. For example, the subscript of 8 in the array {7,8,9} is 2.

Parameter:

- Target array, using oval blocks which return array values.
- subscript of specified element.

Return: value of the element at the specified position in the array.

Get multiple specified character of string

Description: Get multiple elements at the specified subscript position in the specified array. Get the element based on the step value within the range of the start and end subscripts.

Parameter:

- Target array, using oval blocks which return array values.
- Specify the range of elements by start subscript and end subscript.
- Step value is used to determine how often elements are obtained. 1 refers to obtaining all, and 2 refers to obtaining every other element, and so forth.

Return: new array of specified elements.

Set specified character of array

Description: Set the value of the element at the specified position of the array.

Parameter:

- target array, using oval blocks that return array values.
- subscript of the element.
- value of element.

Custom

The custom commands are used for creating and managing custom blocks, and calling global variables.

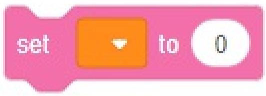
Call global variable

Description: Call global variables set in the control software.

Parameter: Name of a global variable.

Return: Value of the global variable.

Set global variables



Description: Set the value of a specified variable. Please note that the block for setting global variables and setting custom variables are the same in shape, but have slightly different functions.

Parameter:

- Select a variable to be modified.
- Value after modification. You can directly fill the value in the blank, or use other oval blocks.

Create variables

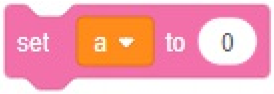
Click to create a variable. The variable name must start with a letter and cannot contain special characters such as Spaces. After creating at least one variable, you will see the following variable blocks in the block list.

Custom number variable

Description: The newly created custom number variable (default value: nil) is recommended to be used after assignment. You can also modify the variable name or delete the variable through the variable drop-down list.

Return: variable value

Set value of custom number variable

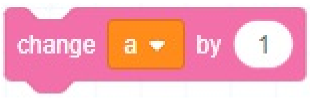


Description: Set the value of a specified number variable. Please note that the block for setting global variables and setting custom variables are the same in shape, but have slightly different functions.

Parameter:

- Select a variable to be modified.
- Value after modification. You can directly fill the value in the blank, or use other oval blocks.

Add value of number variable



Description: Add specified value to a number variable.

Parameter:

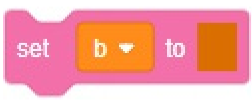
- Select a variable to be modified.
- Added value. You can directly fill the value in the blank, or use other oval blocks. A negative value refers to value decrease.

Custom string variable

Description: The newly created custom string variable (default value: nil) is recommended to be used after assignment. You can also modify the variable name or delete the variable through the variable drop-down list.

Return: variable value

Set value of custom string variable



Description: Set the specified string variable.

Parameter:

- Select a variable to be modified.
- Value after modification. You can directly fill the blank with a string.

Create array

Click to create a custom array. The array name must start with a letter and cannot contain special characters such as Spaces. After creating at least one array, you will see the following array blocks in the block list.

Custom array

Description: The newly created custom array is an empty array by default. It is recommended to use it after assignment. Right-click (PC)/long-press (Android or iOS) the block in the block list to modify the name of the array or delete the array. You can also modify the name of the currently selected array or delete the array through the array drop-down list in other array blocks. The check box on the left side of the array block has no use, which can be ignored.

Return: Array value.

Add variable to array

Description: Add a variable to a specified array. The added variable will be the last item of the array.

Parameter:

- Variable to be added. You can directly fill the variable in the blank, or use other oval blocks.
- Select an array to be modified.

Delete item of array

Description: Delete an item of a specified array.

Parameter:

- Select an array to be modified.

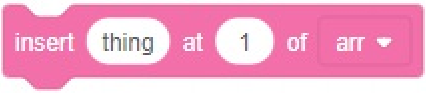
- Item index. You can directly fill the index in the blank, or use other oval blocks that return numeric values.

Delete all items of array

Description: Delete all items of the array.

Parameter: Select an array to be modified.

Insert item into array

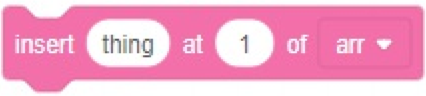


Description: Insert an item to a specified position of the array.

Parameter:

- Select an array to be modified.
- insert position. You can directly fill the index in the blank, or use other oval blocks that return numeric values.
- Variable to be added. You can directly fill the variable in the blank, or use other oval blocks.

Replace items of array



Description: Replace an item of the array with a specified variable.

Parameter:

- Select an array to be modified.
- Item index. You can directly fill the index in the blank, or use other oval blocks that return numeric values.
- Variable after replacement. You can directly fill the variable in the blank, or use other oval blocks.

Get items of array

Description: Get the value of a specified item of the array.

Parameter:

- Select an array.
- Item index. You can directly fill the index in the blank, or use other oval blocks that return numeric values.

Return: value of specified item

Get number of items in array

Description: Get the number of items in an array.

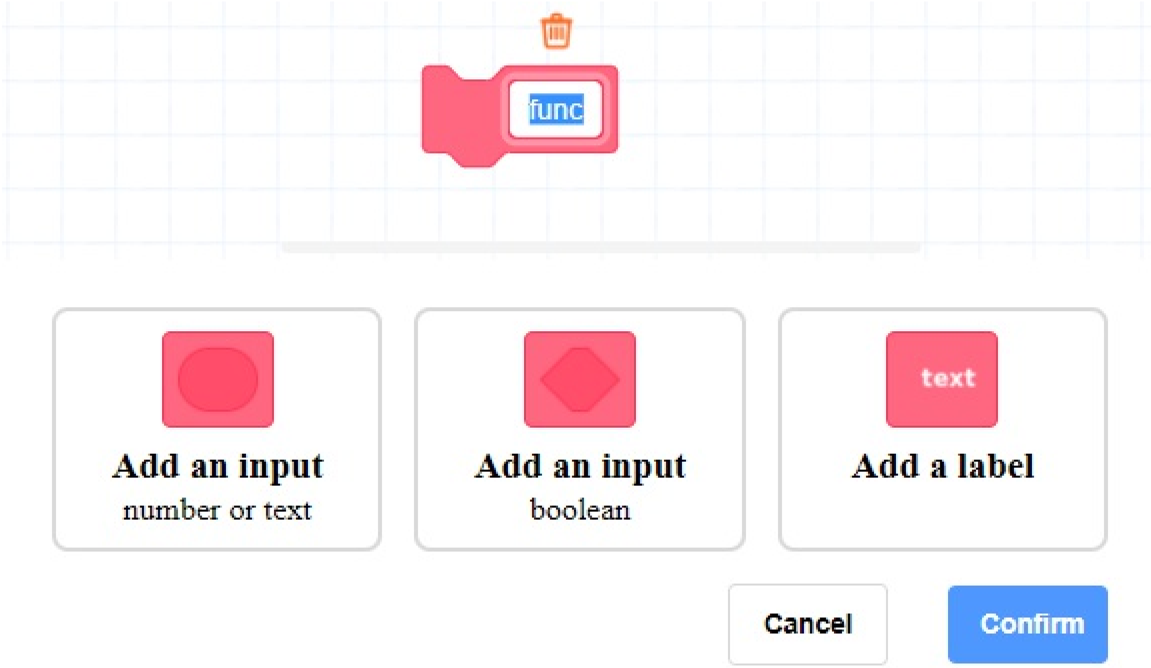
Parameter: Select an array.

Return: Number of items in the array.

Create function

Click to create a new function. A function is a fixed program segment. You can define a group of blocks that implement specific functions as a function. Every time you want to use the function, you only need to call this function with no need to build the same block group repeatedly. A new created function needs to be declared and defined. After the new function is created successfully, the corresponding function block will appear in the block list.

1. **Declare function**



In this interface, you need to define the name of the function, and the type, quantity and name of the input (parameter). The function and parameter names should not contain special characters such as spaces. You can also add labels to functions, which can be used as comments for functions or inputs.

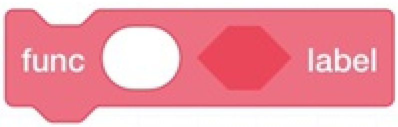
1. **Define function**

After completing the function declaration, you will see the definition header block in the programming area.

You need to program below the header block to define the function.

You can drag out the input in the header block to use in the blocks below, indicating using the input when actually calling the function as a parameter.

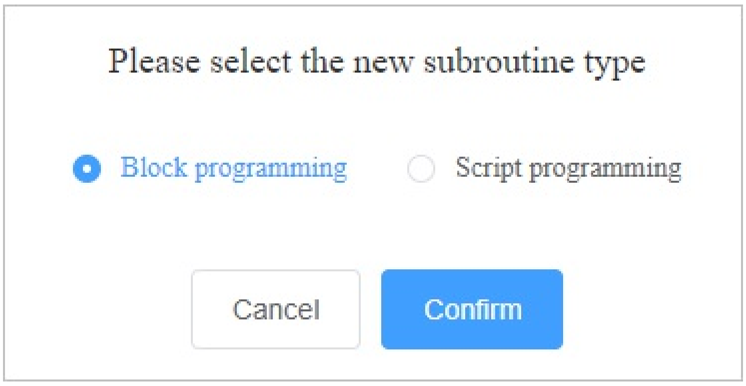
Custom function



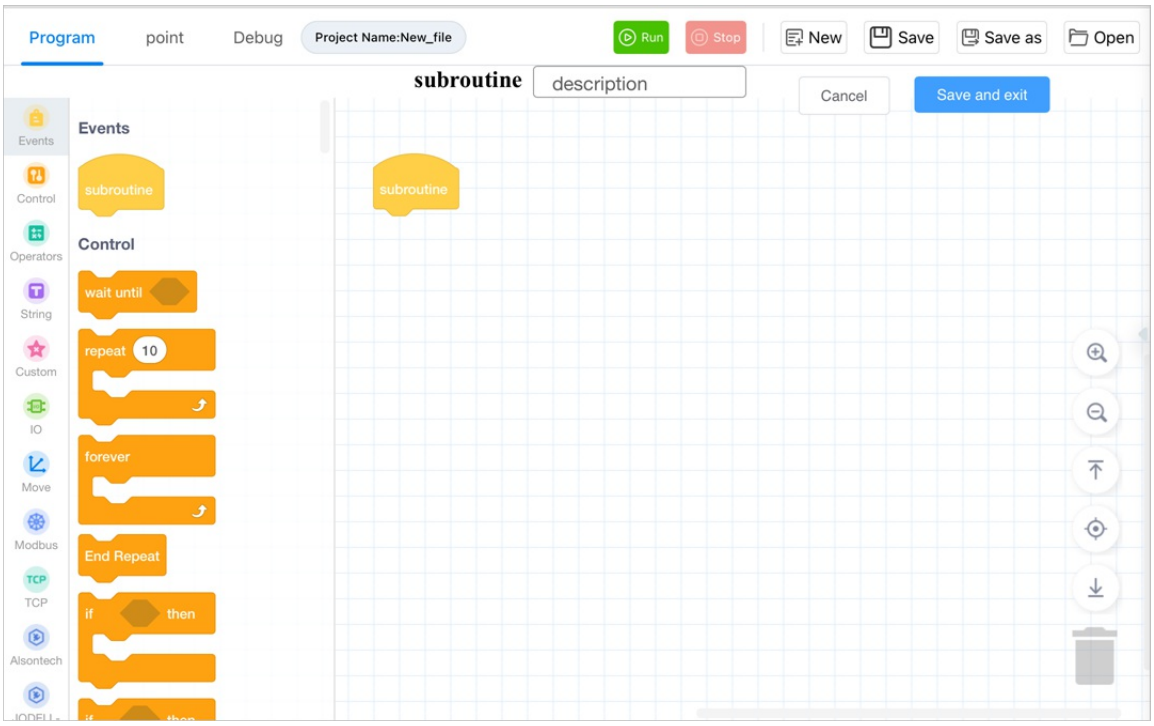
Description: The custom function blocks, of which the name and input parameters are defined by the user, are used to call the defined function. Right-clicking (PC)/long-pressing (App) the block in the block list can modify the declaration of the function. If you need to delete the function, delete the definition header block of the function.

Create sub-routine

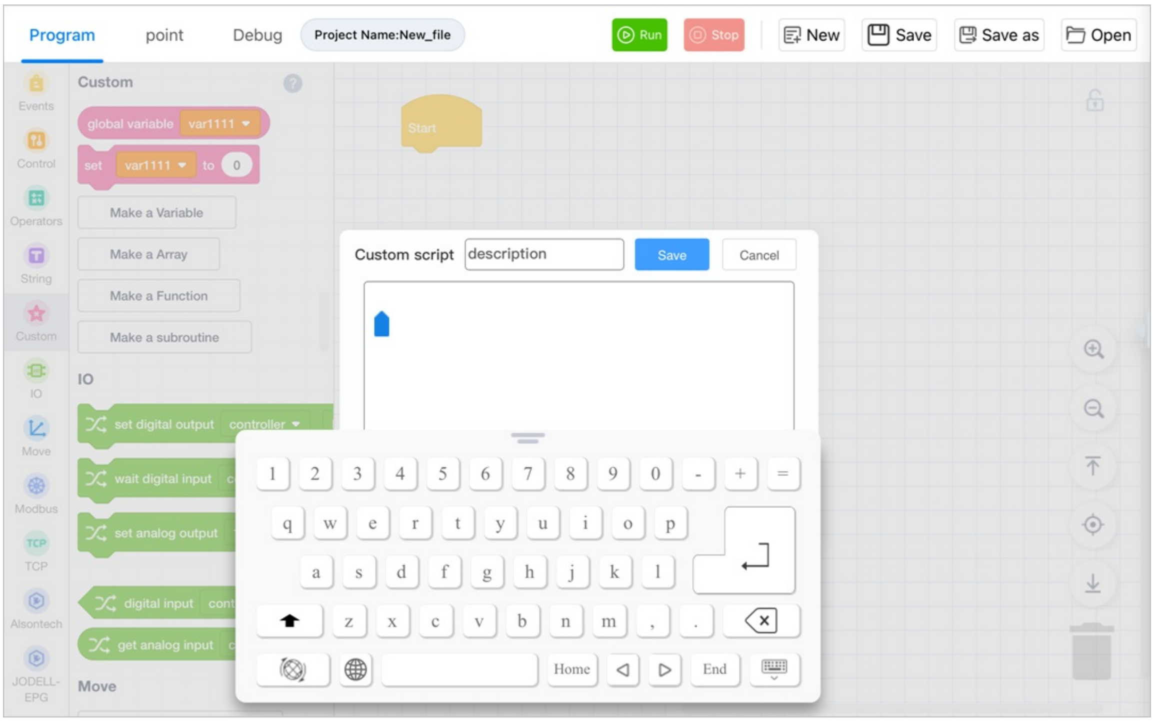
Click to create a new sub-routine. Blockly programming supports embedding and calling sub-routines, which can be blockly programming and script programming, with a maximum of two embedded levels. After the new sub-routine is successfully created, the corresponding sub-routine block will appear in the block list.



- After selecting **Block programming**, you will see the sub-routine block programming page. You can set the sub-routine description and write the subroutine.



- After selecting **Script programming**, you will see the sub-routine script programming window. You can set the sub-routine description and write the subprogram.



Sub-routine

- Blockly sub-routine

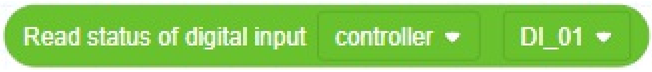
- Script sub-routine

Description: The sub-routine block, which is defined by the user when creating a sub-routine, is used to call the saved sub-routine. Right-clicking (PC)/long-pressing (App) the block in the block list can modify or delete the sub-routine.

IO

The IO blocks are used to manage the input and output of the IO terminals of the robot arm. The value range of the input and output ports is determined by the corresponding number of terminals of the robot arm. Please refer to the hardware guide of the corresponding robot arm.

Get digital input



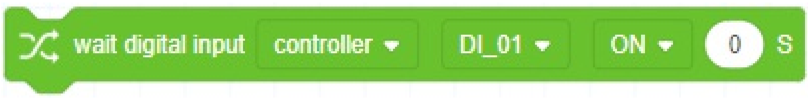
Description: Get the status of the specified DI.

Parameter:

- Select the position of DI port, including controller (base) and tool
- Select DI port index

Return: status of the specified DI. 0 refers to OFF, and 1 refers to ON.

Wait digital input



Description: Wait for the specified DI to meet the condition or wait for timeout before executing subsequent block commands.

Parameter:

- Select the position of DI port, including controller and tool
- Select DI port index
- Select the status (ON or OFF)
- timeout for waiting (0 means waiting until the condition is met)

Set digital output

Description: Set the on/off status of digital output port.

Parameter:

- Select the position of DO port, including controller and tool

- Select DO port index
- Select the output status (ON or OFF)

Set digital output (for sub-thread)

Description: Set the on/off status of digital output port. Please use this block when setting in the sub-thread.

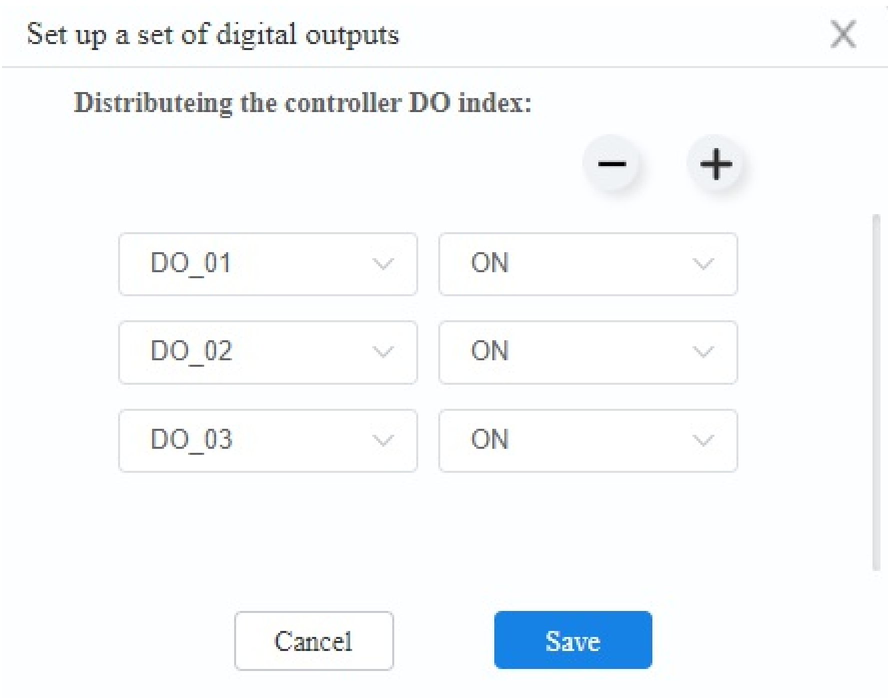
Parameter:

- Select the position of DO port, including controller and tool
- Select DO port index
- Select the output status (ON or OFF)

Set a group of digital output

Description: Set a group of DO. You can drag the block to the programming area and click to set it.

Parameter:



- click + or - to increase or decrease the number of DO
- Select DO port index

- Select the output status (ON or OFF)

Motion

The motion commands are used to control the movement of the robot arm and set motion-related parameters.

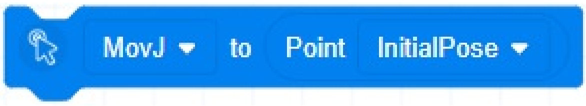
The motion blocks are all asynchronous commands, that is, after the command is successfully delivered, the next command will be executed without waiting for the robot to complete the current movement. You can use **sync** command if you need to wait for the delivered commands to be executed before executing subsequent commands.

The point parameters can be selected here after being added on the "Point" page of the project. The motion blocks also support dragging out the default variable block and replacing it with other oval blocks which return Cartesian point coordinates.

Advanced configuration

When the preset motion block cannot meet the programming requirements, you can create a block that controls the robot motion through advanced configuration. The created block will appear in the programming area. For details, refer to Motion advanced configuration.

Move to target point



Description: Control the robot to move from the current position to the target point. After dragging the blocks to the programming area, double-click to perform advanced configuration. See Motion advanced configuration for details.

Parameter:

- Select a motion mode, including joint motion (MovJ) and linear motion (MovL). For joint motion, the trajectory is non-linear, and all joints complete the motion simultaneously.
- target point

Move to target point (with offset)

Description: Control the robot to move from the current position to a target point after offset.

Parameter:

- Select a motion mode, including relative joint motion (RelMovJ) and relative linear motion (RelMovL).
- offset in the X-axis, Y-axis, Z-axis and R-axis direction relative to the target point under the Cartesian coordinate system. unit: mm

Jump motion

Description: Move from the current position to the target position under the Cartesian coordinate system in a door-shaped mode.

1. The robot arm will first raise the specified height vertically, and then transition to the maximum height.
2. The robot arm moves towards the target point in a linear mode.
3. When the robot arm moves near the target point, transition to the specified height above the target point, and then descend vertically to the target point.

Parameter:

- target point
- lifting height of the starting point, unit: mm
- descent height of the end point, unit: mm
- maximum lifting height, unit: mm

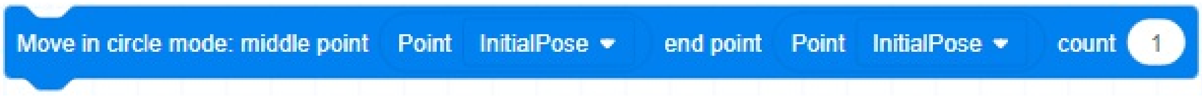
Jump motion (with preset jump parameters)

Description: Move from the current position to the target position under the Cartesian coordinate system in a door-shaped mode (using preset jump parameters).

Parameter:

- target point
- Select the jump motion index. You need to set the corresponding parameters in Settings > Motion parameter > Jump Setting, and enable the robot.

Circle motion



Description: Control the robot arm to move from the current position in an full-circle interpolated mode, and return to the current position after moving a specified number of circles. The coordinates of the current position should not be on the straight line determined by the intermediate point and the end point.

Parameter:

- **Middle point** is an intermediate point to determine the entire circle.
- **End point** is used to determine the entire circle.
- Enter the number of circles for circle movement, range: 1~ 999.

Arc motion

Description: Control the robot to move from the current position to a target position in an arc interpolated mode under Cartesian coordinate system. The coordinates of the current position should not be on the straight line determined by the intermediate point and the end point.

Parameter:

- **Middle point** is an intermediate point to determine the arc.
- **End point** is the target point.

Control aux joint motion

Description: Control the aux joint to move. The command can be used only after you have installed and configured the aux joint in the process.

Parameter:

- Set the angle or distance of motion. The meaning of this parameter depends on the type of motion (joint/linear) set in Advanced Settings in the Aux Joint process. unit: degree (when the type is joint) or mm (when the type is line).
- Set the speed ratio when moving.
- Set the acceleration ratio when moving.
- Set the mode of the command:
  - Sync: execute the next command after the movement is completed.
  - Async: After an command is delivered, execute the next command directly without waiting for the motion to be completed.

Set joint acceleration ratio

Description: Set the acceleration ratio of joint motion.

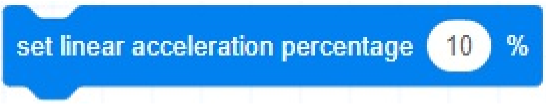
Parameter: joint acceleration ratio, range: 0~100. Actual robot acceleration = percentage set in blocks × acceleration in playback settings × global speed ratio.

Set joint speed ratio

Description: Set the speed ratio of joint motion.

Parameter: joint speed ratio, range: 0~100. Actual robot speed = percentage set in blocks × speed in playback settings × global speed ratio.

Set linear acceleration ratio



Description: Set the acceleration ratio of lineal and arc motion.

Parameter:

- Linear and arc acceleration ratio (value range: 0~100). Actual robot acceleration = set ratio × value in playback settings in software × global speed ratio.

Set linear speed ratio

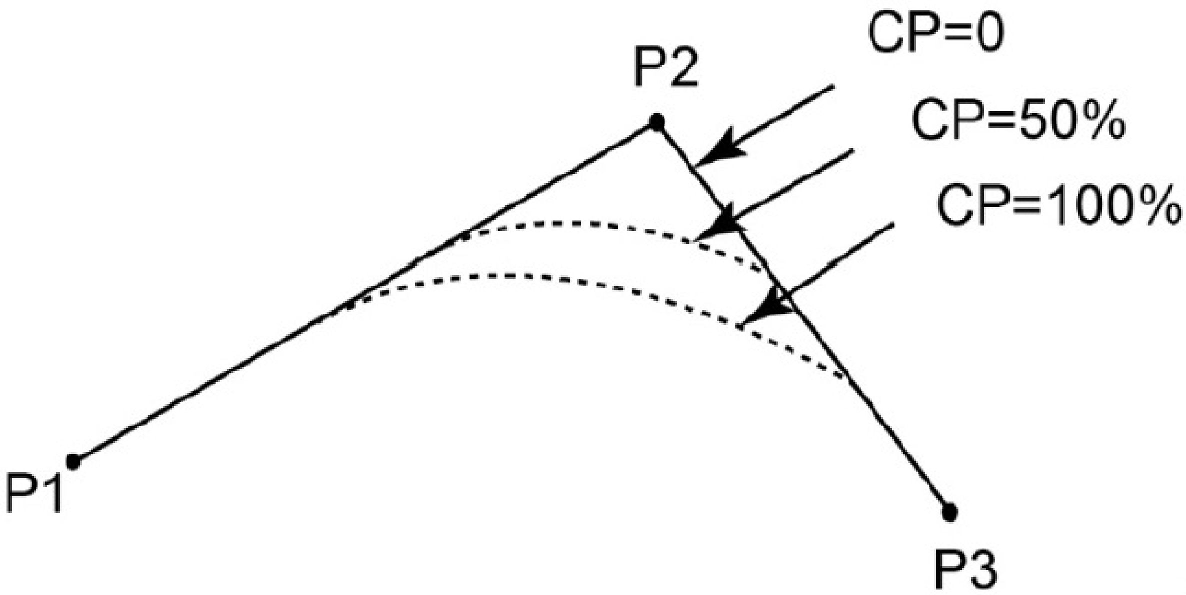
Description: Set the speed ratio of lineal motion. Actual robot speed = percentage set in blocks × speed in playback settings × global speed ratio.

Parameter:

- Linear speed ratio, range: 0~100.

Set CP ratio

Description: Set the continuous path ratio in motion, that is, when the robot moves from the starting point to the end point via the intermediate point, whether it passes the intermediate point through right angle or in curve, as shown below.

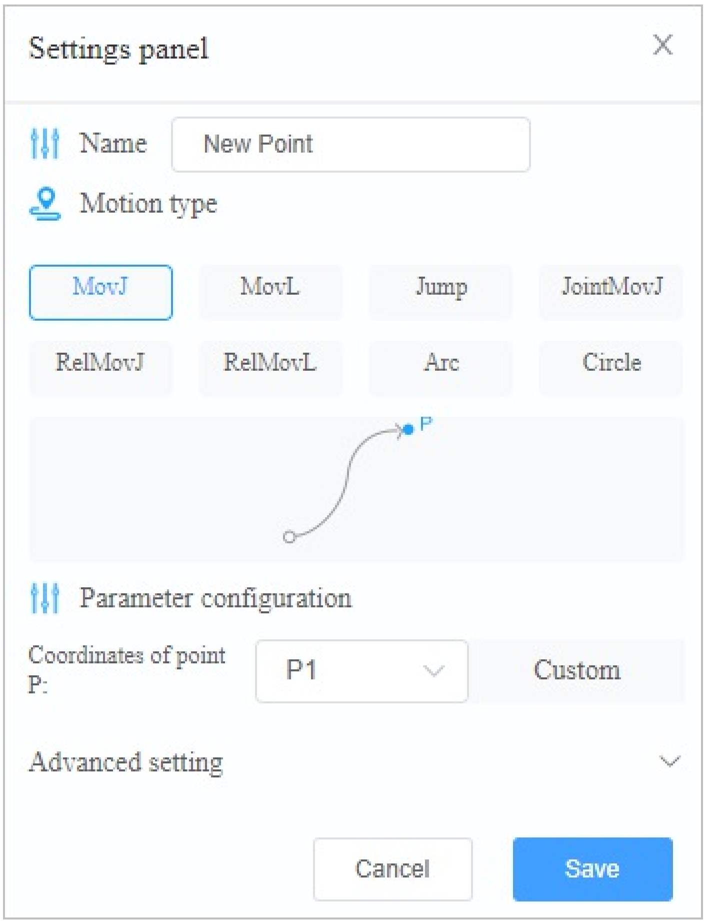


Parameter: Continuous path ratio, range: 0~100.

Sync command

Description: When the program runs to this command, it will wait for the robotic arm to execute all the commands that have been delivered before, and then continue to execute subsequent commands.

Motion advanced configuration

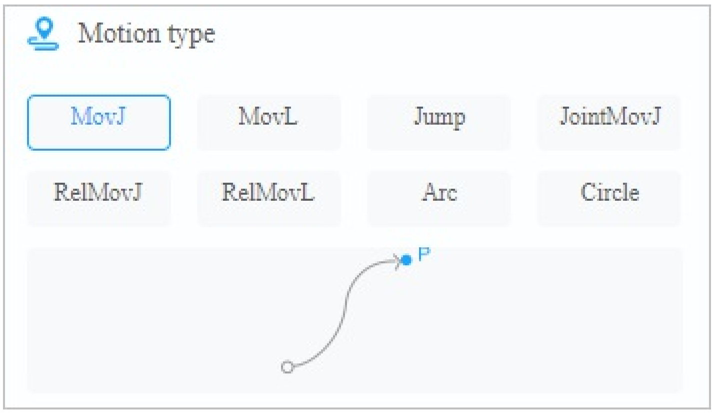


Create a block that controls the movement of the robot through advanced configuration. The configuration includes the block name, motion mode and motion parameters. Different motion modes vary in the motion parameters to be configured.

Actual robot speed/acceleration = percentage set in commands × speed/acceleration in playback settings × global speed ratio.

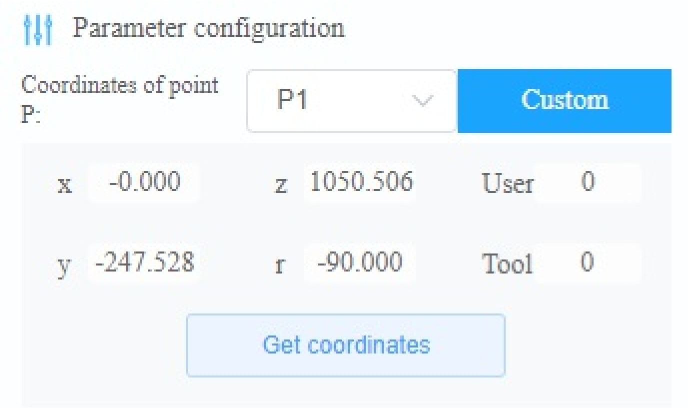
MovJ

Motion mode: Move from the current position to the target position under the Cartesian coordinate system in a joint-interpolated mode.



Basic setting:

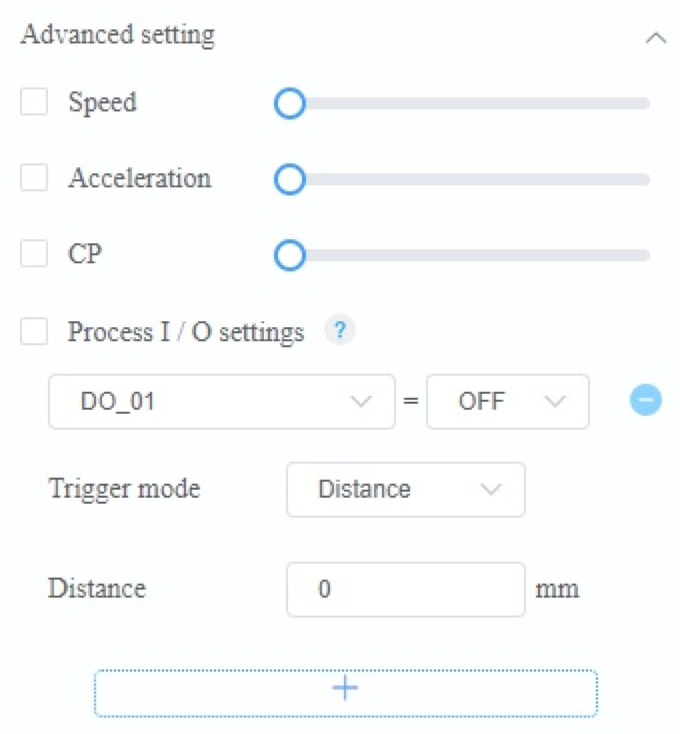
P: target point, which can be selected here after being added in the Point page, or defined in this page.



Advanced setting:

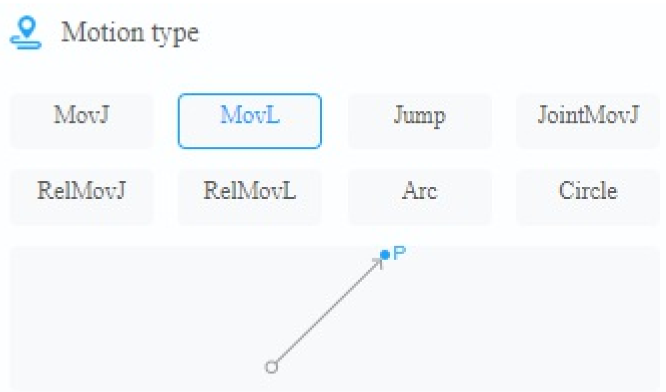
Select and configure the advanced parameters as required.

- Speed: velocity rate, range: 1~100.
- Acceleration (Accel): acceleration rate, range: 1~100.
- CP: set continuous path in motion, range: 0~100. See Continuous path (CP) at the end of this section for details.
- Process I/O settings: When the robot arm moves to the specified distance or percentage, the specified DO will be triggered. When the distance is positive, it refers to the distance away from the starting point; and when the distance is negative, it refers to the distance away from the target point. You can click "+" below to add a process IO, and click "-" on the right to delete the corresponding process IO.

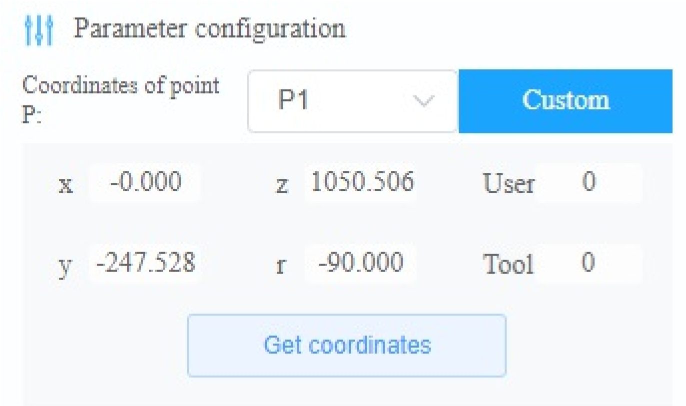


MovL

Motion mode: Move from the current position to the target position under the Cartesian coordinate system in a linear interpolated mode.



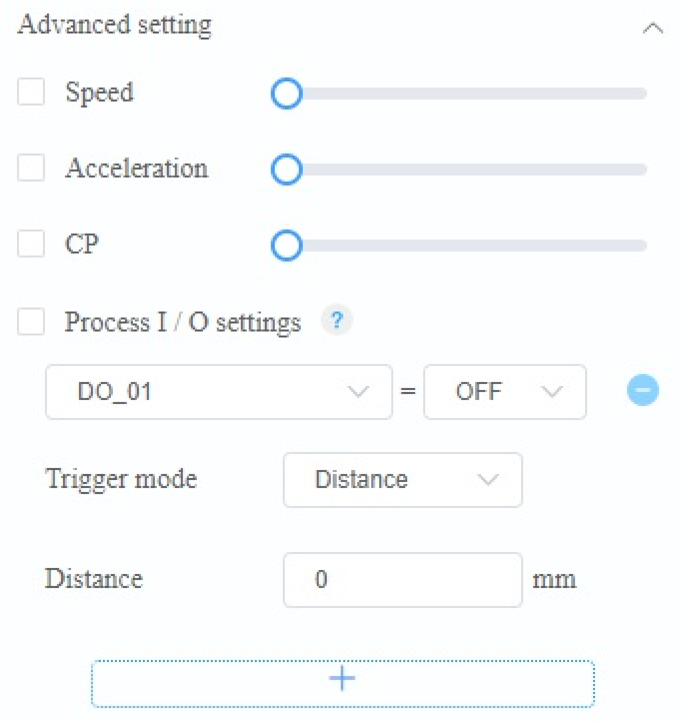
Basic setting: P: target point, which can be selected here after being added in the Point page, or defined in this page.



Advanced setting:

Select and configure the advanced parameters as required.

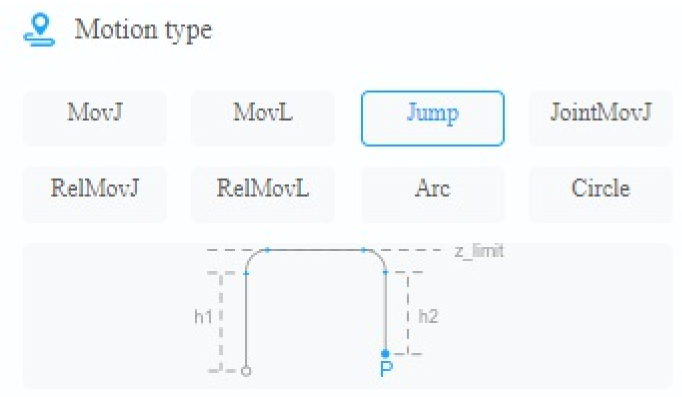
- Speed: velocity rate, range: 1~100.
- Acceleration (Accel): acceleration rate, range: 1~100.
- CP: set continuous path in motion, range: 0~100. See Continuous path (CP) at the end of this section for details.
- Process I/O settings: When the robot arm moves to the specified distance or percentage, the specified DO will be triggered. When the distance is positive, it refers to the distance away from the starting point; and when the distance is negative, it refers to the distance away from the target point. You can click "+" below to add a process IO, and click "-" on the right to delete the corresponding process IO.



Jump

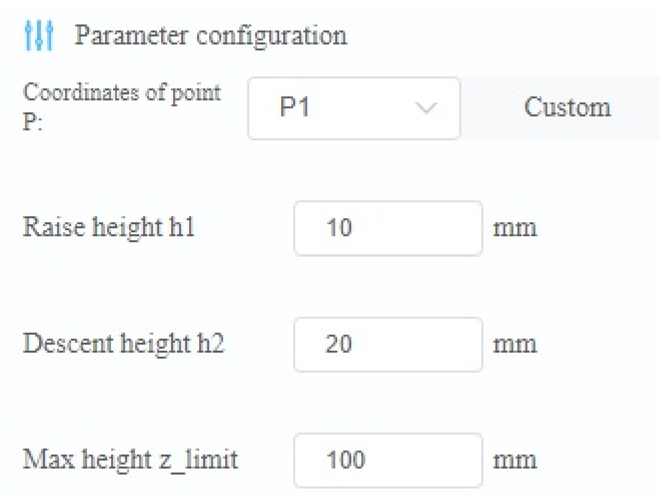
Motion mode: Move from the current position to the target position under the Cartesian coordinate system in a door-shaped mode.

1. The robot arm will first raise the specified height vertically, and then transition to the maximum height.
2. The robot arm moves towards the target point in a linear mode.
3. When moving near the target point, transition to the specified height above the target point, and then descend vertically to the target point.



Basic setting:

- Coordinates of point P: target point coordinates, which can be selected here after being added in the Point page, or defined in this page.
- lifting height (h1): lifting height of the starting point
- descent height (h2): descent height of the end point
- Max height (z_limit): maximum lifting height. You can refer to the diagram above for the relations among the three heights

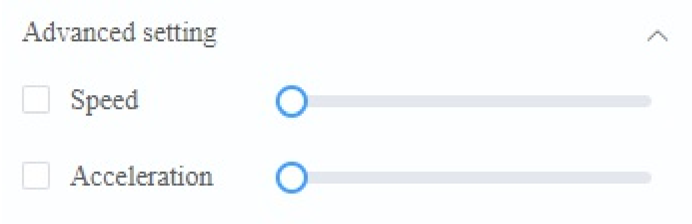


Advanced setting:

Select and configure the advanced parameters as required.

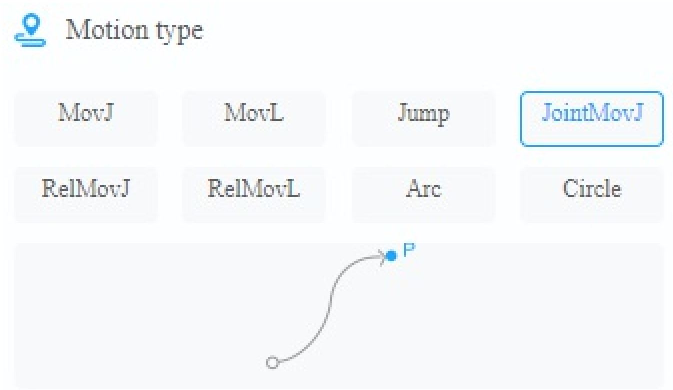
- Speed: velocity rate, range: 1~100.

- Accel: acceleration rate, range: 1~100.

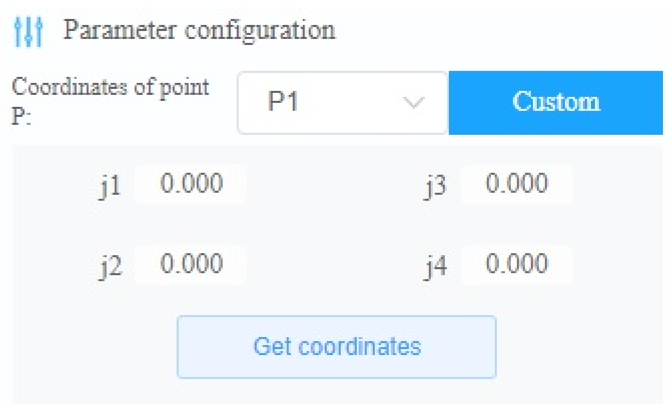


JointMovJ

Motion mode: Move from the current position to the target joint angle in a joint-interpolated mode.



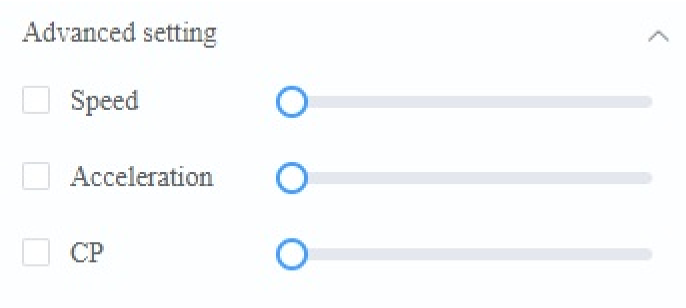
Basic setting: target joint angle, which can be defined through teaching.



Advanced setting:

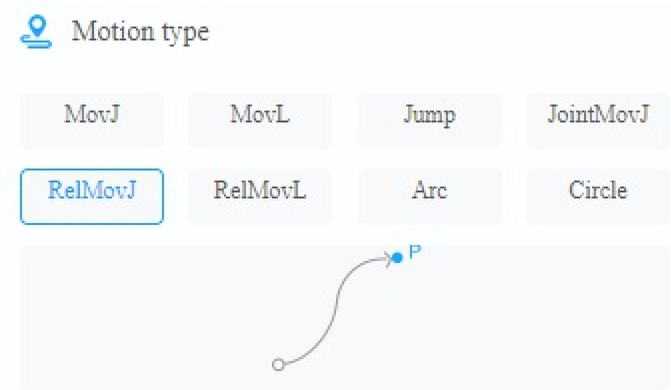
Select and configure the advanced parameters as required.

- Speed: velocity rate, range: 1~100.
- Acceleration (Accel): acceleration rate, range: 1~100.
- CP: set continuous path in motion, range: 0~100. See Continuous path (CP) at the end of this section for details.

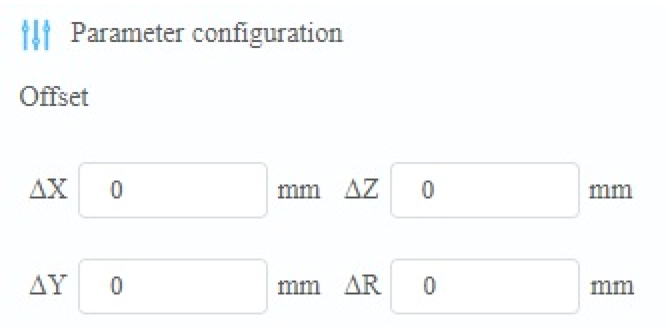


RelMovJ

Motion mode: Move from the current position to the target offset position under the Cartesian coordinate system in a joint-interpolated mode.



Basic setting: X-axis, Y-axis and Z-axis offset under the Cartesian coordinate system, unit: mm

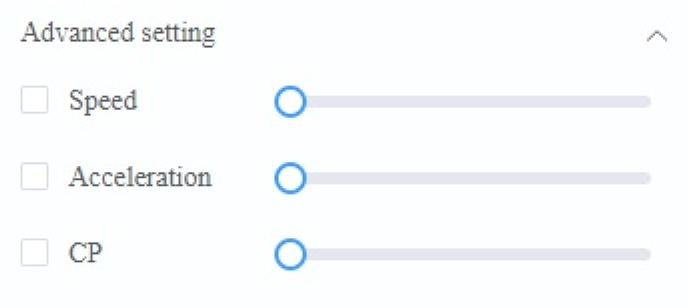


Advanced setting:

Select and configure the advanced parameters as required.

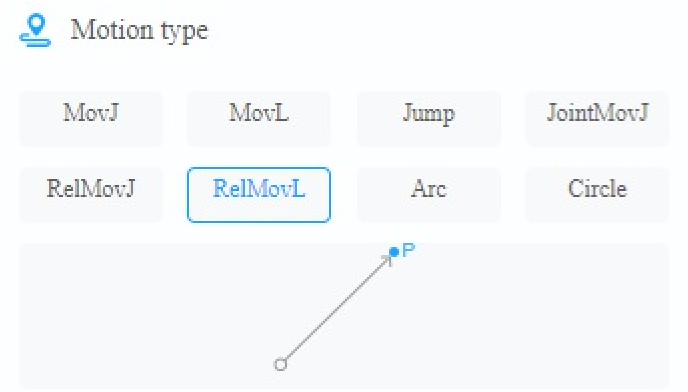
- Speed: velocity rate, range: 1~100.
- Acceleration (Accel): acceleration rate, range: 1~100.
- CP: set continuous path in motion, range: 0~100. See Continuous path (CP) at the end of this section

for details.

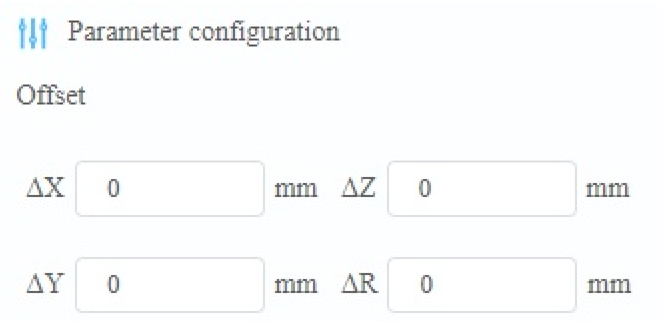


RelMovL

Motion mode: Move from the current position to the target offset position under the Cartesian coordinate system in a linear interpolated mode.



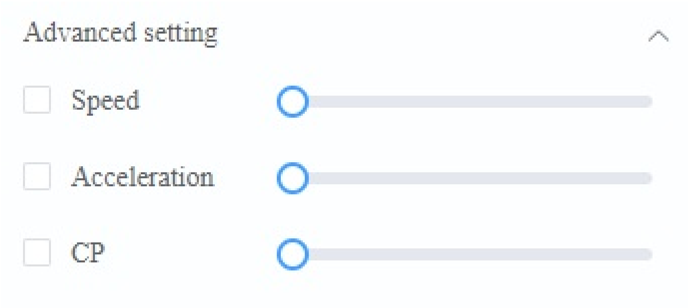
Basic setting: X-axis, Y-axis and Z-axis offset under the Cartesian coordinate system, unit: mm



Advanced setting:

Select and configure the advanced parameters as required.

- Speed: velocity rate, range: 1~100.
- Acceleration (Accel): acceleration rate, range: 1~100.
- CP: set continuous path in motion, range: 0~100. See Continuous path (CP) at the end of this section for details.



Arc

Motion mode: Move from the current position to the target position in an arc interpolated mode under the Cartesian coordinate system. The current position should not be on a straight line determined by point A and point B.

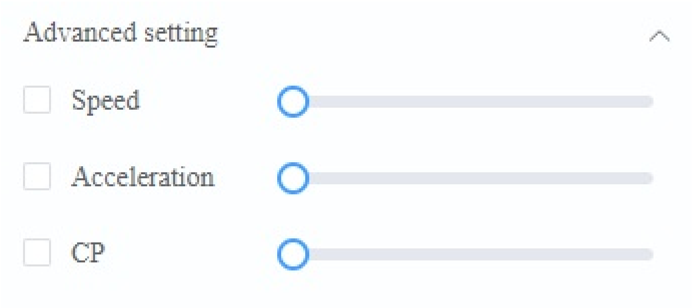
Basic setting:

- Intermediate point A coordinate: intermediate point coordinates of arc
- End point B coordinate: target point coordinates. The two points can be selected here after being added in the Points page, or defined in this page.

Advanced setting:

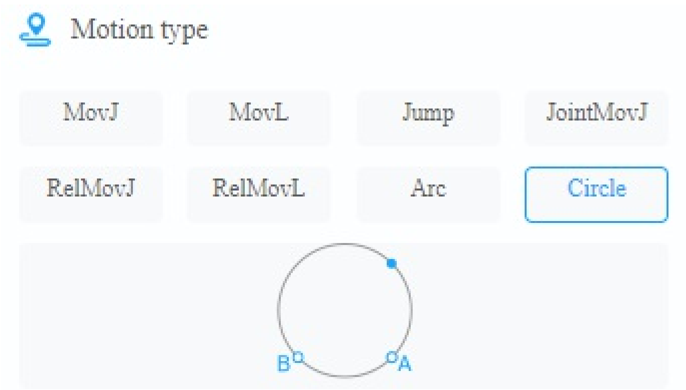
Select and configure the advanced parameters as required.

- Speed: velocity rate, range: 1~100.
- Acceleration (Accel): acceleration rate, range: 1~100.
- CP: set continuous path in motion, range: 0~100. See Continuous path (CP) at the end of this section for details.



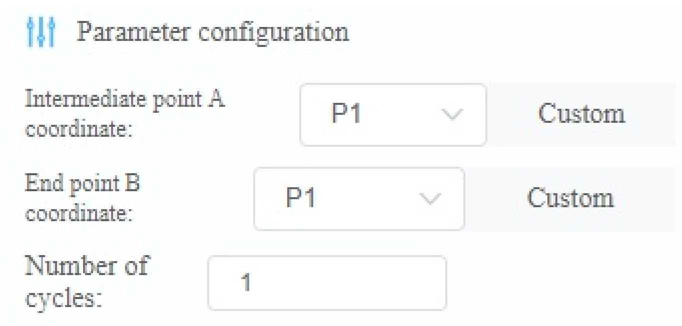
Circle

Motion mode: Move from the current position in a circle interpolated mode, and return to the current position after moving specified circles. The current position should not be on a straight line determined by point A and point B, and the circle determined by the three points cannot exceed the movement range of the robot arm.



Basic setting:

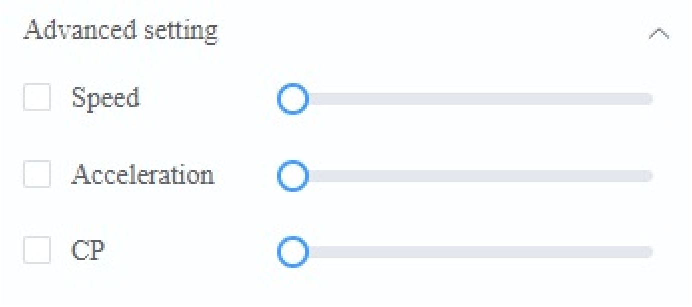
- Intermediate point A coordinate: It is used to determine the intermediate point coordinates of the circle.
- End point B coordinate: It is used to determine the end point coordinates of the circle. The two points can be selected here after being added in the Points page, or defined in this page.
- Number of circles: circles of Circle motion, range: 1~999.



Advanced setting:

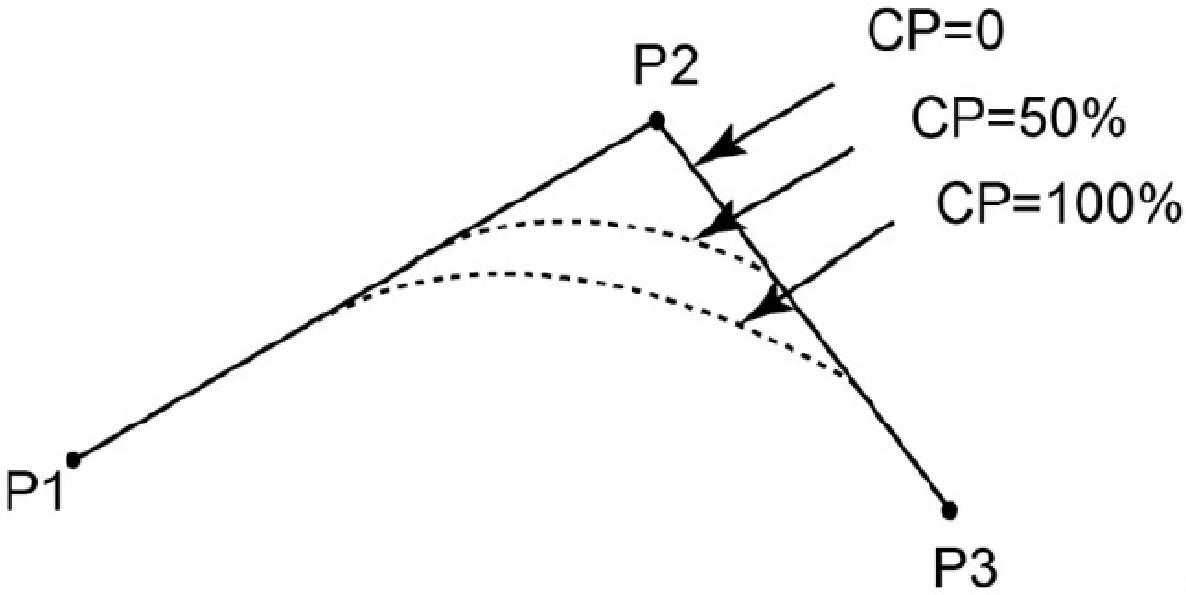
Select and configure the advanced parameters as required.

- Speed: velocity rate, range: 1~100.
- Acceleration (Accel): acceleration rate, range: 1~100.
- CP: set continuous path in motion, range: 0~100. See Continuous path (CP) at the end of this section for details.



Continuous path (CP)

The continuous path (CP) means when the robot arm moves from the starting point to the end point via the middle point, whether it transitions at a right angle or in a curved way when passing through the middle point, as shown below.



Posture

The posture commands are used for operations related to robot postures.

Get Cartesian coordinates of current posture

Description: Get the Cartesian coordinates of current posture.

Return: Cartesian coordinates of current posture

Get specified axis value of Cartesian coordinates of current posture

Description: Get the value of the specified axis of the current posture under the Cartesian coordinate system

Parameter: specified joint

Return: value of the specified axis of the current posture under the Cartesian coordinate system

Get joint coordinates of current posture

Description: Get the joint coordinate of current posture.

Return: joint coordinate of current posture

Get specified joint value of current posture

Description: Get the value of the specified joint of the current posture under the Joint coordinate system

Parameter: specified joint

Return: value of the specified joint of the current posture under the Joint coordinate system

Get coordinates after offset

Description: Get the coordinates of a specified position after a specified offset.

Parameter:

- initial position before offset. You need to use oval blocks which return Cartesian coordinates
- offset on each coordinate axis

Return: Cartesian coordinates after offset

Define Cartesian coordinates

Description: Define the coordinates under the Cartesian coordinate system

Parameter:

- value of the customized point on each coordinate axis.
- index value of the user coordinate system to which the point belongs.
- index value of the tool coordinate system to which the point belongs.

Return: Cartesian coordinates

Get coordinates

Description: Get coordinates of a specified point in Cartesian coordinate system.

Parameter: Select a point to obtain its coordinates.

Return: Cartesian coordinates of the specified point

Get coordinates of a specified axis

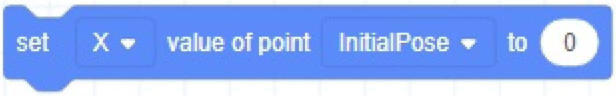
Description: Get the value of the specified point in the specified Cartesian coordinate axis.

Parameter:

- Select a point to get the coordinate value.
- Select the coordinate dimension.

Return: Value of the specified Cartesian coordinate axis

Modify coordinates



Description: Modify the value of the specified point in the specified Cartesian coordinate axis.

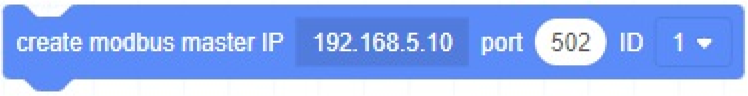
Parameter:

- Select a point.
- Select a coordinate axis.
- Set the value after modification.

Modbus

The Modbus commands are used for operations related to Modbus communication.

Create Modbus master



Description: Create Modbus master, and establish the connection with slave.

Parameter:

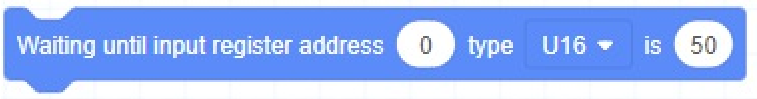
- IP address of Modbus slave
- port of Modbus slave
- ID of Modbus slave, range: 1~4

Get result of creating Modbus master

Description: Get the result of creating Modbus master.

Return: It returns 0 if the Modbus master is created successfully, and 1 if the Modbus master failed to be created.

Wait for input register

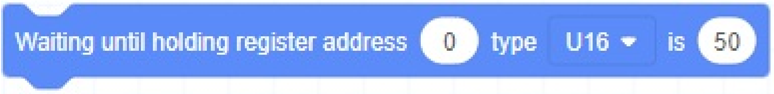


Description: Wait for the value of the specified address of input register to meet the condition before executing the next command.

Parameter:

- Address: Starting address of the input registers. Value range: 0~4095.
- Data type
    - U16: 16-bit unsigned integer (two bytes, occupy one register)
    - U32: 32-bit unsigned integer (four bytes, occupy two registers)
    - F32: 32-bit single-precision float number (four bytes, occupy two registers)
    - F64: 64-bit double-precision float number (eight bytes, occupy four registers).
- condition that the value is required to meet

Wait for holding register

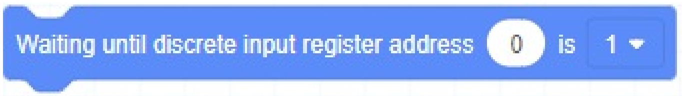


Description: Wait for the value of the specified address of holding register to meet the condition before executing the next command.

Parameter:

- Address: Starting address of the holding registers. Value range: 0~4095.
- Data type
  - U16: 16-bit unsigned integer (two bytes, occupy one register)
  - U32: 32-bit unsigned integer (four bytes, occupy two registers)
  - F32: 32-bit single-precision float number (four bytes, occupy two registers)
  - F64: 64-bit double-precision float number (eight bytes, occupy four registers).
- condition that the value is required to meet

Wait for discrete input register

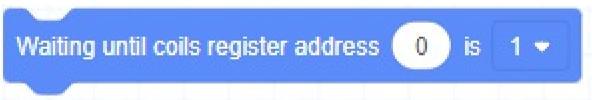


Description: Wait for the value of the specified address of discrete input register to meet the condition before executing the next command.

Parameter:

- Address: Starting address of the discrete input registers. Value range: 0~4095.
- condition that the value is required to meet

Wait for coil register



Description: Wait for the value of the specified address of input register to meet the condition before executing the next command.

Parameter:

- Address: Starting address of the coil registers. Value range: 0~4095.
- condition that the value is required to meet

Get input register

Description: Get the value of the specified address of input register.

Parameter:

- Address: Starting address of the input registers. Value range: 0~4095.
- Data type
  - U16: 16-bit unsigned integer (two bytes, occupy one register)
  - U32: 32-bit unsigned integer (four bytes, occupy two registers)
  - F32: 32-bit single-precision float number (four bytes, occupy two registers)
  - F64: 64-bit double-precision float number (eight bytes, occupy four registers).

Return: input register value

Get holding register

Description: Get the value of the specified address of holding register.

Parameter:

- Address: Starting address of the holding registers. Value range: 0~4095.
- Data type
  - U16: 16-bit unsigned integer (two bytes, occupy one register)
  - U32: 32-bit unsigned integer (four bytes, occupy two registers)
  - F32: 32-bit single-precision float number (four bytes, occupy two registers)
  - F64: 64-bit double-precision float number (eight bytes, occupy four registers).

Return: holding register value

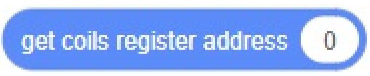
Get discrete input

Description: Get the value of the specified address of discrete input register.

Parameter: Starting address of the discrete input register. Value range: 0~4095

Return: discrete input value

Get coil register

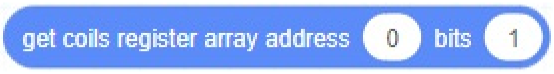


Description: Get the value of the specified address of coil register.

Parameter: Starting address of the coil register. Value range: 0~4095

Return: coil register value

Get multiple values of coil register



Description: Get multiple values of the specified address of coil register.

Parameter:

- Starting address of the coils register. Value range: 0~4095.
- 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

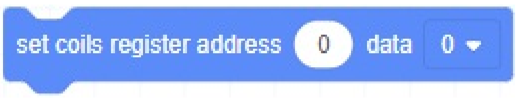
Description: Get multiple values of the specified address of holding register.

Parameter:

- Starting address of the input registers. Value range: 0~4095.
- 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 registers)
  - F32: 32-bit single-precision float number (four bytes, occupy two registers)
  - F64: 64-bit double-precision float number (eight bytes, occupy four registers).

Return: holding register values stored in table. The first value in table corresponds to the value of holding register at the starting address.

Set coil register



Description: Write the value to the specified address of coil register.

Parameter:

- Starting address of the coil register. Value range: 6~4095.
- Values written to the coil register. Value range: 0 or 1.

Set multiple coil register

Description: Write multiple values to the specified address of coil register.

Parameter:

- Starting address of the coil register. Value range: 0~4095.
- Number of value bits to be written.
- Values written to the coil register. Fill in an array with the same length as the number of bits written, each of which can only be 0 or 1.

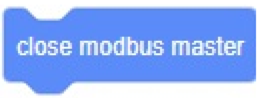
Set holding register

Description: Write the value to the specified address of holding register.

Parameter:

- Starting address of the input registers. Value range: 0~4095.
- Value to be written, which should correspond to the selected data type.
- Data type
  - U16: 16-bit unsigned integer (two bytes, occupy one register)
  - U32: 32-bit unsigned integer (four bytes, occupy two registers)
  - F32: 32-bit single-precision float number (four bytes, occupy two registers)
  - F64: 64-bit double-precision float number (eight bytes, occupy four registers).

Close Modbus master

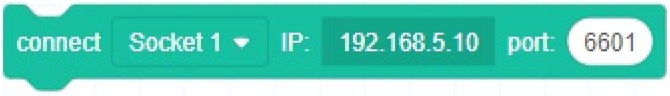


Description: Close the Modbus master, and disconnect from all slaves.

TCP commands

The TCP commands are used for operations related to TCP.

Connect SOCKET



Description: Create a TCP server to communicate with the specified TCP server.

Parameter:

- Select the SOCKET index (4 TCP communication links at most can be established).
- IP address of TCP server.
- TCP server port.

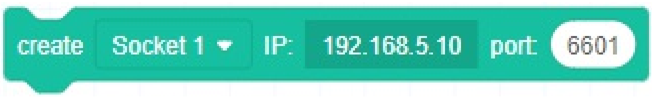
Get result of connecting SOCKET

Description: Get the result of TCP communication connection.

Parameter: Select SOCKET index.

Return: It returns 0 for successful connection, and 1 for failing to be connected.

Create SOCKET



Description: Create a TCP server to wait for connection from the client.

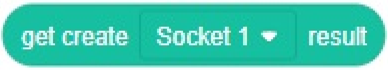
Parameter:

- Socket index (4 TCP communication links at most can be established).
- IP address of TCP server.
- TCP server port: When the robot serves as a server, do not use the following ports that have been occupied by the system.

  22, 23, 502 (0~1024 ports are linux-defined ports, which has a high possibility of being occupied. Please avoid to use),

  5000~5004, 6000, 8080, 11000, 11740, 22000, 22002, 29999, 30003, 30004, 60000, 65500~65515

Get result of creating SOCKET



Description: Get the result of creating TCP server.

Parameter: Select SOCKET index.

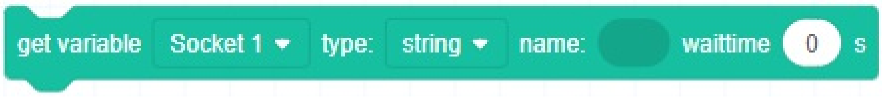
Return: It returns 0 for successful creation, and 1 for failing to be created.

Close SOCKET

Description: Close specified SOCKET, and disconnect the communication link.

Parameter: Select SOCKET index.

Get variables

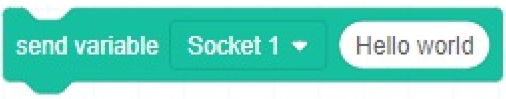


Description: Get variables through TCP communication and save it.

Parameter:

- Socket index
- variable type: string or number.
- Name is used for saving received variables, using created variable blocks
- Waiting time: if the waiting time is 0, it will wait until it gets variables.

Send variables

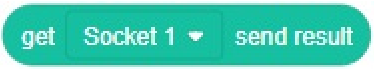


Description: Send variables through TCP communication.

Parameter:

- Socket index.
- data to be sent. You can use oval blocks that return string or numeric values, or directly fill in the blank.

Get result of sending variables



Description: Get the result of sending variables.

Parameter: Socket index.

Return: It returns 0 if the variable is sent, and 1 if the variable failed to be sent

Appendix C Script Commands

- C.1 Lua basic grammar
  - C.1.1 Variable and data type
  - C.1.2 Operator
  - C.1.3 Process control
- C.2 Command description
  - C.2.1 Motion
  - C.2.2 Motion parameter
  - C.2.3 Relative Motion
  - C.2.4 IO
  - C.2.5 TCP/UDP
  - C.2.6 Modbus
  - C.2.7 Program control
  - C.2.8 Vision

Lua basic grammar

Variable and data type

> If you want to learn related knowledge of Lua programming systematically, please search for Lua tutorials on the Internet. This guide only lists some of the basic Lua syntax for your quick reference.

Variables are used to store values, pass values as parameters or return values as results. Variables are assigned with "=".

Variables in Lua are global variables by default unless explicitly declared as local variables using "local". The scope of local variables is from the declaration location to the end of the block in which they are located.

```
a = 5               -- global variable
local b = 5         -- local variable
```

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.

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 can be made up of digits, letters, and/or underscores. Strings can be represented in three ways：

- Characters between single quotes.
- Characters between double quotes.
- characters between **[[** and **]]**

When performing arithmetic operations on a string of numbers, Lua attempts to convert the string of numbers into a number.

Lua provides many functions to support the operations of strings.

Function	Description
string.upper (argument)	Convert to uppercase letters
string.lower (argument)	Convert to lowercase letters
string.gsub (mainString, findString,replaceString,num)	Replace characters in a string. **MainString** is the source string, **findString** is the characters to be replaced, replaceString is the replacement characters, and **num** is the number of substitutions (can be ignored)
string.find (str, substr, [init, [end]])	Search for the specified content **substr** in a target string **str**. If a matching substring is found, the starting and ending indexes of the substring are returned, and nil is returned if none exists
string.reverse(arg)	The string is reversed
string.format(...)	Returns a formatted string similar to **printf**
string.char(arg) and string.byte(arg[,int])	**char** is used to convert integer numbers to characters and concatenate them **byte** is used to convert characters to integer values
string.len(arg)	Get the length of a string
string.rep(string, n)	Copy the string, n indicates the number of replication
..	Used to link two strings
string.gmatch(str, pattern)	It's an iterator function. Each time this function is called, it returns the next substring found in the **str** that matches the pattern description. If the substring described by pattern is not found, the iterator returns nil
string.match(str, pattern, init)	Search for the specified content that matches the description of Pattern in a target string **str**. Init is an optional parameter that specifies the starting index for the search, which defaults to 1. Only the first matching in the source **str** is found. If a matching character is found, the matching string is returned. If no capture flag is set, the entire matching string is returned. Return nil if there is no successful matching.
string.sub(s, i [, j])	Used to intercept strings. **s** is the source string to be truncated, **i** is the start index, **j** is the end index, and the default is -1, indicating the last character.

Example:

```
str = "Lua"
print(string.upper(str))       --Convert to uppercase letters, and print the result: LUA
print(string.lower(str))       --Convert to lowercase letters, and print the result: lua
print(string.reverse(str))     --The string is reversed, and print the result: aul
print(string.len("abc"))       --Calculate the length of the string ABC, and print the result:
 3
print(string.format("the value is: %d",4))     --Print the result: the value is:4
print(string.rep(str,2))       --Copy the string twice and print the result: LuaLua
string1 = "cn."
string2 = "dobot"
string3 = ".cc"
print("Address: ",string1..string2..string3)    --Use .. to connect strings, and print the resu
lt: Address: cn.dobot.cc

string1 = [[aaaa]]
print(string.gsub(string1,"a","z",3))          --Replace in a string and print the result: zzza


print(string.find("Hello Lua user", "Lua", 1))    --Search for Lua in the string and return th
e starting and ending index of the substring, printing the result: 7, 9

sourcestr = "prefix--runoobgoogletaobao--suffix"
sub = string.sub(sourcestr, 1, 8)               --Get the first through eighth characters of t
he string
print("\n result", string.format("%q", sub))    --Print: result: "prefix--"
```

Table

A table is a group of data with indexes.

- The simplest way to create a table is to use {}, which creates an empty table. This method initializes the table directly.
- A table can use associative arrays. The index of an array can be any type of data, but the value cannot be nil.
- The size of a table is not fixed and can be expanded as required.
- The symbol "#" can be used to obtain the length of a table.

```
tbl = {[1] = 2, [2] = 6, [3] = 34, [4] =5}
print("tbl length", #tbl)  -- Print the result: 4
```

Lua provides many functions to support the operation of table.

Function	Description
table.concat (table [, sep [, start [, end]]])	The table.concat () function lists all elements of the specified array from start to end, separated by the specified separator (sep)
table.insert	

(table, [pos,] value)	optional parameter, which defaults to the end of the table.
table.remove (table [, pos])	Return the element in the table at the specified position (pos), the element that follows will be moved forward. **pos** is an optional parameter and defaults to the table length, which is deleted from the last element.
table.sort (table [, comp])	The elements in the table are sorted in ascending order.

- Example 1:

```
fruits = {}                           --initialize an table
fruits = {"banana","orange","apple"}  --assign for the table

  print("String after concatenation",table.concat(fruits,", ", 2,3))   --Gets the element of t
he specified index from the table and concatenate them, String after concatenation orange, app
le

  --Insert element at the end
  table.insert(fruits,"mango")
  print("The element with index 4 is",fruits[4])     --print the result: The element with inde
x 4 is mango

  -- Insert the element at index 2
  table.insert(fruits,2,"grapes")
  print("The element with index 2 is",fruits[2])     --print the result: The element with inde
x 2 is grapes

  print("The last element is",fruits[5])        --print the result: The last element is mango
  table.remove(fruits)
  print("The last element after removal is",fruits[5])   --print the result: The last element
after removal is nil
```

- Example 2:

```
fruits = "banana","orange","apple","grapes"}
print("Before")
for k,v in ipairs(fruits) do
        print(k,v)                      --print the result: banana orange apple grapes
end
--In ascending order
table.sort(fruits)
print("After")
for k,v in ipairs(fruits) do
        print(k,v)                     --print the result: apple banana grapes orange
end
```

Array

An array is a collection of elements of the same data type arranged in a certain order. It can be one-dimensional or multidimensional. The index of an array can be represented as an integer, and the size of the array is not fixed.

- One-dimensional array: The simplest array with a logical structure of a linear table.
- Multidimensional array: An array contains an array or the index of a one-dimensional array corresponds to an array.

Example 1: One-dimensional array can be assigned or read through the **for** loop command. An integer index is used to access an array element. If the index has no value then the array returns nil.

```
array = {"Lua", "Tutorial"}  --Create a one-dimensional array
for i= 0, 2 do
     print(array[i])           --Print the result: nil Lua Tutorial
end
```

In Lua, array indexes start at 1 or 0. Alternatively, you can use a negative number as an index of an array.

```
array = {}
for i= -2, 2 do
     array[i] = i*2+1           --Assign values to a one-dimensional array
end
  for i = -2,2 do
     print(array[i])           --Print the result: -3 -1 1 3 5
  end
```

Example 2: An array of 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])     --Print the result: 1 2 3 2 4 6 3 6 9
    end
  end
```

Operator

Arithmetic Operator

Command	Description
+	Addition
-	Subtraction
*	Multiplication
/	Floating point division
//	Floor division
%	Remainder
^	Exponentiation
&	And operator
\	OR operator
~	XOR operator
<<	Left shift operator
>>	Right shift operat

- Example

```
a=20
b=5
print(a+b)          --Print the results for 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 results for the b-power of a: 3200000
print(a&b)          --Print the results of a And b: 4
print(a|b)          --Print the results of a OR b: 21
print(a~b)          --Print the results 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 Operator

Command	Description
==	Equal

~=	Not equal
<=	Equal or less than
>=	Equal or greater than
<	Less than
>	Greater than

- Example

```
a=20
b=5
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 Operator

Command	Description
and	Logical AND operator, the result is true if both sides are true, and false if either side is false
or	Logical OR operator, the result is true if one side is true, or false if either side is false
not	Logical NOT operator, that is, the judgment result is directly negative

- Example

```
a=true
b=false
print(a and b)            --True and false, the result is false
print(a or b)            --True or false, the result is true
print(20 > 5 not true)    --True and untrue, the result is false
```

Process control

Command	Description
if…then… elseif… then… else…end	Conditional command (if). Determine whether the conditions are valid from top to bottom. If a condition judgment is true, the corresponding code block is executed, and the subsequent condition judgments are directly ignored and no longer executed
while… do…end	Loop command (while). When the condition is true, make the program execute the corresponding code block repeatedly. The condition is checked for true before the statement is executed
for…do… end	Loop command (for), execute the specified statement repeatedly, and the number of repetitions can be controlled in the for statement
repeat… until()	Loop command (repeat), the loop repeats until the specified condition is true

- Example

1.Conditional command (if)

```
a = 100;
b = 200;
if(a == 100)
then
   if(b == 200)
   then
      print("This is a: ", a );
  print("This is b: ", b );
   end
end
```

2.Loop command (while)

```
a=10
while( a < 20 )
do
   print("This is a: ", a)
   a = a+1
end
```

3.Loop command (for)

```
for i=10,1,-1 do
    print(i)
end
```

4.Loop command (repeat)

```
a = 10
repeat
   print("This is a: ", a)
   a = a + 1
until(a > 15)
```

Command description

Motion

The motion commands is used to control the movement of the robot arm. The motion speed ratio/acceleration ratio can also be set in Motion parameter. If the parameters are set in both motion commands and motion parameter commands, the value of the motion commands will take precedence.

Actual robot speed/acceleration = set percentage in commands × speed/acceleration in playback settings × global speed ratio.

MovJ

Command:

```
MovJ(P, {CP=1, SpeedJ=50, AccJ=20, SYNC=0})
```

Description:

Move from the current position to a target position under the Cartesian coordinate system in a point-to-point mode (joint motion). The trajectory of joint motion is not linear, and all joints complete the motion at the same time.

Required parameter:

P: target point, which is user-defined or obtained from the Point page. Only Cartesian coordinate points are supported.

Optional parameter:

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0~100.
- SpeedJ: velocity rate, range: 1~100.
- AccJ: acceleration rate, range: 1~100.
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
    - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
    - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
MovJ(P1)
```

The robot moves to P1 in the point-to point mode with the default setting.

MovL

Command:

```
MovL(P, {CP=1, SpeedL=50, AccL=20, SYNC=0})
```

Description:

Move from the current position to a target position under the Cartesian coordinate system in a linear mode.

Required parameter:

P: target point, which is user-defined or obtained from the Point page. Only Cartesian coordinate points are supported.

Optional parameter:

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0~100.
- SpeedL: Velocity rate. Value range: 1~100
- AccL: Acceleration rate. Value range: 1~100
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
  - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
  - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
Move(P1)
```

The robot arm moves to P1 in a linear mode with the default setting.

Jump

Command:

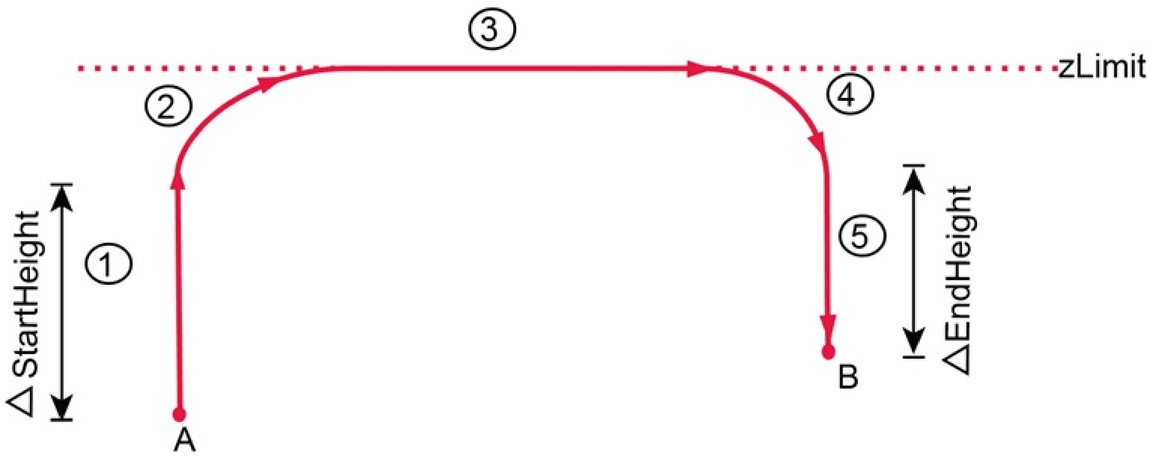
```
Jump(P,{SpeedL=50, AccL=20, Start=10, ZLimit=100, End=20, SYNC=0})
Jump(P,{SpeedL=50, AccL=20, Arch=1, SYNC=0})
```

Description:

Move from the current position to the target position under the Cartesian coordinate system in a door-shaped mode.

1. The robot arm will first raise the specified height vertically, and then
2. Transition to the maximum height.

3. Move towards the target point in a linear mode.
4. When the robot arm moves near the target point, transition to the specified height above the target point
5. Descend vertically to the target point.



Required parameter:

- P: target point, which is user-defined or obtained from the Point page. Only Cartesian coordinate points are supported.The height of P cannot exceed ZLimit.

Optional parameter:

- SpeedL: Velocity rate. Value range: 1~100
- AccL: Acceleration rate. Value range: 1~100
- Start: lifting height of the starting point
- ZLimit: maximum lifting height
- End: descent height of the end point
- Arch: Jump parameter index, which is set in the software
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
  - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
  - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
Go(P4)
Jump(P5,"Start=10 ZLimit=600 End=10")
```

The robot arm moves to P4, and then the door type moves to P5 through jump motion.

JointMovJ

Command:

```
JointMovJ(P,{CP=1, SpeedJ=50, AccJ=20, SYNC=0})
```

Description:

Move from the current position to a target joint angle in a point-to-point mode (joint motion).

Required parameter:

P: target point, which can only be defined through joint angle.

Optional parameter:

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0~100.
- SpeedJ: velocity rate, range: 1~100.
- Accl: acceleration rate, range: 1~100.
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
  - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
  - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
local P = {joint={0,-0.0674194,0,0}}
JointMovJ(P)
```

Define the joint coordinate point P. Move the robot to P with the default setting.

Circle

Command:

```
Circle(P1,P2,Count,{CP=1, SpeedL=50, AccL=20, SYNC=0})
```

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 on a straight line determined by P1 and P2, and the circle determined by the three points cannot exceed the movement range of the robot arm.

Required parameter:

- P1: middle point, which is user-defined or obtained from the Point page. Only Cartesian coordinate

points are supported.
- P2: end point, which is user-defined or obtained from the Point page. Only Cartesian coordinate points are supported.
- Count: number of circles, range: 1~999.

Optional parameter:

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0~100.
- SpeedL: Velocity rate. Value range: 1~100
- AccL: Acceleration rate. Value range: 1~100
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
  - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
  - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
Go(P1)
Circle3(P2,P3,1)
```

The robot arm moves to P1, and then moves a full circle determined by P1, P2 and P3.

Arc

Command:

```
Arc3(P1,P2,{CP=1, SpeedL=50, AccL=20, SYNC=0})
```

Description:

Move from the current position to a target position in an arc interpolated mode under the Cartesian coordinate system. As the arc needs to be determined through the current position, P1 and P2, the current position should not be on a straight line determined by P1 and P2.

Required parameter:

- P1: middle point, which is user-defined or obtained from the Point page. Only Cartesian coordinate points are supported.
- P2: target point, which is user-defined or obtained from the Point page. Only Cartesian coordinate points are supported.

Optional parameter:

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0~100.
- SpeedL: Velocity rate. Value range: 1~100

- AccL: Acceleration rate. Value range: 1~100
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
  - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
  - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
Go(P1)
Arc3(P2,P3)
```

The robot moves to P1, and then moves to P3 via P2 in the arc interpolated mode.

MovJIO

Command:

```
MovJIO(P, { {Mode, Distance, Index, Status},{Mode, Distance, Index, Status}...},{CP=1, SpeedJ=
50, AccJ=20, SYNC=0})
```

Description:

Move from the current position to a target position in a point-to-point mode (joint motion) under the Cartesian coordinate system, and set the status of digital output port when the robot is moving.

Required parameter:

- P: target point, which is user-defined or obtained from the Point page. Only Cartesian coordinate points are supported.
- Digital output parameters: Set the specified DO to be triggered when the robot arm moves to a specified distance or percentage. You can set multiple groups, each of which contains the following parameters:
  - Mode: trigger mode. 0: distance percentage; 1: distance value
  - Distance: specified distance
    - If Distance is positive, it refers to the distance away from the starting point
    - If Distance is negative, it refers to the distance away 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; 1: ON

Optional parameter:

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0~100.

- Speed: velocity rate, range: 1~100.
- Accel: acceleration rate, range: 1~100.
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
  - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
  - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
MovJIO(P1, {0, 10, 1, 1})
```

The robot arm moves towards P1 with the default setting. When it moves 10% distance away from the starting point, set DO2 to ON.

MovLIO

Command:

```
MovLIO(P, { {Mode, Distance, Index, Status},{Mode, Distance, Index, Status}...},{CP=1, SpeedL=
50, AccL=20, SYNC=0})
```

Description:

Move from the current position to a target position in a linear mode under the Cartesian coordinate system, and set the status of digital output port when the robot is moving.

Required parameter:

- P: target point, which is user-defined or obtained from the Point page. Only Cartesian coordinate points are supported.
- Digital output parameters: Set the specified DO to be triggered when the robot arm moves a specified distance or percentage. You can set multiple groups, each of which contains the following parameters:
  - Mode: trigger mode. 0: distance percentage; 1: distance value
  - Distance: specified distance
    - If Distance is positive, it refers to the distance away from the starting point
    - If Distance is negative, it refers to the distance away 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; 1: ON

Optional parameter:

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0~100.

- SpeedL: velocity rate, range: 1~100.
- AccL: acceleration rate, range: 1~100.
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
    - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
    - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
MovLIO(P1, {0, 10, 1, 1})
```

The robot moves towards P1 with the default setting. When it moves 10% distance away from the starting point, set DO2 to ON.

MovJExt

Command:

```
MovJExt(AD, {SpeedE=50, AccE=20, SYNC=0})
```

Description:

Control the aux joint to move to the target angle and position.

Required parameter:

AD: angle or distance of motion. The meaning of this parameter depends on the type of motion (joint/linear) set in Advanced Settings in the Aux Joint process. unit: degree (when the type is joint) or mm (when the type is line).

Optional parameter:

- SpeedE: velocity rate, range: 1~100.
- AccE: acceleration rate, range: 1~100.
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
    - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
    - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
MovJExt(20)
```

Move the aux joint to the specified position 20.

Motion parameter

The motion parameters are used to set or obtain relevant motion parameters of the robot.

Sync

Command:

```
Sync()
```

Description:

The command is used to block the program to execute the queue commands. It returns until all the queue commands have been executed, and then executes subsequent commands. Generally it is used to wait for the robot arm to complete the movement.

Example:

```
MovJ(P1)
MovJ(P2)
Sync()
```

The robot arm moves to P1, and then moves to P2 before it returns to execute subsequent commands.

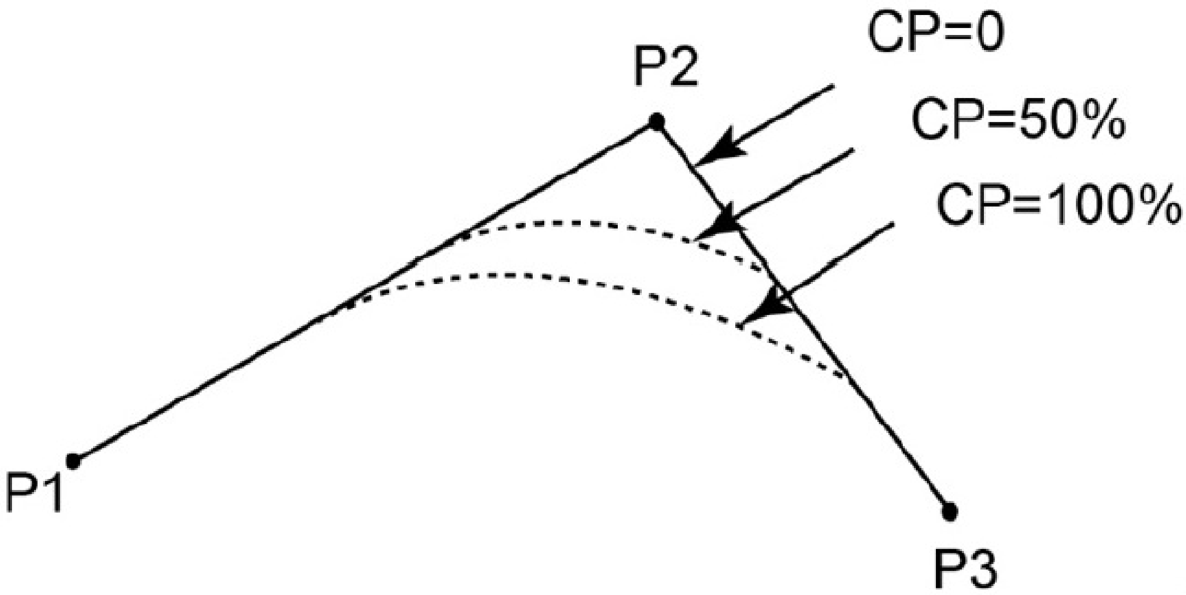
CP

Command:

```
CP(R)
```

Description:

Set the continuous path (CP) ratio, that is, when the robot arm moves continuously via multiple points, whether it transitions at a right angle or in a curved way when passing through the middle point.



Required parameter:

R: continuous path ratio, range: 0~100

Example:

```
CP(50)
MovL(P1)
MovL(P2)
MovL(P3)
```

The robot moves from P1 to P3 via P2 with 50% continuous path ratio.

SpeedJ

Command:

```
SpeedJ(R)
```

Description:

Set the velocity rate of joint motion. Actual robot speed = percentage set in commands × speed in playback settings × global speed ratio.

Required parameter:

R: velocity rate, range: 0~100

Example:

```
SpeedJ(20)
MovJ(P1)
```

The robot moves to P1 with 20% velocity rate.

AccJ

Command:

```
AccJ(R)
```

Description:

Set the acceleration rate of joint motion. Actual robot acceleration = percentage set in commands × acceleration in playback settings × global speed ratio.

Required parameter:

R: acceleration rate, range: 0~100

Example:

```
AccJ(50)
MovJ(P1)
```

The robot moves to P1 with 50% acceleration rate.

SpeedL

Command:

```
SpeedL(R)
```

Description:

Set the velocity ratio of linear and arc motion. Actual robot speed = percentage set in commands × speed in playback settings × global speed ratio.

Required parameter:

R: velocity rate, range: 0~100

Example:

```
SpeedL(20)
MovL(P1)
```

The robot moves to P1 with 20% velocity rate.

AccL

Command:

```
AccL(R)
```

Description:

Set the acceleration ratio of linear and arc motion. Actual robot acceleration = percentage set in commands × acceleration in playback settings × global speed ratio.

Required parameter:

R: acceleration ratio, range: 0~100

Example:

```
AccL(50)
MovL(P1)
```

The robot moves to P1 with 50% acceleration ratio.

GetPose

Command:

```
GetPose()
```

Description:

Get the real-time posture of the robot arm under the Cartesian coordinate system. If you have set a user coordinate system or tool coordinate system, the obtained posture is under the current coordinate system.

Return:

Cartesian coordinates of the current posture

Example:

```
local currentPose = GetPose()
MovJ(P1)
MovJ(currentPose)
```

The robot moves to P1, and then returns to the current posture.

GetAngle

Command:

```
GetAngle()
```

Description:

Get the real-time posture of the robot arm under the Joint coordinate system.

Return:

Joint coordinates of the current posture

Example:

```
local currentAngle = GetAngle()
MovJ(P1)
JointMovJ(currentAngle)
```

The robot moves to P1, and then returns to the current posture.

Relative motion

The motion commands is used to control the movement of the robot arm. The motion speed ratio/acceleration ratio can also be set in Motion parameter. If the parameters are set in both motion commands and motion parameter commands, the value of the motion commands will take precedence.

Actual robot speed/acceleration = set percentage in commands × speed/acceleration in playback settings × global speed ratio.

RelJoint

Command:

```
RelJoint(P, {Offset1, Offset2, Offset3, Offset4})
```

Description:

Set the angle offset of J1~J4 axes of a specified point under the Joint coordinate system, and return a new joint coordinate point.

Required parameter:

- P1: Point before offset, which cannot be obtained from the TeachPoint page. Only Cartesian coordinate points are supported
- Offset1~Offset4: J1~J4 axes offset. unit: °

Return:

Joint coordinate point after offset

Example:

```
JointMovJ(RelJoint(P1, {60,50,32,30}))
```

Set the angle offset of J1~J4 axes of P1, and move the robot arm to the target point after offset.

RelPoint

Command:

```
RelPoint(P, {OffsetX, OffsetY, OffsetZ, OffsetR})
```

Description:

Set the X-axis, Y-axis, Z-axis and R-axis offset of a point under the Cartesian coordinate system to return a new Cartesian coordinate point.

Required parameter:

- Point before offset, which is user-defined or obtained from the Point page. Only Cartesian coordinate points are supported
- OffsetX, OffsetY, OffsetZ, OffsetR: X-axis, Y-axis, Z-axis and R-axis offset in the Cartesian coordinate system. unit: mm (X, Y, Z) or degree (R)

Return:

Cartesian coordinate point after offset

Example:

```
Go(RP(P1, {30,50,10,0}))
```

Displace P1 by a certain distance on the X, Y, and Z axes respectively, and then move to the point after the offset.

RelMovJ

Command:

```
RelMovJ({OffsetX, OffsetY, OffsetZ, OffsetR}, {CP=1, SpeedJ=50, AccJ=20, SYNC=0})
```

Description:

Move from the current position to the offset position in a point-to-point mode (joint motion) under the Cartesian coordinate system. The trajectory of joint motion is not linear, and all joints complete the motion at the same time.

Required parameter:

OffsetX, OffsetY, OffsetZ, OffsetR: X-axis, Y-axis, Z-axis and R-axis offset under the Cartesian coordinate system. unit: mm (X, Y, Z) or degree (R)

Optional parameter:

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0~100.
- SpeedJ: velocity rate, range: 1~100.
- AccJ: acceleration rate, range: 1~100.
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
  - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
  - SYNC=1: synchronous execution. which means not returning after being called until the

command is executed completely.

Example:

```
RelMovJ({10,10,10,0})
```

The robot arm moves to the target offset point in a joint-to-joint mode with the default setting.

RelMovL

Command:

```
RelMovL({OffsetX, OffsetY, OffsetZ, OffsetR}, {CP=1, SpeedL=50, AccL=20, SYNC=0})
```

Description:

Move from the current position to the offset position in a linear mode under the Cartesian coordinate system.

Required parameter:

OffsetX, OffsetY, OffsetZ, OffsetR: X-axis, Y-axis, Z-axis and R-axis offset under the Cartesian coordinate system. unit: mm (X, Y, Z) or degree (R)

Optional parameter:

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0~100.
- SpeedL: velocity rate, range: 1~100.
- AccL: acceleration rate, range: 1~100.
- SYNC: synchronization flag, range: 0 or 1 (default value: 0).
  - SYNC=0: asynchronous execution, which means returning immediately after being called, regardless of the execution process.
  - SYNC=1: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

```
RelMovL({10,10,10,0})
```

The robot arm moves to the target offset point through linear motion with the default setting.

IO

The IO commands are used to read and write system IO and set relevant parameters

DI

Command:

```
DI(index)
```

Description:

Get the status of the digital input port.

Required parameter:

index: DI index

Return:

Level (ON/OFF) of corresponding DI port

Example:

```
if (DI(1)==ON) then
MovL(P1)
end
```

The robot moves to P1 through linear motion when the status of DI1 is ON.

DO

Command:

```
DO(index,ON|OFF)
```

Description:

Set the status of digital output port.

Required parameter:

- index: DO index
- ON/OFF: status of the DO port. ON: High level; OFF: Low level

Example:

```
DO(1,ON)
```

Set the status of DO1 to ON.

DOInstant

Command:

```
DOInstant(index,ON|OFF)
```

Description:

Set the status of digital output port immediately regardless of the current command queue.

Required parameter:

- index: DO index
- ON/OFF: status of the DO port. ON: High level; OFF: Low level

Example:

```
DOInstant(1,ON)
```

Set the status of DO1 to ON immediately regardless of the current command queue.

ToolDI

Command:

```
ToolDI(index)
```

Description:

Get the status of tool digital input port.

Required parameter:

index: tool DI index

Return:

Level (ON/OFF) of corresponding DI port

Example:

```
if (ToolDI(1)==ON) then
MovL(P1)
end
```

The robot moves to P1 in a linear mode when the status of tool DI1 is ON.

TCP/UDP

TCP/UDP commands are used for TCP/UDP communication.

TCPCreate

Command:

```
TCPCreate(isServer, IP, port)
```

Description:

Create a TCP network. Only one TCP network is supported.

Required parameter:

- isServer: whether to create a server. true: create a server; false: create a client
- IP: IP address of the server, which is in the same network segment of the client without conflict. It is the IP address of the robot arm when a server is created, and the address of the peer when a client is created.
- port: server port. When the robot serves as a server, do not use the following ports that have been occupied by the system.

  22, 23, 502 (0~1024 ports are linux-defined ports, which has a high possibility of being occupied. Please avoid to use),

  5000~5004, 6000, 8080, 11000, 11740, 22000, 22002, 29999, 30003, 30004, 60000, 65500~65515

Return:

- err:
  - 0: TCP network has been created successfully
  - 1: TCP network failed to be created
- socket: socket object

Example 1:

```
local ip="192.168.5.1" // Set the IP address of the robot as the server
local port=6001 // Server port
local err=0
local socket=0
err, socket = TCPCreate(true, ip, port)
```

Create a TCP server.

Example 2:

```
local ip="192.168.5.25" //Set the IP address of external equipment such as a camera as the ser
ver
local port=6001 //Server port
local err=0
local socket=0
err, socket = TCPCreate(false, ip, port)
```

Create a TCP client.

TCPStart

Command:

```
TCPStart(socket, timeout)
```

Description:

Establish TCP connection. The robot arm waits to be connected with the client when serving as a server, and connects the server when serving as a client.

Required parameter:

- socket: socket object
- timeout: waiting timeout. unit: s. If timeout is 0, wait until the connection is established successfully. If not, return connection failure after exceeding the timeout,

Return:

Connection result.

- 0: TCP connection is successful
- 1: input parameters are incorrect
- 2: socket object is not found
- 3: timeout setting is incorrect
- 4: connection failure

Example:

```
err = TCPStart(socket, 0) // socket is the socket object returned by TCPCreate
```

Start to establish TCP connection until the connection is successful.

TCPRead

Command:

```
TCPRead(socket, timeout, type)
```

Description:

Receive data from a TCP peer.

Required parameter:

- socket: socket object

Optional parameter:

- timeout: waiting timeout. unit: s. If timeout is 0, wait until the data is completely read before running; if not, continue to run after exceeding the timeout.
- type: type of return value. If type is not set, the buffer format of RecBuf is a table. If type is set to string, the buffer format is a string.

Return:

- err:
  - 0: Data has been received successfully
  - 1: Data failed to be received.
- Recbuf: data buffer

Example:

```
// socket is the socket 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
```

Receive TCP data, and save the data as string and table format respectively.

TCPWrite

Command:

```
TCPWrite(socket, buf, timeout)
```

Description:

Send data to TCP peer.

Required parameter:

- socket: socket object

- buf: data sent by the robot

Optional parameter:

timeout: waiting timeout. unit: s. If timeout is 0, the program will not continue to run until the peer receives the data. If timeout is not 0, the program will continue to run after exceeding the timeout

Return:

Result of sending data.

- 0: Data has been sent successfully
- 1: Data failed to be sent.

Example:

```
TCPWrite(socket, "test") // socket is the socket object returned by TCPCreate
```

Send TCP data "test".

TCPDestroy

Command:

```
TCPDestroy(socket)
```

Description:

Disconnect the TCP network and destroy the socket object.

Required parameter:

socket: socket object

Return:

Execution result.

- 0: It has been executed successfully.
- 1: It failed to be executed.

Example:

```
TCPDestroy(socket) // socket is the socket object returned by TCPCreate
```

Disconnect with the TCP peer.

UDPCreate

Command:

```
UDPCreate(isServer, IP, port)
```

Description:

Create a UDP network. Only one UDP network is supported.

Required parameter:

- isServer: false
- IP: IP address of the peer, which is in the same network segment of the client without conflict
- port: peer port

Return:

- err:
  - 0: The UDP network has been created successfully
  - 1: The UDP network failed to be created
- socket: socket object

Example:

```
local ip="192.168.5.25" //Set the IP of an external device such as a camera as the IP address
of the peer
local port=6001 //peer port
local err=0
local socket=0
err, socket = UDPCreate(false, ip, port)
```

Create a UDP network.

UDPRead

Command:

```
UDPRead(socket, timeout, type)
```

Description:

Receive data from the UDP peer.

Required parameter:

- socket: socket object

Optional parameter:

- timeout: waiting timeout. unit: s. If timeout is 0, wait until the data is completely read before running; if not, continue to run after exceeding the timeout.
- type: type of return value. If type is not set, the buffer format of RecBuf is a table. If type is set to string, the buffer format is a string.

Return:

- err:
  - 0: Data has been received successfully
  - 1: Data failed to be received.
- Recbuf: data buffer

Example:

```
// socket is the socket 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
```

Receive UDP data, and save the data as string and table format respectively.

UDPWrite

Command:

```
UDPWrite(socket, buf, timeout)
```

Description:

Send data to UDP peer.

Required parameter:

- socket: socket object
- buf: data sent by the robot

Optional parameter:

timeout: waiting timeout. unit: s. If timeout is 0, the program will not continue to run until the peer receives the data. If timeout is not 0, the program will continue to run after exceeding the timeout

Return:

Result of sending data.

- 0: Data has been sent successfully
- 1: Data failed to be sent.

Example:

```
UDPWrite(socket, "test") // socket is the socket object returned by UDPCreate
```

Send UDP data "test".

Modbus

Modbus commands are used for Modbus communication.

ModbusCreate

Command:

```
ModbusCreate()
```

Description:

Create Modbus master station, and establish connection with the slave station.

Required parameter:

- IP: IP address of slave station. When IP is not specified, or is 127.0.0.1 or 0.0.0.1, it indicates connecting the local Modbus slave.
- port: slave station port
- slave_id: ID of slave station. range: 1~4

Return:

- err:
  - 0: Modbus master station has been created successfully.
  - 1: Modbus master station failed to be created.
- id: device ID of slave station

Example 1:

```
local ip="192.168.5.123" //slave ID
local port=503 //slave port
local err=0
local id=0
err, id = ModbusCreate(ip, port, 1)
```

Create the Modbus master, and connect with the specified slave.

Example 1:

The following commands all indicates connecting Modbus slave station.

```
ModbusCreate()
```

```
ModbusCreate("127.0.0.1")
```

```
ModbusCreate("0.0.0.1")
```

```
ModbusCreate("127.0.0.1", xxx,xxx) // xxx arbitrary value
```

```
ModbusCreate("0.0.0.1", xxx,xxx) // xxx arbitrary value
```

GetInBits

Command:

```
GetInBits(id, addr, count)
```

Description:

Read the discrete input value from Modbus slave

Required parameter:

- id: slave ID
- addr: starting address of the discrete inputs, range: 0~4095
- count: number of the discrete inputs

Return:

Discrete input value stored in a table, where the first value in the table corresponds to the discrete input value at the starting address

Example:

```
inBits = GetInBits(id,0,5)
```

Read 5 discrete inputs starting from address 0.

GetInRegs

Command:

```
GetInRegs(id, addr, count, type)
```

Description:

Read the input register value with the specified data type from the Modbus slave.

Required parameter:

- id: slave ID
- addr: starting address of the input registers, range: 0 - 4095
- count: number of input register values, range: 0 ~ 4096

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 registers)
- 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:

Input register values stored in a table, where the first value corresponds to the Input register value at the starting address.

Example:

```
data = GetInRegs(id, 2048, 1, “U32”)
```

Read a 32-bit unsigned integer starting from address 2048.

GetCoils

Command:

```
GetCoils(id, addr, count)
```

Description:

Read the coil register value from the Modbus slave.

Required parameter:

- id: slave ID
- addr: starting address of the coil register, range: 0~4095
- count: number of coil register values

Return:

Coil register value stored in a table, where the first value corresponds to the coil register value at the starting address.

Example:

```
Coils = GetCoils(id,0,5)
```

Read 5 values in succession starting from address 0.

GetHoldRegs

Command:

```
GetHoldRegs(id, addr, count, type)
```

Description:

Read the holding register value with the specified data type from the Modbus slave.

Required parameter:

- id: slave ID
- addr: starting address of the holding register, range: 0~4095
- count: number of holding register values

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 registers)
- 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:

Holding register value stored in a table, where the first value corresponds to the holding register value at the starting address

Example:

```
data = GetHoldRegs(id, 2048, 1, “U32”)
```

Read a 32-bit unsigned integer starting from address 2048.

SetCoils

Command:

```
SetCoils(id, addr, count, table)
```

Description:

Write the specified value to the specified address of coil register.

Required parameter:

- id: slave ID
- addr: starting address of the coil register, range: 6~4095
- count: number of values to be written to the coil register, range: 0 to 4096
- table: store the values to be written to the coil register. The first value of the table corresponds to the starting address of coil register

Example:

```
local Coils = {0,1,1,1,0}
SetCoils(id, 1024, #coils, Coils)
```

Starting from address 1024, write 5 values in succession to the coil register.

SetHoldRegs

Command:

```
SetHoldRegs(id, addr, count, table, type)
```

Description:

Write the specified value according to the specified data type to the specified address of holding register.

Required parameter:

- id: slave ID
- addr: starting address of the holding register, range: 0~4095
- count: number of values to be written to the holding register
- table: store the values to be written to the coil register. The first value of the table corresponds to the starting address of holding register

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 registers)
- 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:

```
local data = {95.32105}
SetHoldRegs(id, 2048, #data, data, “F64”)
```

Starting from address 2048, write a double-precision floating-point number to the holding register.

ModbusClose

Command:

```
ModbusClose(id)
```

Description:

Disconnect with Modbus slave station.

Optional parameter:

id: slave ID

Return:

- 0: The Modbus slave has been disconnected successfully.
- 1: The Modbus slave failed to be disconnected.

Example:

```
ModbusClose(id) // id is the slave ID returned by ModbusCreate
```

Disconnect with the Modbus slave.

Program Control

The program control commands are general commands related to program control. The **while**, **if** and **for** are flow control commands of Lua. Please refer to Lua basic grammar - Process control. The **print** is used to output information to the console.

Sleep

Command:

```
Sleep(time)
```

Description:

Delay the execution of the next command.

Required parameter:

time: delay time, unit: ms

Example:

```
DO(1,ON)
Sleep(100)
DO(1,OFF)
```

Set DO1 to ON, wait 100ms and then set DO1 to OFF.

Wait

Command:

```
Wait(time)
```

Description:

Deliver the motion command with a delay, or deliver the next command with a delay after the current motion is completed.

Required parameter:

time: delay time, unit: ms

Example:

```
DO(1,ON)
Wait(100)
MovJ(P1)
Wait(100)
DO(1,OFF)
```

Set DO1 to ON, wait 100ms and then move the robot to P1. Delay 100ms, and then set DO1 to OFF.

Pause

Command:

```
Pause()
```

Description:

Pause running the program. The program can continue to run only through software control or remote control.

Example:

```
MovJ(P1)
Pause()
MovJ(P2)
```

The robot moves to P1 and then pauses running. It can continue to move to P2 only through external control.

SetCollisionLevel

Command:

```
SetCollisionLevel(level)
```

Description:

Set the level of collision detection.

Required parameter:

level: collision detection level, range: 0~5. 0 means turning off collision detection. The higher the level from 1 to 5, the more sensitive the collision detection is.

Example:

```
SetCollisionLevel(2)
```

Set the collision detection to Level 2.

ResetElapsedTime

Command:

```
ResetElapsedTime()
```

Description:

Start timing after all commands before this command are executed completely. This command should be used combined with ElapsedTime() command for calculating the operating time.

Example:

Refer to the example of ElapsedTime.

ElapsedTime

Command:

```
ElapsedTime()
```

Description:

Stop timing and return the time difference. The command should be used combined with ResetElapsedTime() command.

Return:

time between the start and the end of timing.

Example:

```
MovJ(P2)
ResetElapsedTime()
for i=1,10 do
MovL(P1)
MovL(P2)
end
print (ElapsedTime())
```

Calculate the time for the robot arm to move back and forth 10 times between P1 and P2, and print it to the console.

Systime

Command:

```
Systime()
```

Description:

Get the current system time.

Return:

Unix time stamp of the current time

Example:

```
local time = Systime()
```

Get the current system time and save it to the variable "time".

SetPayload

Command:

```
SetPayload(payload, {x, y}, index)
```

Description:

Set the load weight, eccentric coordinates and servo parameter index. For specific instructions, refer to 4.5.2 Terminal load.

Required parameter:

- payload: load weight. range: 0~1000, unit: g
- {x, y}: eccentric coordinates

Optional parameter:

- index: servo parameter index. Please set it under the guidance of technical support.

Example:

```
SetPayload(100, {0, 0})
```

Set the load weight to 100g without eccentricity.

SetTool485

Command:

```
SetTool485(baud,parity,stopbit)
```

Description:

Set the data format corresponding to the RS485 interface of the end tool.

Required parameter:

- baud: baud rate of RS485 interface
- parity: whether there are parity bits. "O" means odd, "E" means even, and "N" means no parity bits.
- stopbit: stop bit length. range: 1, 1.5, 2.

Example:

```
SetTool485(115200,"N",1)
```

Set the baud rate corresponding to the RS485 interface of the end tool to 115200Hz, parity bit to N, and stopbit to 1.

SetUser

Command:

```
SetUser(index,table)
```

Description:

Modify the specified user coordinate system.

Required parameter:

- index: index of user coordinate system, range: 0~9 (0 is default user coordinate system)
- table: matrix for user coordinate system, in {x, y, z, r} format

Example:

```
SetUser(1,{10,10,10,0})
```

Modify the user coordinate system 1 to "X=10, Y=10, Z=10, R=0".

SetTool

Command:

```
SetTool(index,table)
```

Description:

Modify the specified tool coordinate system.

Required parameter:

- index: index of tool coordinate system, range: 0~9 (0 is default user coordinate system)
- table: matrix for tool coordinate system, in {x, y, z, r} format

Example:

```
SetTool(1,{10,10,10,0})
```

Modify the tool coordinate system 1 to "X=10, Y=10, Z=10, R=0".

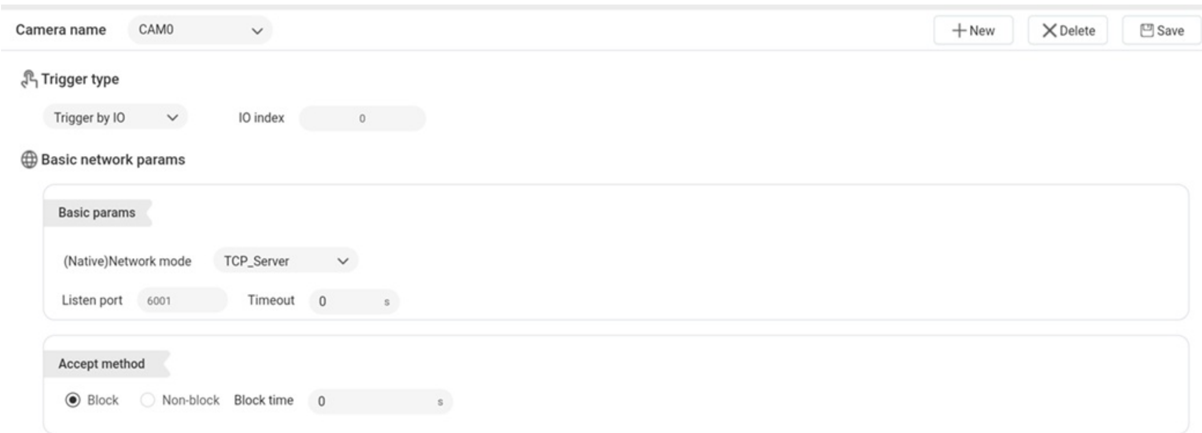
Vision

The vision module is used to configure relevant camera settings. The camera is fixed within the working range of the robot. Its position and vision field are fixed. The camera acts as the eye of the robot and interacts with the robot through Ethernet communication or I/O triggering.

The camera installation and configuration methods vary according to different cameras. This section will not describe in details.

Configuring vision process

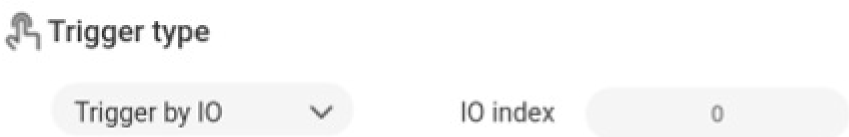
Click **Vision Config** on the right side of the **Vision** commands to start configuring the camera. If you configure the camera for the first time, click **New** and enter a camera name to create a camera configuration. Then the following page will be displayed.



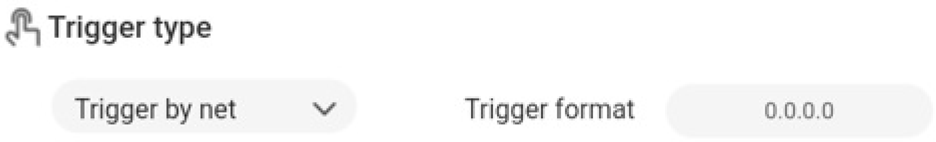
Trigger type

Set a type to trigger the camera.

- Trigger by IO: Connect the camera to the DO interface of the robot. You need to configure the corresponding output port according to electrical wiring port.



- Trigger by net: Connect the camera to the Ethernet port of the robot. You need to configure the strings that the robot sends through the network to trigger the camera.



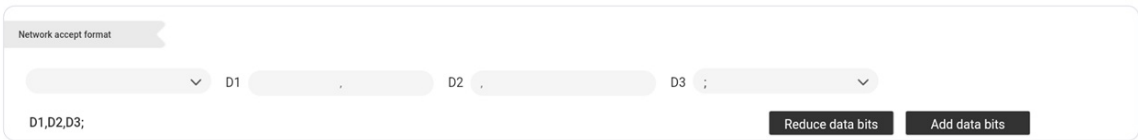
Basic network parameters

The basic network parameters are used to set the communication mode between the camera and the robot, including the following modes.

- TCP_Client: TCP communication. The robot serves as the client and the camera as the server. You need to configure the IP address, port and timeout of the camera.
- TCP_Server: TCP communication. The robot serves as the server and the camera as the client. You need to configure the port and the timeout of the camera.

The receiving method includes two modes: block and non-block. Please select according to the project script.

- Block: After sending the trigger signal, the program will stay at the data-receiving line during the block time, and the program will continue to execute until the data sent by the camera is received; If the blocking time is set to 0, the program will wait at the data receiving line until it receives the data sent by the camera.
- Non-block: After sending the trigger signal, the program continues to execute no matter whether the data from the camera is received or not.



The network accept format refers to the data type sent by the camera used for parse. If the current default data bit is not enough, you can click **Add data bits** to increase the length of received data to a maximum of 8 bits: No, D1, D2, D3, D4, D5, D6, STA, where **No** indicates the start bit template number, and **STA** indicates the end bit (status bit).

You can set a variety of data formats, such as:

- Without start bit and end bit: XX, YY, CC;
- With a start bit but no end bit: No, XX, YY, CC;
- With no start bit but an end bit: XX, YY, CC, STA;
- With a start bit and end bit: No, XX, YY, CC, STA;

Click **Save** on the right-top corner after configuration.

InitCam

Command:

```
InitCam(CAM)
```

Description:

Connect to the specified camera and initialize it.

Required parameter:

CAM: Name of the camera, which should be consistent with the camera configured in the vision process

Return:

Initialization result.

- 0: Initialization successful
- 1: Failed to be initialized

Example:

```
InitCam("CAM0")
```

Connect to the CAM0 camera and initialize it.

TriggerCam

Command:

```
TriggerCam(CAM)
```

Description:

Trigger the initialized camera to take a picture.

Required parameter:

CAM: Name of the camera, which should be consistent with the camera configured in the vision process

Return:

Trigger result.

- 0: Trigger successfully
- 1: Fail to trigger

Example:

```
TriggerCam("CAM0")
```

Trigger the CAM0 camera to take a picture.

SendCam

Command:

```
SendCam(CAM,data)
```

Description:

Send data to the initialized camera.

Required parameter:

- CAM: Name of the camera, which should be consistent with the camera configured in the vision process
- data: data sent to camera

Return:

Result of send data.

- 0: Send successfully
- 1: Failed to send

Example:

```
SendCam(“CAM0”,“0,0,0,0”)
```

Send data (“0,0,0,0”) to the CAM0 camera.

RecvCam

Command:

```
RecvCam(CAM,type)
```

Description:

Receive data from the initialized camera.

Required parameter:

CAM: Name of the camera, which should be consistent with the camera configured in the vision process

Optional parameter:

type: data type, value range: number or string (number by default)

Return:

- err: error code
  - 0: Receive data correctly
  - 1: Time out
  - 2: Incorrect data format which cannot be parsed
  - 3: Network disconnection
- n: number of data groups sent by the camera.
- data: data sent by the camera is stored in a two-dimensional array.

Example:

```
local err,n,data = RecvCam("CAM0","number")
```

Receive data from the CAM0 camera, and the data type is number.

DestroyCam

Command:

```
DestroyCam(CAM)
```

Description:

Release the connection with the camera.

Required parameter:

CAM: Name of the camera, which should be consistent with the camera configured in the vision process

Return:

- 0: The camera has been disconnected.
- 1: The camera failed to be disconnected.

Example:

```
DestroyCam(“CAM0”)
```

Release the connection with the camera CAM0.

Example

After setting the vision parameters, you can call vision APIs for programming to receive data from the camera. The demo below is about obtaining the data from CAM0 and assigning the value to point 2.

```
while true do
  ::create_camera::
  resultInit = InitCam("CAM0")                              --Connect CAM0 camera
  if resultInit ~= 0 then
      print("Connect camera failed, code:", resultInit)
      Sleep(1000)
      goto create_camera
  end
  while true do
      TriggerCam("CAM0")                                    --Trigger CAM0 camera  photo
      SendCam("CAM0","1,2,3,0;")                            --Send data to CAM0 camera
      err,visionNum,visionData  = RecvCam("CAM0","number")  --Receive CAM0 camera data
      if err ~= 0 then
          print("Failed to read data")
          Sleep(1000)
          break
      end

      print("(visionNum):",(visionNum))                     --Print how many sets of CAM0 camera data received
      print("(visionData[1][1]):",(visionData[1][1]))       --Print the first data of the first group received
      i = 1
      while not ((visionNum)<i) do
      print(type(P2.coordinate[1]))
      print(P2)
      P2.coordinate[1]=(visionData[i][1])                    --visionData[i][1] is assigned to P2.X
      P2.coordinate[2]=(visionData[i][2])                   --visionData[i][2] is assigned to P2.Y
      Go(P2,"SYNC=1")
      i = i + 1
      end
      Sleep(10)
  end
  Sleep(10)
end
```