DOBOT

DobotStudio Pro User Guide
(CR & Nova)

Shenzhen Yuejiang Technology Co.,Ltd.|China

Table of Contents

Preface

Purpose

This document describes the functions and operations of DobotStudio Pro for controlling six-axis robots (CR and Nova series), 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
2024/02/06	Update based on V2.8.0
2023/09/22	Optimize the contents of lua commands
2023/05/29	Update based on V2.7.1
2023/05/16	Update based on V2.7.0
2023/01/12	Improve the functions of Dobotblockly programming (V2.6.0)
2022/11/30	Improve the functions of collision detection, remote I/O and blockly programming in the latest software (V2.5.0) Add best practice Add description on DobotBlockly commands and Script commands
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, robot arm damage
 NOTICE	Indicates a potentially hazardous situation which, if not avoided, can result in robot 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 six-axis robot arm (CR and Nova series).

DobotStudio Pro supports the following operation systems:

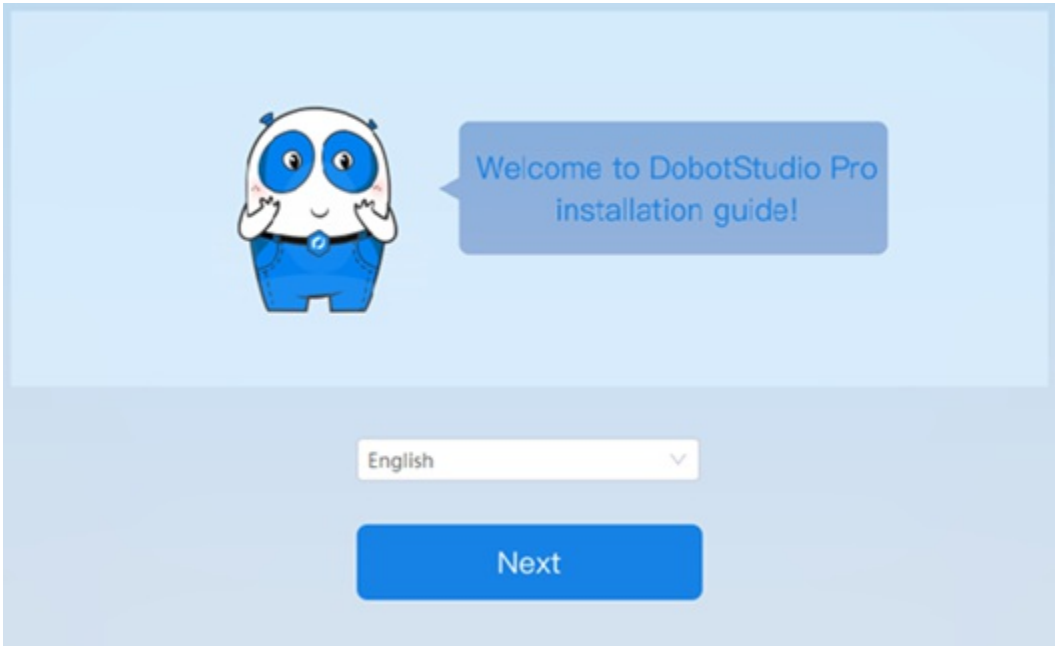
- Win7
- Win10
- Win11

DobotStudio Pro Installation

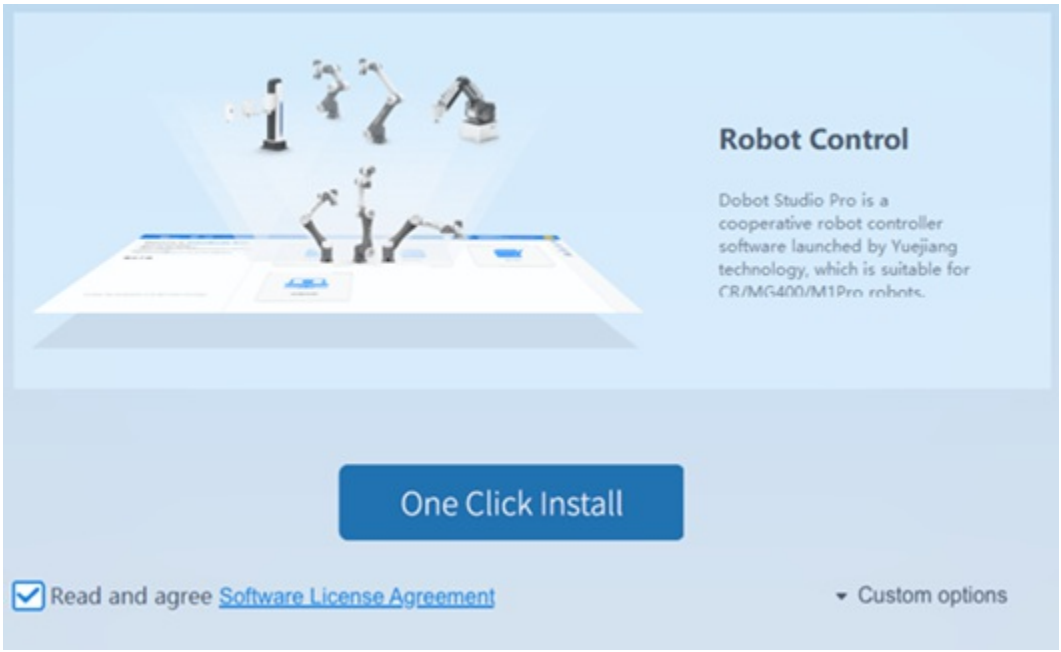
Please visit Dobot website to download the latest DobotStudio Pro installation package.

Procedure

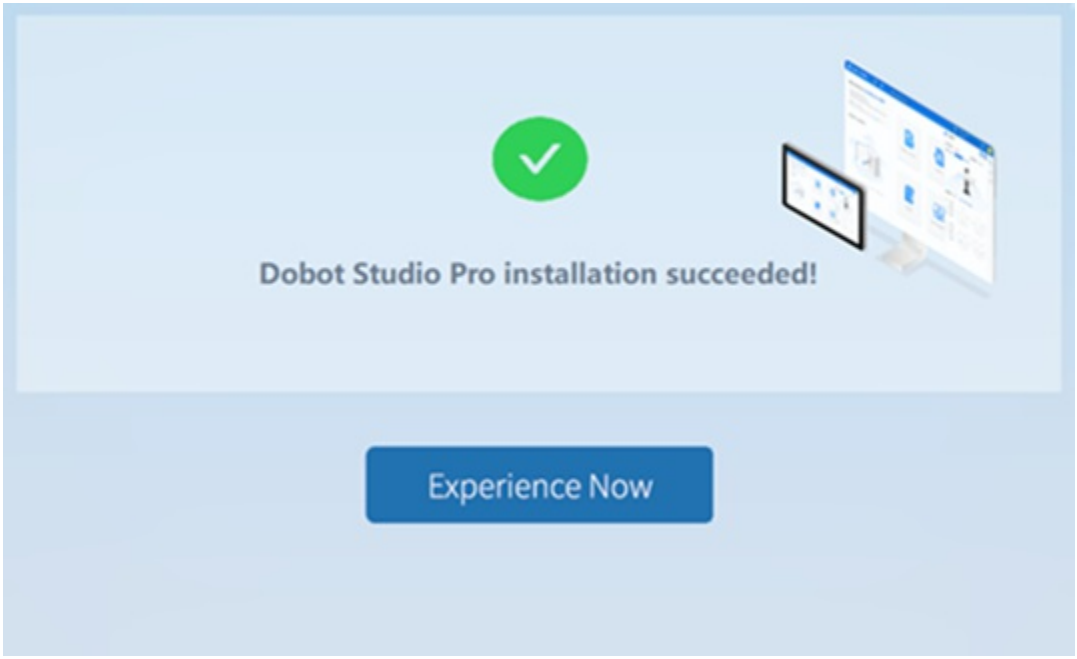
Step 1: Double-click DobotStudio Pro installation package. Select a language for installation. Click **Next**.



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. If the robot is installed in an in-ceiling or wall-mounting mode or at a certain angle, you need to set the rotation angle and tilt angle

under the disabled state. See Installation for details.

4. I/O Monitoring: Know about the monitoring function provided by DobotStudio Pro. If you need to install end plug-ins, see Dobot+ for configuration.
5. Programming/Process: 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 3.5.4.0, you need to open SMB1 protocol before connecting to the robot. See (Optional) Open SMB protocol in this chapter for details.

Wired connection

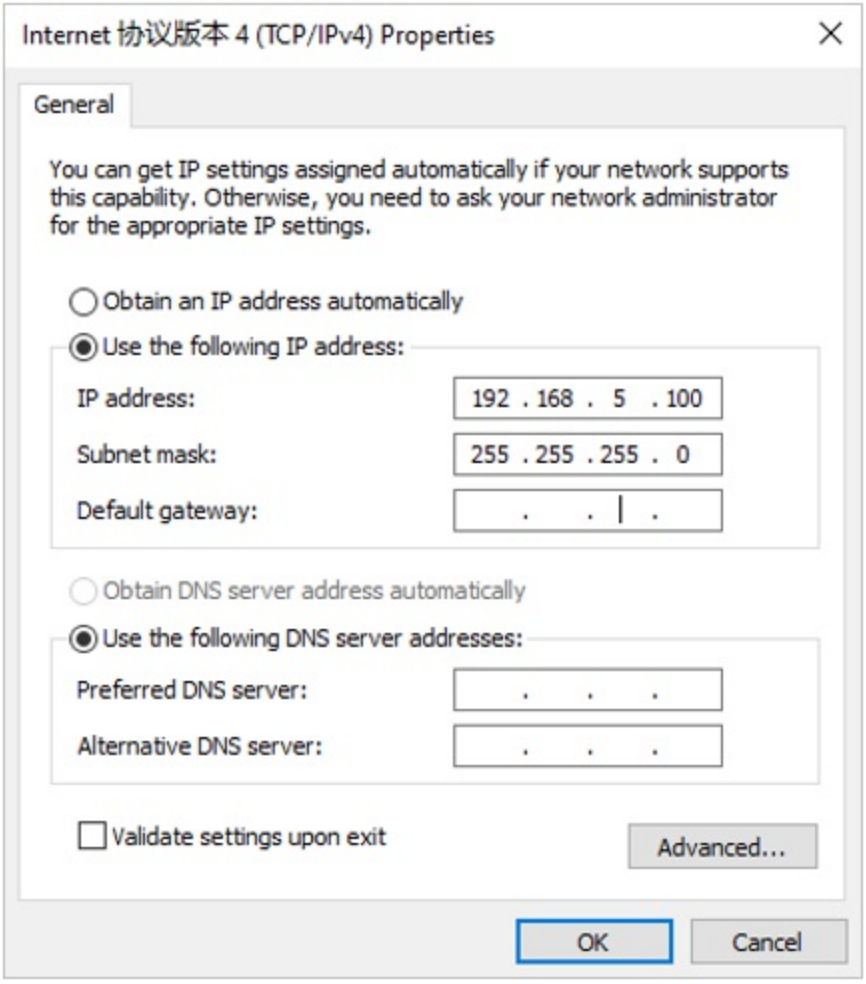
Connect one end of the network cable to the LAN interface on the controller and the other end to the PC. Change the IP address of the PC to make it on the same network segment as that of the controller. The default IP address of the controller is 192.168.5.1, which can be modified in Communication settings.

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

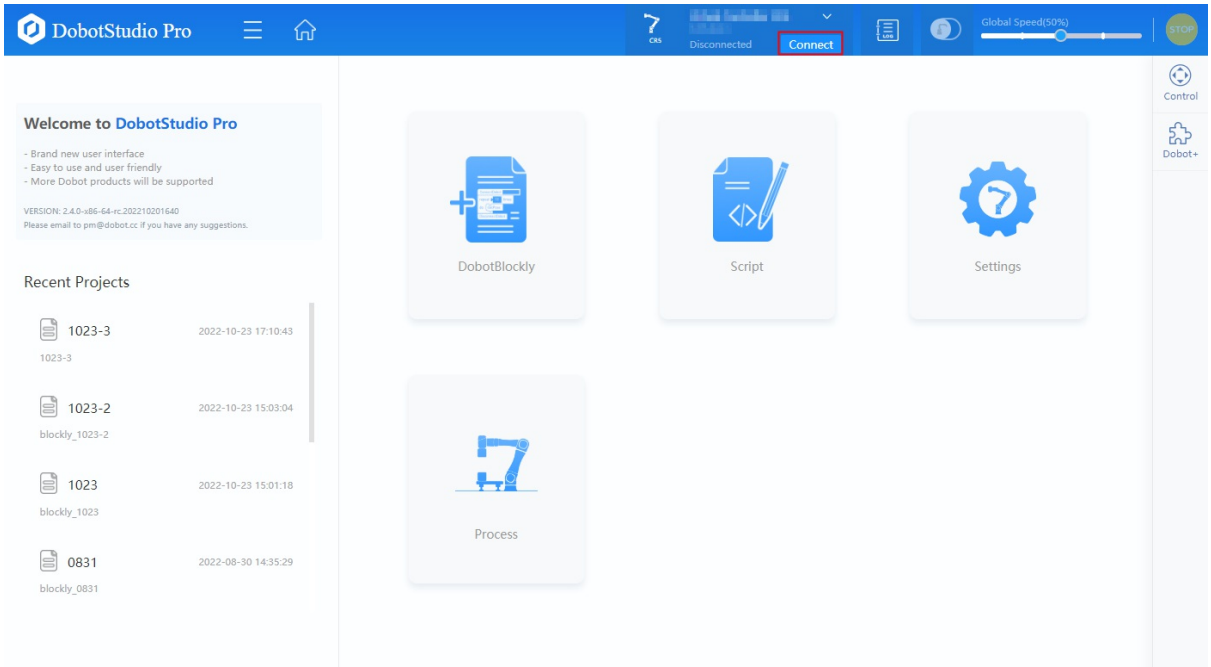
Step 1: Search **View network connections**, and click **Open**.

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

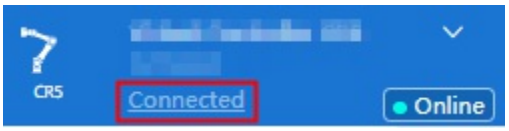
Step 3: Select **Use the following IP address** in "Internet Protocol Version 4(TCP/IPv4)" page, and change the IP address, subnet mask and gateway of the PC. You can change the IP address of the PC to make it in the same network segment as that of the controller without conflict. The subnet mask and gateway of the PC must be the same as that of controller. For example, set the IP address to 192.168.5.100, 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**.



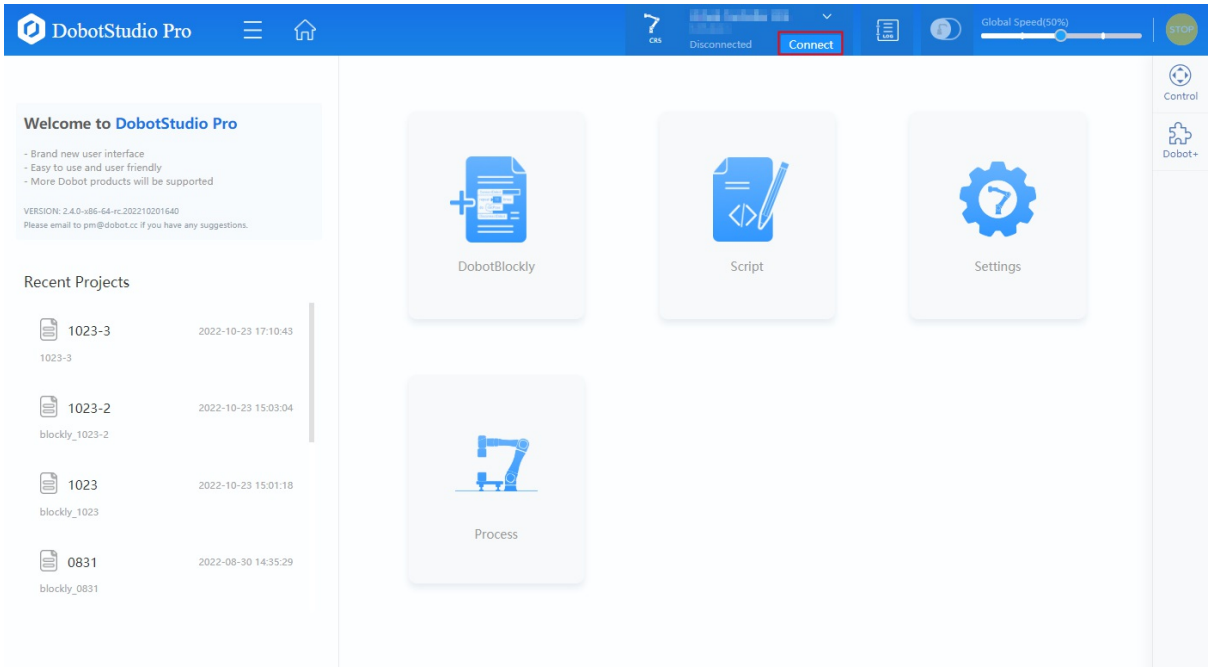
Wireless Connection

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

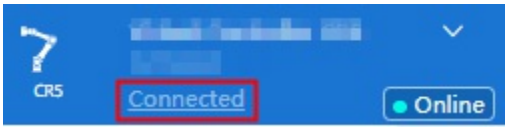
Step 1: Search Dobot controller WiFi name and connect it. The default SSID of WiFi is "Dobot {Product Name}-{Serial Number}", where the serial number is the two 4-digit numbers connected by "-" in the robot's S/N code (the S/N code is affixed to the nameplate on the robot base. For example， the default SSID is "DobotCR5-4025-0731" for the CR5 with SN "4025-0731"), and WiFi password is 1234567890 by default. You can modify the WIFI SSID and password in Communication settings.

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

> **NOTE**
>
> Due to the internal network architecture of the controller, DobotStudio Pro may search for two different IPs of the robot (e.g. 192.168.1.6 and 192.168.5.1) when connecting to the robot via WiFi. No matter which IP is used, the robot can be connected normally. It is recommended to use 192.168.1.6 for connection.

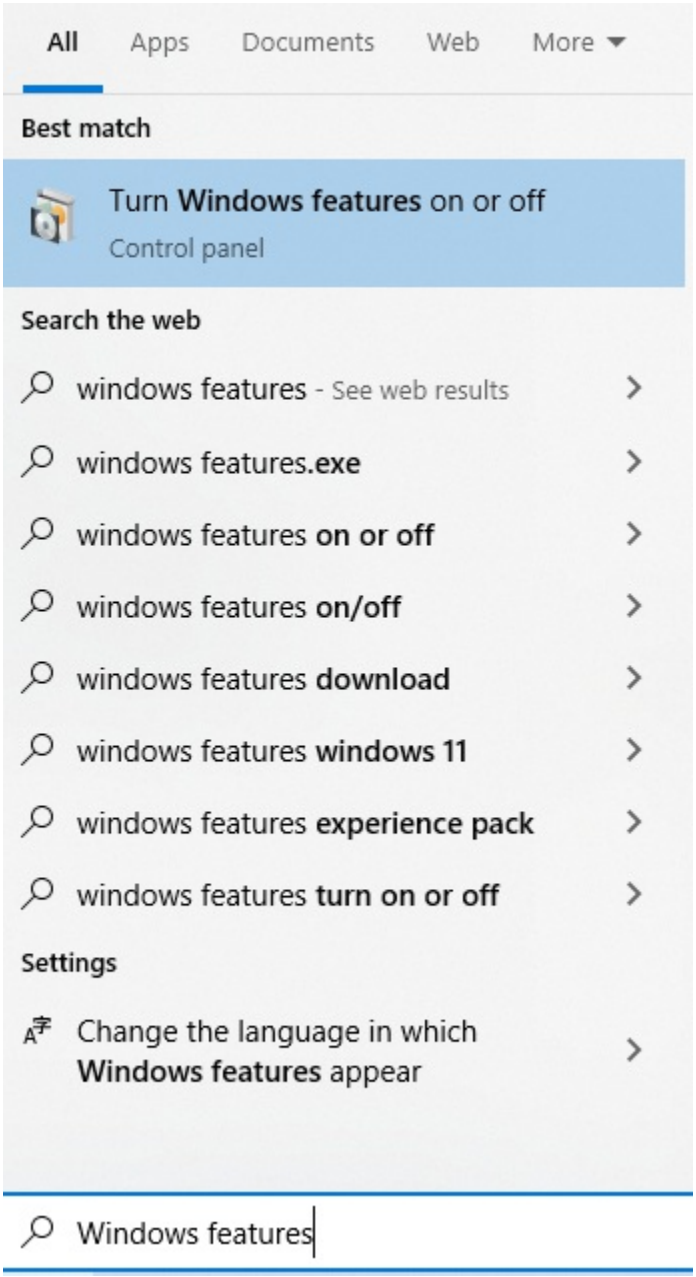


Step 3: After connection, you will see the pop-up prompt information in DobotStudio Pro interface, and the connection status turns to **Connected**. If you want to disconnect the robot, click **Connected**.

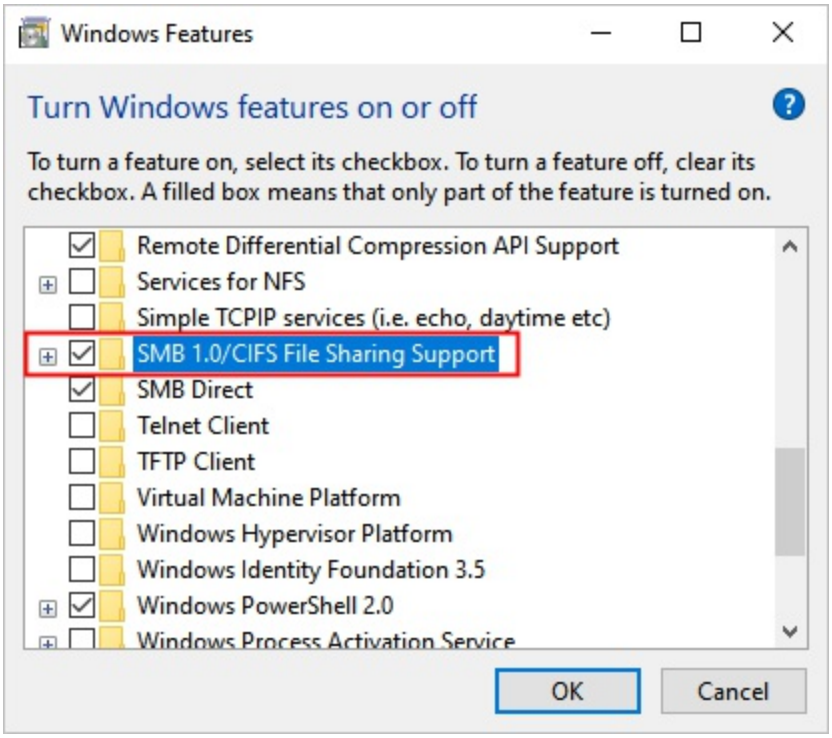


(Optional) Open SMB protocol

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

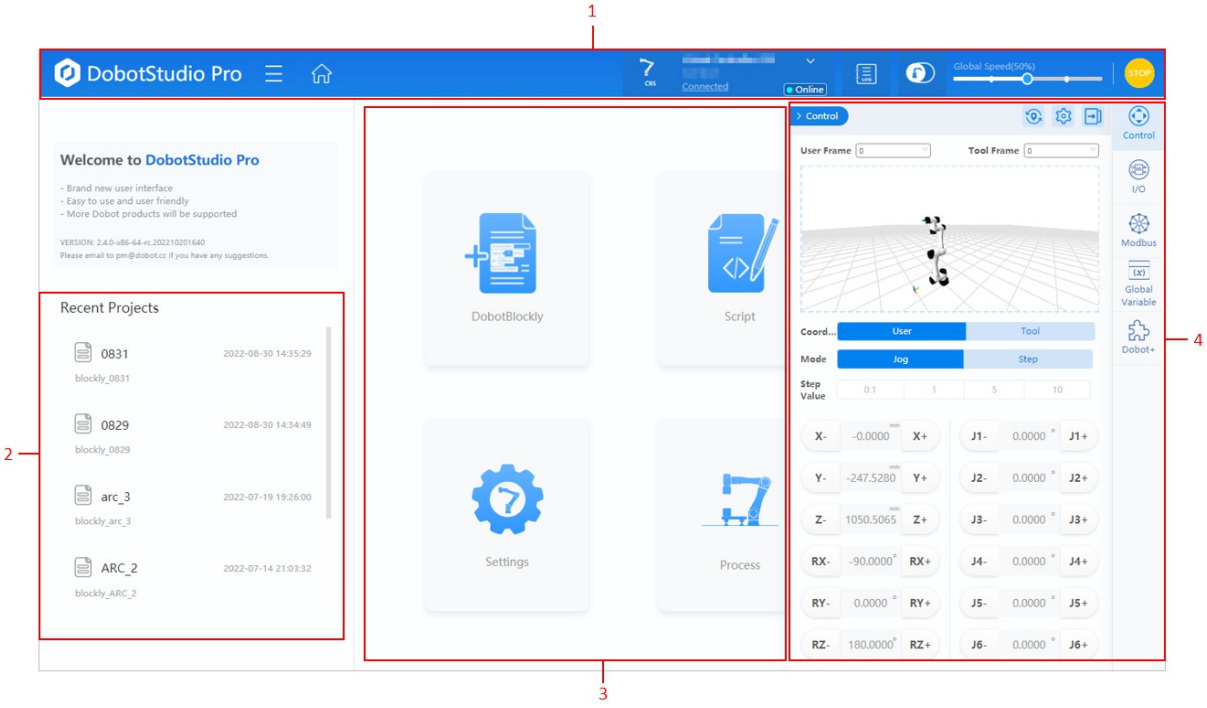


1. Select **SMB 1.0/CIFS File Sharing Support**, and click **OK**.


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



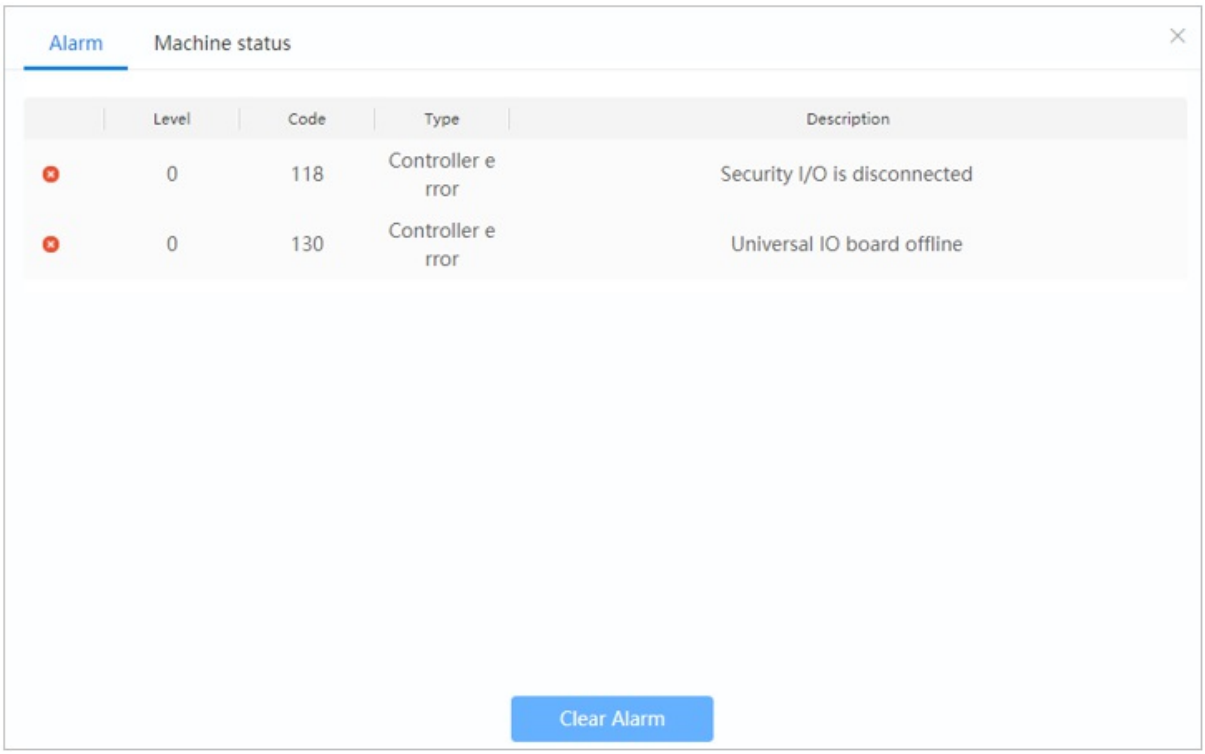
No.	Description
1	Top toolbar
2	Display the recent projects, which you can click to open quickly.
3	Major functions, including DobotBlockly, Script, Settings and Process
4	Click the icon on the right toolbar to display or hide the corresponding panels, including Control, I/O, Modbus, Global Variable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

3.2 Top toolbar

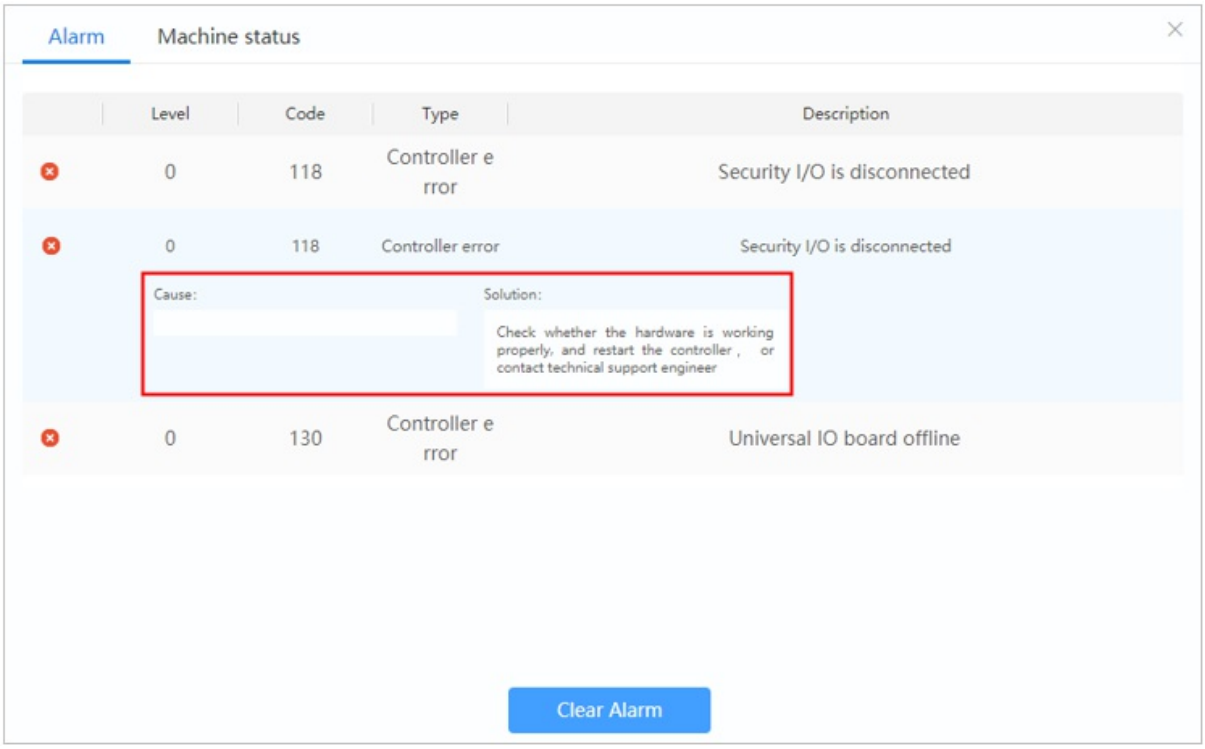
No.	Description
1	Click the icon, and the following items will pop up: • Settings: Click to open Settings page • Language: Select a language • Help: Access help functions, such as help documents, debugging tools, etc. • Check updates: View the version information of the software • About: View the components 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	Enabling button. Click to switch the enabling status of the robot arm. See Enabling status for details. The button is blue in disabled status, green in enabled status, and white in power-off status
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**.

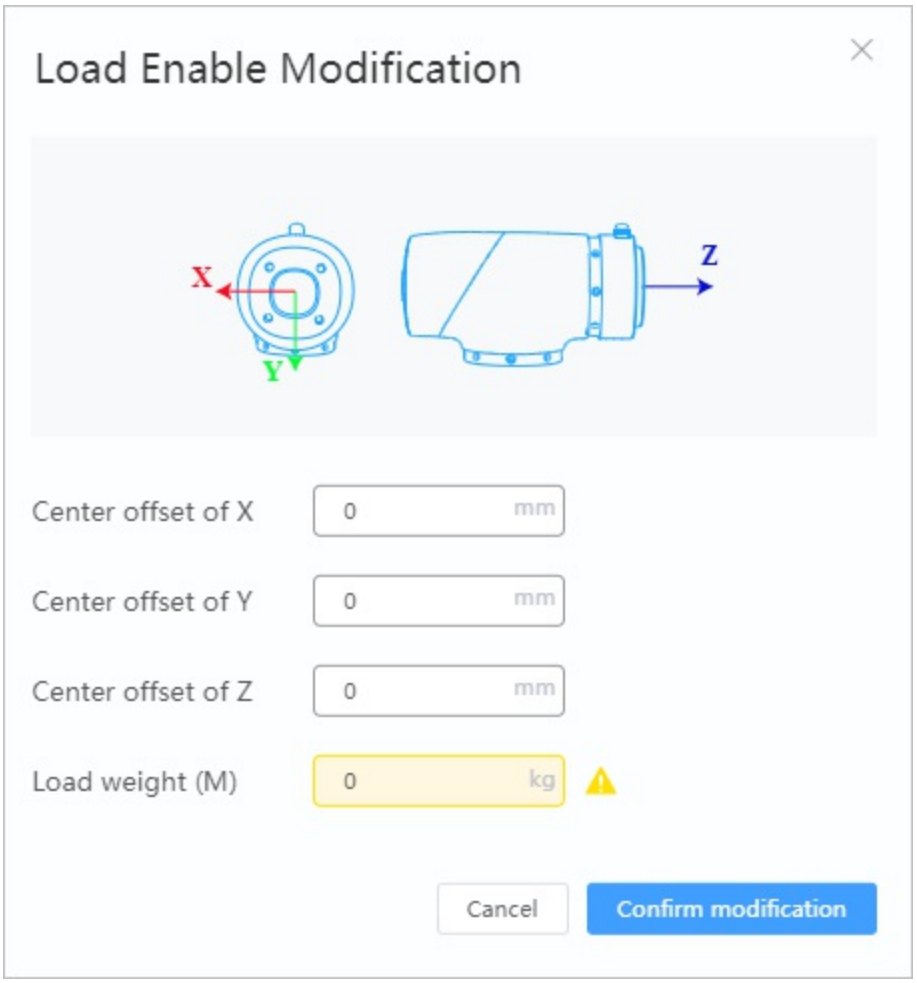


Enabling status

The robot arm can work only in the enabled state.

- When the Enabling button is blue (), the robot arm is in the disabled status. Click the button, and the "Load Enable Modification" window will pop up (the eccentric coordinate of the end load should

be set when the J6 axis is 0°, and the load value should not exceed the maximum allowable load weight of the robot). After setting the parameters, click **Confirm modification** to enable the robot. The robot arm starts to move slightly, and then the end indicator light turns green, indicating that the robot arm is enabled. At the same time, the Enabling button moves to the right side, and the icon turns green ( ) (yellow exclamation mark in the lower right corner indicates that the load is 0kg)



- When the Enabling button is green, the robot arm is in the enabled status. Click the button, and a confirmation box will be displayed. After your confirmation, the robot arm starts to be disabled. The indicator light at the end of the robot arm turns blue, indicating that the robot arm is disabled. At the same time, the Enabling button also 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

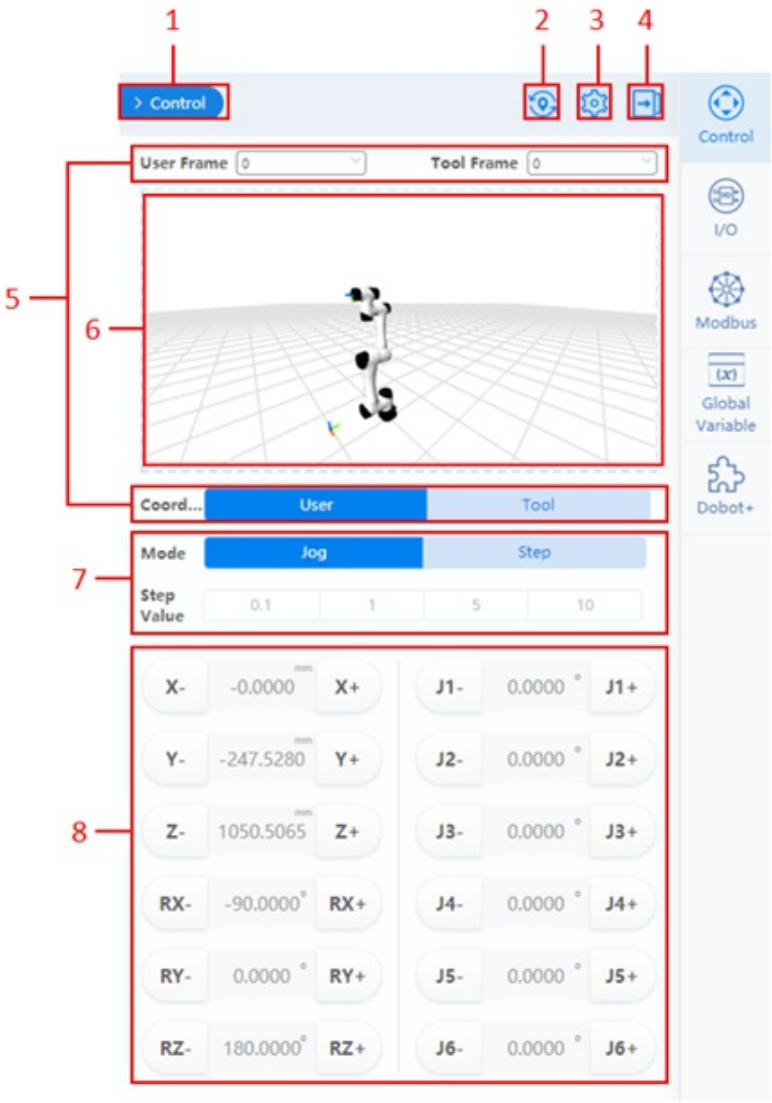
Once the emergency stop button is pressed, the robot arm will stop running and be powered off, and the emergency stop icon will turn red.

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



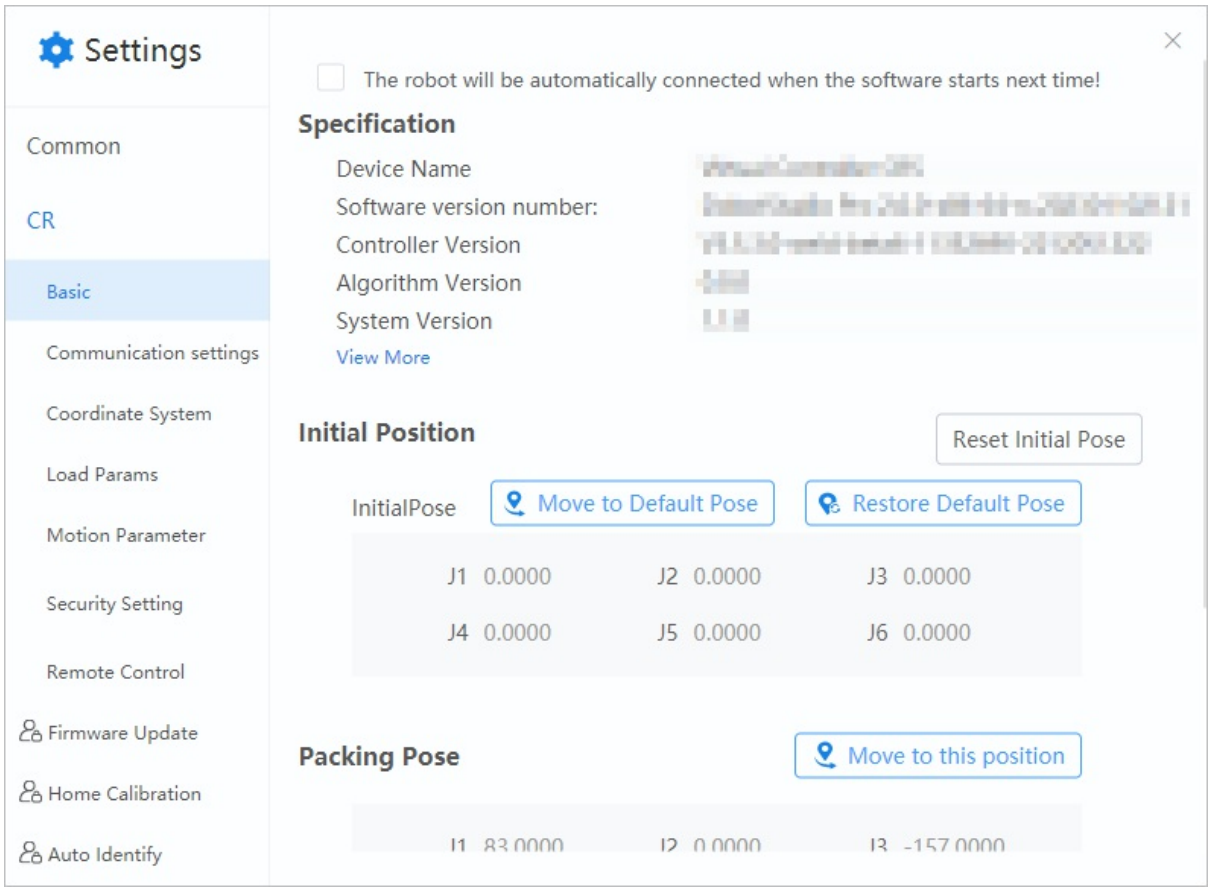
No.	Description
1	Click to hide the panel. You can click **Control** in the right toolbar to display the panel.
2	Long-press the button to move the robot to its initial posture, which can be set in Settings.
3	Click to open Settings page.
4	Click to fold the control panel, and click it again to unfold 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.
	Select the motion mode of the robot • Jog: the robot keeps moving when you press and hold the jog button, and stops

7	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 left column is jog buttons for Cartesian coordinate system, and the right column 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

The Basic Settings page is used to see 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 point **View More** to see more firmware version information.

Initial posture

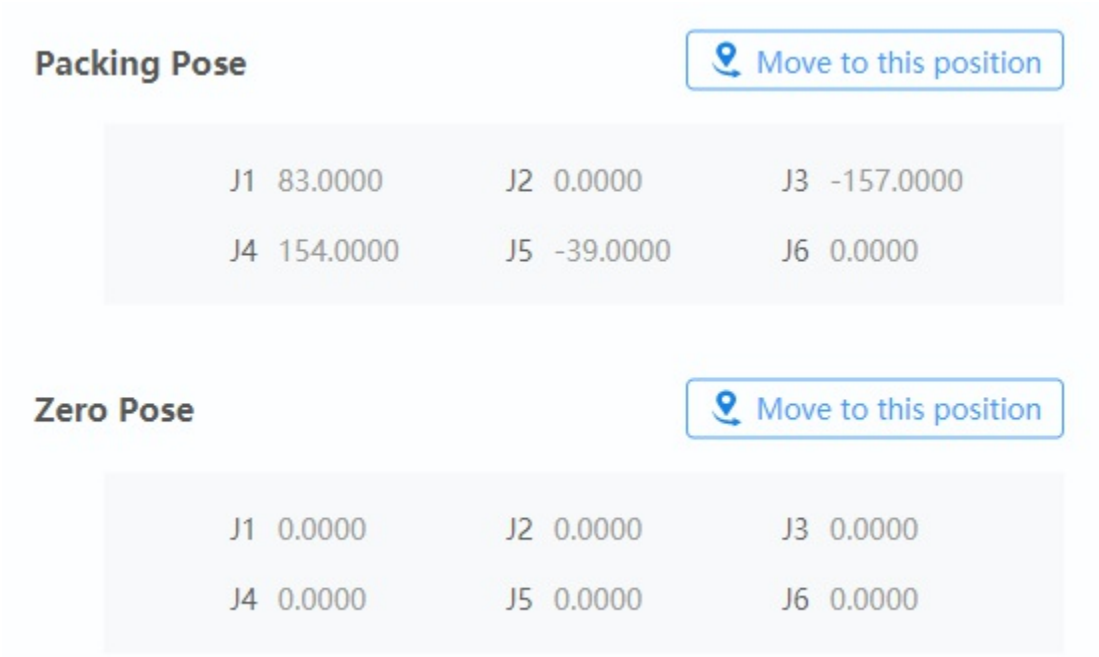
The initial posture is a self-defined posture, which is the home posture by default, namely, all joint angles are 0.

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.

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.

Package posture and home posture



Moving the robot to the package posture can reduce the robot space, making it easy to pack and transport.

In home posture, all joints angles are 0.

Long press **Move to this position** to move the robot to the corresponding posture.

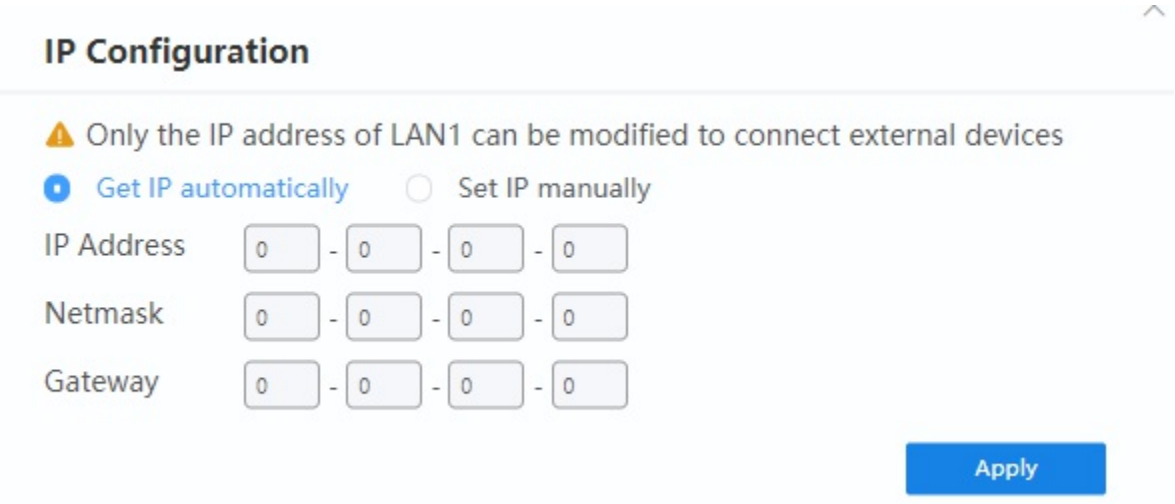
4.2 Communication settings

IP Setting

The robot can communicate with external device thourgh the LAN interface which supports TCP, UDP and Modbus protocols. You can modify the IP address, netmask and gateway. The IP address of the robot must be within the same network segment as that of the external device without conflict.

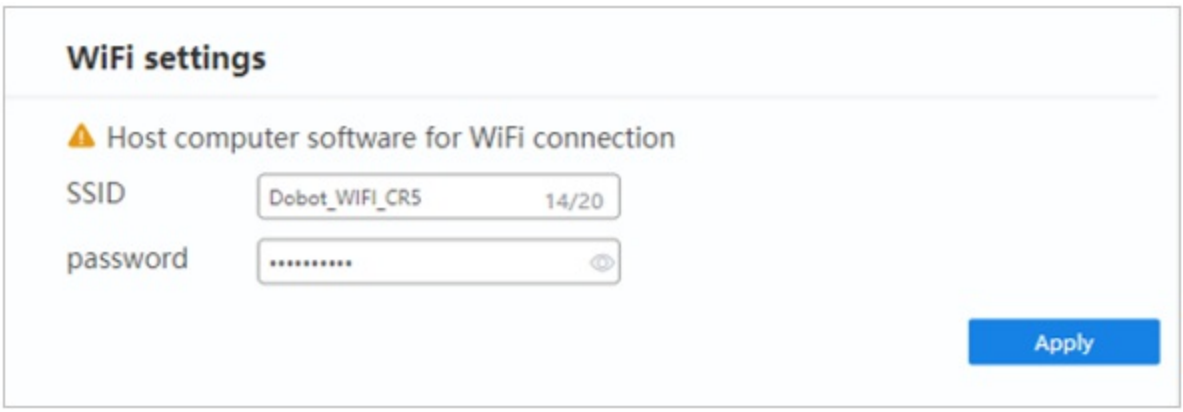
When the controller is CC162, there is only one LAN interface, and the address of this interface is modified. When the controller is CCBOX, there are two LAN interfaces, and the address of the LAN1 interface is modified. The default IP address of the LAN1 interface of both controllers is 192.168.5.1.

- If the robot arm is connected to the external device directly or through a switch, you need to check **Set IP manually**, modify the IP address, netmask and gateway and click **Apply**.
- If the robot arm is connected to the external device through a router, check **Get IP automatically** (the IP address is automatically assigned by the router), and then click **Apply**.



WiFi Setting

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


WiFi settings
Host computer software for WiFi connection
SSID
Dobot_WIFI_CR5
14/20
password
Apply

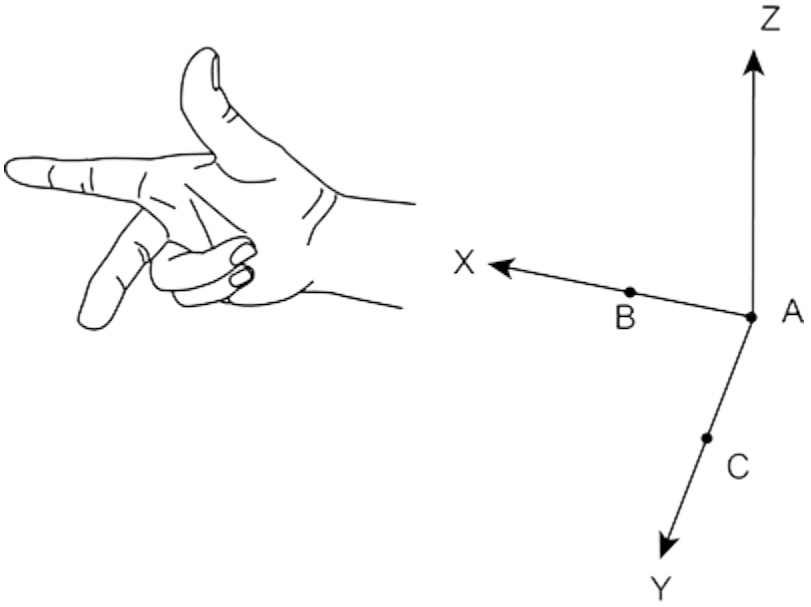
4.3 Coordinate system

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 to simplify 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.

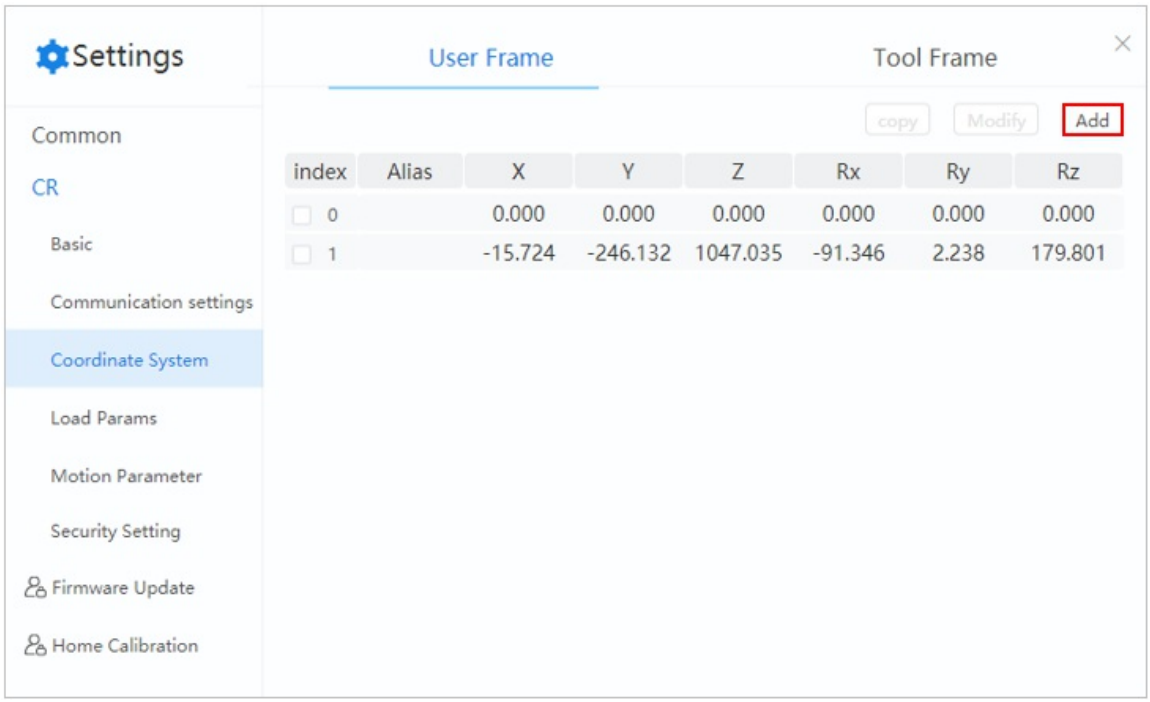
> NOTE
> When creating a user coordinate system, make sure that the reference coordinate system is the base coordinate system.

The user coordinate system is created by three-point calibration method. Move the robot to three points: **A**, **B**, and **C**. Point **A** is defined as the origin and the line from point **A** to point **B** is defined as the positive direction of X-axis. The line that point **C** is perpendicular to X-axis is defined as the positive direction of Y-axis. The Z-axis can be defined based on the right-hand rule.

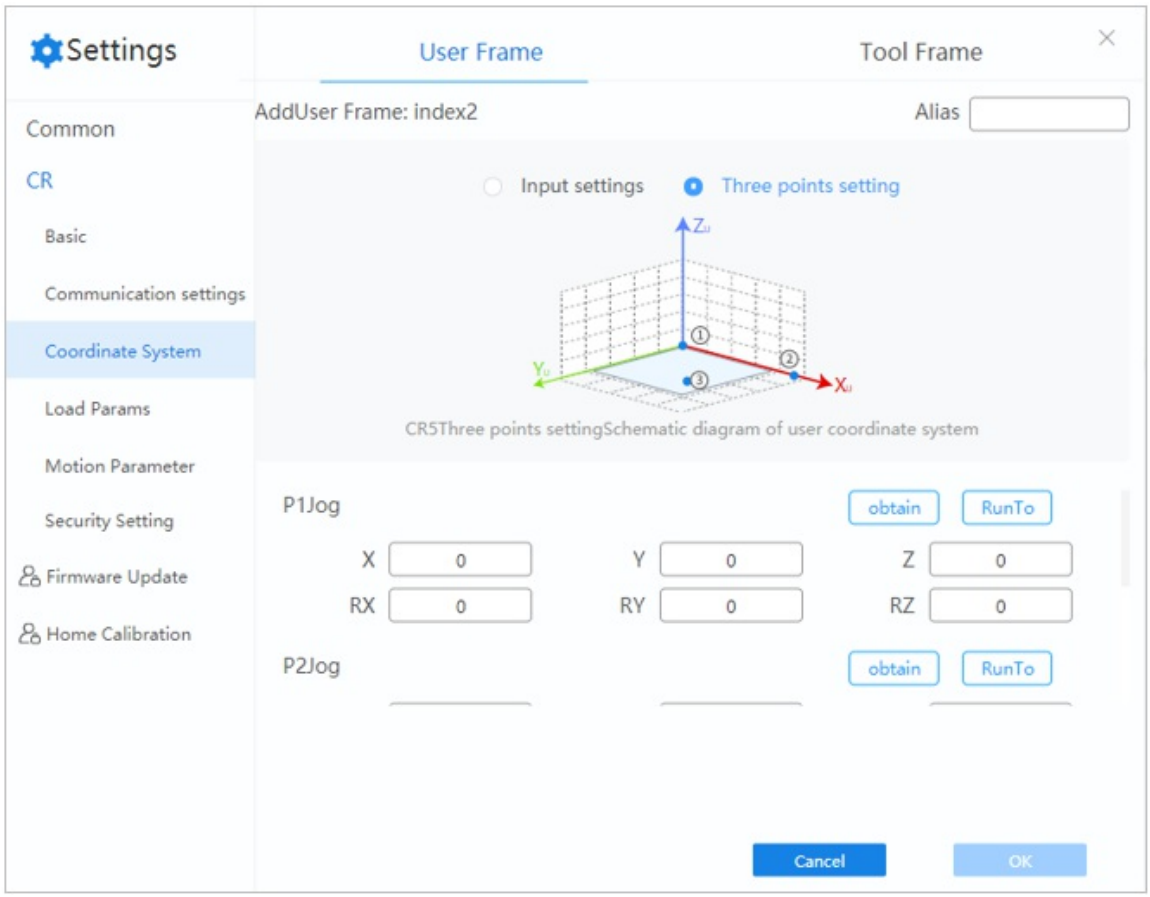


Creating user coordinate system

1. Click **Add**.



2. Select **Three points setting** in "Add User Frame: index2" page.



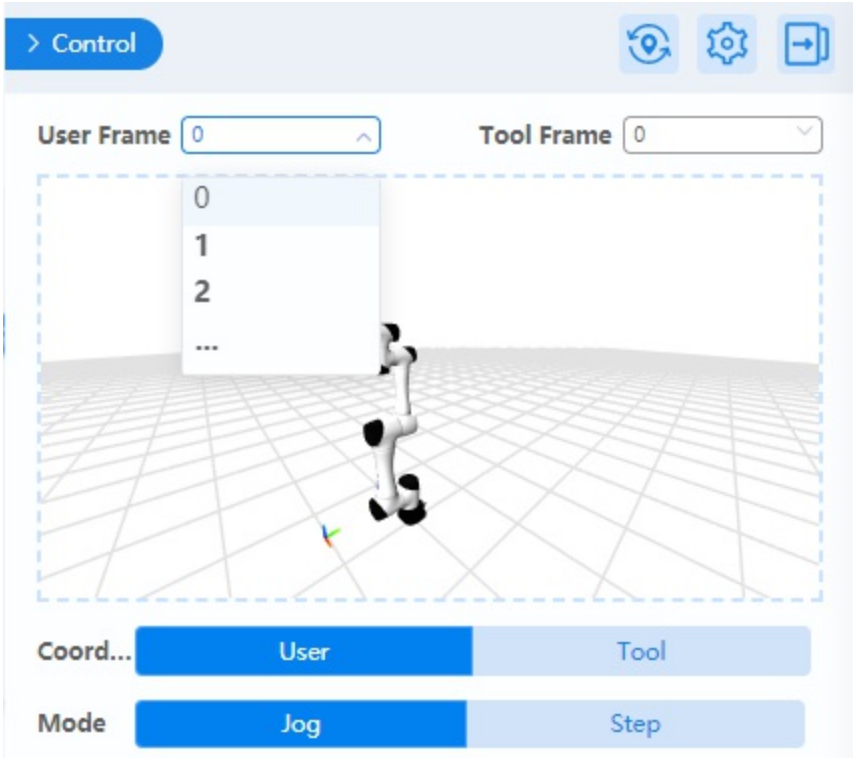
NOTE

- 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. Jog the robot to the point P3 and click **obtain** on the P3 panel.
4. Click **OK**. The user coordinate system is created successfully.

Now you can select a user coordinate system 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, Rx, Ry and Rz 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.

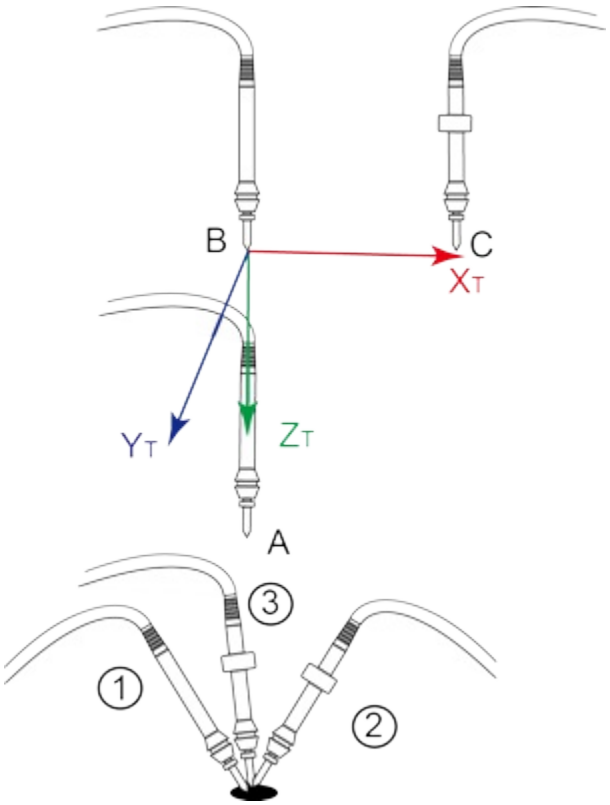
4.3.2 Tool coordinate system

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.

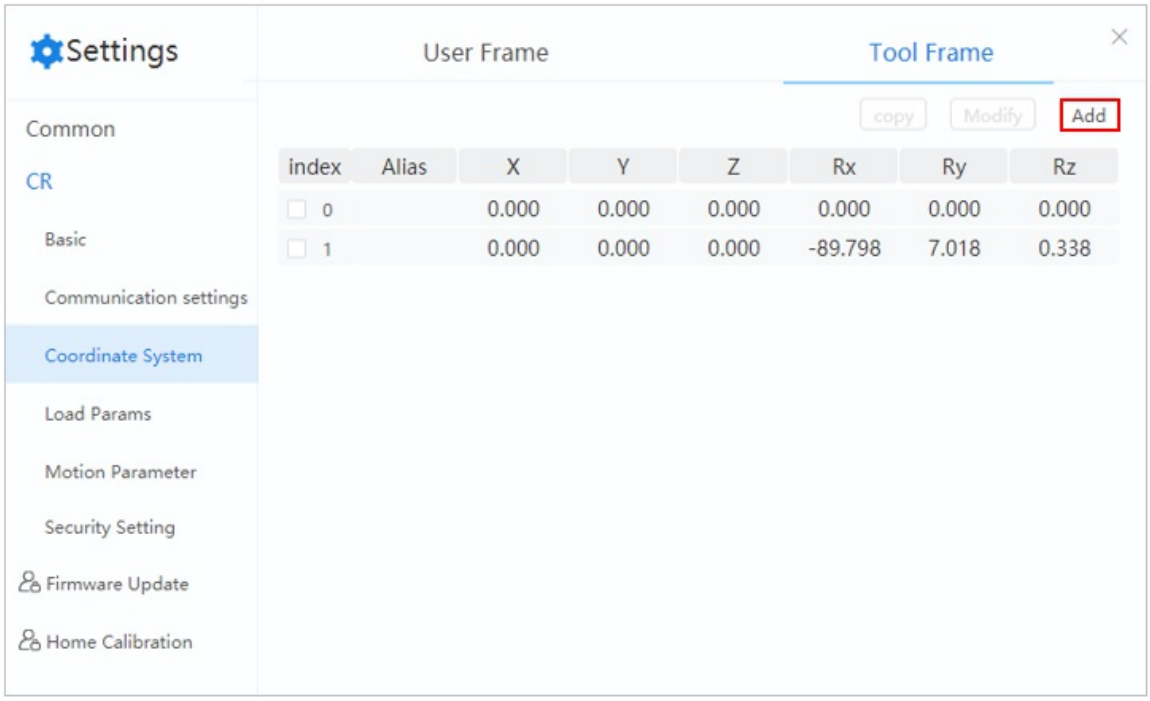
The tool coordinate system of six-axis robot is created by three-point calibration method (TCP +ZX): After mounting the end effector, adjust the direction of the end effector to make TCP (Tool Center Point) align with the same point (reference point) in three different directions for obtaining the position offset of the end effector. Then jog the robot to three other points (A, B, C) for obtaining the angle offset.



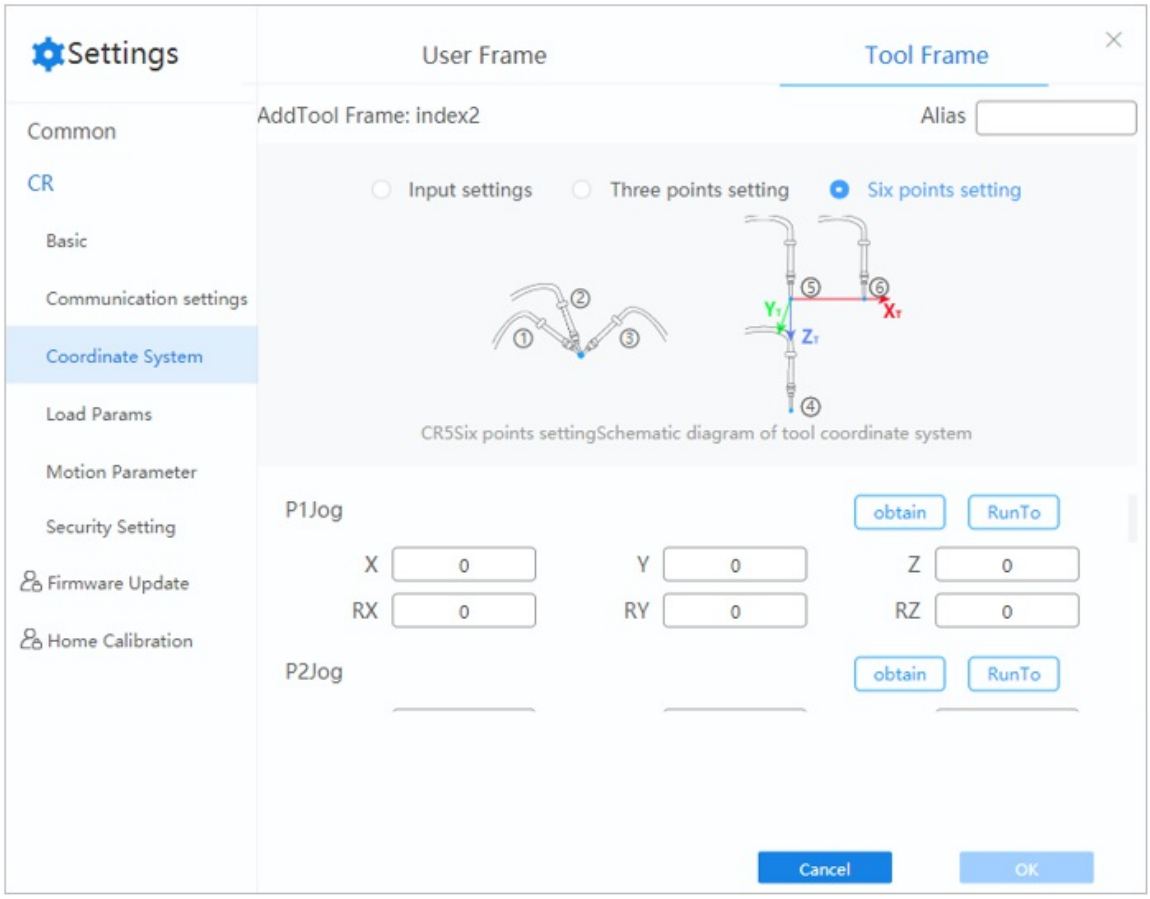
Creating tool coordinate system

1. Mount an end effector on the robot.

2. Click **Add**.



3. Select **Six points setting** in "Add Tool Frame: index2" page.



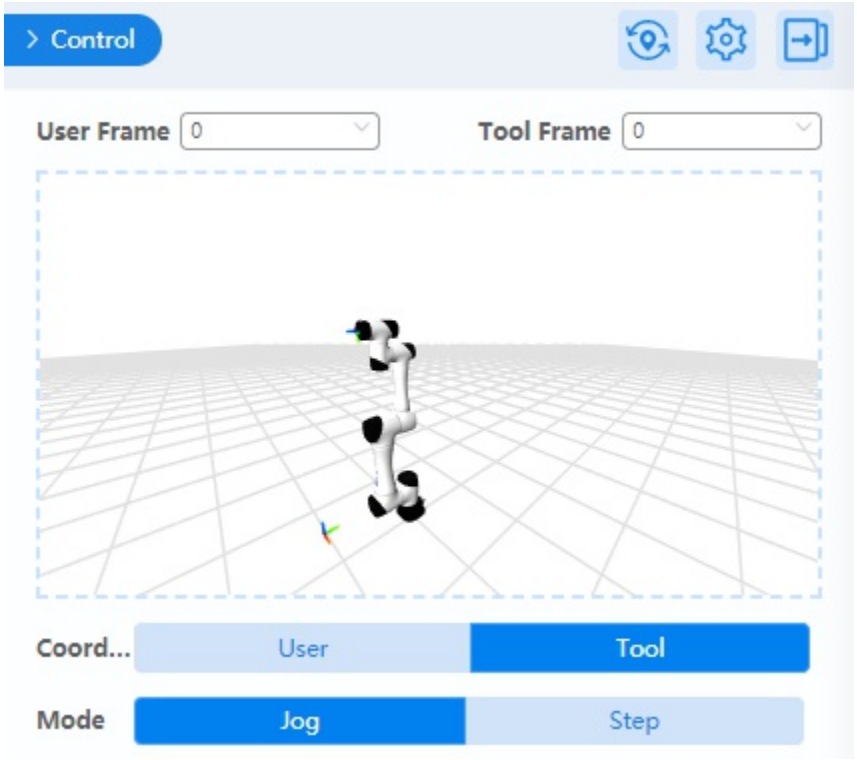
NOTE

- 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. Jog the robot to the reference point in the third direction, then click **obtain** on the P3 panel.
4. Jog the robot to the reference point in the vertical direction, then click **obtain** on the P4 panel.
5. Jog Z-axis along the positive direction to move the robot to another point, then click **obtain** on the P5 panel.
6. Jog X-axis along the positive direction to move the robot to point P6 (not in the same line with P4 and P5). Click **obtain** on the P6 panel.
7. 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** or **Three points setting** in Step 3.

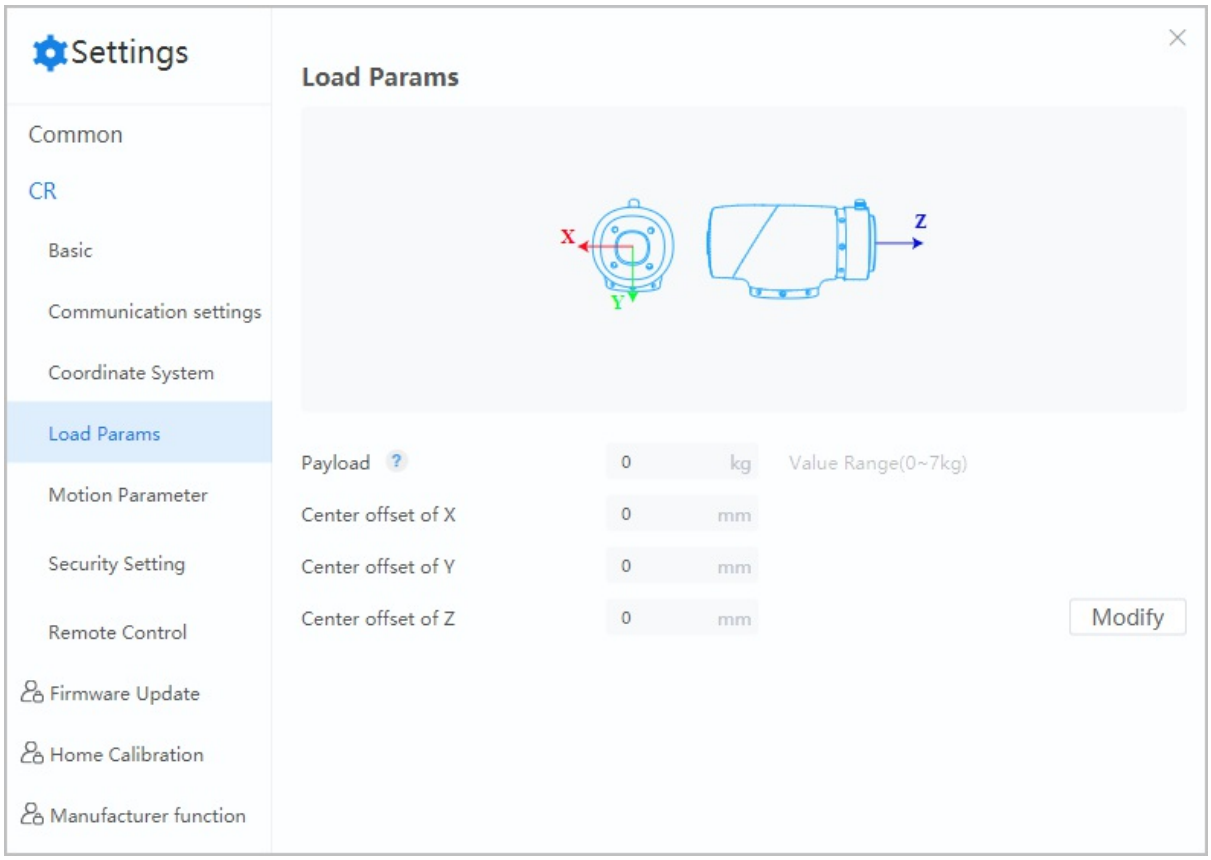
Other operations

- 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 6 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 "Load Enable Modification" window will pop up which requires you to set the load parameters. The parameters you set will be synchronized to the "Load Params" page.



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

- You need to set the eccentric coordinate of the load when J6 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 settings may cause abnormal collision detection alarms or the robot arm to be out of control during dragging.

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

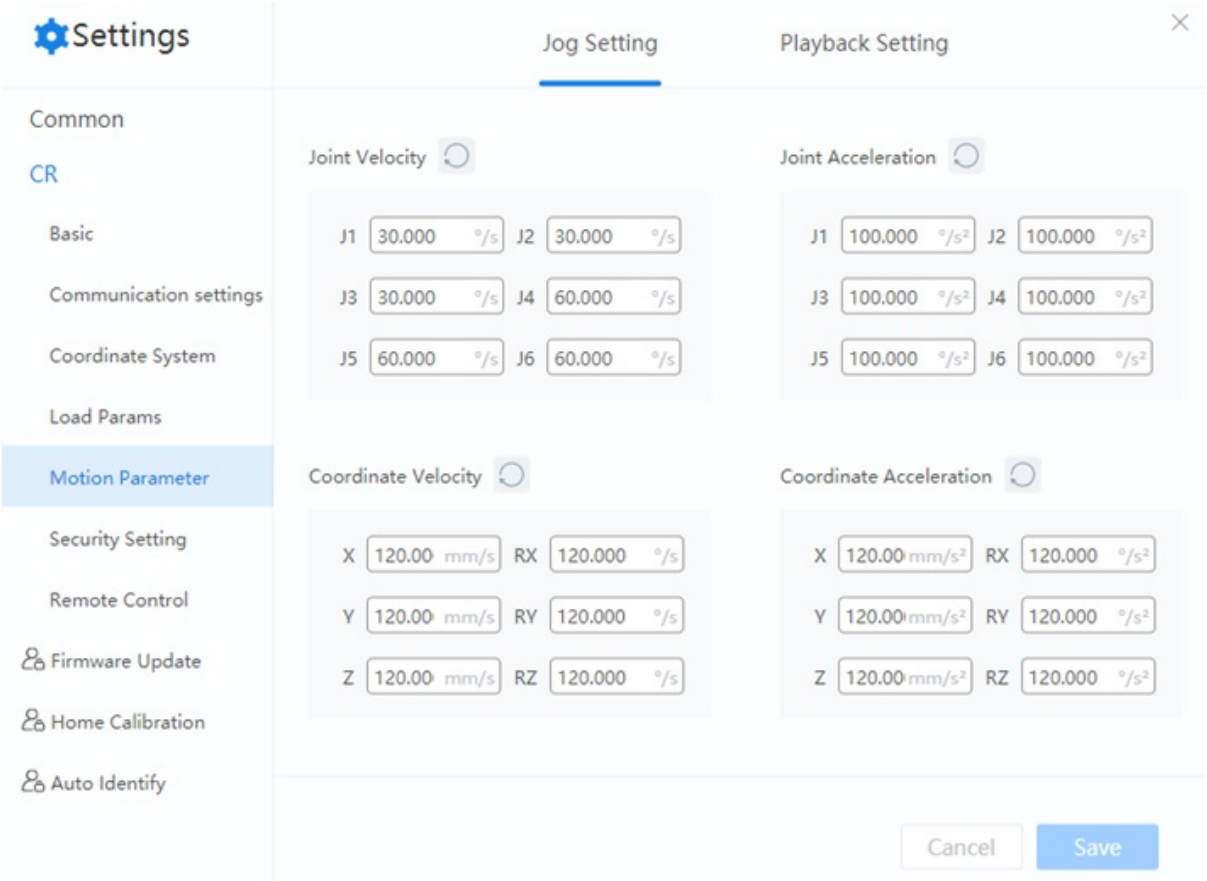
Please set load and eccentric coordinates properly. Otherwise, it may cause errors or excessive shock, and shorten the life cycle of parts.

4.5 Motion Parameters

The optimal motion parameters have been set before delivery, and are not recommended to be modified without special requirements. If you feel that the robot speed is too high, you can adjust it downwards according to your actual needs. If a higher speed is required, please contact technical support for a speed increase solution.

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.

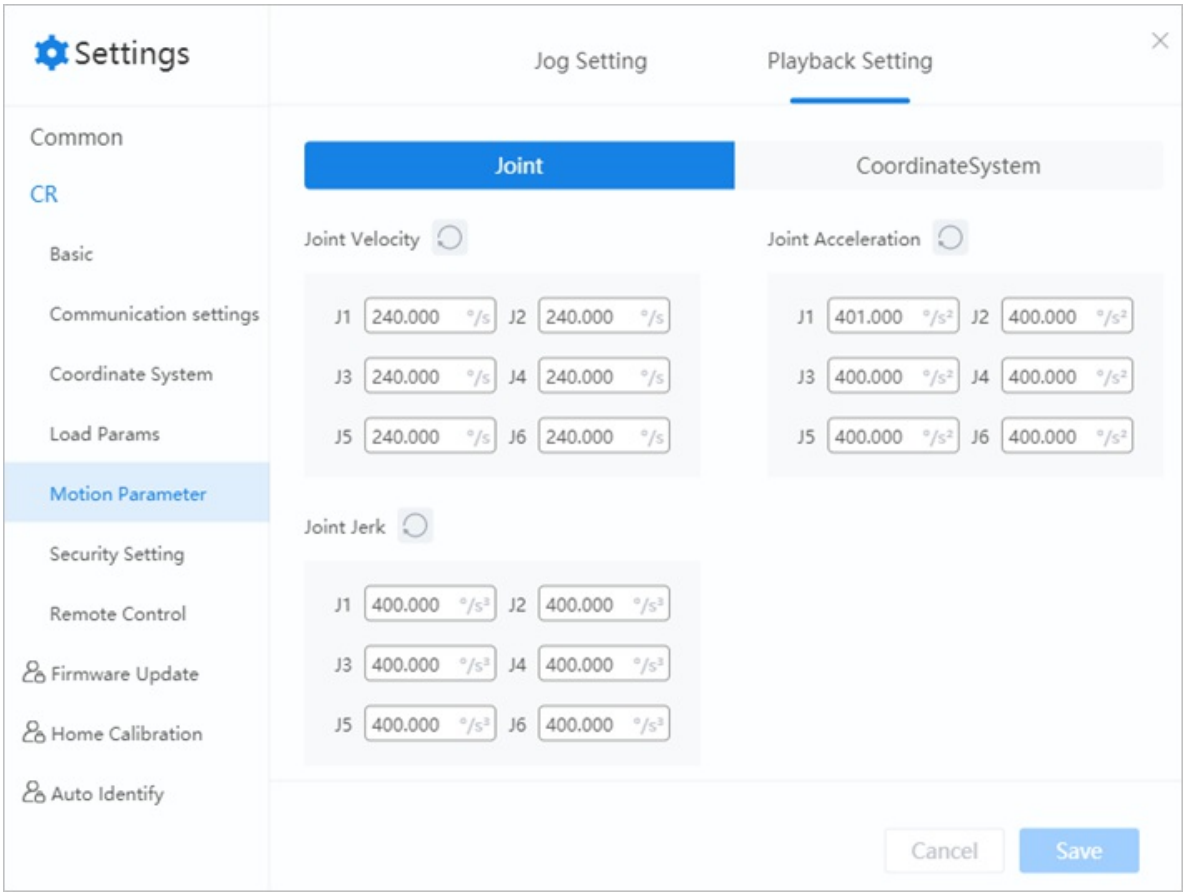


Actual robot speed/acceleration = set speed/acceleration × global speed ratio.

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.



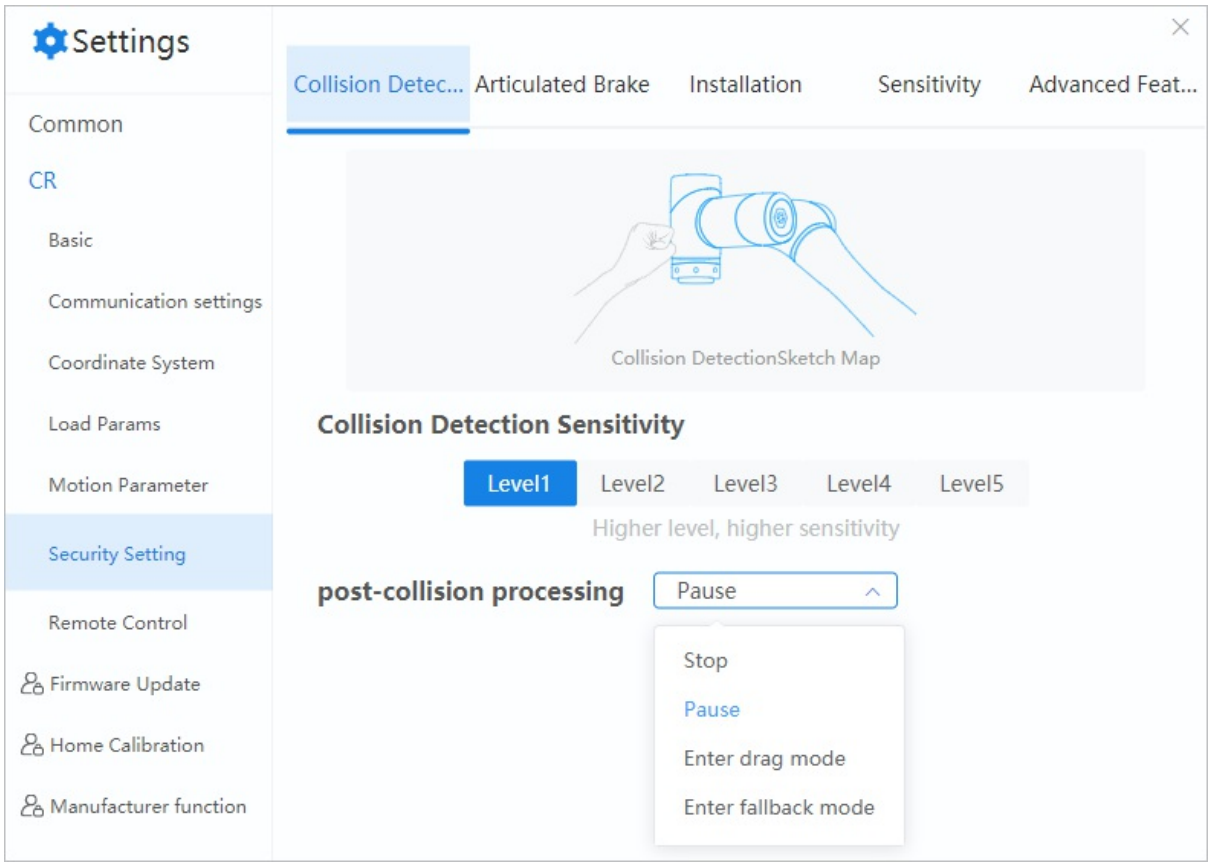
Actual robot speed/acceleration = set speed/acceleration × global speed ratio × set percentage in speed commands when programming.

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

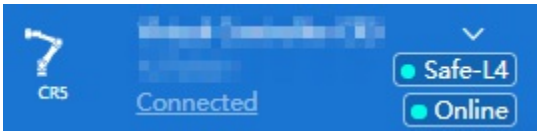
4.6 Security setting

4.6.1 Collision detection

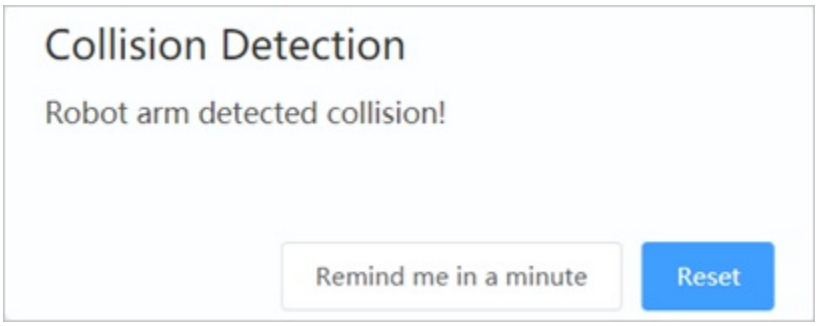
Collision detection is mainly used for reducing the impact on the robot to avoid damage to the robot or external equipment. If collision detection is activated, the robot arm will suspend running automatically when hitting an obstacle.



The higher sensitivity level you select, the smaller force the robot requires to stop after collision detection. The safety level will be displayed in the connection panel on the top toolbar.

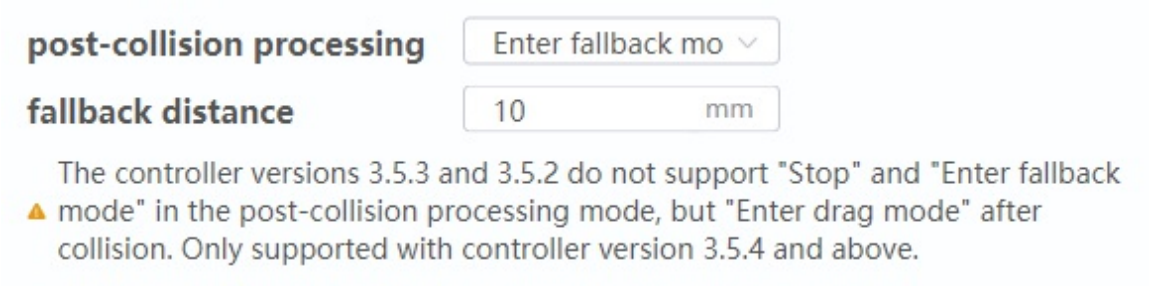


When the force required to stop is detected when you jog the robot, the "Collision Detection" window will pop up. In this case, you need to resolve the cause of the collision and click **Reset**. If you need to operate the software to resolve the collision cause, click **Remind me in a minute** to temporarily close the pop-up window (a pop-up message will be displayed again in one minute).



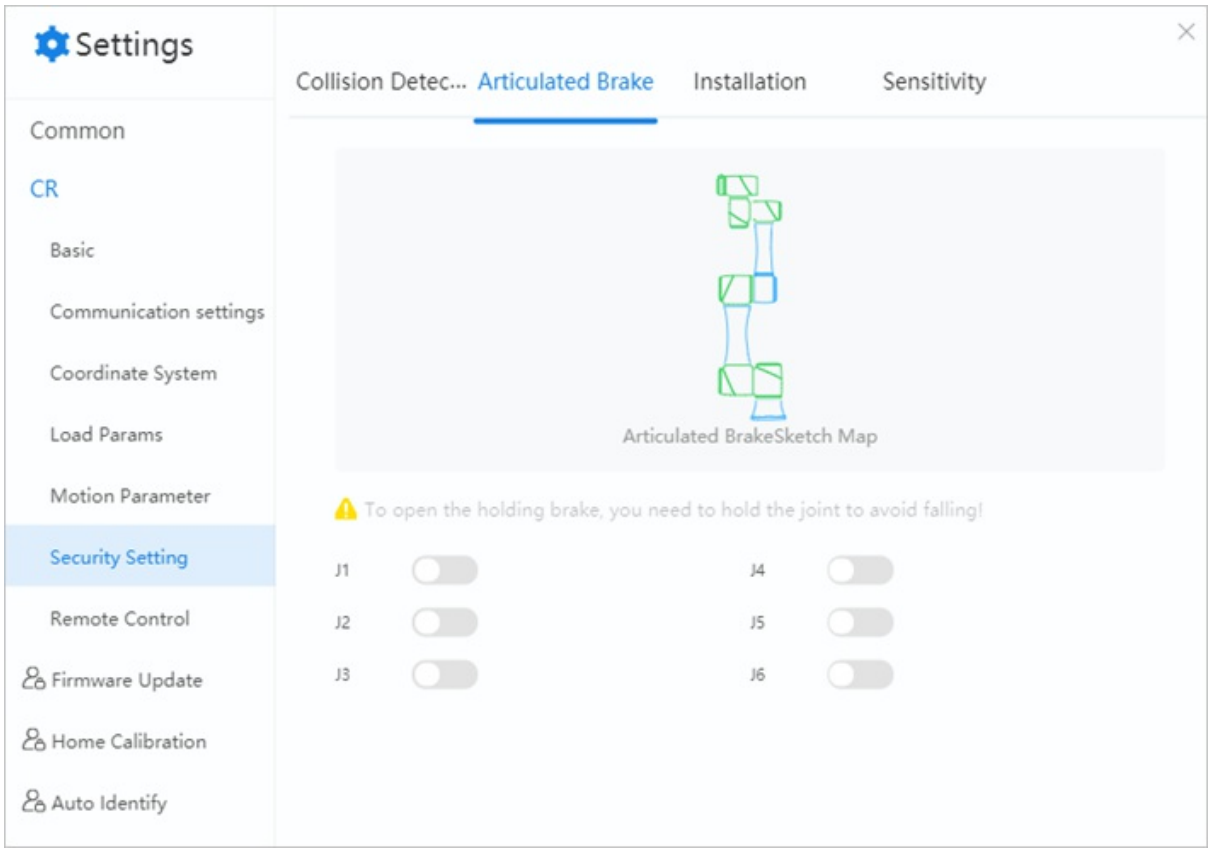
Post-collision processing refers to the treatment after collision while the robot arm is running the project:

- Stop: The robot arm stops running the project.
- Pause: The robot arm pauses. You need to select whether to resume the operation after solving the cause of the collision according to the actual condition, or stop the operation.
- Enter drag mode: The robot arm stops running the project and automatically enters drag mode.
- Enter fallback mode: The robot arm automatically back off the specified distance according to the trajectory before the collision. The range of the back-off distance is 0~50mm.



4.6.2 Joint brake

Braking prevents the servo motor shaft from moving when the servo driver is not in operation, so that the motor keeps its position locked and ensures that the moving part of the machine will not move because of its self weight or external force.

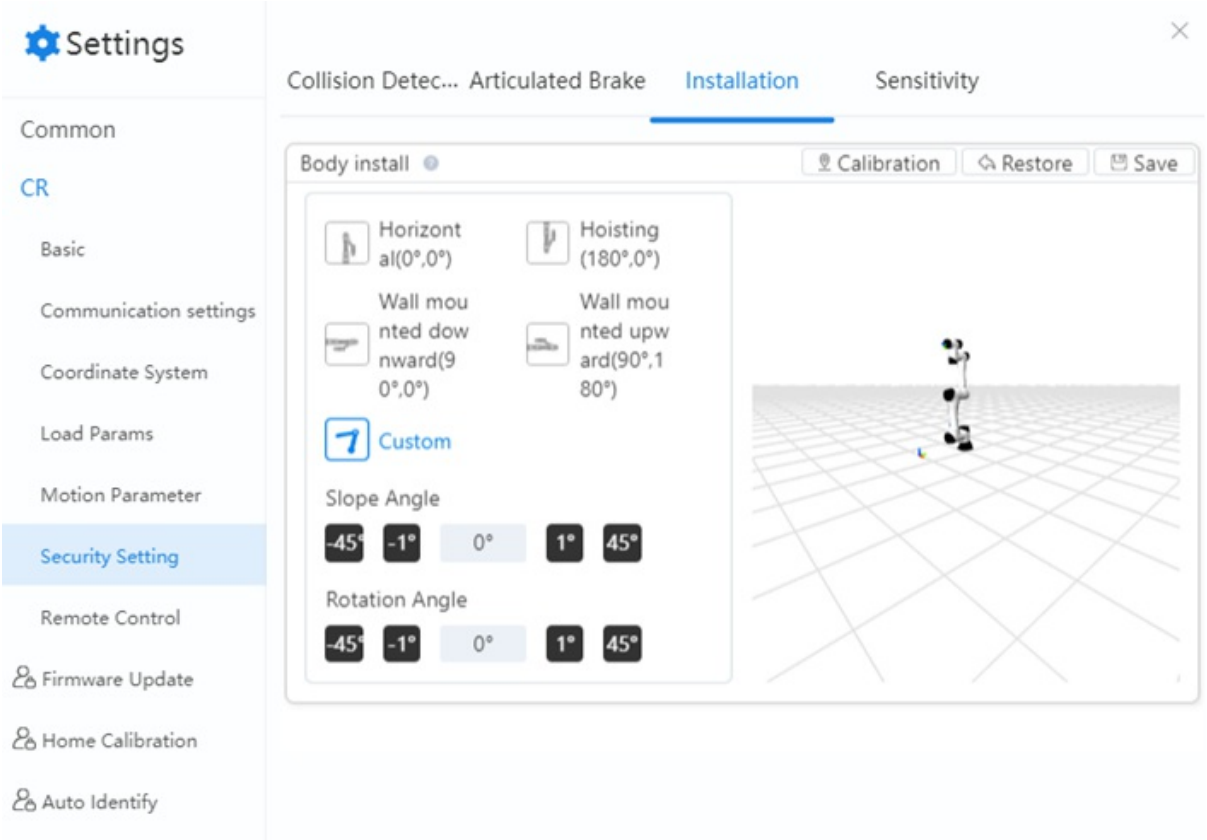


If you want to drag joints manually, you can enable the brake function, that is, hold the joint manually after the robot arm is disabled, then click the button of the corresponding joint.

> ⚠ NOTICE
> When enabling the function, hold the joint with your hand to prevent it from moving.

4.6.3 Installation

If the robot is mounted on a flat table or floor, you do not need to set in this page. However, if the robot is ceiling mounted, wall mounted or mounted at an angle, you need to set the rotation angle and slop angle in the disabled status.



You can calibrate via **Manual calibration** or **Automatic calibration**.

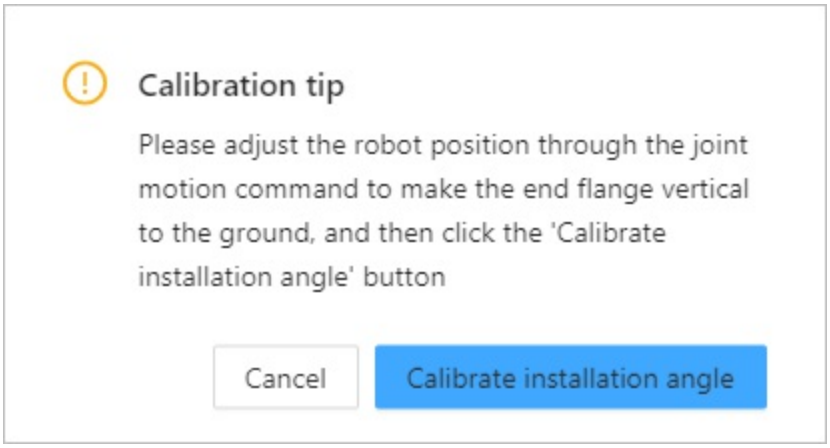
Manual calibration

You can select a proper installation posture based on the actual condition.

- Slop angle is the angle that a robot rotates counterclockwise around X-axis at the origin point.
- Rotation angle is the angle that a robot rotates counterclockwise around Z-axis at the origin point.

Automatic calibration

After installing and enabling the robot, you can click **Calibration** and operate according to the pop-up window to obtain the slop angle and rotation angle.

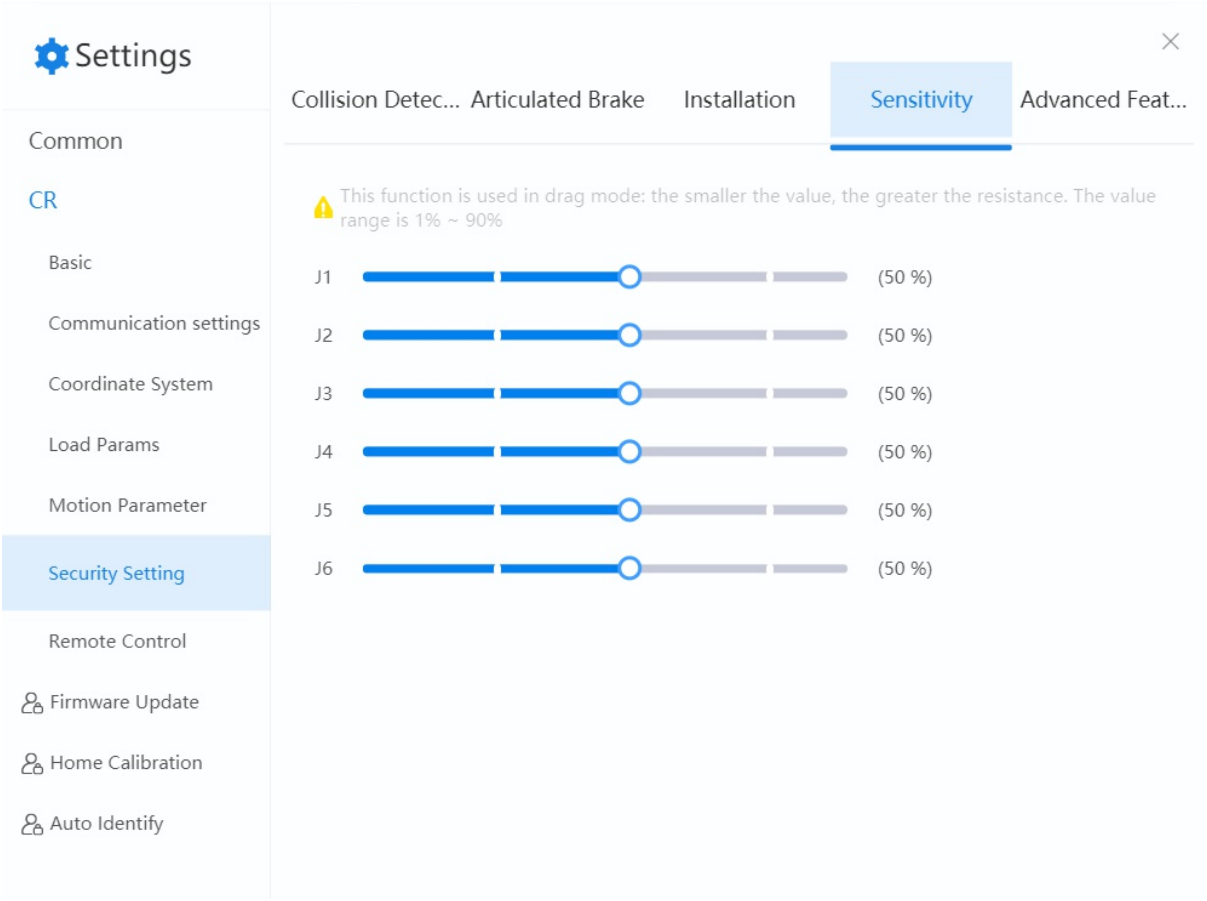


Click **Save** after calibration to save the settings.

Click **Restore** to restore the calibrated angle to the default value.

4.6.4 Sensitivity

Sensitivity setting is mainly used to adjust the sensitivity of joints during running and dragging.

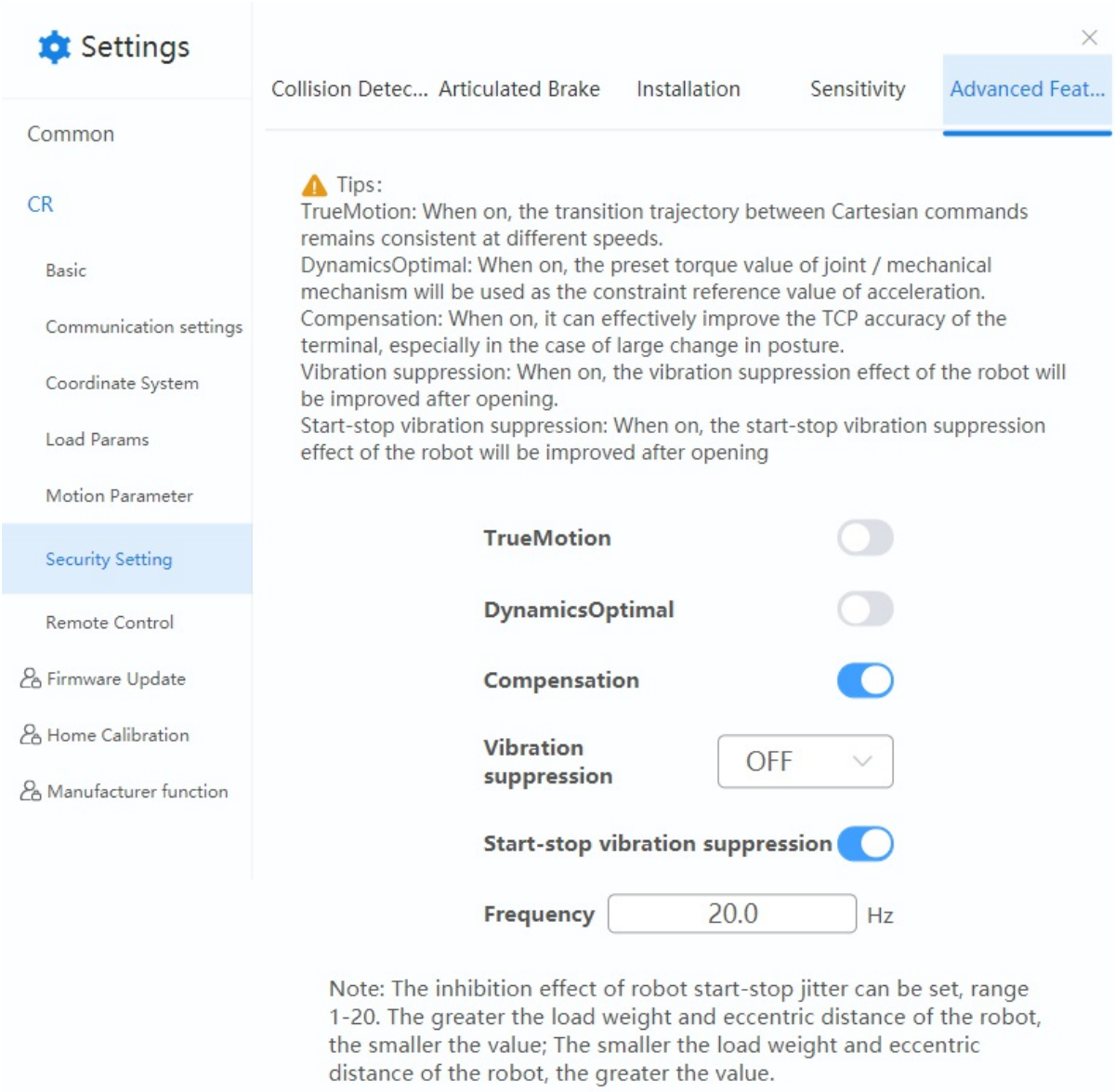


The lower the sensitivity is, the greater resistance there is during dragging.

4.6.5 Advanced functions

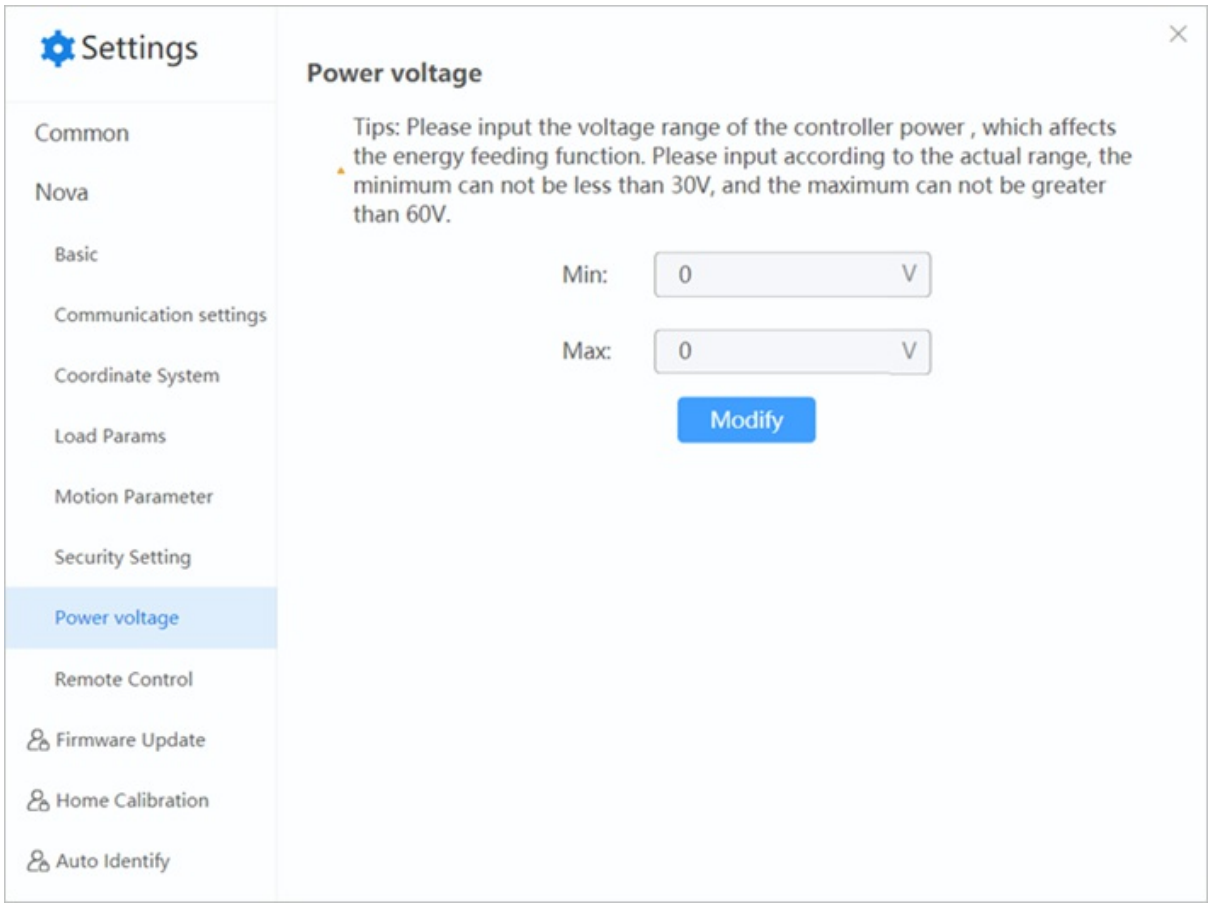
If you need to enable or disable the advanced functions, you can configure them in this page. It is recommended to keep the default value if you do not have special requirements.

> NOTE
>
> The function requires manager authority (default password: 888888).



4.7 Poewer voltage (CCBOX)

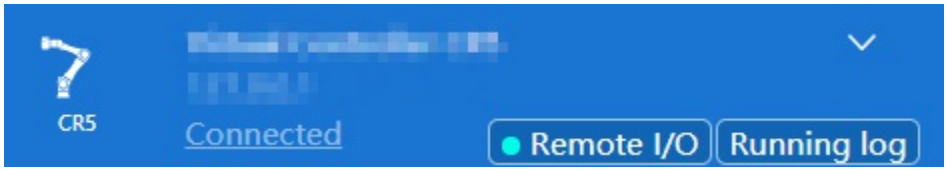
When the type of the controller is CCBOX, you need to set the input voltage range of the controller.



Please set according to the actual range of input voltage, with maximum voltage no more than 60V and minimum voltage no less than 30V.

4.8 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.

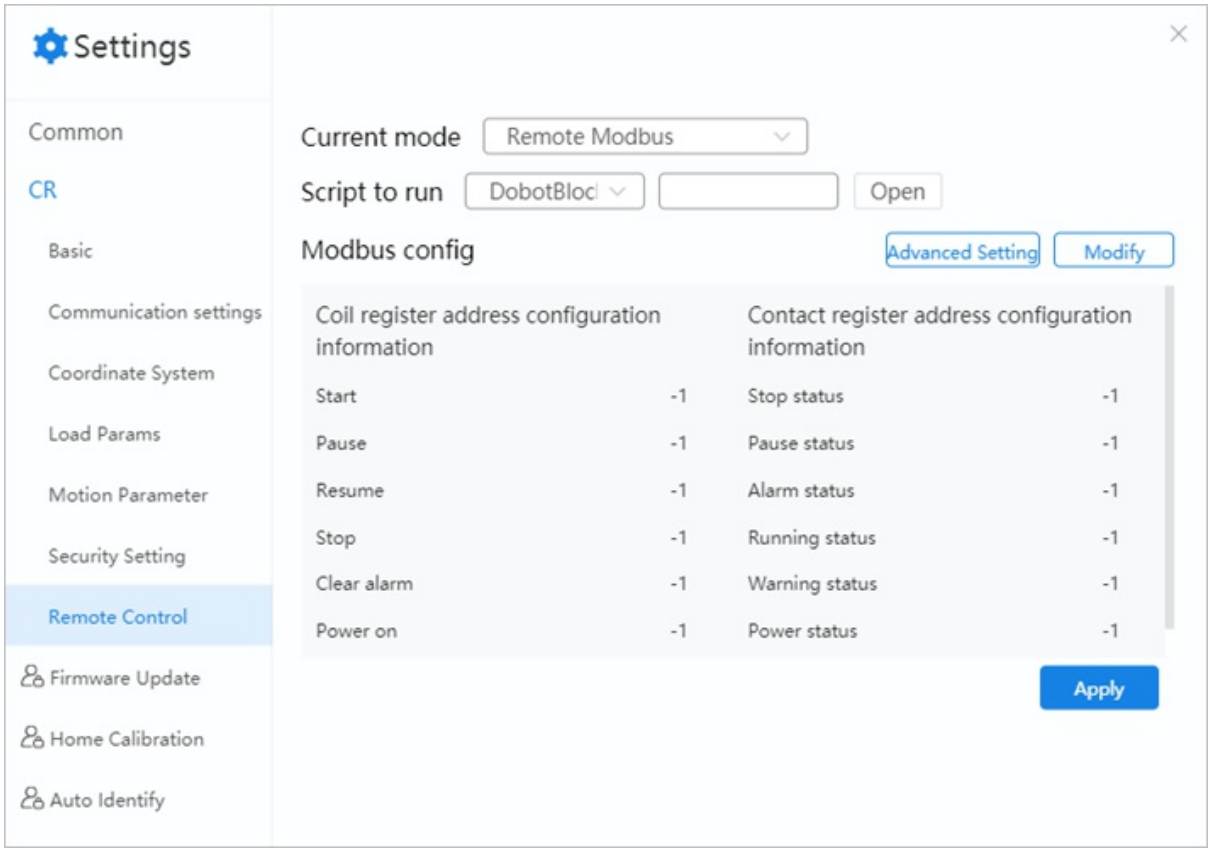
> NOTE
>
> - You do not need to restart the robot control system when switching remote control mode.
> - No matter what mode the robot control system is in, the emergency stop switch is always effective.
> - If the robot is running in the remote control mode, the project will stop running automatically when you switch to other working modes.
> - Do not send control signals before the robot is powered on and initialized, otherwise it may cause the robot to move abnormally.
> - After entering the remote control mode, the top toolbar will display the current mode. Click "Running log" to view the running log of the remote mode. The following figure takes the Remote I/O mode as an example.
>
> 
>

Online mode

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

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.

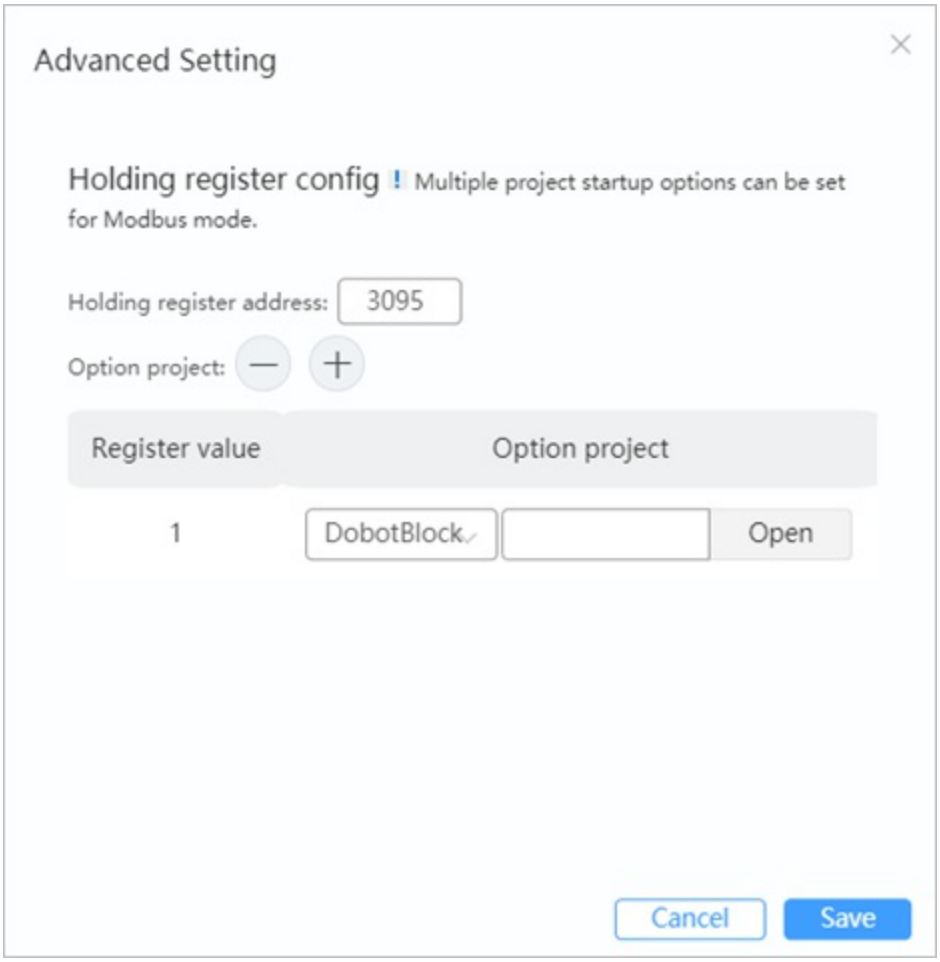
Prerequisites

- The project to be running in the remote mode has been prepared.
- The robot has been connected to the external equipment through the LAN 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 192.168.5.1. 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.

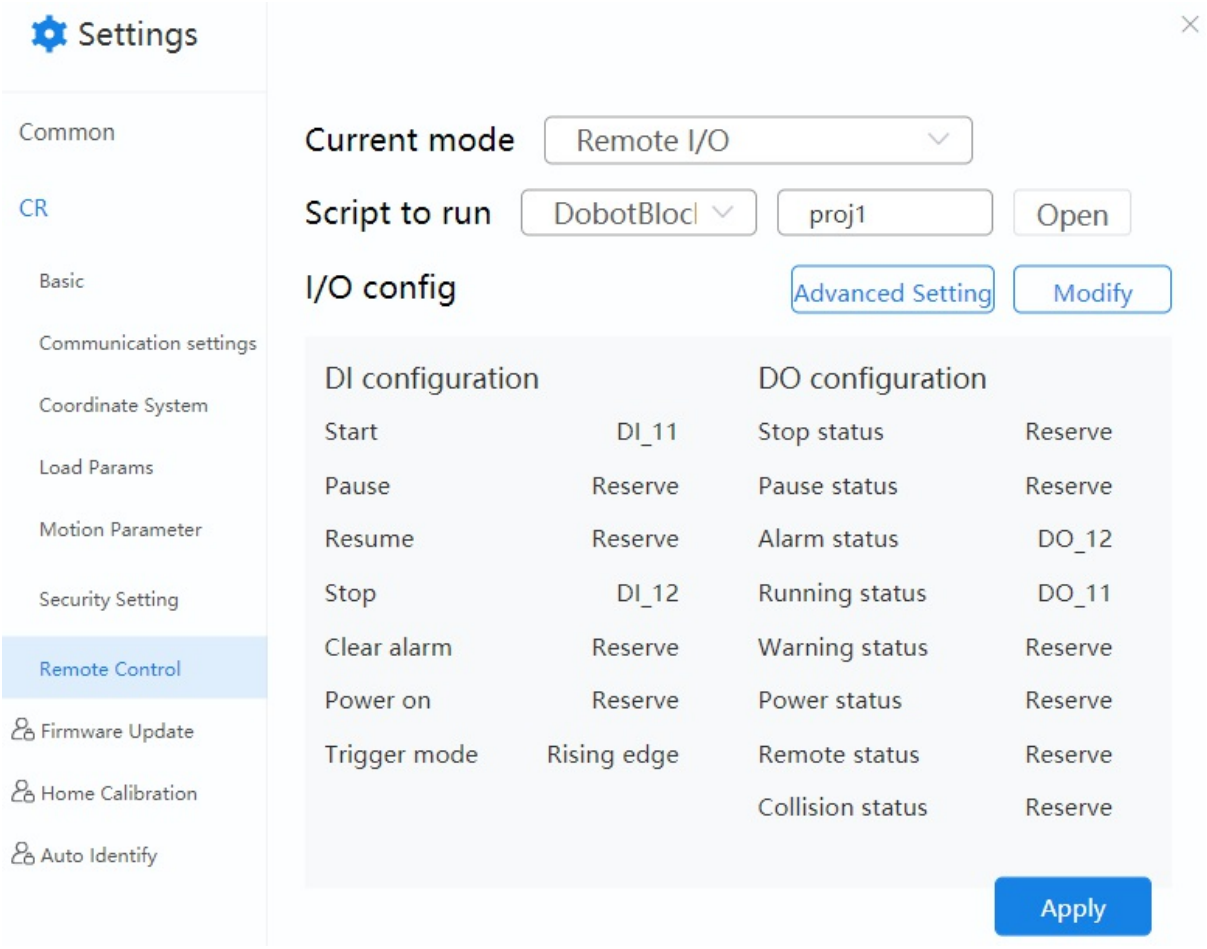
2. If you need to start multiple different projects through Modbus, click **Advanced Setting**. In Advanced Settings, you can set **Hold register address** of the option project and configure the list of option projects, as shown in the following figure.



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.

Remote IO

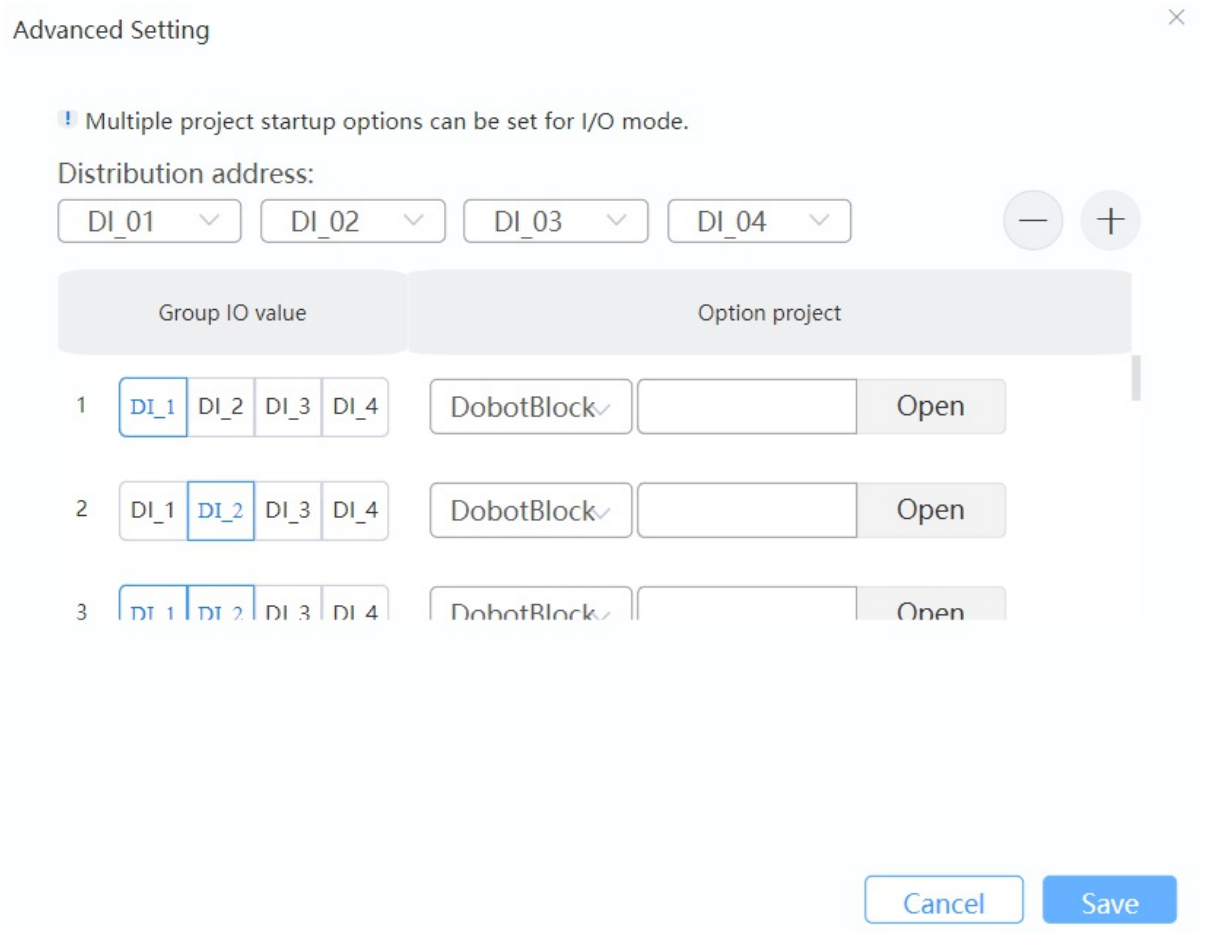
External equipment can control the robot arm in the remote IO mode.



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

When the type of control cabinet is CCBOX, safe I/O and universal I/O share the same terminal. The terminal configured as safe I/O cannot be configured as remote I/O.

Clicking **Advanced Setting** can configure multiple projects through group I/O.



1. Click + or - to increase or decrease the number of addresses assigned to group I/O. The more the assigned addresses are, the more configurable projects are.

   0 address: Optional projects cannot be configured.
   1 address: 1 optional project can be configured.
   2 addresses: 3 optional projects can be configured.
   3 addresses: 7 optional projects can be configured.
   4 addresses: 15 optional projects can be configured.

2. You can modify the assigned address through the drop-down list. The address assigned to group I/O cannot be duplicated with remote I/O or safe I/O (CCBOX).

3. After assigning addresses, you can set optional projects (at least one) for each group IO value.
   - In remote I/O mode, before running the project, select the corresponding optional project by setting the corresponding (blue refers to ON, and gray refers to OFF) group IO value.
   - Take assigning DI1~ DI4 in the figure above as an example:
     - DI1 is ON, DI2, DI3 and DI4 are OFF: select the first optional project;
     - DI1 and DI2 are ON, DI3 and DI4 are OFF: select the third optional project;
     - DI1~ DI4 are all ON: select the 15th optional project.
   - When all group IOs are set to OFF, it means selecting the main project configured in the previous page.
4. Click **Save** to complete the configuration.

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. (Optional) Trigger group I/O on external devices and select a project for running.
4. Trigger the starting signal. The robot will move according to the selected project file.
5. If the stop signal is triggered, the robot arm will stop moving.

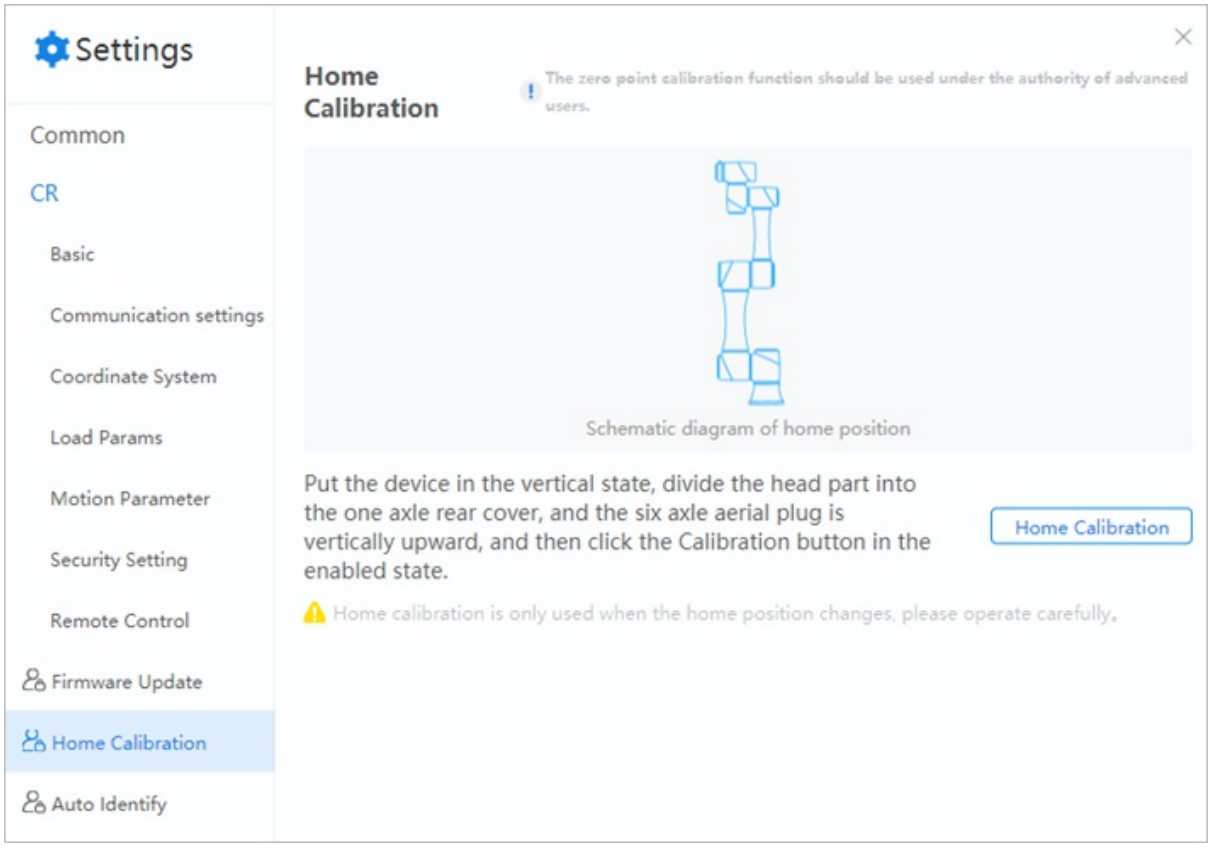
TCP/IP secondary development

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)

4.9 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 password (default password: 888888)



Move the robot to the home point according to the prompt in the page. Click **Home Calibration** in the enabled status.

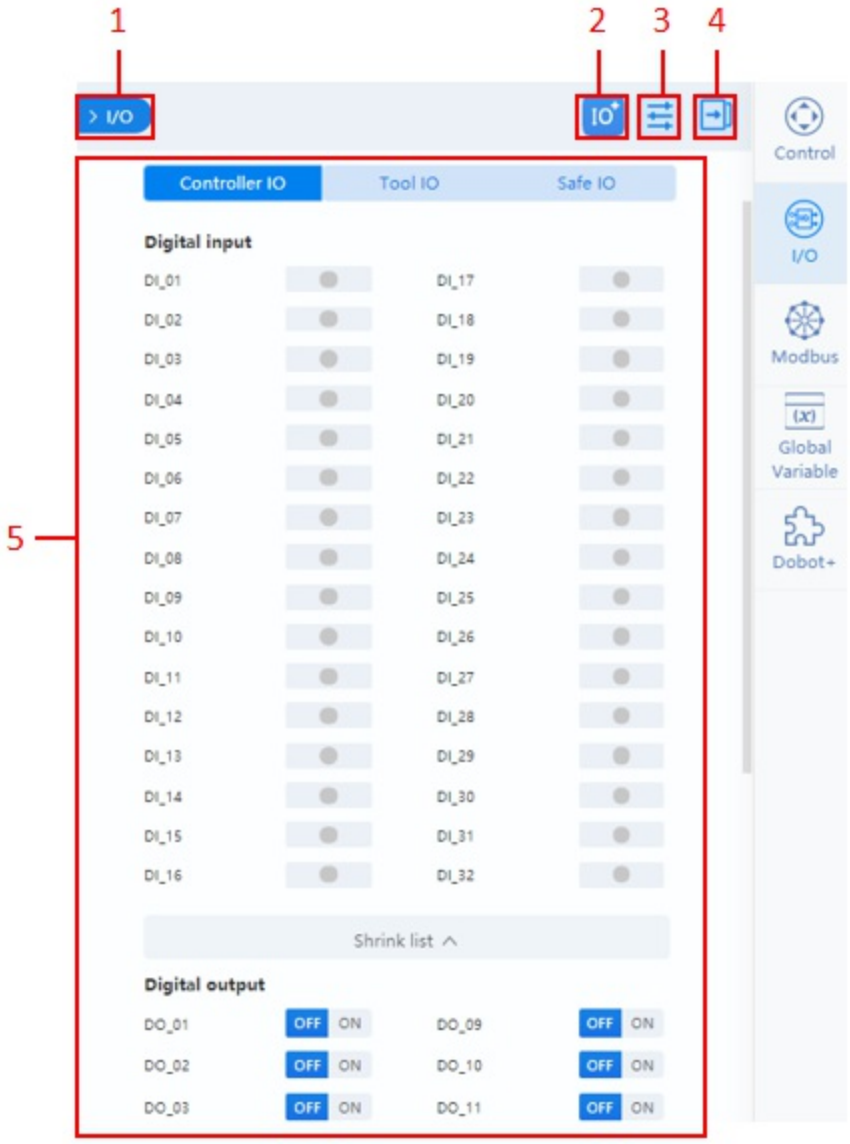
After operating the homing calibration, you can check the joint coordinates in the control panel. Now all joint coordinates (J1-J6) are zero.

4.10 Manufacturer function

Manufacturer functions include automatic identification and collision detection switch. Improper operation may cause robot anomaly and security risks. Please use it under the guidance of technical support.

5 I/O Monitoring

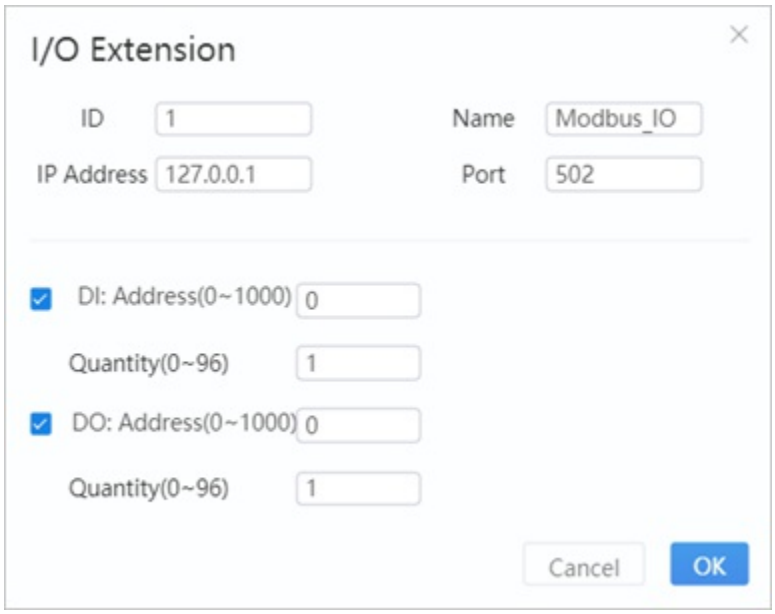
You can monitor and set the I/O status of the controller and the end tool in the I/O page. For the I/O definition, refer to the IO description in the corresponding robot hardware guide. As different controllers vary in the number of I/O, the screenshots in this document are for reference only.



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

I/O extension



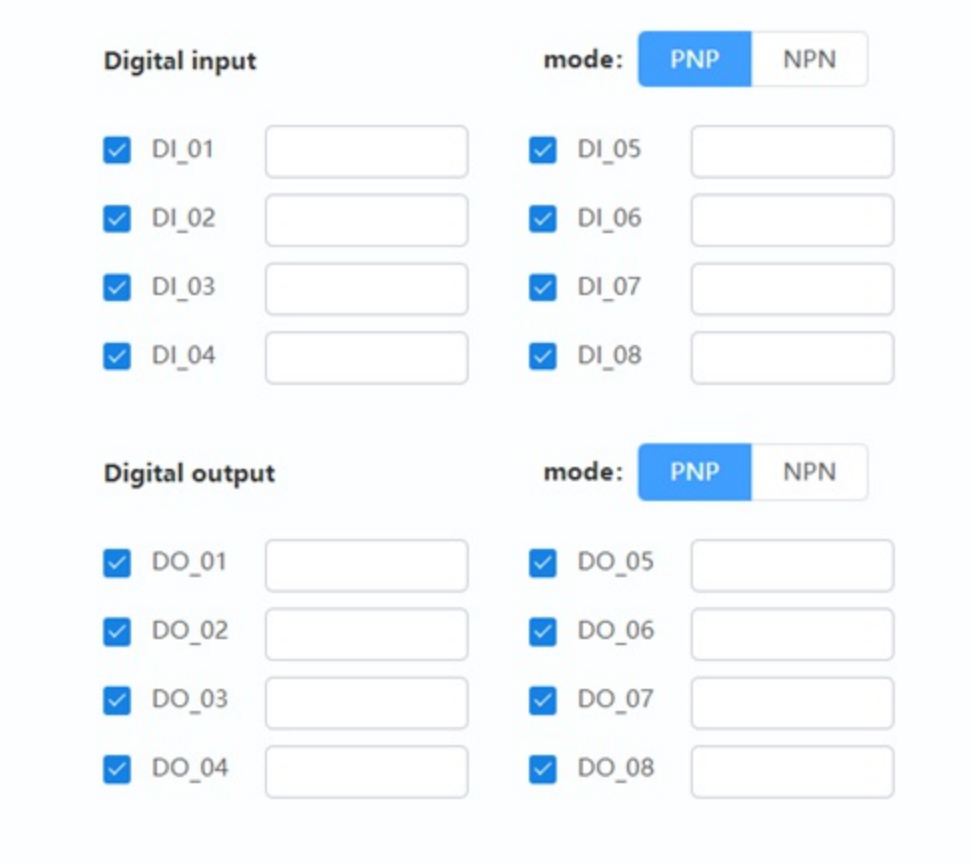
- ID: Slave device ID.
- Name: Name of the slave device.
- IP address: Address of the Modbus device. It cannot be the same as the IP address of the controller LAN interface (for example, the default 192.168.5.1), otherwise it may cause functional abnormalities.
- Port: Port number of Modbus communication.
- DI/DO: Configure the register address and number of DI/DO after selecting the function. The address should be assigned from 0.

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 extended I/O can delete the I/O.

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.
- When the type of control cabinet is CCBOX, you can set digital input/output type to PNP (high-level active) or NPN (low-level active), as shown in the figure below.



Monitoring

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

Output: Set the digital output or analog output. For digital output, you can click the corresponding switch on the right side to switch its status. For analog output, you need to click **Modify** first, and then click **Confirm Modification** after modifying the output value.

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.

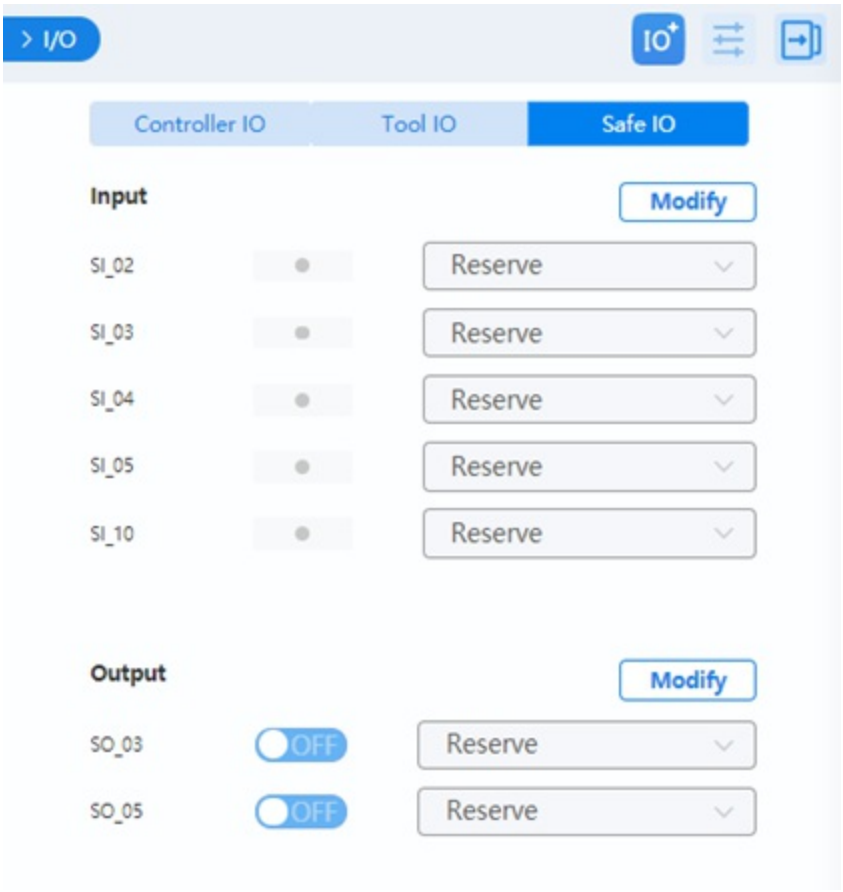
Simulation: Simulate the status of digital input to facilitate debugging and running programs. Click the status display area of the corresponding DI, and a setting window will pop up. Click **Fictitious** and select **DI Transformation**, and the DI will turn to virtual trigger status (green dot), which is regarded as ON logically. If you do not select **DI Transformation**, the DI will maintain its real status.

When the type of control cabinet is CC162, the analog input/output supports voltage or current mode. Click **Modify** to select the corresponding mode, and click **Confirm modification** to switch the mode. The unit of the value on the right side will change in real time. Note that the value of the analog input can only be viewed rather than modified.

> NOTE
> This function needs to be used with the DIP switch, which is located inside the control cabinet. if you need to set it, please contact Dobot technical support.

You can set the functions of each safe I/O interface in the Safe IO page. For details on the safe I/O, refer to the IO description in the corresponding robot hardware guide.

When the type of control cabinet is CCBOX, safe I/O and universal I/O share the same terminal. The terminal configured as remote IO cannot be configured as safe I/O.



NOTE

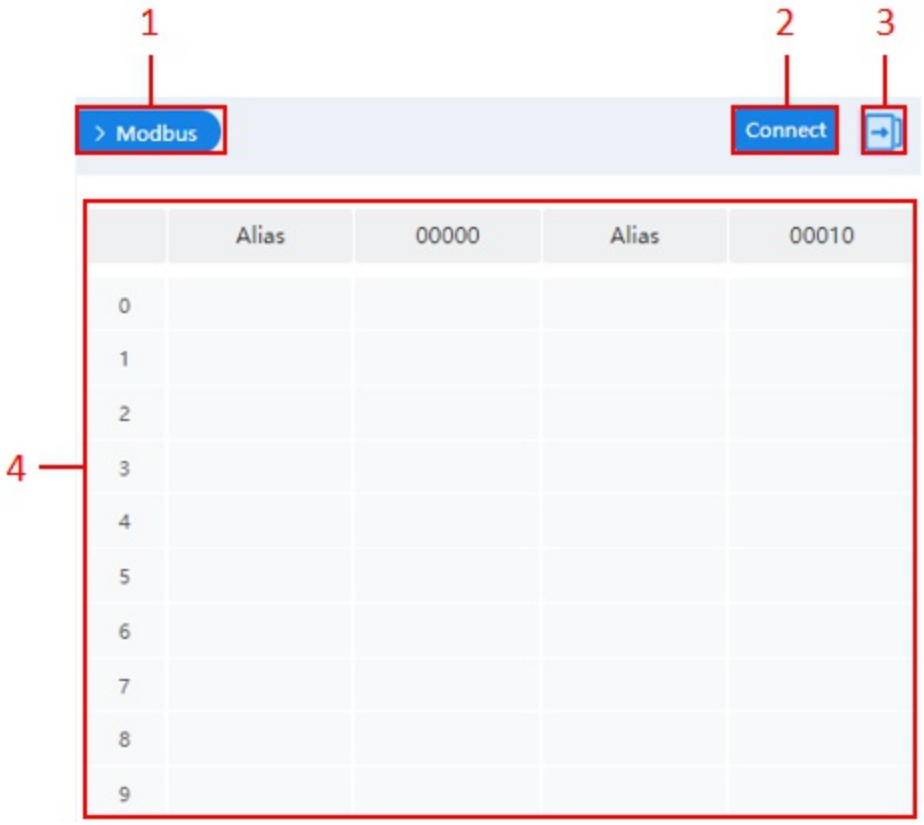
In actual use, please try to ensure that SI signal changes at an interval of over 150ms, otherwise SI signal jump may cause anomaly in robot running status.

For example, when the protective stop reset input is not configured, the jump of the protective stop input signal (the change interval is less than 150ms) may cause the robot not to automatically resume operation after being paused. In this case, you need to solve the problem through the method below.

1. Pause the project through the software, and then continue to run the project.
2. Trigger the protective stop input signal again and remain over 150ms.

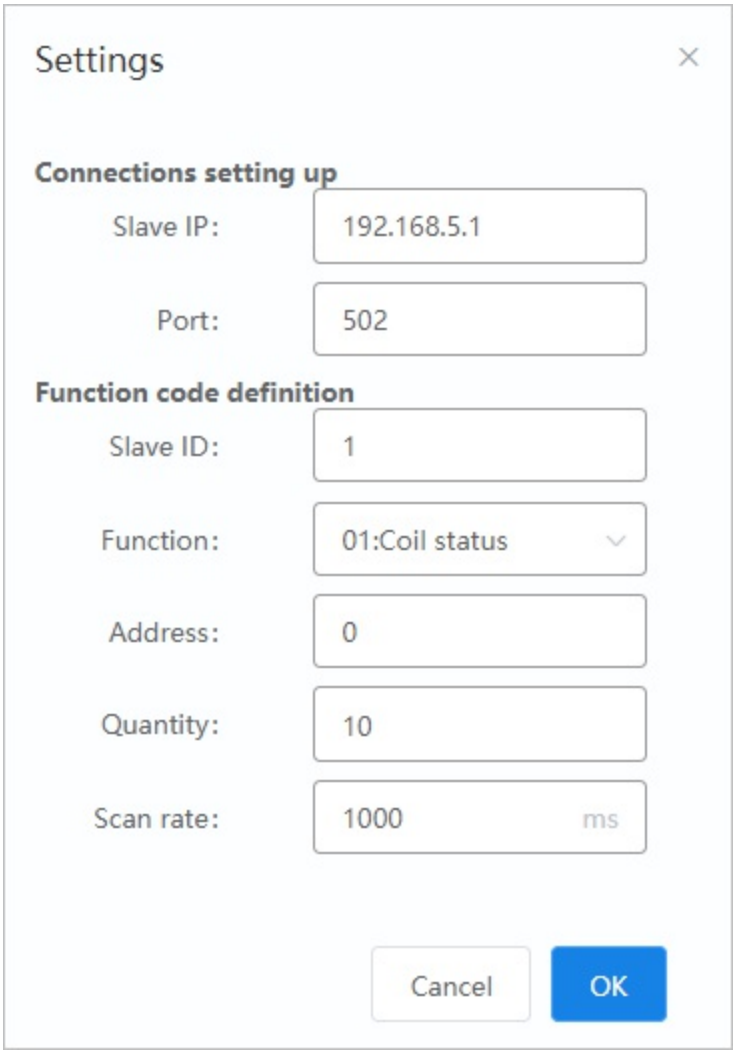
6 Modbus

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 restore it
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

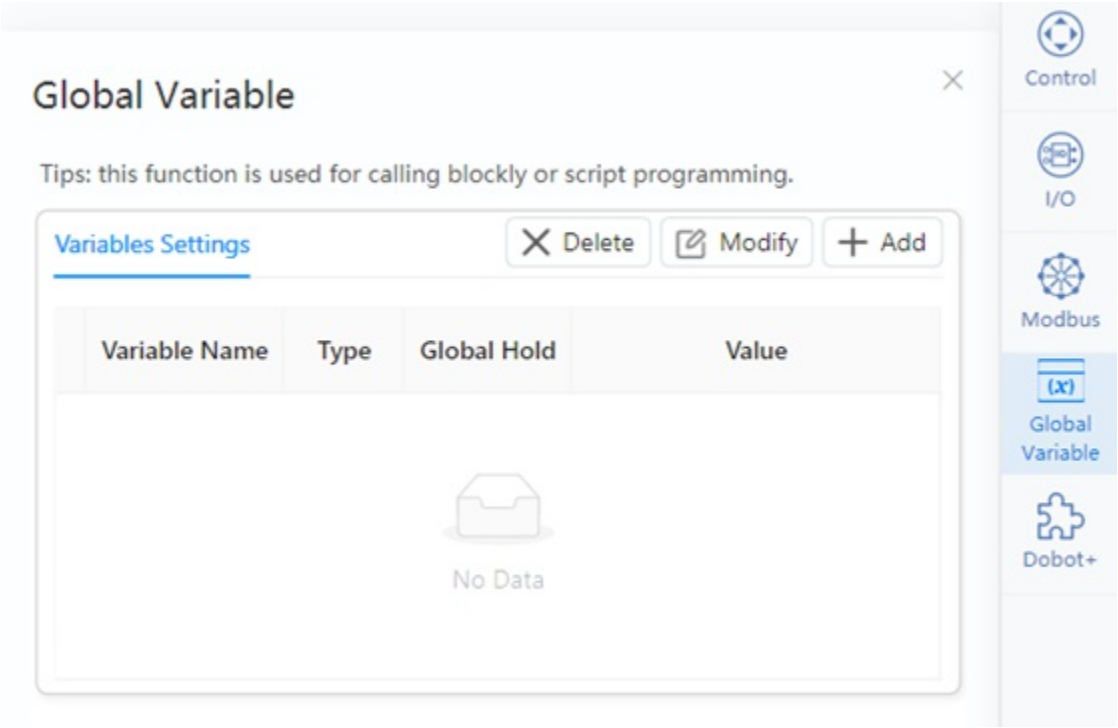


- Slave IP: address of Modbus device.
- Port: port number of Modbus communication.
- Slave ID: slave device ID.
- Function: select the function type of the slave device.
- Address/Quantity: address and number of registers.
- Scanning rate: time interval of scanning the slave station by the robot arm.

7 Global Variable

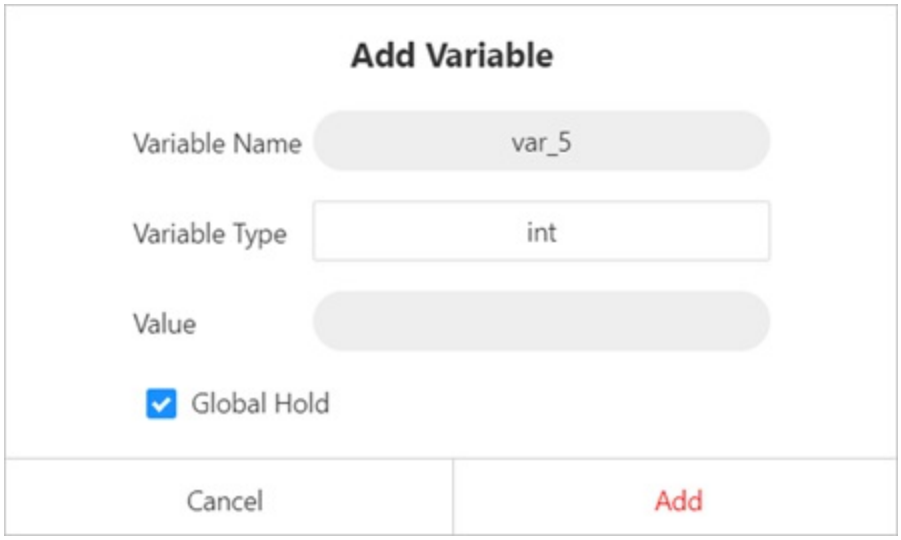
The module is used to configure and check the global variables.

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.



Global Hold:

- When **Global Hold** is checked, any modification to the variable will be saved, and the modified value will be saved regardless of exiting the script or powering off and restarting the robot.
- When **Global Hold** is not checked, modifications to the variable are only effective when the script is running. It will be restored to the initial setting value when exiting the script.

Naming recommendations for global variable

It is recommended to use the format of `g_project abbreviation_variable meaning` to name global variables and avoid using names that are too short.

It should be noted that global variable names cannot duplicate lua's reserved keywords and the functions and variable names already defined by Dobot.

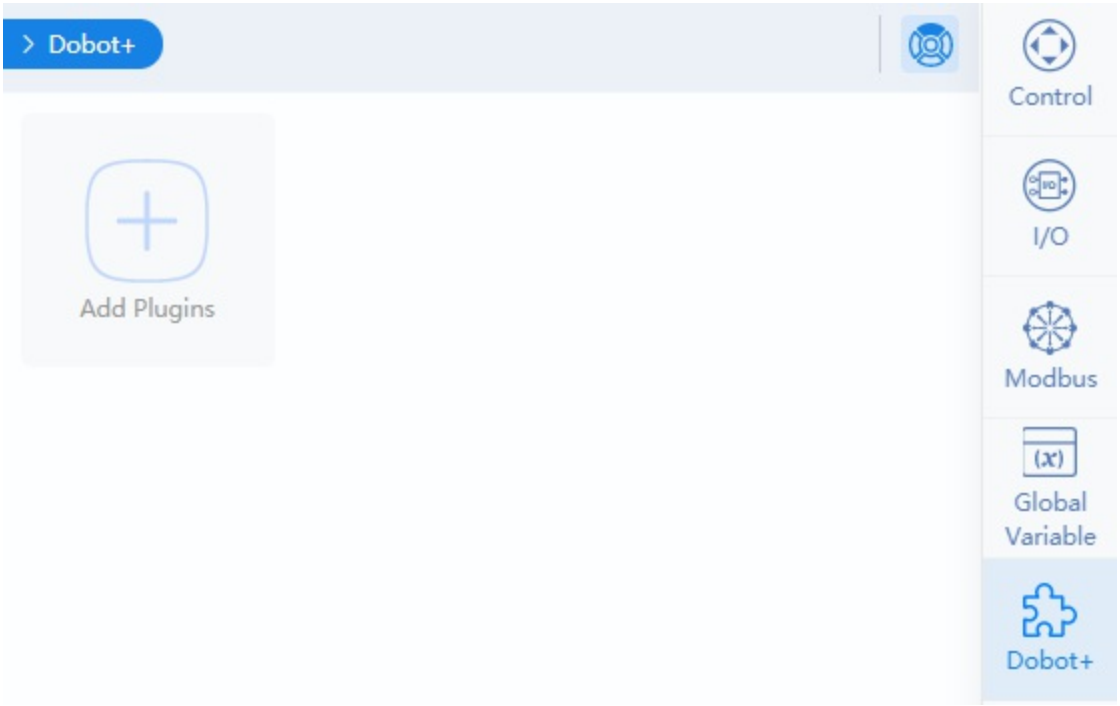
```
-- lua's reserved keywords
and, break, do, else, elseif, end, false, for,
function, if, in, local, nil, not, or, repeat,
return, then, true, until, while, goto

-- Dobot's defined functions and variables (including internally reserved functions)
Accel, AccelR, AccelS, AI, AO, AOExecute, Arc3,
Arc3Weave, Arch, BlockHighlight, CalcTool, CalcUser,
CheckGo, CheckMove, Circle3, CnvVison, CollisionLogInfo,
CP, CRC16, CyHelix, DestroyCam, DI, DIGroup, DO,
dobot_hook, DOExecute, DOGroup, ECP, ECPSet, ElapsedTime,
EventLog, FC, FCAct, FCCalib, FCCondDisplac, FCCondDisplace,
FCCondForce, FCCondTCPSpeed, FCCondTorque, FCCondTorque,
FCDeact, FCForce, FCLine, FCLoadID, FCRefDirect, FCRefForce,
FCRefLine, FCRefRot, FCRefSpiral, FCRefTorAdjust, FCRefTorque,
FCRel, FCRot, FCSetCompliance, FCSetGain, FCSpiral, FCTorque,
GetABZ, GetAngle, GetCnvObject, GetCoils, GetHoldRegs,
GetInBits, GetInRegs, GetLayerIndex, GetPartIndex, getPathPose,
GetPathStartPose, GetPose, GetRunningData, GetTraceStartPose,
Go, GoIO, GoR, GoToolR, GoUserR, GRead, GWrite, IceCreamGO,
IceCreamP, IceCreamR, init(pix), InitCam, InverseSolution,IsDone,
```

```
Jerk, JerkR, JerkS, Jump, LimZ, Linear, Linear, LoadSet, LoadSwitch,
Log, LogFormat, MatrixPallet, MD, ModbusClose, ModbusCreate,
Move, MoveIn, MoveIO, MoveJ, MoveJIO, MoveJR, MoveOut, MoveR,
MoveToolR, MoveUserR, MoveWeave, NewPose, OFF, ON, PalletSync,
Pause, print, QuitScript, ReadDB, RecvCam, Release, RelJointMovJ,
RelMovJTool, RelMovJUser , RelMovLTool, RelMovLUser, Reset,
ResetElapsedTime, RJ, Rotation, RP, ScrewCommitErrInfo,
ScrewCommitInfo, ScrewCommitProductInfo, ScrewDriverGetError,
ScrewDriverGetResult, ScrewGetSelectedSN, ScrewSetCurrentSN,
SendCam, ServoJ, ServoP, SetABZPPC, SetBackDistance,
SetCnvPointOffset, SetCnvTimeCompensation, SetCoils, SetCollisionLevel,
SetDOMode, setExcitMode, SetExicitCmd, SetExicitCmdZ, SetHoldRegs,
SetObstacleAvoid, SetPartIndex, SetSafeSkin, SetStackMode,
SetTool, SetTool485, SetToolBaudRate, SetToolPower, SetUnstackMode,
SetUser, SetWeldParams, SixForceHome, Sleep, Sleep(period), Speed,
SpeedFactor, SpeedR, SpeedS, Spiral, StartCamera, StartPath,
StartTrace, StopMove, StopSyncCnv, Sync, SyncCnv, SyncExit, Systime,
TCPCreate, TCPDestroy, TCPRead, TCPReadMultiVision, TCPReadVision,
TCPSpeed, TCPSpeedEnd, TCPStart, TCPWrite, TeachPallet, ToolAI,
ToolAnalogMode, ToolDI, ToolDO, ToolDOExecute, TriggerCam,
UDPCreate, UDPRead, UDPWrite, Wait
```

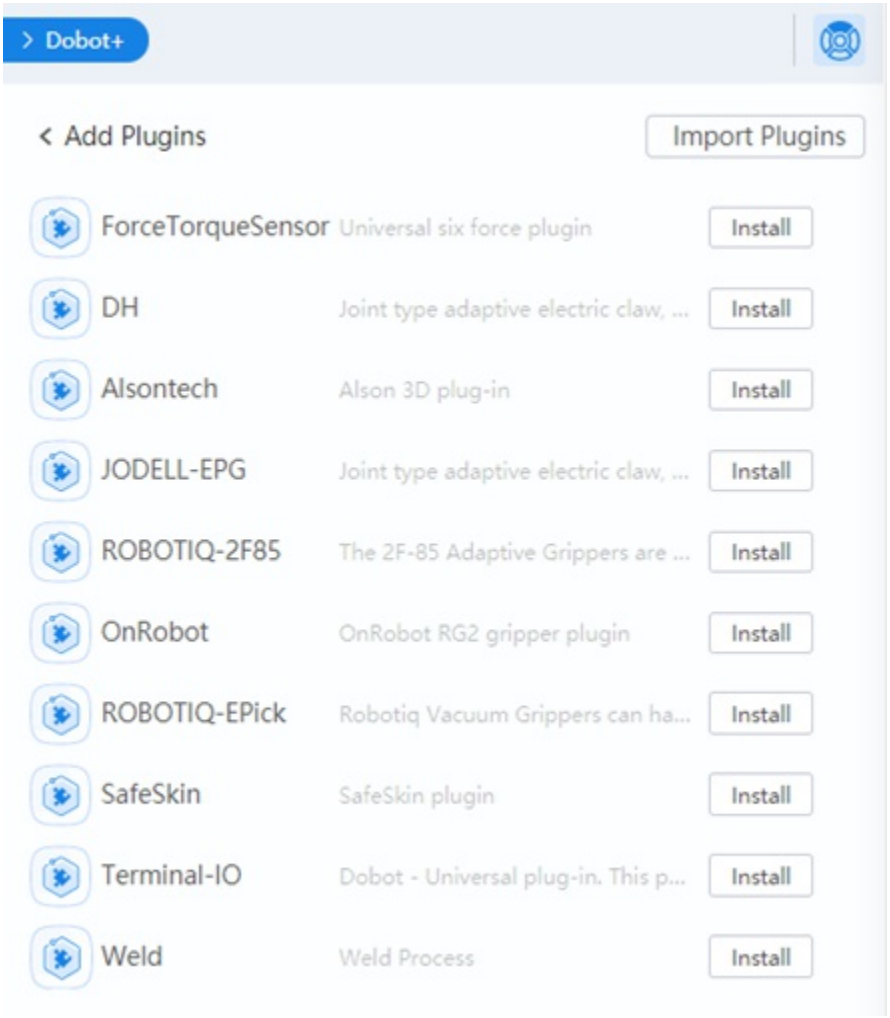
8 Dobot+

The page is used to install and configure the end plug-in for controlling the end tool of the robot.

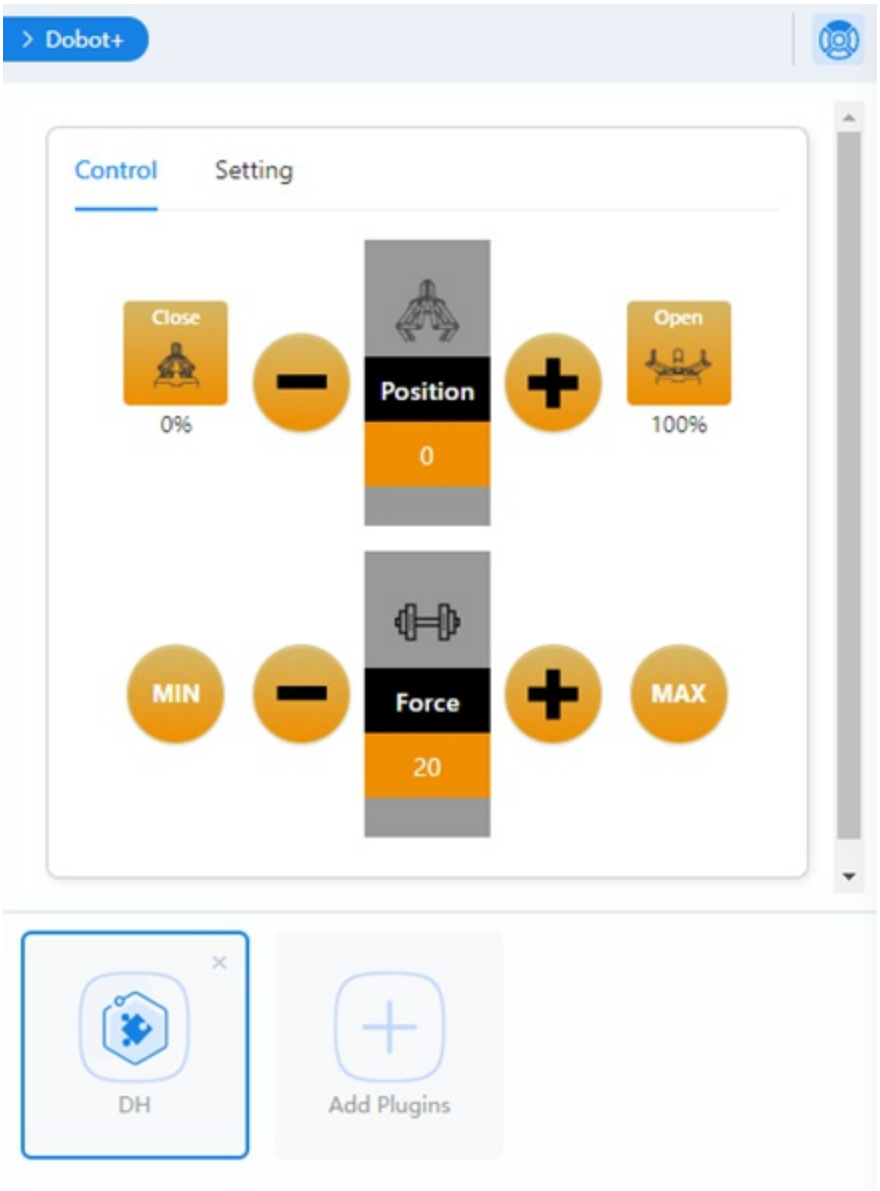
You can clicking **Dobot +** on the upper left of the panel to hide the panel, and restore the panel by clicking **Dobot +** on the right toolbar.



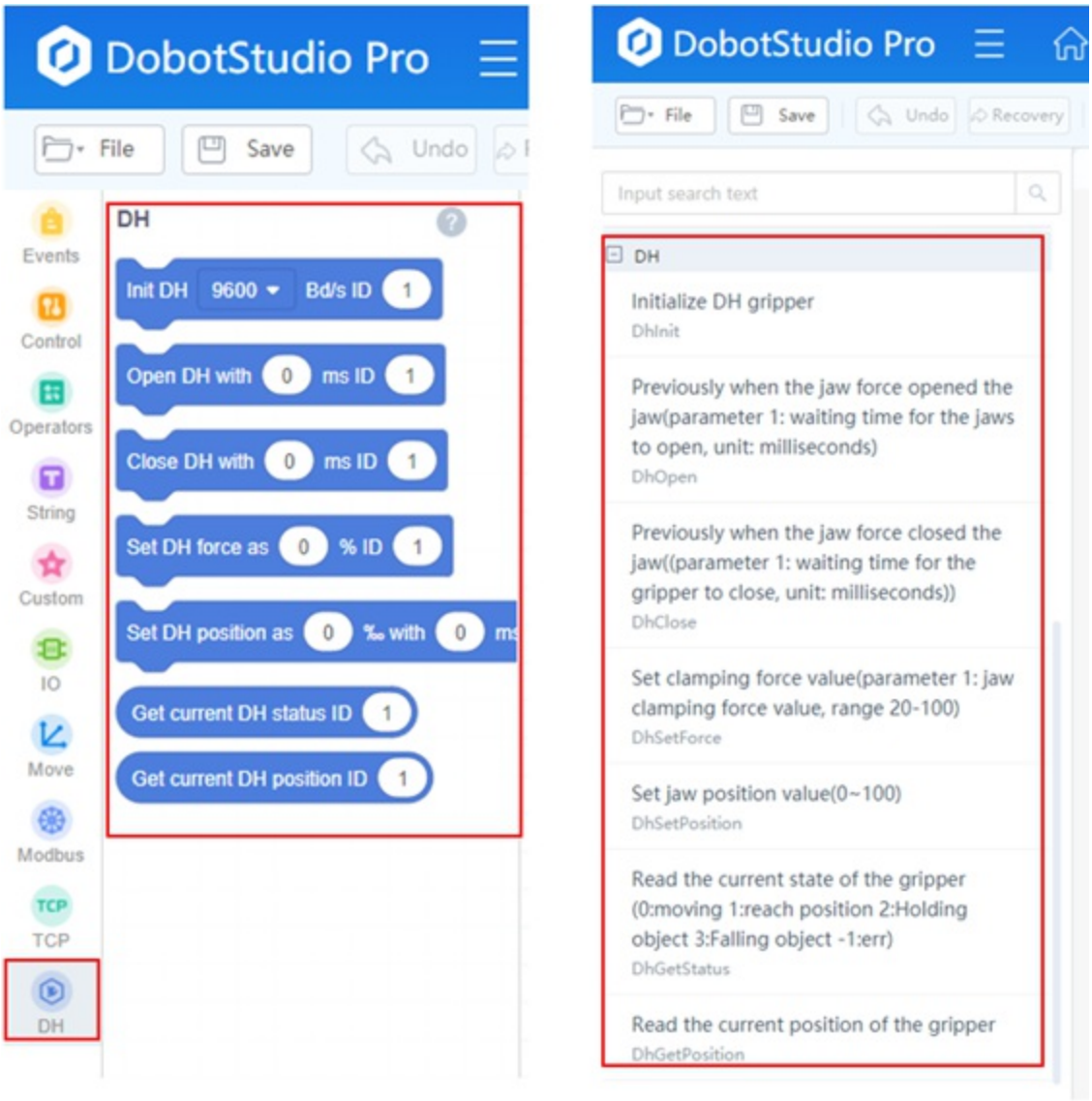
Click **Add Plugins** to view the plug-in list. You can click **Install** to install the plug-in, or click **Import Plugins** to upload a plug-in.



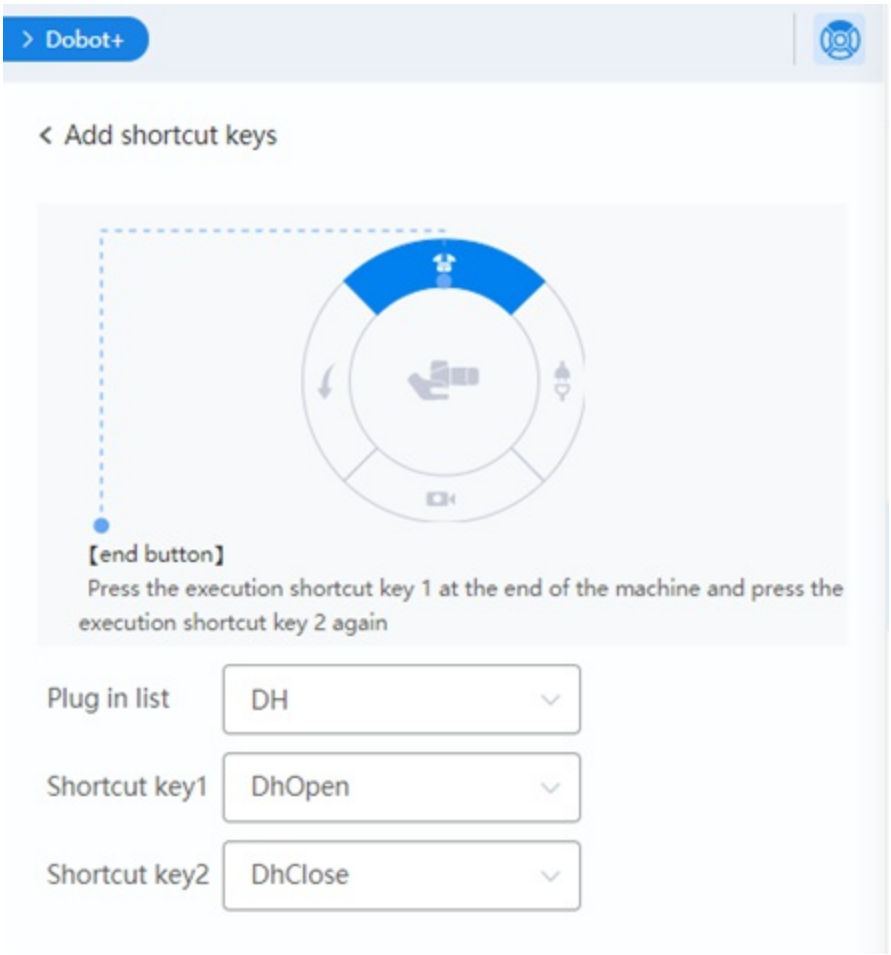
The plug-in will be displayed in the Dobot+ main page after installation. After selecting the corresponding plug-in, you can configure the basic functions of the plug-in. Different plug-ins vary in their configuration methods, which will not be described in this Guide. If you want to delete the plug-in, click **x** in the upper right corner of the plug-in.



After installing a plug-in, the relevant blocks/commands will be added in DobotBlockly and Script module. Now take DH gripper as an example, as shown in the figure below.



You can click  on the right-top corner of the page to set a hotkey for the plug-in. After saving it, you can control the end tool through the end button. For example, select **DH**. Set **Shortcut key1** to **DhOpen**, and **Shortcut key2** to **DhClose**, and click **Save**. Then press the button at the end of the robot, as shown below, and the gripper will open. Press the button again, and the gripper will close.

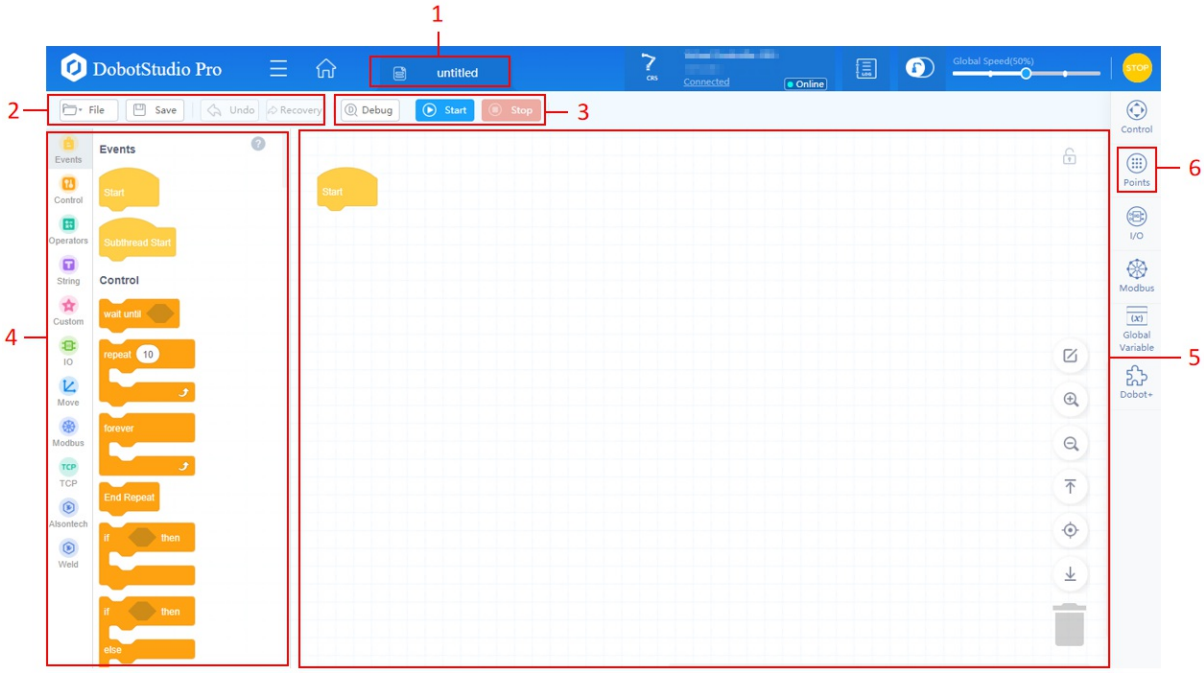

Dobot+
Add shortcut keys
【end button】
Press the execution shortcut key 1 at the end of the machine and press the execution shortcut key 2 again
Plug in list
DH
Shortcut key1
DhOpen
Shortcut key2
DhClose

9 Programming

9.1 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 Appendix B.



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 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

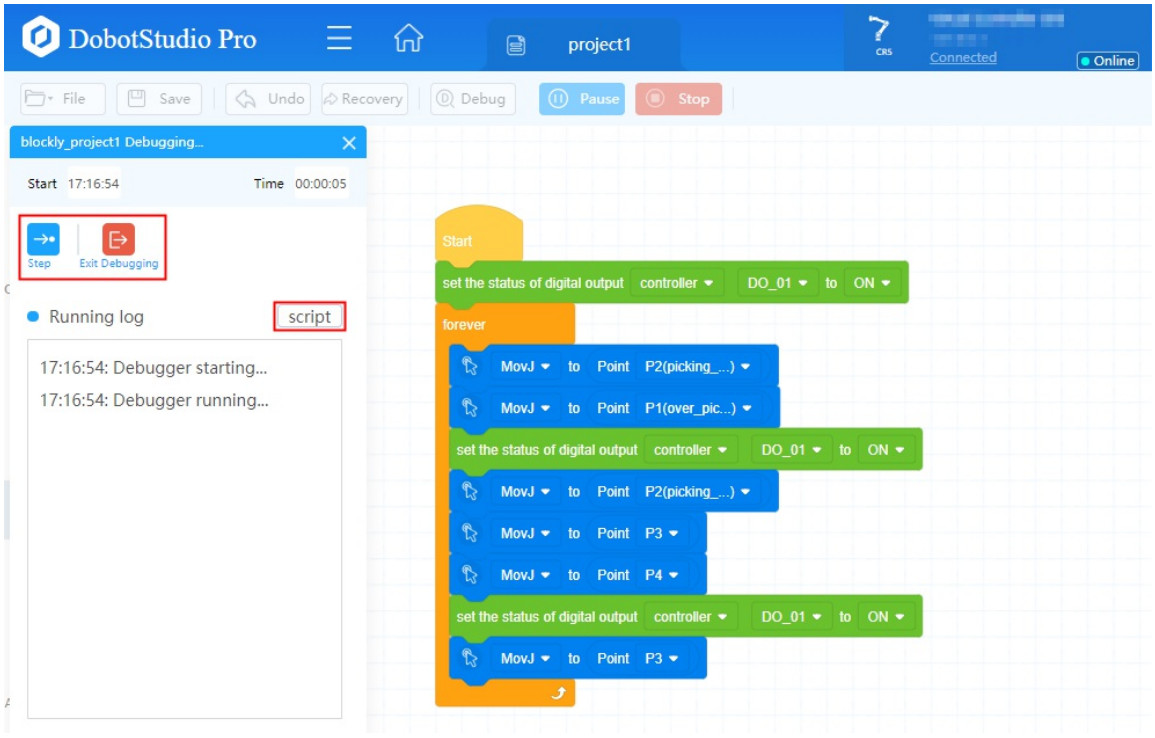
The Point interface is used to manage the points in programming, as shown below.

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 Points *.
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

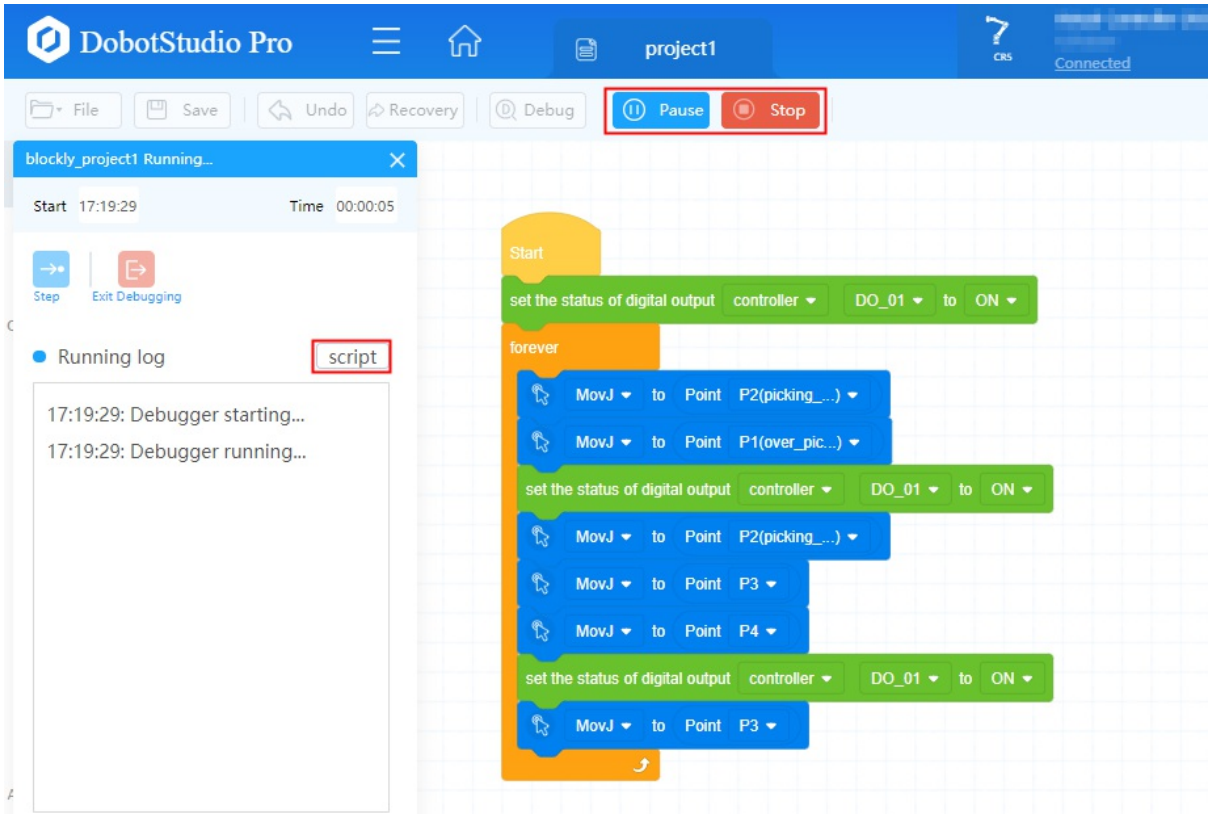
Click **Debug** after saving the project, and the project will start running step by step. You can view view the operation log in this process.

- 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.



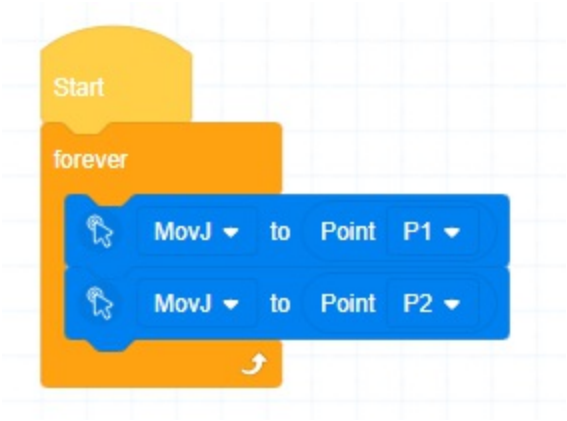
Operation procedure

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.

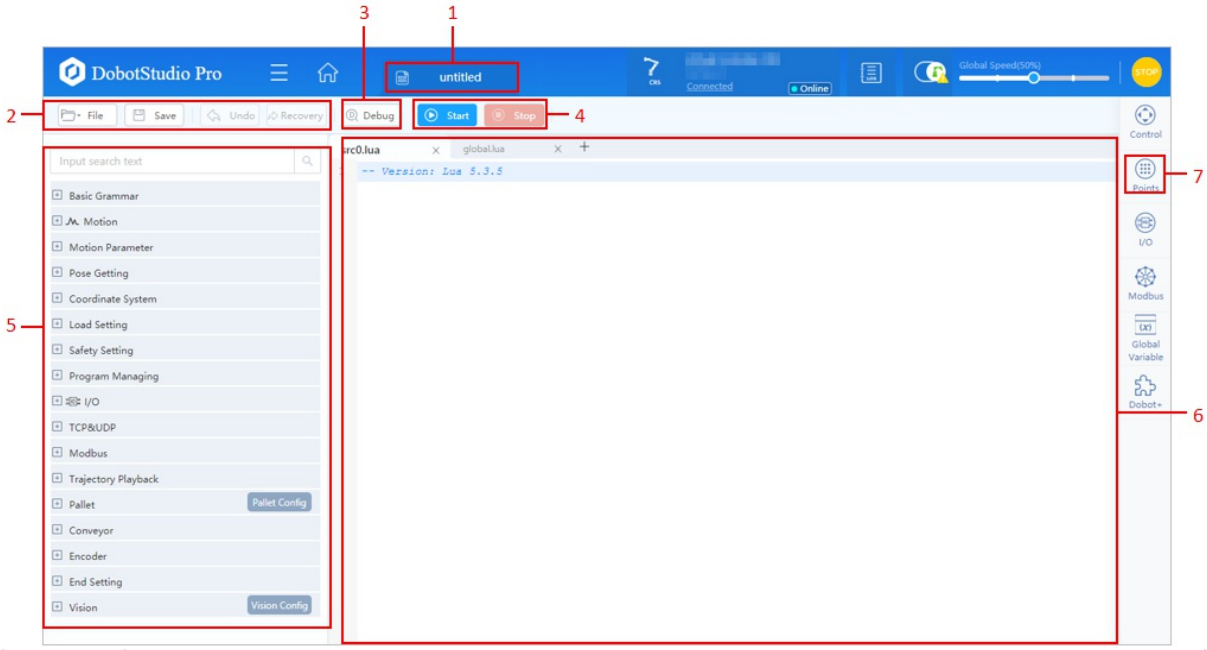


6. Click **Save**, enter the project name and click **OK**.
7. Click **Start**, and the robot starts to move.

9.2 Script

Dobot robots providess 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)*.

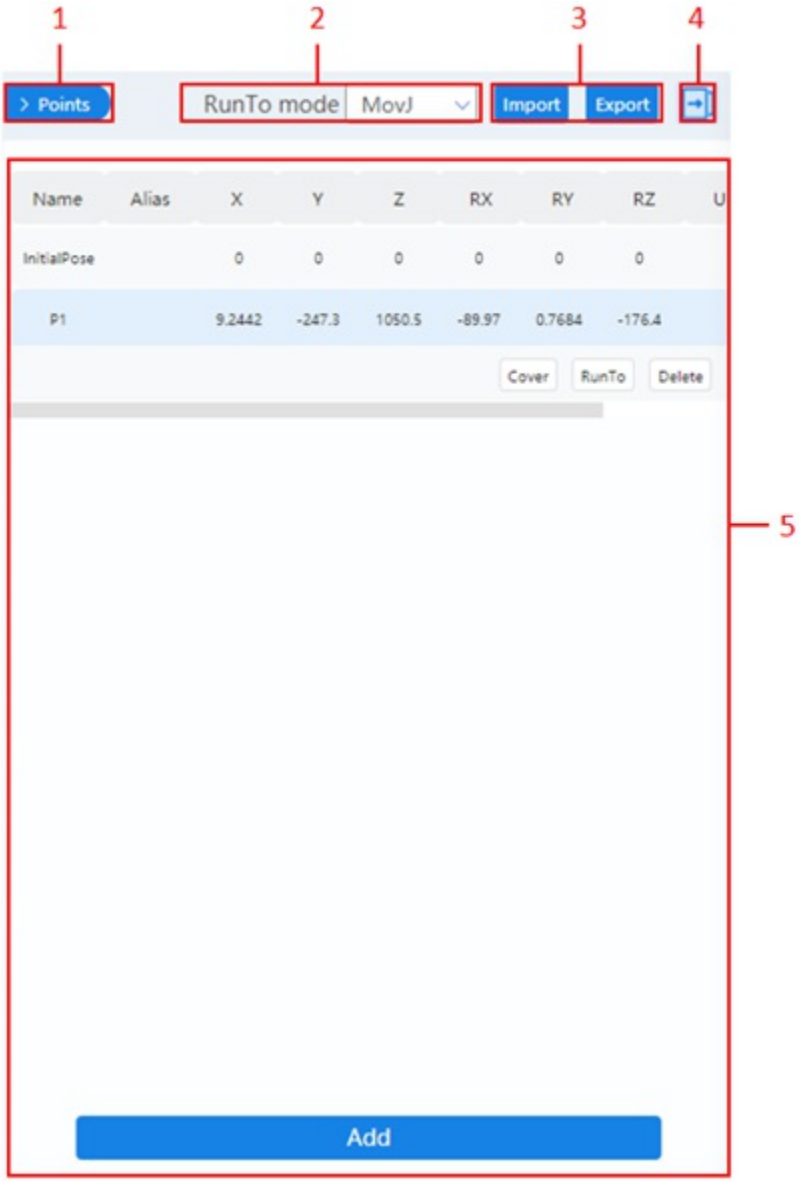


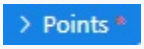
No.	Description
1	Display the current project name
2	It is used to manage project files and undo or restore programming operations.
3	Open the debug page. See Debugging project for details.
4	Control the running of the project. See Running project for details.
5	Command list. • Click on the left side 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.
	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.

6	• 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.

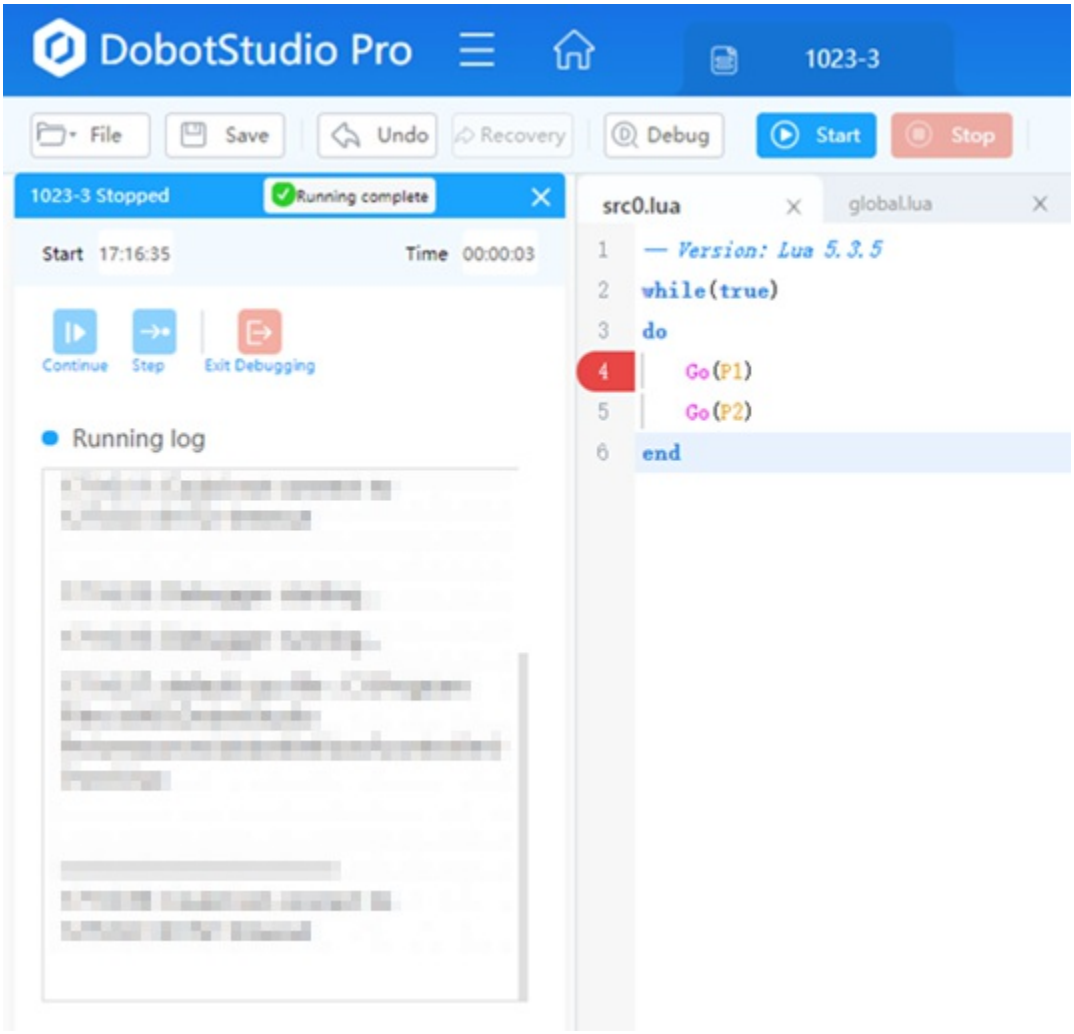


No.	Description
1	Click to hide **Points** panel. You can click **Points** on the right toolbar to restore it. If there are unsaved points, the icon will turn to  .
2	Set the motion mode of **Run To**
3	Import a point list file or export the current point list to a file.

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, 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 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.

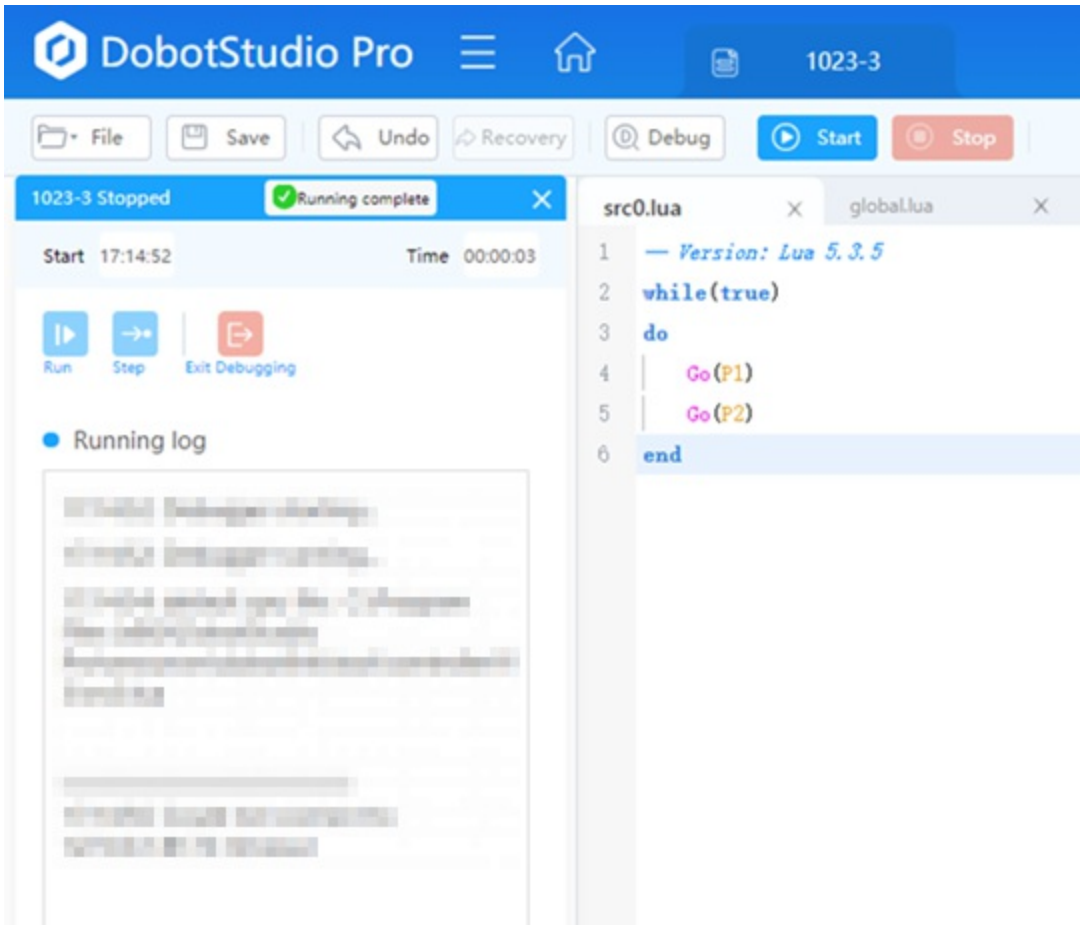


Running project

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

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

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

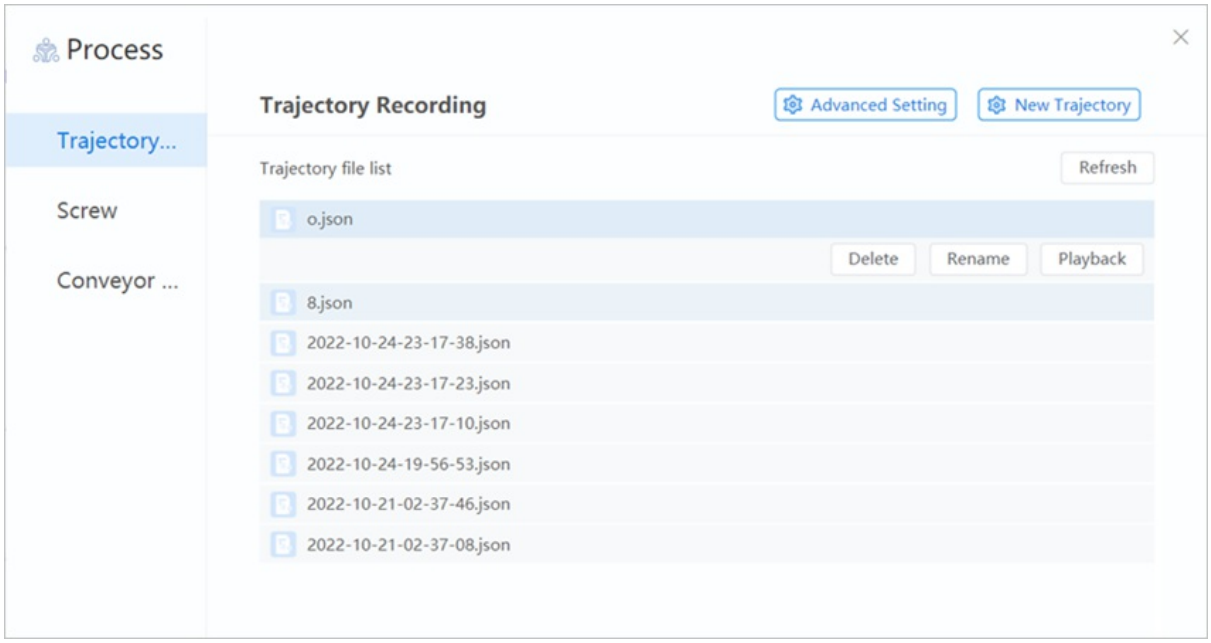
```
while(true)
do
    Go(P1)
    Go(P2)
end
```

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

10 Process

10.1 Trajectory playback

The trajectory playback is used to play back the trajectory recorded during manually dragging the robot arm. The main interface is shown below.



Click **New Trajectory** on the right top corner, and the indicator light at the end of the robot turns yellow. Now you can drag the robot. The trajectory will be recorded for playback.

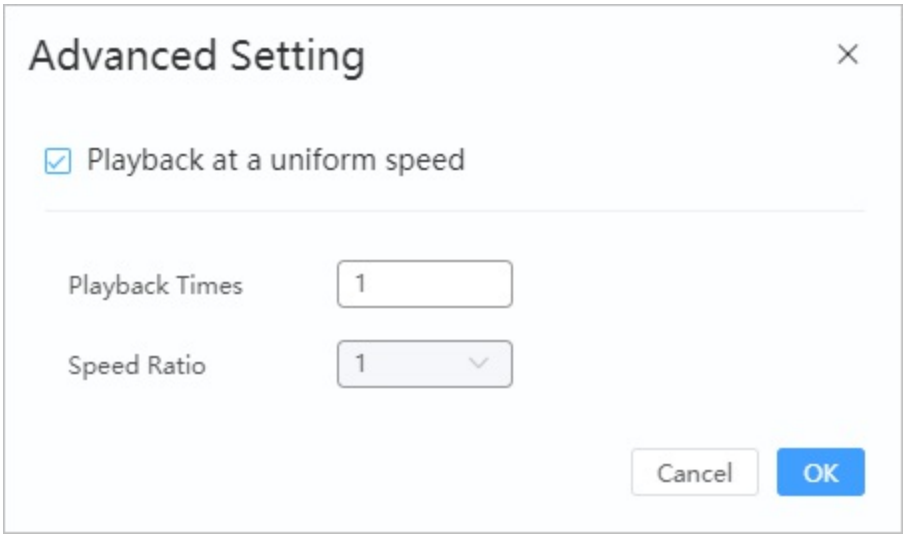
After recording the trajectory, click **Save path** on the right top corner. Now the indicator light turns green, and a new record named after the recording time is added in the Trajectory file list.

Click the saved trajectory, and you will see three buttons: **Delete**, **Rename** and **Playback**.

- Click **Playback** to play back the recorded trajectory of the robot. In this case the indicator light at the end of the robot flashes yellow, and the robot starts to move. After playback, the robot stops moving, and the indicator light turns green.
- Click **Rename** to modify the name of the trajectory file.
- Click **Delete** to delete this trajectory.

> NOTE
>
> - The saved trajectory file can be called through relevant playback commands in blockly and script programming.
> - Trajectory playback is not affected by the safety IO signal.

Click **Advanced** to set the playback mode. You can set the speed ratio of playback if you do not select **Playback at a uniform speed**.

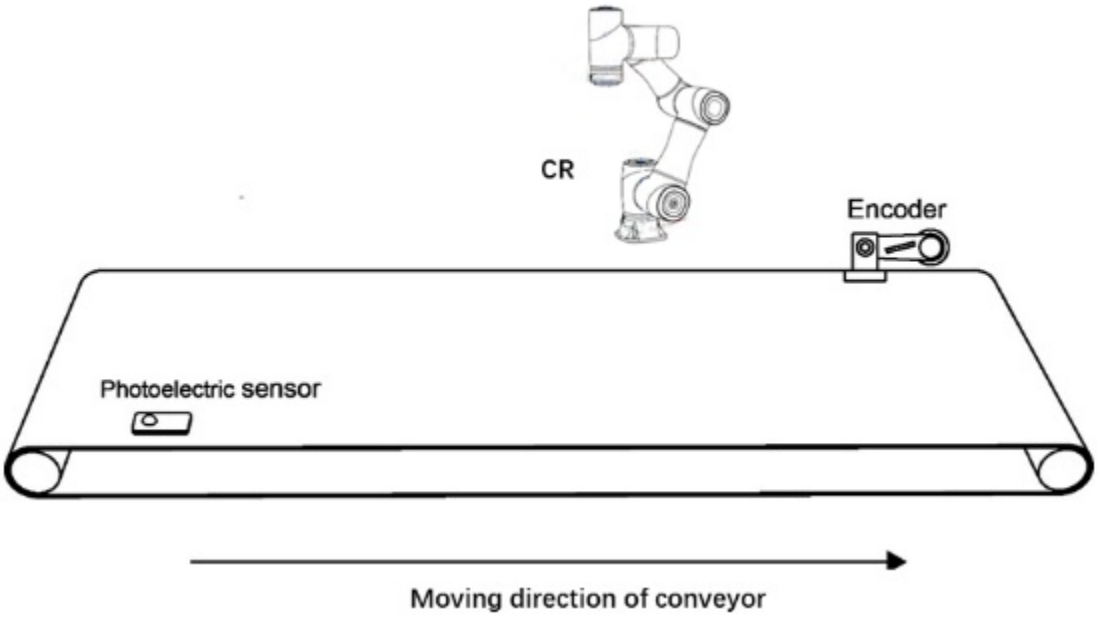

Advanced Setting
Playback at a uniform speed
Playback Times
1
Speed Ratio
1
Cancel
OK

10.2 Conveyor tracking

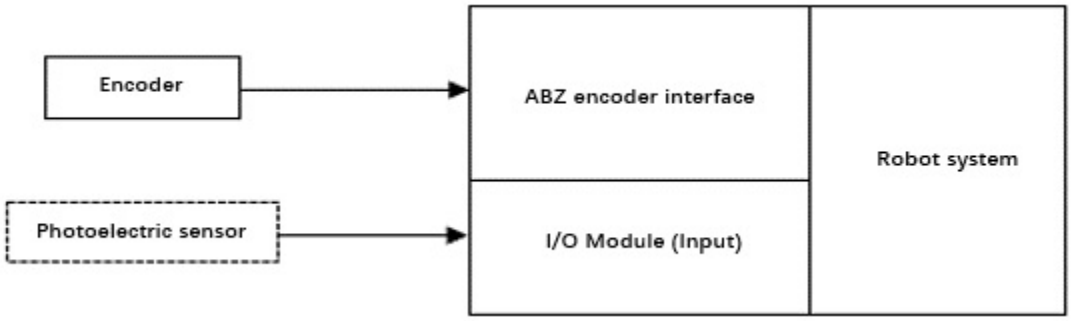
Conveyor process is that the photoelectric sensor detects the objects on the conveyor when the conveyor is moving, and the robot picks them up as they move.

Building Environment

The figure below shows a full process environment of conveyor tracking.



The communication diagram is shown below.



Encoder An encoder is used to record the conveyor moving distance and the workpiece position, and report them to the robot. The encoder should be connected to the ABZ encoder interface on the controller. It is recommended to use E6B2-CWZ1X (1000P/R) encoder. Its pin connection is shown in the table below.

Color	Port
Black	A+
Black and red	A-

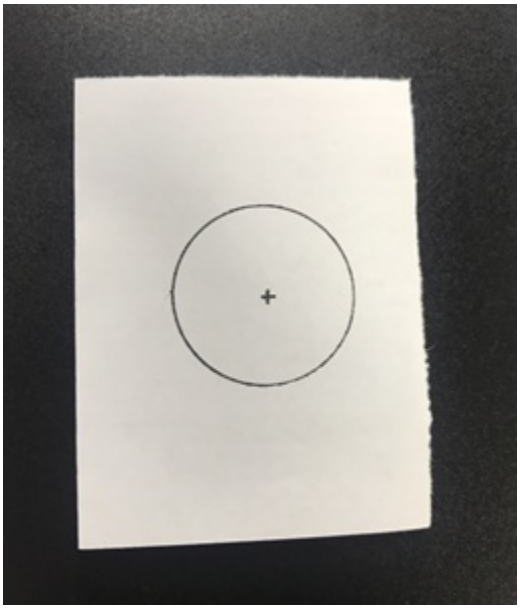
White	B+
White and red	B-
Orange	Z+
Orange and red	Z-
Brown	I/O 5V
Blue	I/O 0V

Photoelectric Sensor

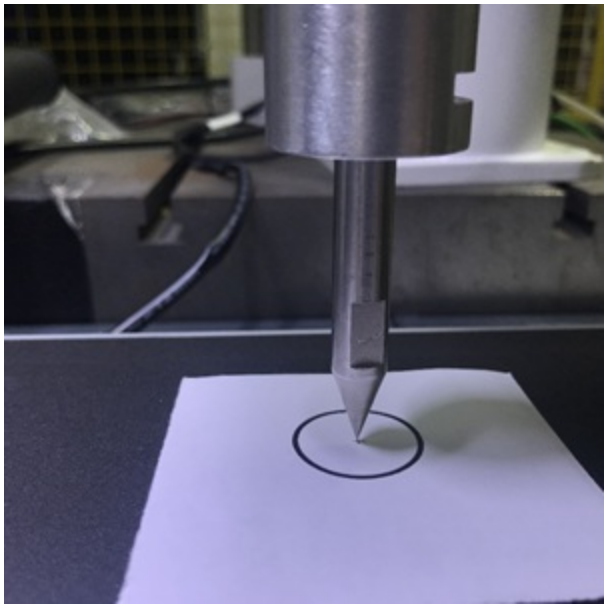
The photoelectric sensor outputs different level signals according to whether the workpiece is detected or not. The control system can detect the workpiece through the signal edge. You need to connect the photoelectric sensor to a DI interface on the controller, as shown below.

Calibrating Conveyor Before configuring the conveyor process, you need to calibrate the conveyor for obtaining the positional relationship between the conveyor and the robot. In the following steps, the user coordinate system is used for calibration. Before calibration, you need to install a calibration needle at the end of the robot, and prepare a label for calibration.

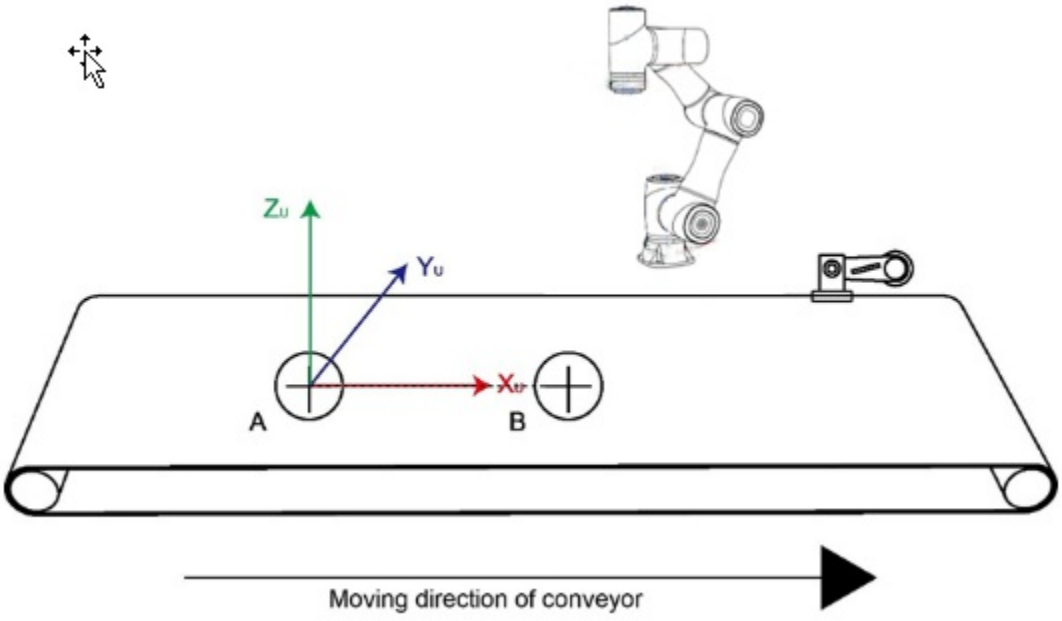
1. Paste a label on the conveyor, as shown below.



1. Enter User coordinate system page.
2. Enable the robot. Jog the robot to the label position on the conveyor to obtain the first point (point A, the origin of the User coordinate system).



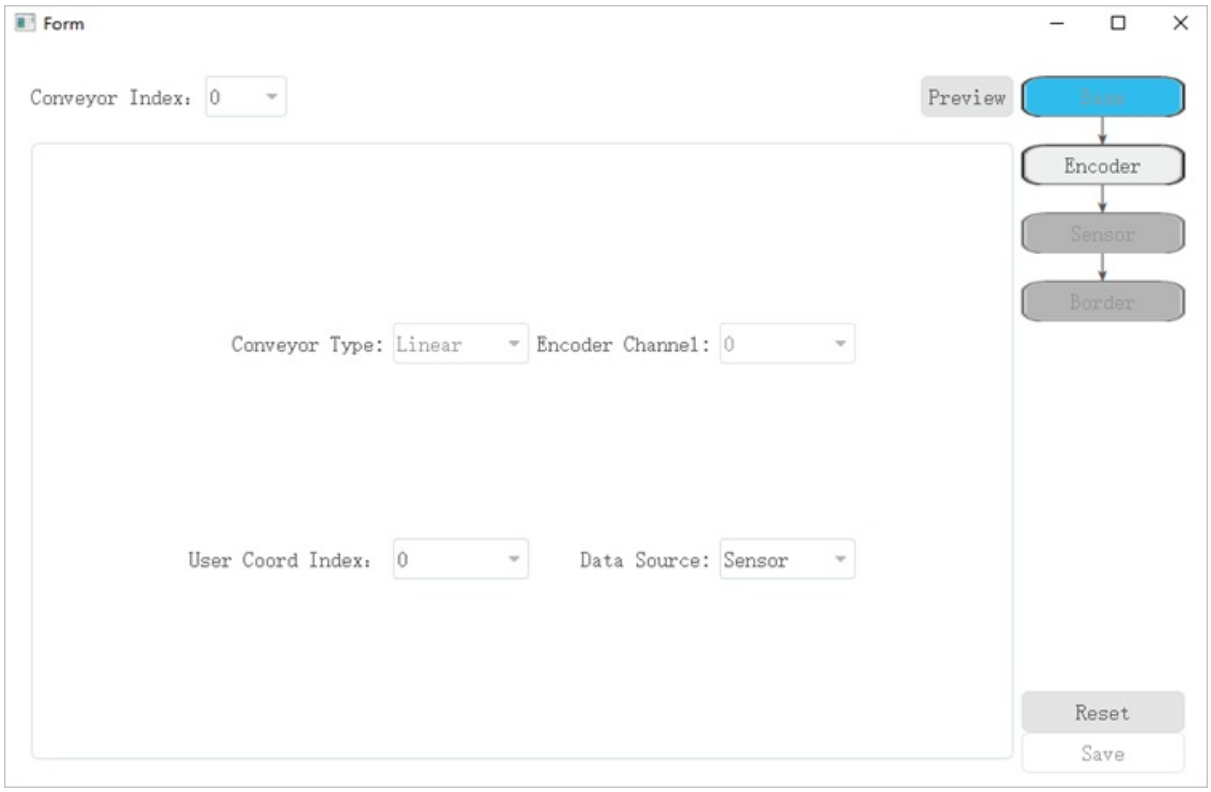
1. Control the conveyor to move a specified distance.
2. Jog the robot to the label position on the conveyor to obtain the second point (point B, to determine the x-axis)
3. Jog the robot to a point not in the line determined by point A and B on the conveyor so as to obtain the third point (point C, to determine the y-axis)



1. Add or cover a coordinate system, and save it.

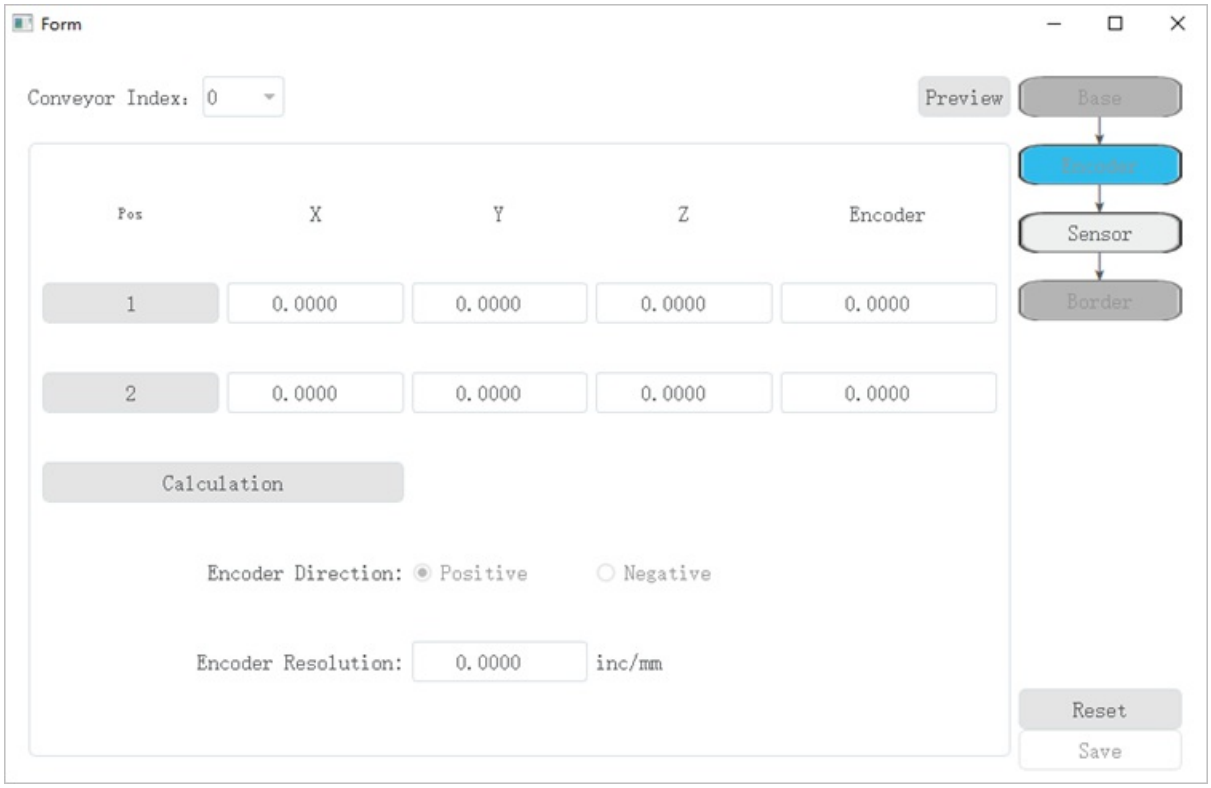
Configuring conveyor

Click **Conveyor Tracking** in Process page. Click **Open Tool**, and you will see the following page.



- Conveyor Index: Select the index of the conveyor to be configured, multiple conveyor are supported.
- Conveyor Type: Only linear type is supported (cannot be configured).
- Encoder Channel: The channel is default (cannot be configured).
- User Coord Index: Select the conveyor coordinate system saved in the last step.
- Data Source: Only sensor is supported (cannot be configured).

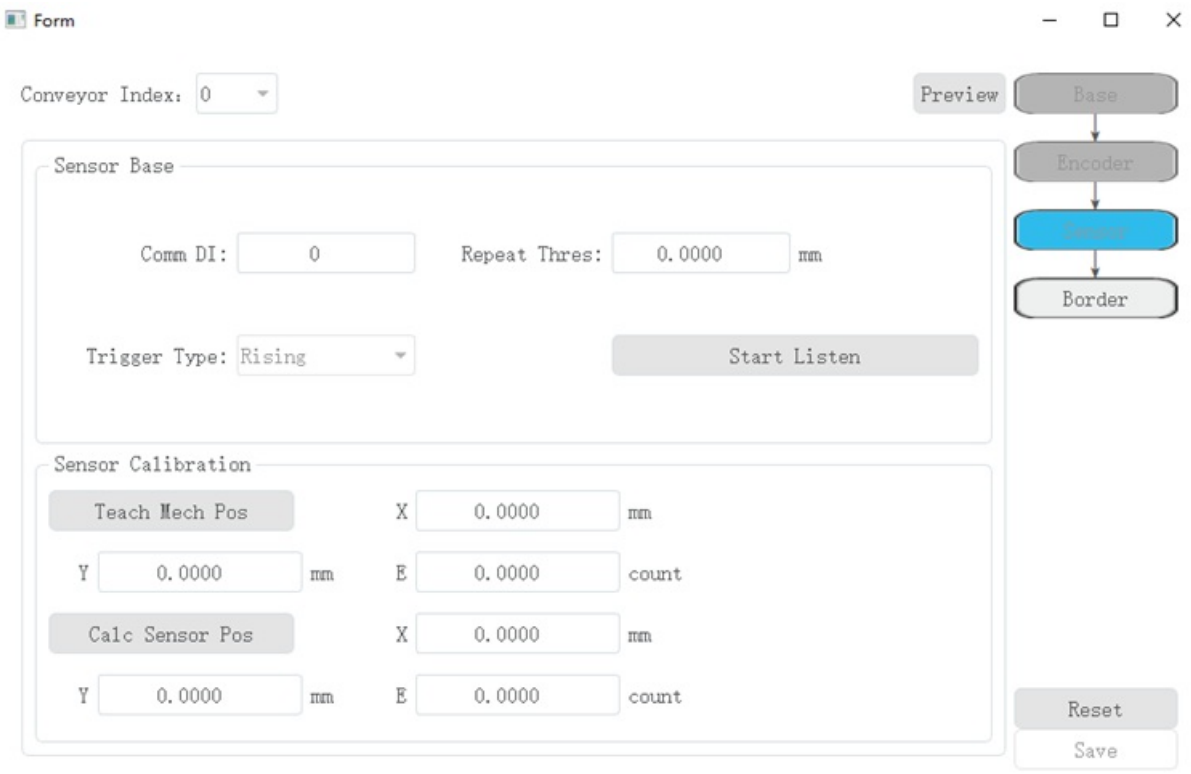
Encoder



Calibrate the encoder resolution. The encoder resolution is the pulse increment of the encoder per unit length that the conveyor moves.

1. Paste a label on the conveyor. Jog the robot to the label position on the conveyor, and click **1** in **Pos** column to obtain the value of Position 1.
2. Control the conveyor to move a specified distance and stop it.
3. Jog the robot to the label position on the conveyor. Click **2** in **Pos** column to obtain the value of Position 2.
4. Set the direction of the encoder according to the actual condition (taking **Positive** as an example).
5. Click **Calculation** to obtain the encoder resolution.

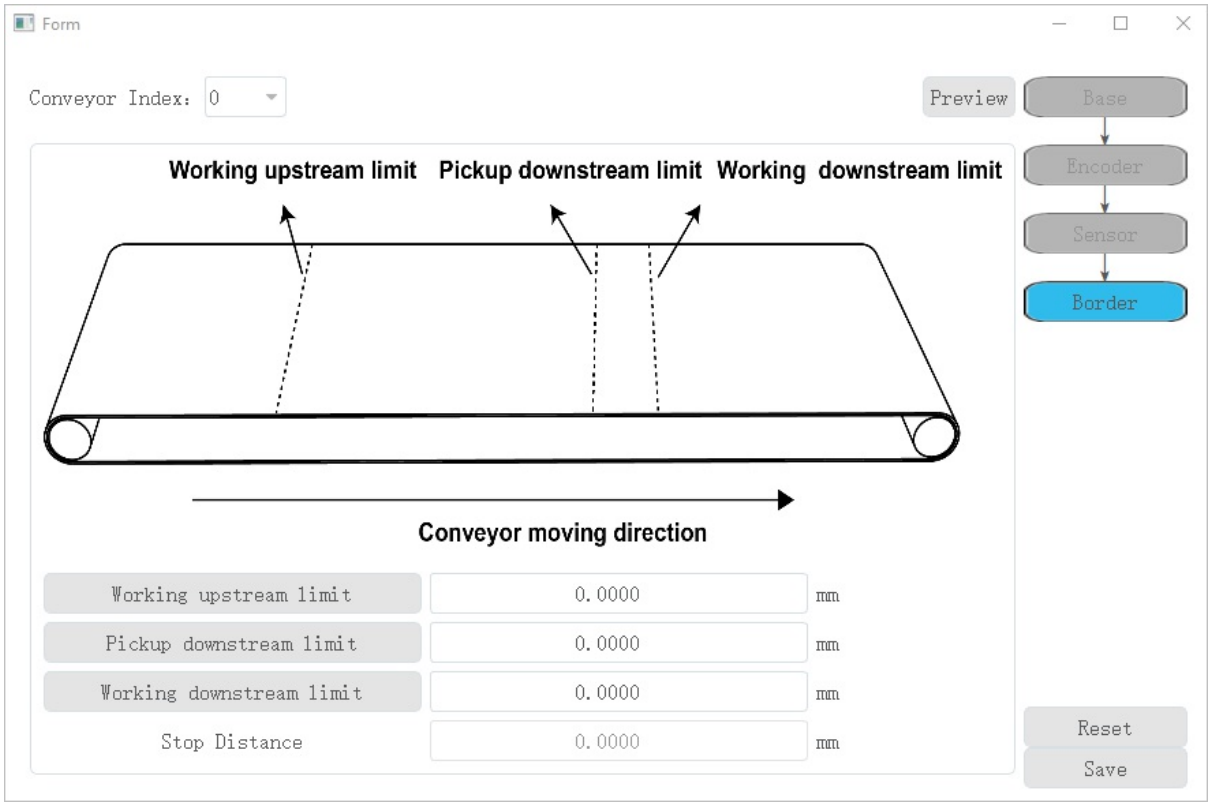
Sensor settings



- Trigger input IO: Set the DI port index in the controller to which the sensor is connected.
- Deduplication distance: After a valid signal is detected, if the signal changes within a subsequent distance, the signal will be considered as invalid and automatically eliminated. The value can be set to 2mm ~ 5mm based on the actual condition.
- Trigger type: Only rising edge trigger is supported (cannot be configured).

The sensor calibration aims to obtain the position where the sensor detects the workpiece so that the position of the workpiece under the User coordinate system at every moment can be calculated based on the coordinate offset when the workpiece moves along with the conveyor.

1. Click **Start Listen** to start listening the sensor signal.
2. Place a workpiece on the upsteam of the conveyor. Turn on the conveyor, and the workpiece moves with the conveyor. After the sensor detects the workpiece and the workpiece is within the workspace of the robot, stop the conveyor.
3. Jog the robot to the workpiece center, then click **Teach Mech Pos** to obtain the current position.
4. Click **Calc Sensor Pos** to obtain the position of the sensor.



Setting working border

The working border is used to set the working area of the robot on the conveyor.

- Work enter border: After the workpiece crosses this border, the robot arm starts to track and pick up the workpiece.
- Pick max border: When the workpiece crosses the border, If the robot arm has not started picking the workpiece, it is regarded that it cannot finish the pickup of the workpiece, so it will not pick up the workpiece. The border should be set based on the moving speed of the conveyor and the actual experience. It is recommended to debug multiple times before obtaining the optimal value.
- Work off border: After the workpiece crosses the border, the robot arm will stop tracking the workpiece (to prevent the robot from tracking out of the working area), and trigger alarms.

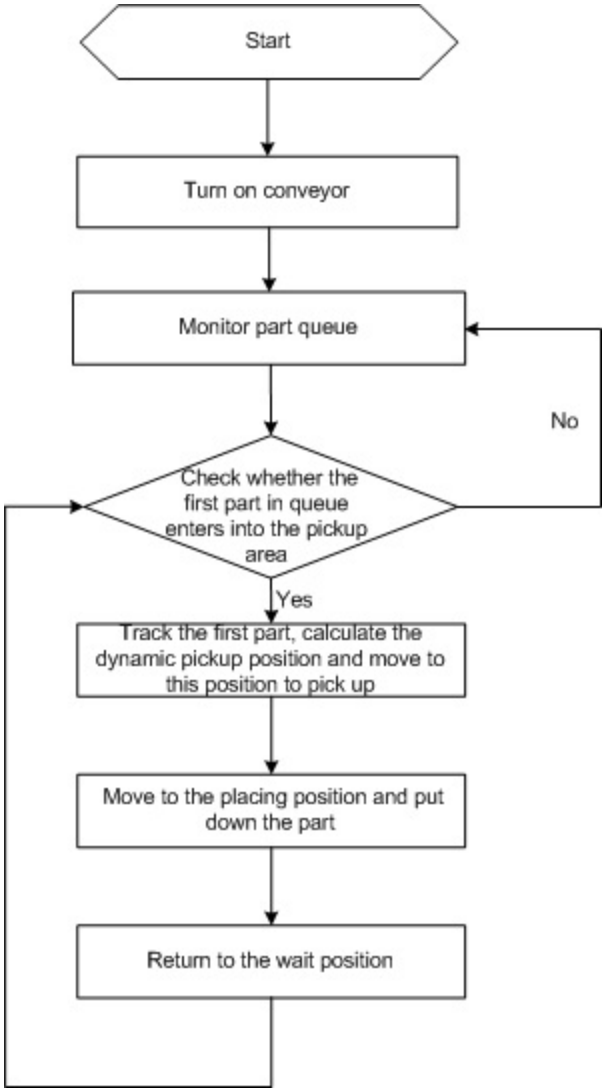
Jog the robot to the corresponding position of each border, and click the buttons to set the values.

Click **Save** on the right top corner. Now you can use the conveyor configuration in the project.

Example

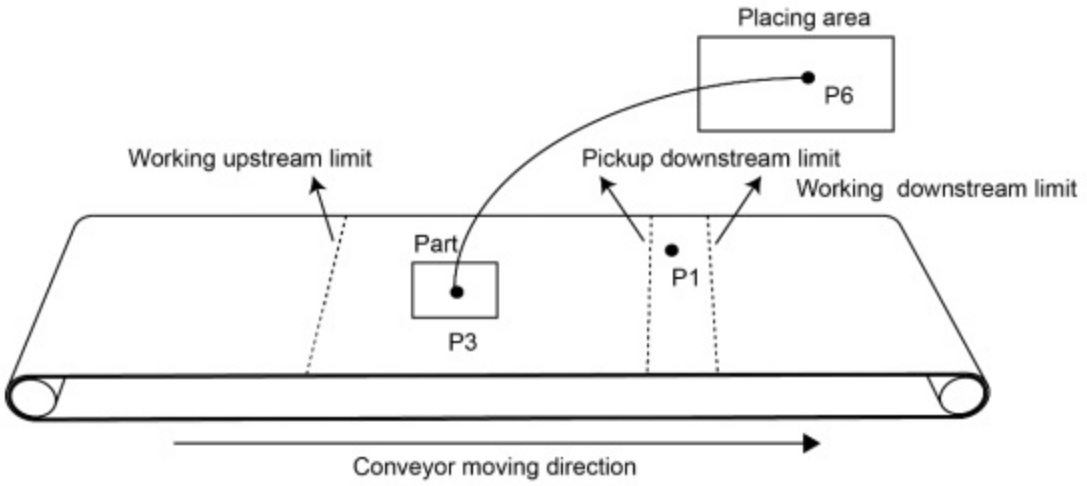
After configuring the parameters of the conveyor belt, you can edit a script by calling the conveyor belt APIs to realize conveyor tracking.

The process of the conveyor tracking is shown as follows:



In this example, you need to teach six points under the user coordinate system set when you configure the conveyor belt.

- Waiting point: P1
- Tracking point: P2
- Pickup point: P3
- Lifting point: P4
- Point above the placing point: P5
- Placing point: P6



The script is as follows.

```
CnvVison(0) ----------------------//Start the conveyor

DO(9,0) ----------------------//Control the start-up and status of air pump through DO1 and DO2

DO(2,0)

local flag  ----------------------//Whether there is a workpiece label

local typeObject  ----------------------//Type of workpiece

local point = {0,0,0}  ----------------------//Queue Workpiece coordinates

while true do

      Go(P1,"Speed=100 Accel=100 SYNC=1") ---//Waiting point, generally set within the workspace

      print("Test")

      while true do

                flag,typeObject,point = GetCnvObject(0,0) --//Check whether there is a workpiece. If there is, break.

                if flag == true then

                          break

                end

                Sleep(20)

      end

      SyncCnv(0) ----------------------//Synchronize the conveyor belt and start tracking
```

```
    Move(P2,"SpeedS=100 AccelS=100") ------------------//Over the pickup point

    Move(P3,"SpeedS=100 AccelS=100") ------------------//Pickup point

    Wait(100)

    DO(9,1) ----------------------//Turn on the air pump to pick up the workpiece

    --DO(2,1)

    Wait(100)

    Move(P4,"SpeedS=100 AccelS=100 SYNC=1")  ---------//Lift the workpiece

    StopSyncCnv() ----------------------//Stop conveyor tracking

    Sleep(20)

    Go(P5,"Speed=100 Accel=100")  ----------------//Over the placing point

    Go(P6,"Speed=100 Accel=100 SYNC=1")  --------//Placing point

    Sleep(1000)

    DO(1,0) ----------------------//Turn off the air pump

    DO(2,0,"SYNC=1")

    Sleep(1000)

    Go(P5,"Speed=100 Accel=100")

End
```

11 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 (taking DH gripper as an example, please refer to DH gripper user guide for its installation), 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:

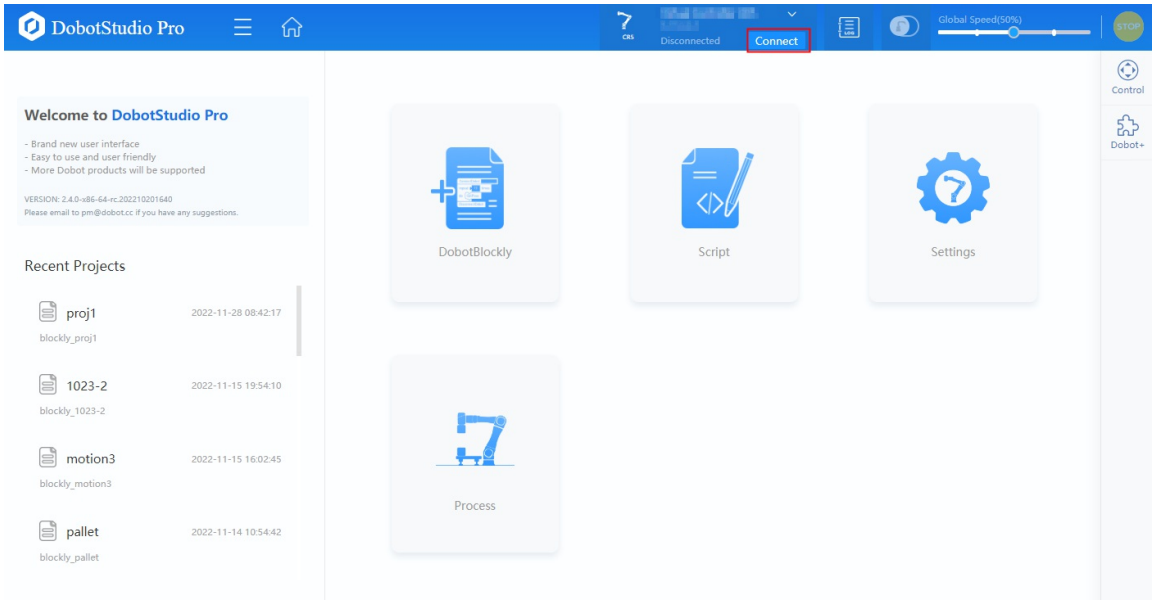
1. Connect the robot
2. Install Dobot+ plug-in (optional)
3. Set and select a tool coordinate system
4. Edit the project file
5. Configure and enter remote I/O mode

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 Dobot_WIFI_XXX (XXX is the robot index located on the base), 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.

Installing Dobot+ plug-in

For details about Dobot+ plugin, refer to Dobot+. Here takes DH plugin as an example.

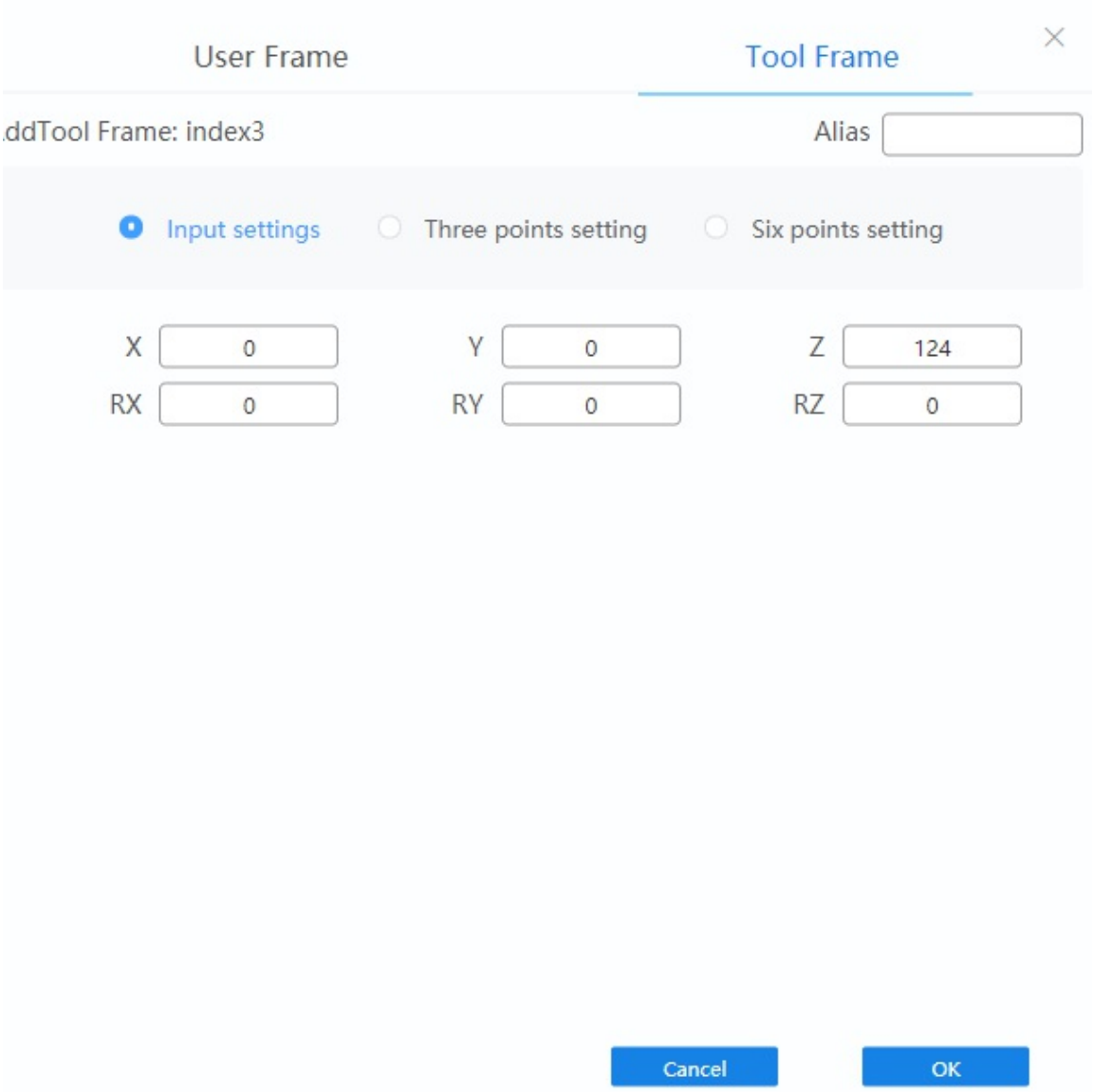
If the installed end tool has no corresponding plug-in, skip this step and directly control the end tool through I/O commands in subsequent programming.

1. Click **Dobot+** on the right side of the main interface to open the Dobot+ panel.
2. Click **Add Plugins**, and then click **Install** of the DH plug-in to complete the installation.

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**.



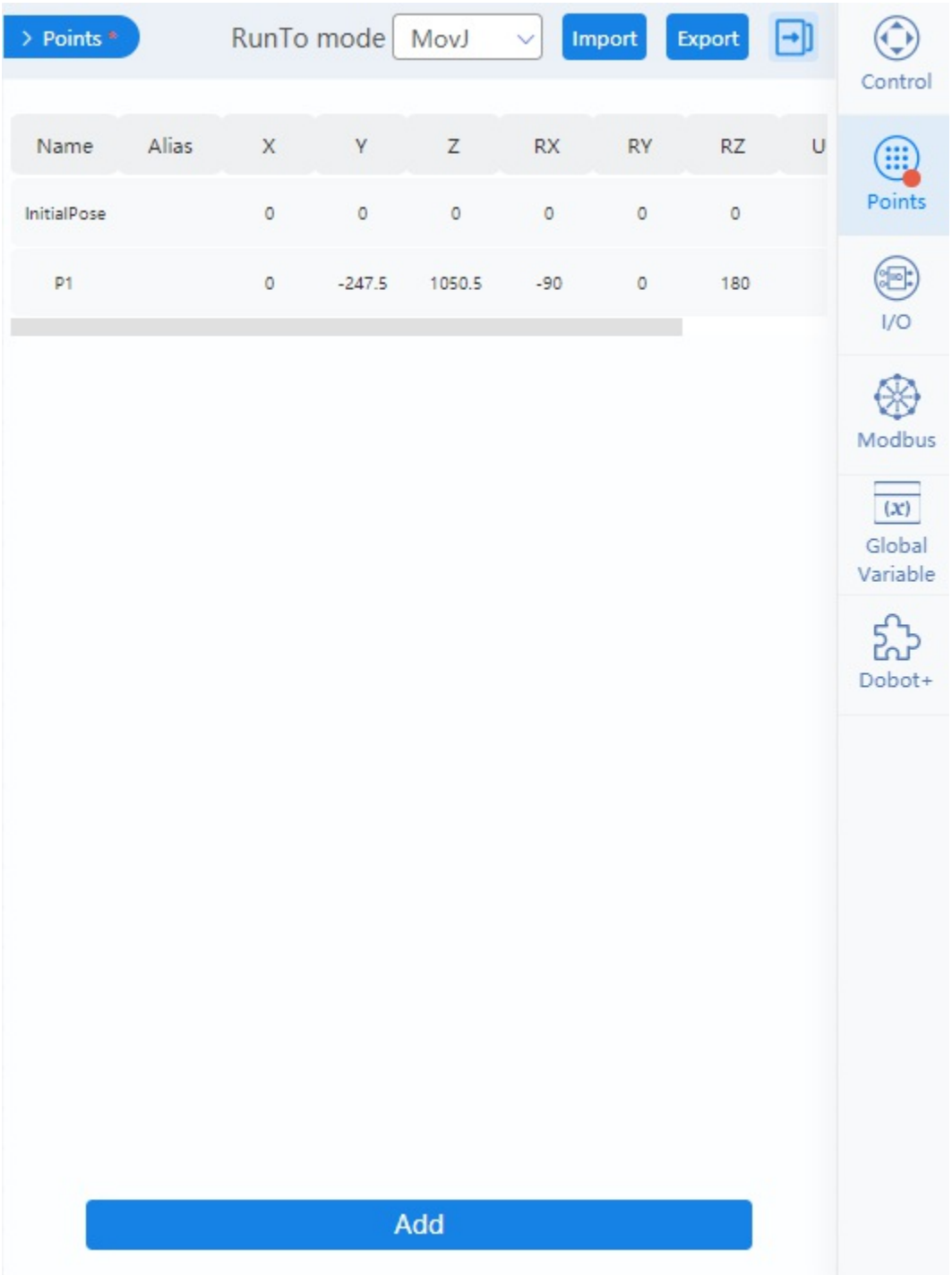
3. Select the tool coordinate system that you set in the last step in the control panel.

Editting project

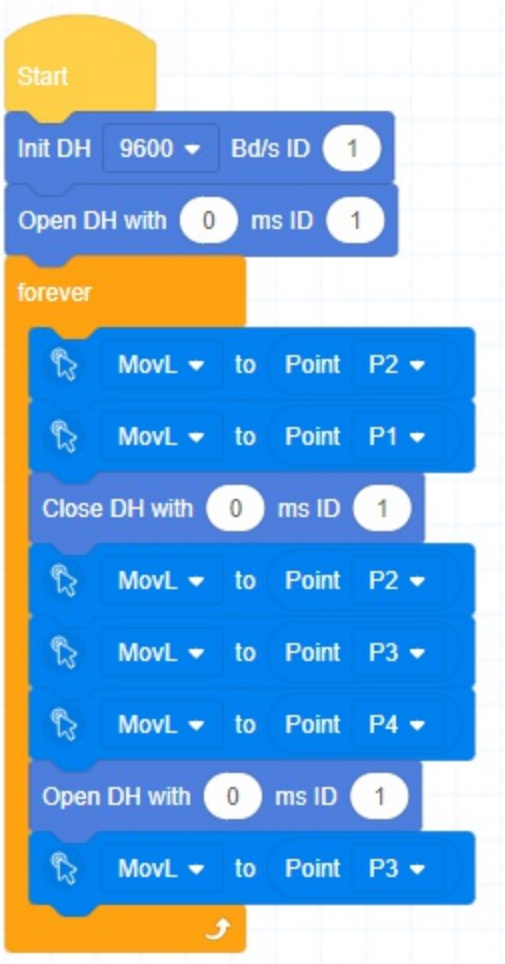
For details on programming, refer to DobotBlockly and Script. Here takes DobotBlockly as an example.

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.

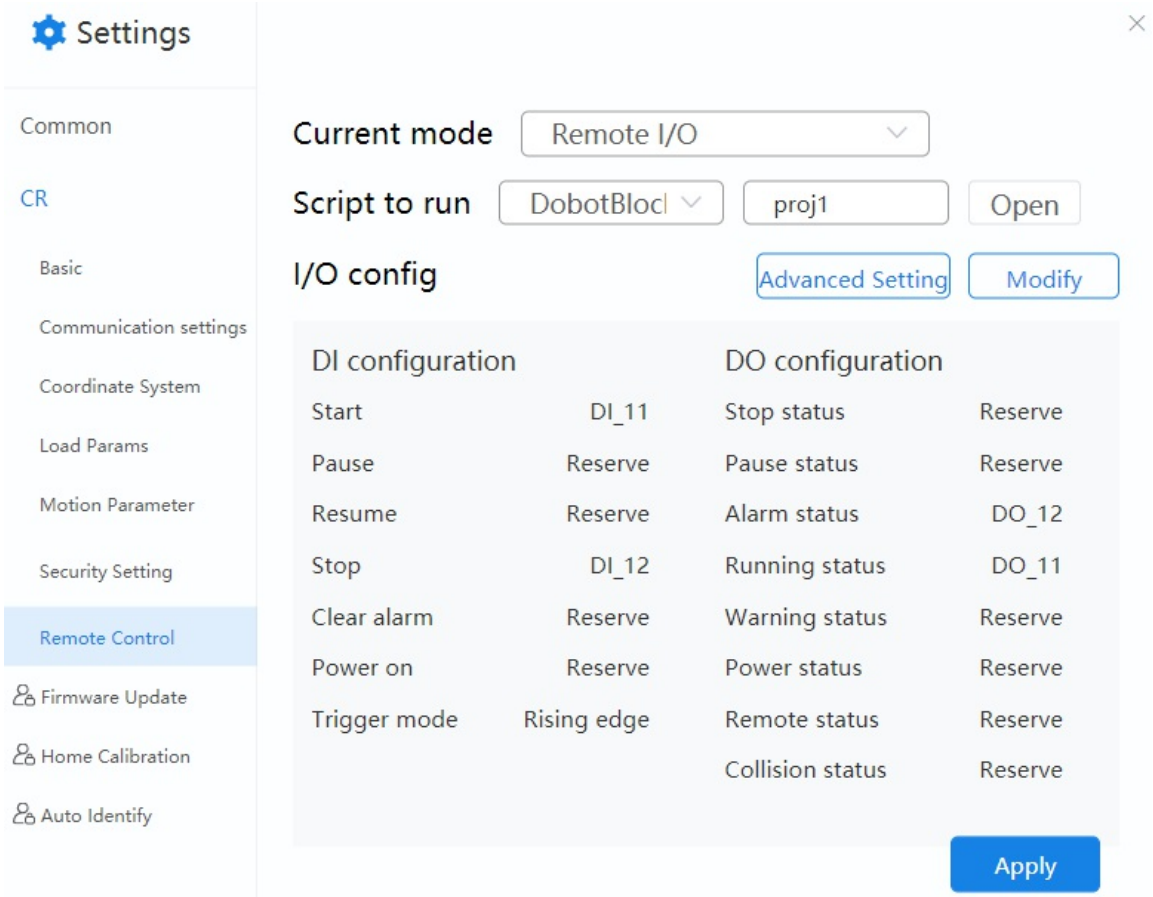


4. Save the project.

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

1 Modbus Introduction

Modbus protocol is a serial communication protocol. The robot system can communicate with external equipment through this protocol. The external equipment, such as a PLC, is set as the Modbus master, and the robot system is set as the slave.

Currently Modbus has the following versions:

- Modbus-RTU: It is compact, and adopts binary representation of data, using a checksum mode of cyclic redundancy check.
- Modbus-ASCII: It adopts ASCII string representation, which is readable and redundant, using checksum mode of longitudinal redundancy check.
- Modbus-TCP: It adopts TCP connection over TCP/IP. This mode does not require checksum calculation.

Based on our existing hardware interface, we currently adopt Modbus-TCP scheme.

A Modbus address typically consists of five digits, including the starting datatype code, followed by the offset address. The Modbus Master protocol library maps standard Modbus addresses to Modbus function numbers, and reads and writes data from slave stations.

Modbus Master protocol library supports the following addresses:

- 00001-09999: Digital output (coil)
- 10001-19999: Digital input (contact)
- 30001-39999: Data input register (generally analog input)
- 40001-49999: Data holding register

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.

2 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	Power on
00101	100	Bit	Enable
00102	101	Bit	Disable
03096 – 10000	3095 – 9999	Bit	User-defined

3 Discrete input (robot status)

PLC address	Script address (GetInBits)	Register type	Function
10002	1	Bit	Stop status
10003	2	Bit	Paused status
10004	3	Bit	Running status
10005	4	Bit	Alarm status
10006	5	Bit	Reserved
10008	7	Bit	Remote mode
13096 – 20000	3095 – 9999	Bit	User-defined

4 Input register

PLC address	Data type	Number of values	Byte	Script address (GetInRegs)	Register type	Function
31000	unsigned short	1	2	999	U16	Data validity
31001	unsigned short	1	2	1000	U16	Total byte length of message
31002 – 31004	unsigned short	3	6	1001 – 1003	U16	Reserved
31005 – 31008	uint64	1	8	1004 – 1007	U64	Digital input
31009 – 31012	uint64	1	8	1008 – 1011	U64	Digital output

31013 – 31016	uint64	1	8	1012 – 1015	U64	Robot mode
31017 – 31020	uint64	1	8	1016 – 1019	U64	Timestamp
31021 – 31024	uint64	1	8	1020 – 1023	U64	Reserved
31025 – 31028	uint64	1	8	1024 – 1027	U64	Memory structure test standard value 0x0123 4567 89AB CDEF
31029 – 31032	double	1	8	1028 – 1031	F64	Reserved
31033 – 31036	double	1	8	1032 – 1035	F64	Speed rate
31037 – 31040	double	1	8	1036 – 1039	F64	Current momentum of robot
31041 – 31044	double	1	8	1040 – 1043	F64	Control board voltage (Not supported by CCBOX)
31045 – 31048	double	1	8	1044 – 1047	F64	Robot voltage (Not supported by CCBOX)
31049 – 31052	double	1	8	1048 – 1051	F64	Robot current (Not supported by CCBOX)
31053 – 31056	double	1	8	1052 – 1055	F64	Reserved
31057 – 31060	double	1	8	1056 – 1059	F64	Reserved
31061 – 31072	double	3	24	1060 – 1071	F64	Reserved
31073 – 31084	double	3	24	1072 – 1083	F64	Reserved
31085 – 31096	double	3	24	1084 – 1095	F64	Reserved
31097 – 31120	double	6	48	1096 – 1119	F64	Target joint position
31121 – 31144	double	6	48	1120 – 1143	F64	Target joint speed
31145 – 31168	double	6	48	1144 – 1167	F64	Target joint acceleration
31169 –	double	6	48	1168 – 1191	F64	Target joint current

31192	double	6	48	1168 – 1191	F64	Target joint current
31193 – 31216	double	6	48	1192 – 1215	F64	Target joint torque
31217 – 31240	double	6	48	1216 – 1239	F64	Actual joint position
31241 – 31264	double	6	48	1240 – 1263	F64	Actual joint speed
31265 – 31288	double	6	48	1264 – 1287	F64	Actual joint current
31289 – 31312	double	6	48	1288 – 1311	F64	TCP sensor force (Six-dimensional force sensor needs to be installed)
31313 – 31336	double	6	48	1312 – 1335	F64	TCP actual Cartesian coordinates
31337 – 31360	double	6	48	1336 – 1359	F64	TCP actual Cartesian speed
31361 – 31384	double	6	48	1360 – 1383	F64	TCP force (calculate through joint current)
31384 – 31408	double	6	48	1384 – 1407	F64	TCP target Cartesian coordinates
31409 – 31432	double	6	48	1408 – 1431	F64	TCP target Cartesian speed
31433 – 31456	double	6	48	1432 – 1455	F64	Joint temperature
31456 – 31480	double	6	48	1456 – 1479	F64	Joint control mode
31481 – 31504	double	6	48	1480 – 1503	F64	Joint voltage
31505 – 31506	char	4	4	1504 – 1505	U16	Hand coordinate system
31507	char	1	1	1506 low byte	U8	User coordinates
31507	char	1	1	1506 high byte	U8	Tool coordinates
31508	char	1	1	1507 low byte	U8	Algorithm queue running signal
31508	char	1	1	1507 high byte	U8	Algorithm queue pause signal

31509	char	1	1	1508 low byte	U8	Joint speed rate
31509	char	1	1	1508 high byte	U8	Joint acceleration rate
31510	char	1	1	1509 low byte	U8	Joint jerk rate
31510	char	1	1	1509 high byte	U8	Cartesian position speed ratio
31511	char	1	1	1510 low byte	U8	Cartesian posture speed ratio
31511	char	1	1	1510 high byte	U8	Cartesian position acceleration ratio
31512	char	1	1	1511 low byte	U8	Cartesian posture acceleration ratio
31512	char	1	1	1511 high byte	U8	Cartesian position jerk ratio
31513	char	1	1	1512 low byte	U8	Cartesian posture jerk ratio
31513	char	1	1	1512 high byte	U8	Robot brake status
31514	char	1	1	1513 low byte	U8	Robot enabling status
31514	char	1	1	1513 high byte	U8	Robot drag status
31515	char	1	1	1514 low byte	U8	Robot drag status
31515	char	1	1	1514 high byte	U8	Robot alarm status
31516	char	1	1	1515 low byte	U8	Robot jog status
31516	char	1	1	1515 high byte	U8	Robot type
31517	char	1	1	1516 low byte	U8	Button board drag signal
31517	char	1	1	1516 high byte	U8	Button board enabling signal (Not supported by Nova series)
31518	char	1	1	1517 low byte	U8	Button board record signal (Not supported by Nova series)

31518	char	1	1	1517 high byte	U8	Button board playback signal (Not supported by Nova series)
31519	char	1	1	1518 low byte	U8	Button board gripper control signal (Not supported by Nova series)
31519	char	1	1	1518 high byte	U8	Six-dimensional force online status (Six-dimensional force sensor needs to be installed)
31520	char	1	1	1519 low byte	I8	Collision status
31520	char	1	1	1519 high byte	I8	(SafeSkin) arm close to pause status
31521	char	1	1	1520 low byte	I8	(SafeSkin) J4 close to pause status
31521	char	1	1	1520 high byte	I8	(SafeSkin) J5 close to pause status
31522	char	1	1	1521 low byte	I8	(SafeSkin) J6 close to pause status
31522	char	1	1	1521 high byte	I8	Reserved
31523 – 31552	-	-	-	1522 – 1551	-	Reserved
31553 – 31556	double	1	8	1552 – 1555	F64	Reserved
31557 – 31560	uint64	1	8	1556 – 1559	U64	Reserved
31561 – 31584	double	6	48	1560 – 1583	F64	Actual torque
31585 – 31588	double	1	8	1584 – 1587	F64	Load weight (kg)
31589 – 31592	double	1	8	1588 – 1591	F64	X-directional eccentric distance (mm)
31593 – 31596	double	1	8	1592 – 1595	F64	Y-directional eccentric distance (mm)
31597 –						Z-directional

31600						(mm)
31601 – 31624	double	6	48	1600 – 1623	F64	User coordinates
31625 – 31648	double	6	48	1624 – 1647	F64	Tool coordinates
31649 – 31652	double	1	8	1648 – 1651	F64	Trajectory playback running index
31653 – 31676	double	6	48	1652 – 1675	F64	Original value of current six-dimentional force (Six-dimensional force sensor needs to be installed)
31677 – 31692	double	4	32	1676 – 1691	F64	[qw,qx,qy,qz] target quaternion
31693 – 31708	double	4	32	1692 – 1707	F64	[qw,qx,qy,qz] actual quaternion
31709 – 31721	double	1	24	1708 – 1720	F64	Reserved
			1440			Totally 1440 bytes

5 Holding register (interaction between robot and PLC)

PLC address	Script address (Get/SetHoldRegs)	Register type	Function
40101	100	U16	Set the global speed, range: 1–100 (the value beyond the range will not take effect)
40001 – 41281	0 – 1280	U16	Reserved for palletizing
43095 – 49000	3095 – 8999	U16	User-defined
49001	9000	U16	Current screw index
49002	9001	U16	Specify the screw number to be executed: The number starts with 1, and 0 means no screw number is specified
49003	9002	U16	Screw locking result: Support at most 50 groups; 0 represents failure; 1 represents success; 2 represents None
49004	9003	U16	Number of screws on the locking product (working station)

49004	9003	U16	(working station)
49005 – 49054	9004 – 9053	U16	Generate locking result: Support at most 50 groups; 0 represents failure; 1 represents success; 2 represents None
49055	9054	U16	Alarm code: 0: No alarm; 1: No material in the screw machine when taking the screws; 2: There are screws in the screwdriver before screw taking; 3: Screw taking failure; 4: Abnormal screw locking; 5: The electrical screwdriver is abnormal
49056	9055	U16	Total number of locked screws
49057	9056	U16	Total number of non-performing locked screws
49058	9057	U16	Total number of locked products
49201	9220	F64	Screw locking torsion (Nm)
49205	9224	F64	Screw locking angle (°)
49209	9228	F64	Screw locking cycle time (s)
49213 – 49412	9232 – 9431	F64	Product locking torsion (Nm); Support at most 50 groups
49413 – 49612	9432 – 9631	F64	Product locking angle (°) Support at most 50 groups

Appendix B Blockly Commands

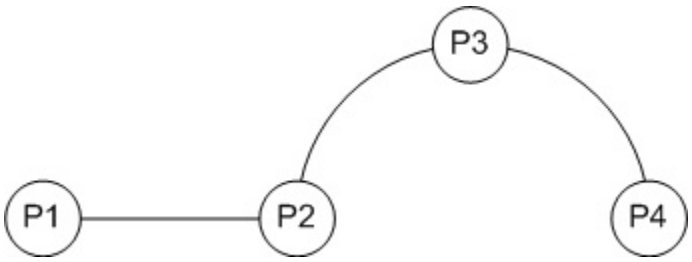
Quick start

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:

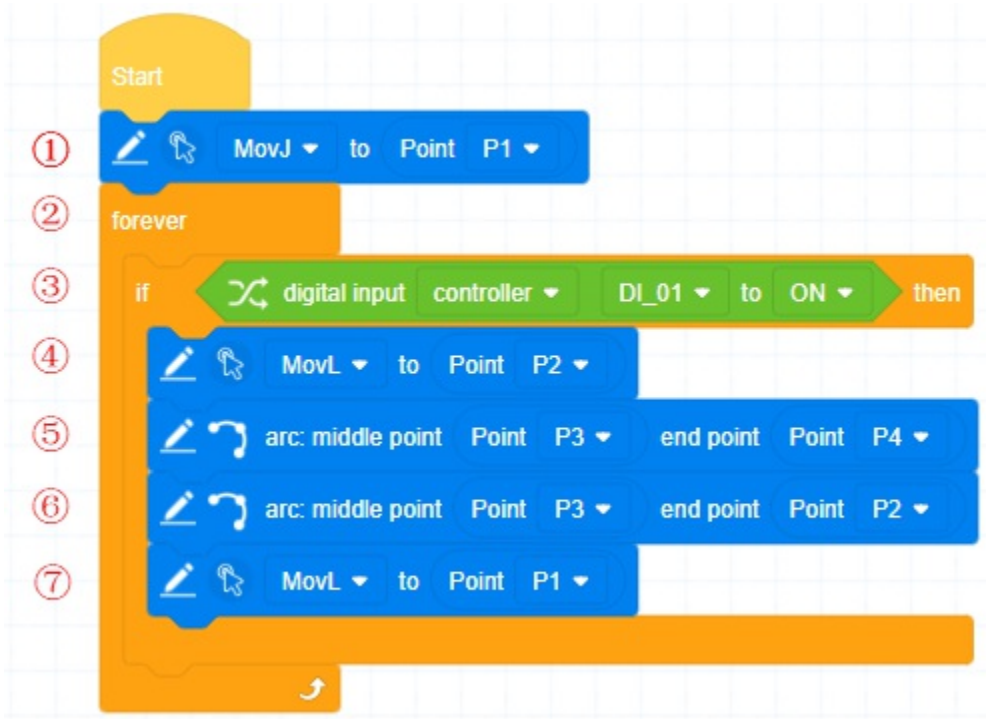
When the controller DI1 is ON, the robot moves from P1 to P2 in a linear mode, moves to P4 via P3 in a arc mode, and then returns along the same way. When the controller DI1 is OFF, the robot arm does not move.



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

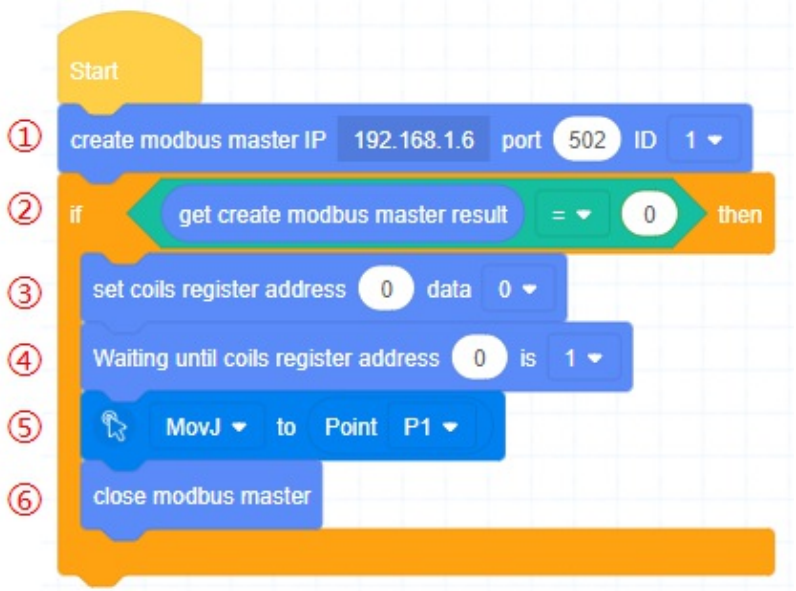
Scene description

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

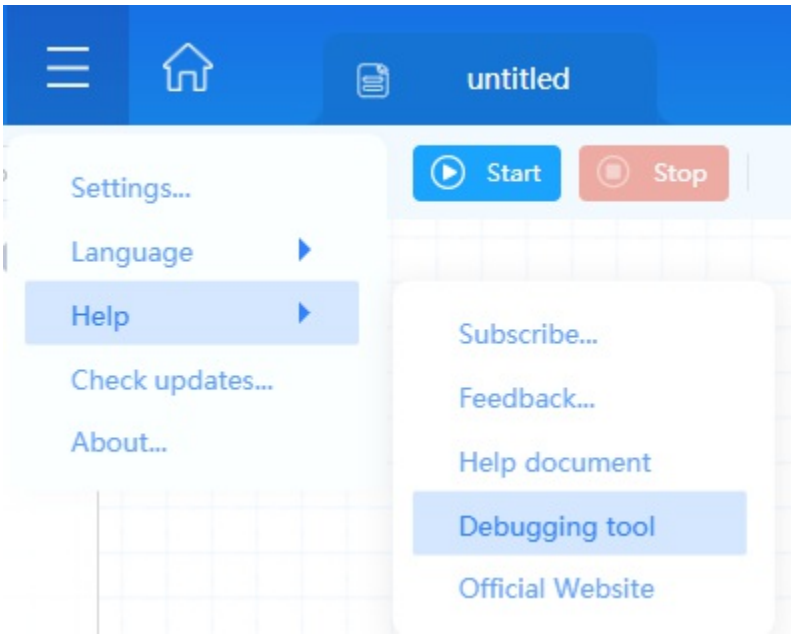
To achieve this scene, you need to edit the program as shown in the figure below.



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.

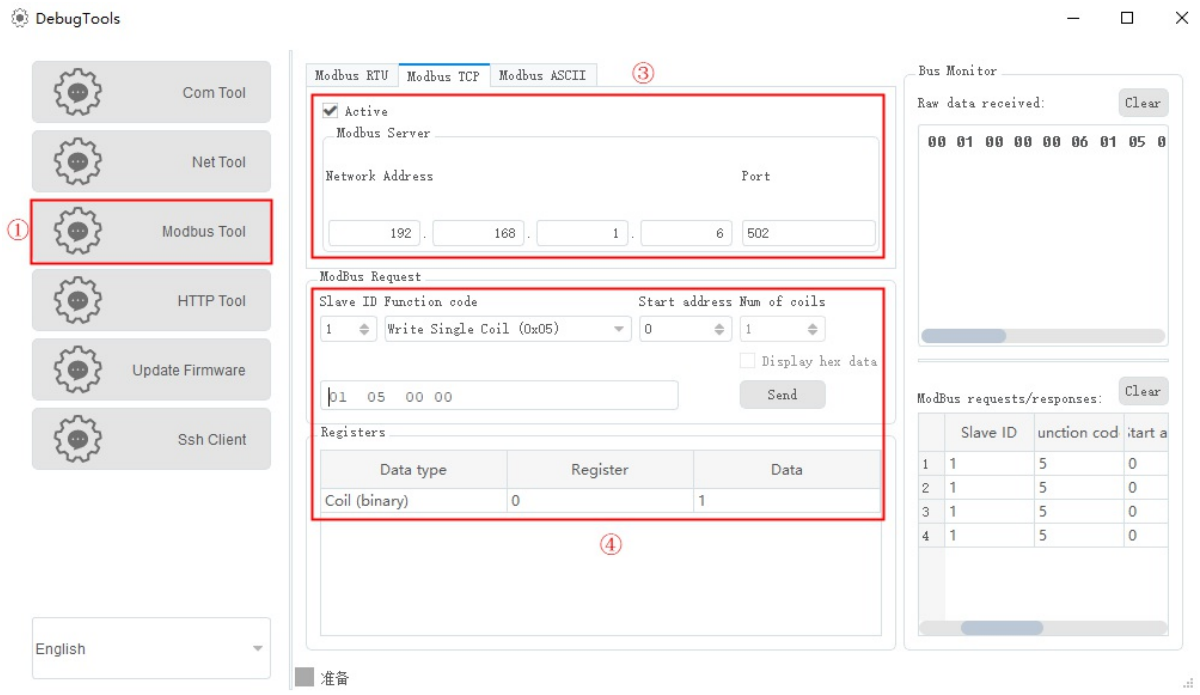
Run program

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.


DebugTools
Com Tool
Net Tool
①
Modbus Tool
HTTP Tool
Update Firmware
Ssh Client
English
Modbus RTU
Modbus TCP
Modbus ASCII
③
Active
Modbus Server
Network Address
Port
192
168
1
6
502
ModBus Request
Slave ID
Function code
Start address
Num of coils
1
Write Single Coil (0x05)
0
1
Display hex data
01 05 00 00
Send
Registers
Data type
Register
Data
Coil (binary)
0
1
④
Bus Monitor
Raw data received:
Clear
00 01 00 00 00 06 01 05 0
ModBus requests/responses:
Clear
Slave ID
unction cod
tart a
1 1 5 0
2 1 5 0
3 1 5 0
4 1 5 0
准备

Transmit data by TCP communication

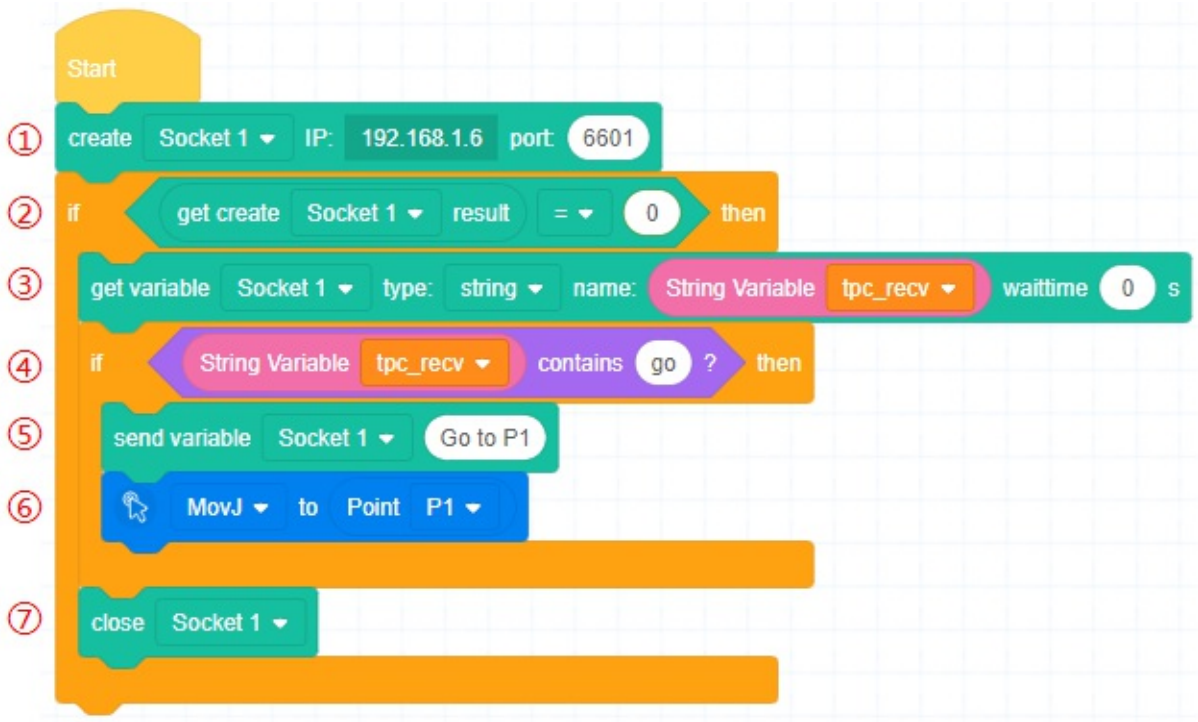
Scene description

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.

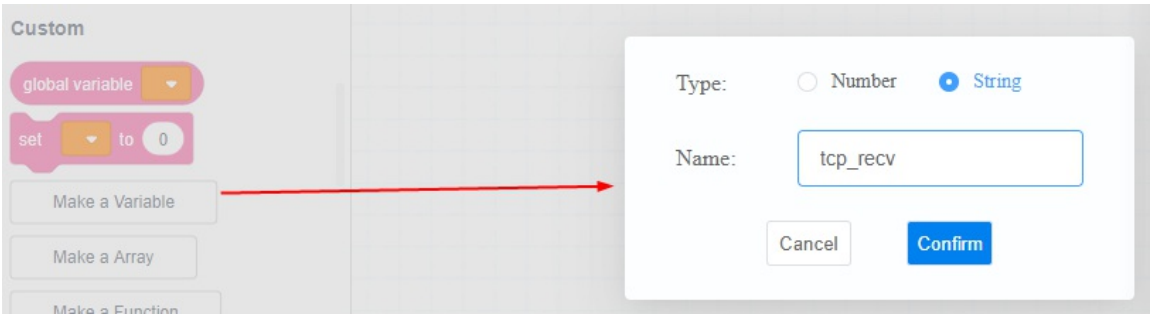
Steps for programming

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.

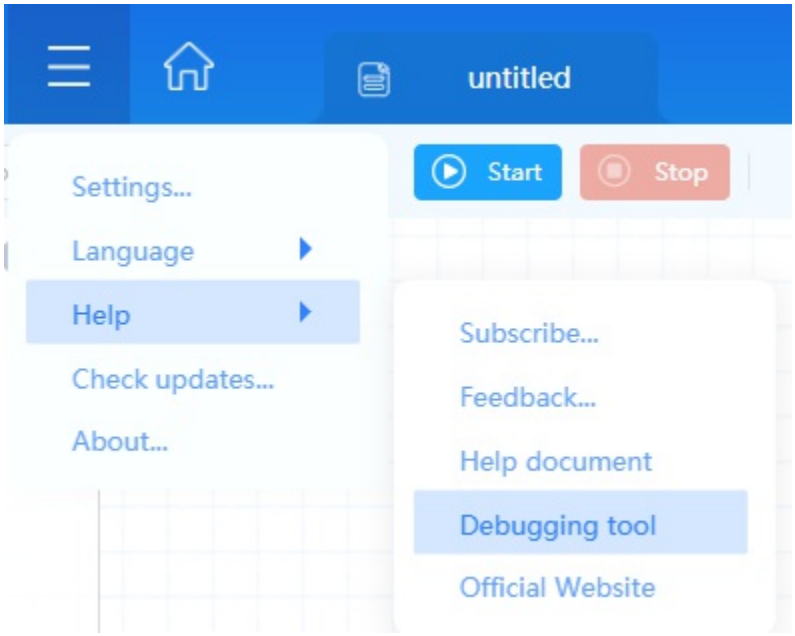
3. Wait for the client to connect and send the string. Save the received string to the string variable "tcp_recv". You need to create the string variable in advance.



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.

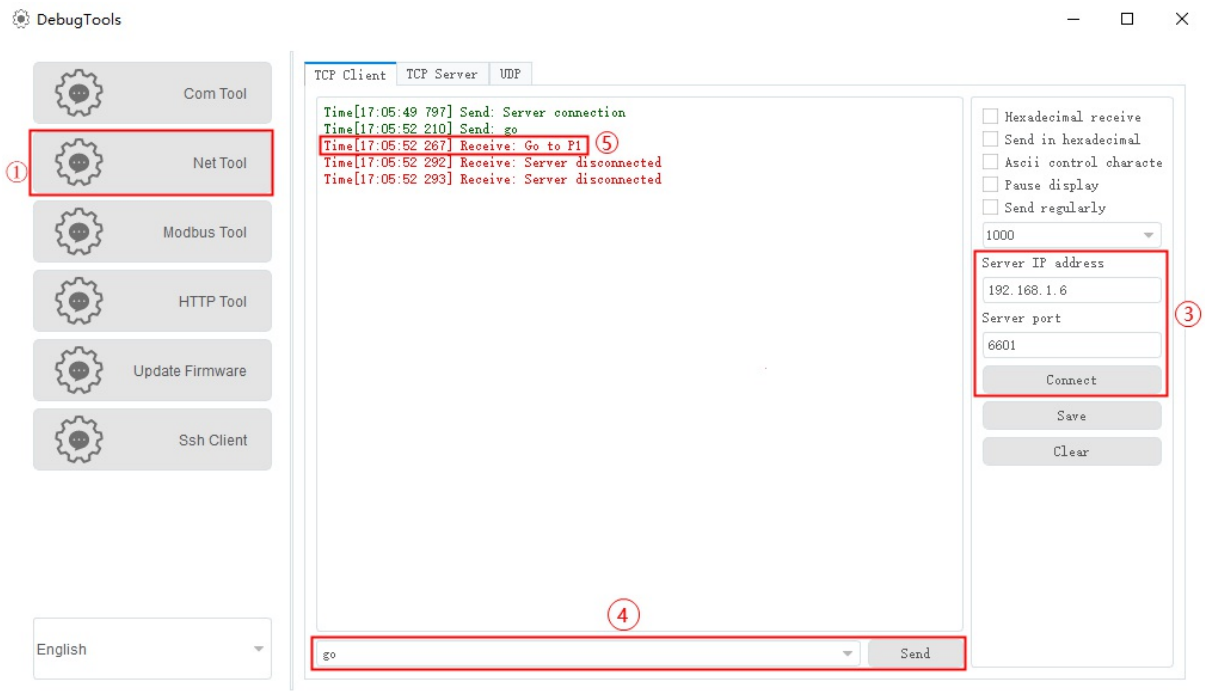
Run program

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.

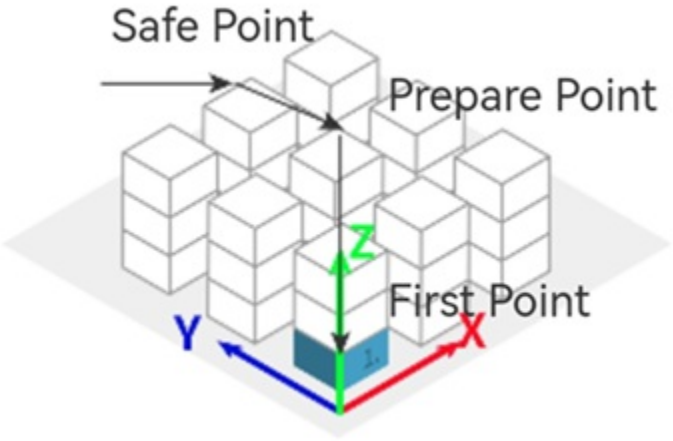


Palletize

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:

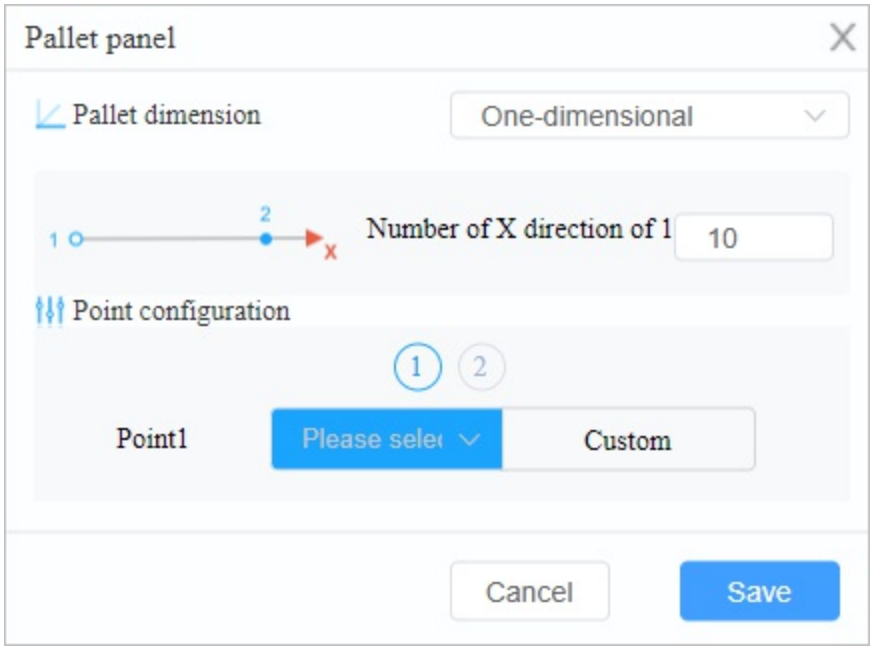


- Safe point (P1): A point the robot must move to when assembling or dismantling stacks for safe transition. It can be set to a point over the picking point.
- Picking point (P2).
- Preparation point and target point do not need to be taught one by one. Please refer to Configuring stack type.

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.

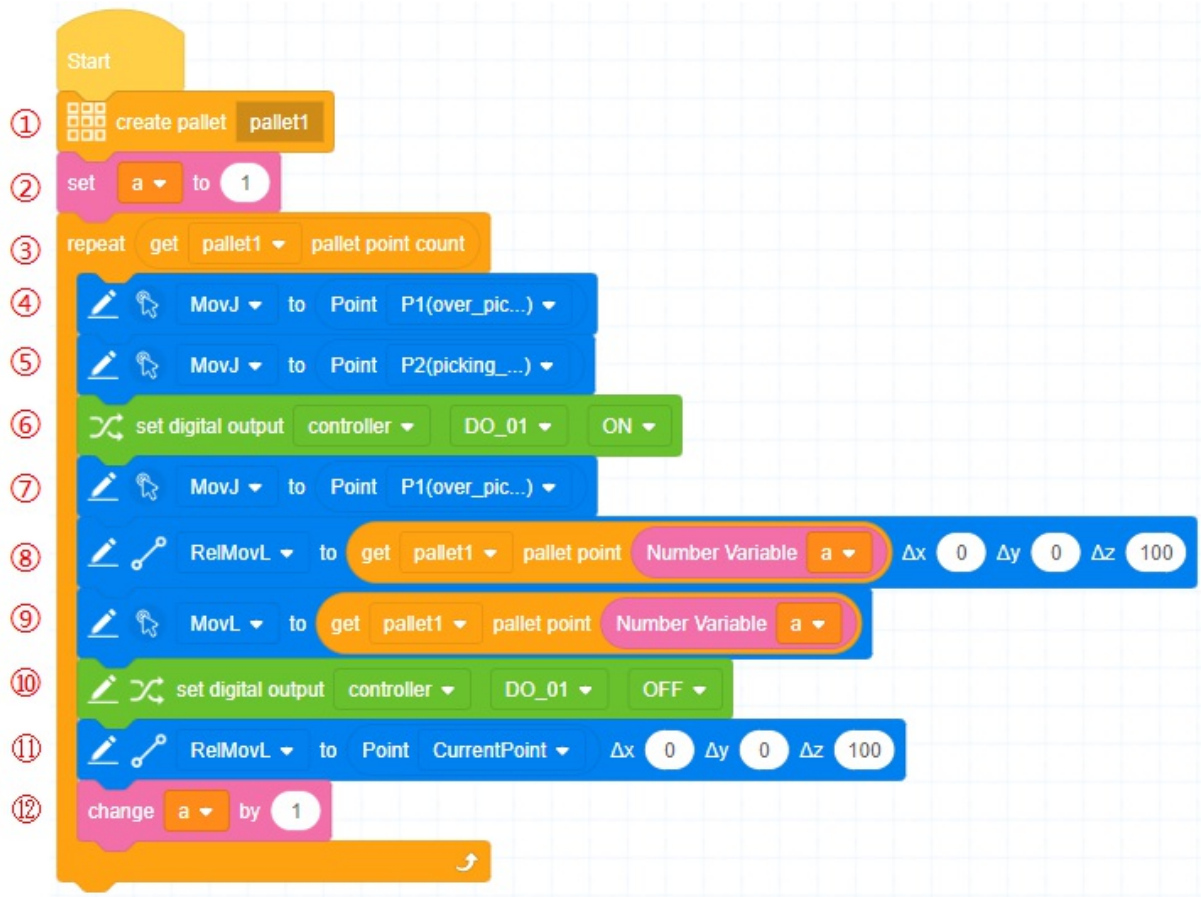
This section takes the three-dimensional stacking as an example. Here the number of materials in each direction is set to 10, so this demo contains 1000 materials.

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.

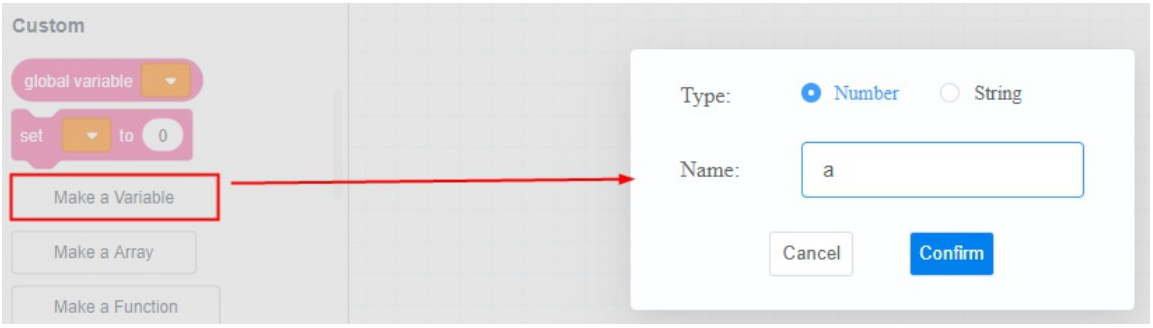
When configuring points, you can select the points that have been taught in the project, or you can click **Custom** to obtain the current point of the robot arm. The configured point icon will turn green.

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

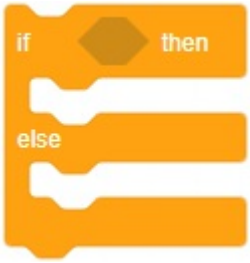
Description: It is used to be embedded inside the execution-repeating blocks. When the program runs to this block, it will directly end the repetition and execute the blocks after the execution-repeating block.

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".

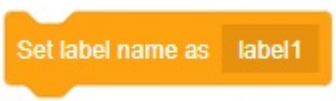
Parameter: Use other hexagonal blocks which return a Boolean value (true or false) as the parameter.

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

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

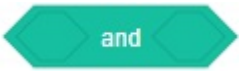
Description: Compare the parameters.

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

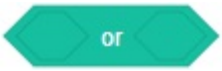


Description: Perform **and** operation to the parameters.

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 xommand



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

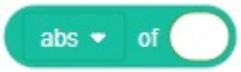
Round-off operation

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



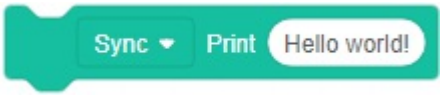
Description: Perform various Monadic operations to parameters.

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



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

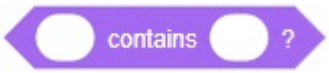
Description: Get the character in the specified position of the string.

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

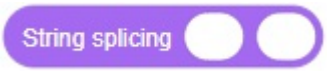


Description: Determine whether the first string contains the second string.

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

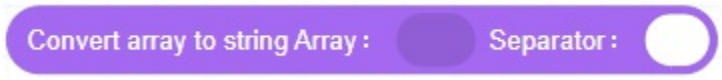
Compare two strings

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



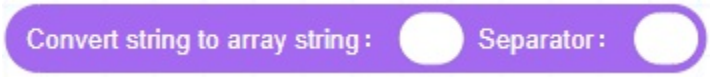
Description: Convert the specified array to a string, and the different array elements in the string are separated by the specified delimiter. For example, if the array is {1,2,3} and the delimiter is |, then the converted string is "1|2|3".

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

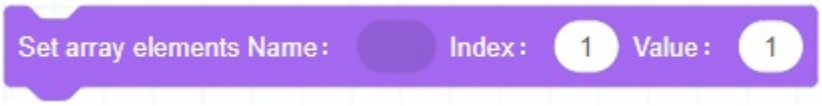
Description: Get multiple elements at the specified subscript position in the specified array. Get the element based on the step value within the range of the start and end subscripts.

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.

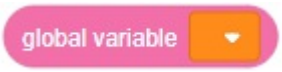
Parameter:

- 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

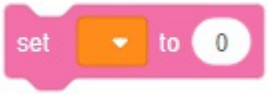


Description: Call global variables set in the control software.

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

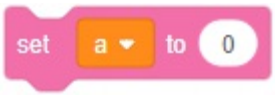
Click to create a variable. The variable name must start with a letter and cannot contain special characters such as Spaces. After creating at least one variable, you will see the following variable blocks in the block list.

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

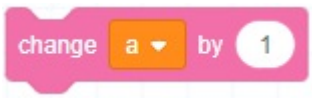


Description: Set the value of a specified number variable. Please note that the block for setting global variables and setting custom variables are the same in shape, but have slightly different functions.

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

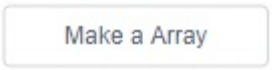
Set value of custom string variable

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



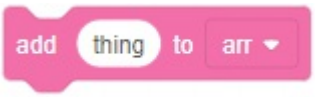
Click to create a custom array. The array name must start with a letter and cannot contain special characters such as Spaces. After creating at least one array, you will see the following array blocks in the block list.

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.

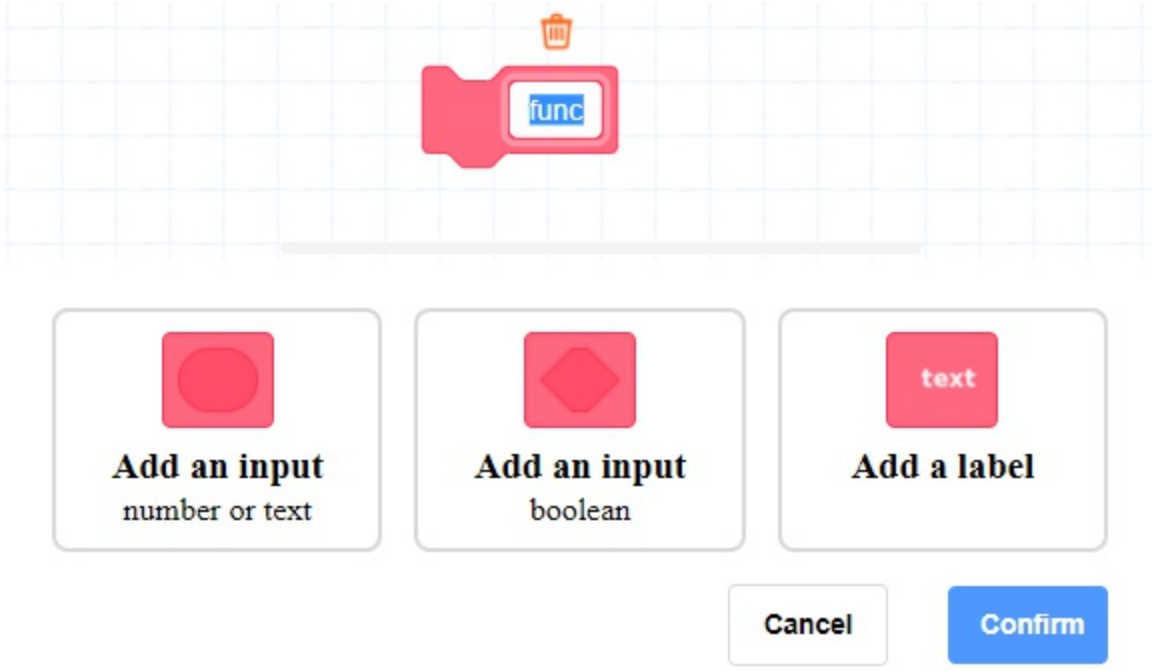
Parameter: Select an array.

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.

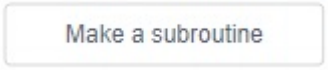
You need to program below the header block to define the function.

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.

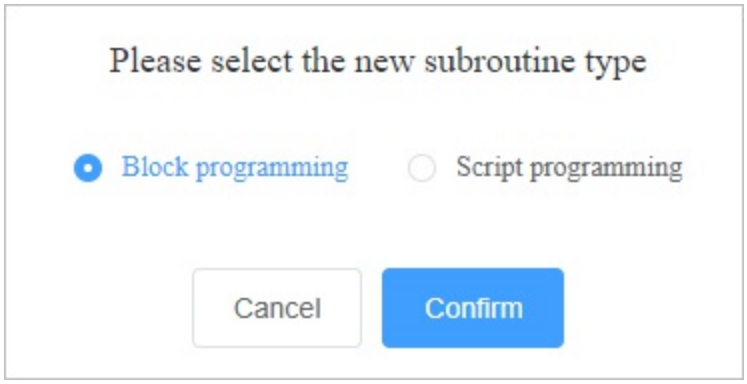
Custom function

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.

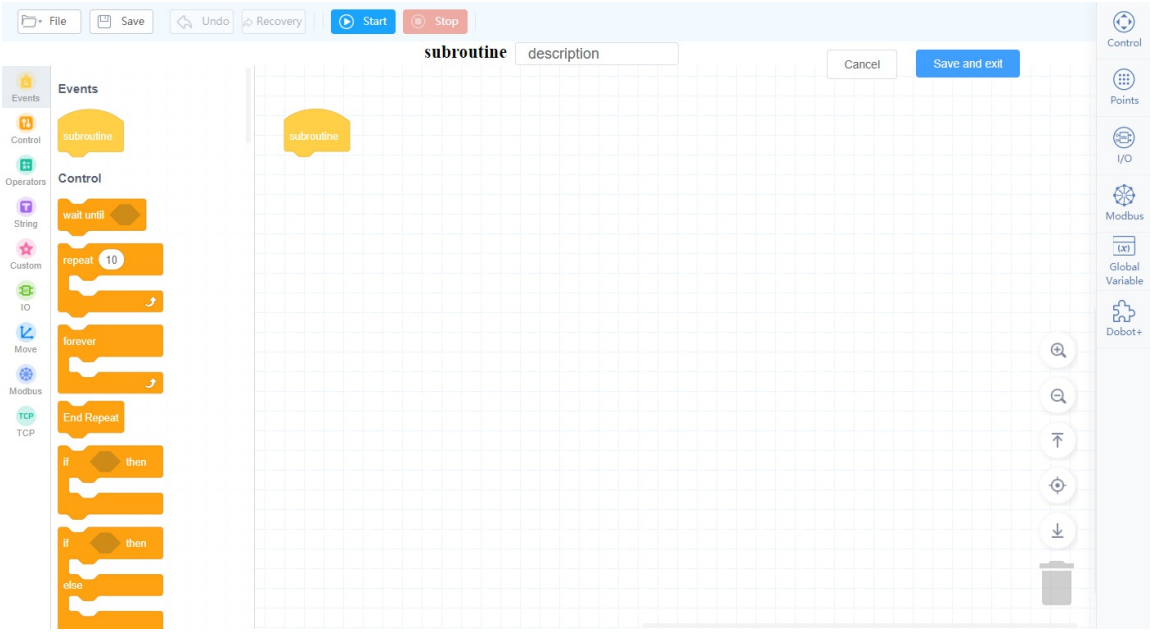
Create sub-routine



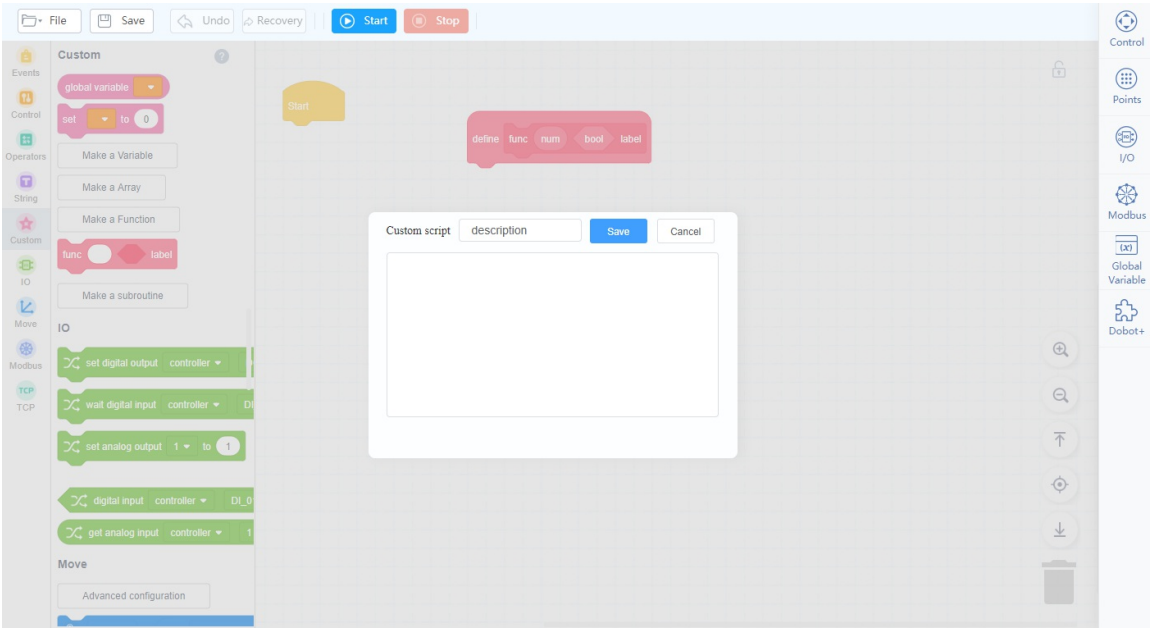
Click to create a new sub-routine. Blockly programming supports embedding and calling sub-routines, which can be blockly programming and script programming, with a maximum of two embedded levels. After the new sub-routine is successfully created, the corresponding sub-routine block will appear in the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

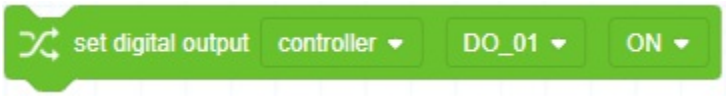
- Script sub-routine

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.

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.

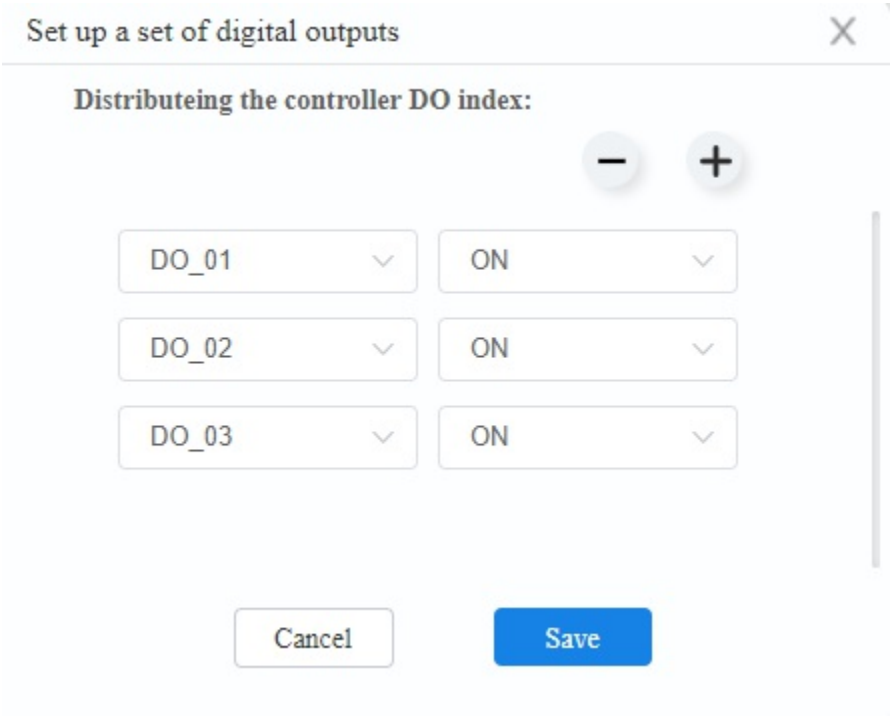
Parameter:

- 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

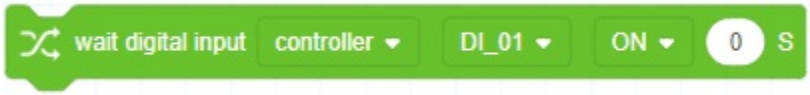
Description: Set a group of DO. You can drag the block to the programming area and click to set it.

Parameter:



- click + or - to increase or decrease the number of DO
- Select DO port index
- Select the output status (ON or OFF)

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 analog output

Description: Set the value of analog output port.

Parameter:

- Select analog output port index.
- Analog output value. You can directly enter the value in the blank, or use oval blocks that return numeric values.

Determine digital input status

Description: Determine whether the current status of the specified DI meets the condition.

Parameter:

- Select the position of DI port, including controller and tool.
- Select DI port index.
- Select the status which is regarded as true

Return: If the current status of the specified DI meets the condition, it returns **true**, otherwise, it returns **false**.

Get analog input

Description: Get the value of analog input.

Parameter:

- Select a position of analog input, including controller or tool.
- Select a port index.

Return: analog input value

Motion commands

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

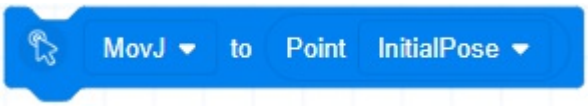
The motion blocks are all asynchronous commands, that is, after the command is successfully delivered, the next command will be executed without waiting for the robot to complete the current movement. You can use **sync** command if you need to wait for the delivered commands to be executed before executing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

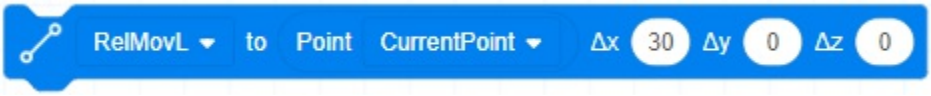


Description: Control the robot to move from the current position to the target point. After dragging the blocks to the programming area, double-click to perform advanced configuration. See Motion advanced configuration for details.

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. You can set the current point as the target point.

Parameter:

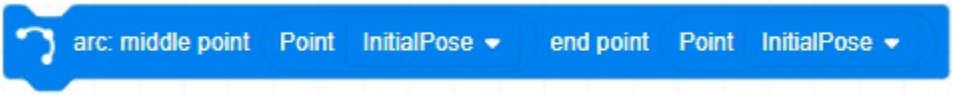
- Select a motion mode, including relative joint motion (RelMovJ) and relative linear motion (RelMovL).
- target point
- offset in the X-axis, Y-axis, and Z-axis direction relative to the target point under the Cartesian coordinate system. unit: mm

Joint offset motion

Description: Control the robot to move a specified offset from the current position.

Parameter: joint offset. unit: °

Arc motion

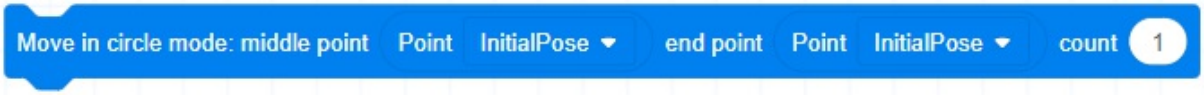


Description: Control the robot to move from the current position to a target position in an arc interpolated mode under Cartesian coordinate system. The coordinates of the current position should not be on the straight line determined by the intermediate point and the end point.

Parameter:

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

Circle motion



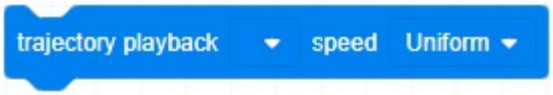
Description: Control the robot arm to move from the current position in an full-circle interpolated mode, and return to the current position after moving a specified number of circles. The coordinates of the current position should not be on the straight line determined by the intermediate point and 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. Value range: 1~ 999.

Trajectory playback



Description: Control the robot to play back the trajectory. The trajectory should be recorded in Trajectory playback process.

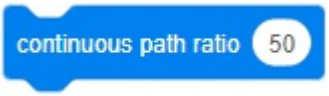
Parameter:

- Select a trajectory file.
- Select the moving speed in playback:
  - uniform speed
  - 0.25x speed
  - 0.5x speed
  - 1x speed
  - 2x speed

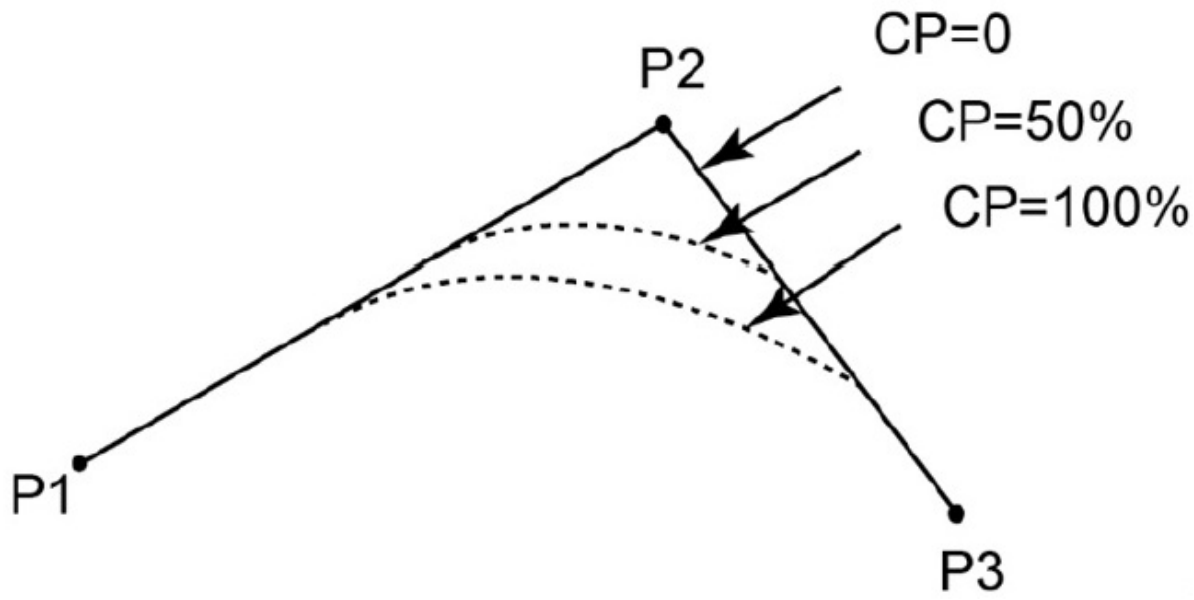
Sync command

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.

Set CP ratio

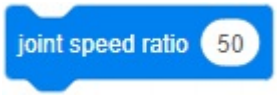


Description: Set the continuous path ratio in motion, that is, when the robot moves from the starting point to the end point via the intermediate point, whether it passes the intermediate point through right angle or in curve, as shown below.



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

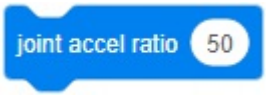
Set joint speed ratio



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.

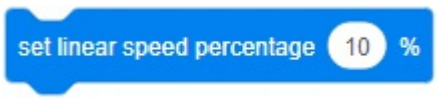
Set joint acceleration ratio



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 linear speed ratio

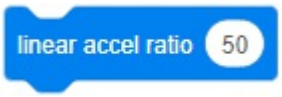


Description: Set the speed ratio of lineal and arc motion.

Parameter:

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

Set linear acceleration ratio

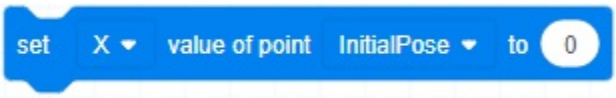


Description: Set the acceleration ratio of lineal and arc motion.

Parameter:

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

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 coordinate value.

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.

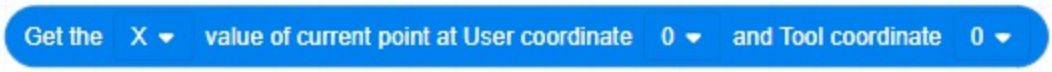
Parameter:

- Select the point to get the coordinate value.

- Select the coordinate dimension.

Return: Value of the specified Cartesian coordinate axis

Get coordinates of current position



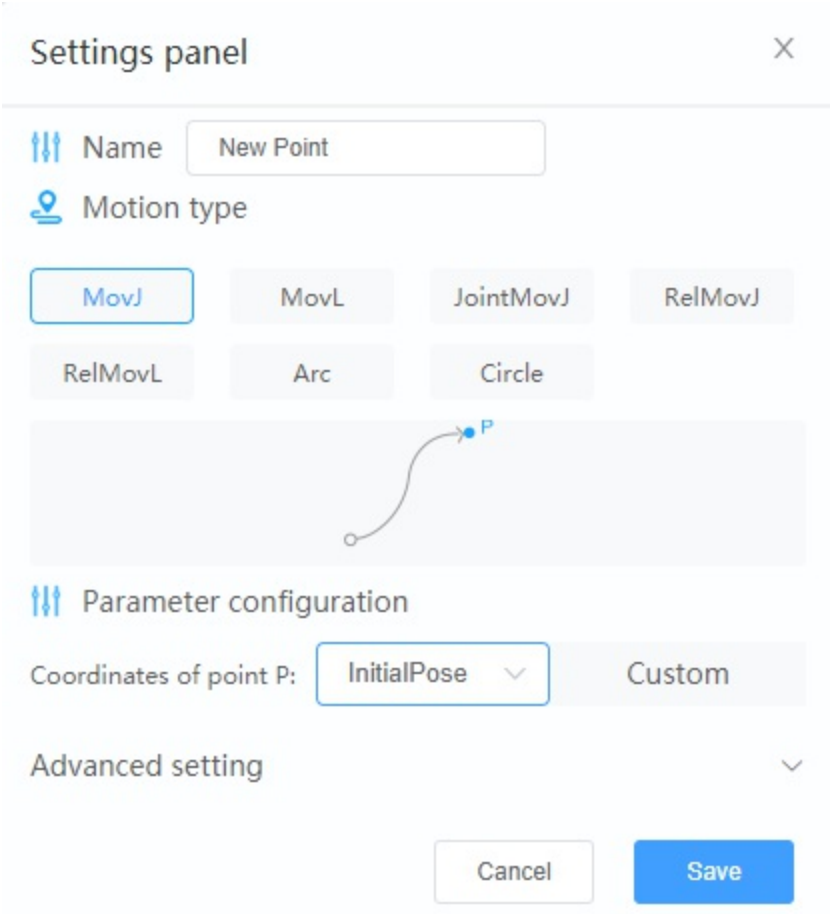
Description: Get the value of the TCP current position in the specified coordinate axis under the specified coordinate system.

Parameter:

- Select the coordinate dimension.
- Select the user coordinate system and tool coordinate system. The returned value will be transformed to the value in the corresponding coordinate system.

Return: Value of the TCP current position in the specified coordinate axis under the specified coordinate system.

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.

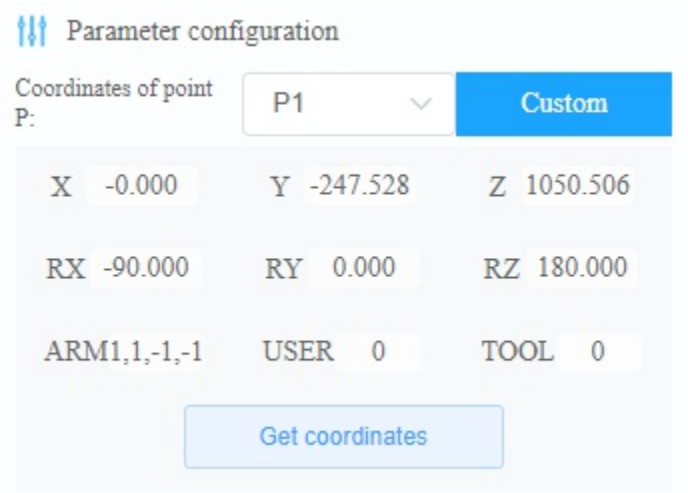
Actual robot speed/acceleration = percentage set in commands × speed/acceleration in playback settings × global speed ratio.

MovJ

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

Basic setting:

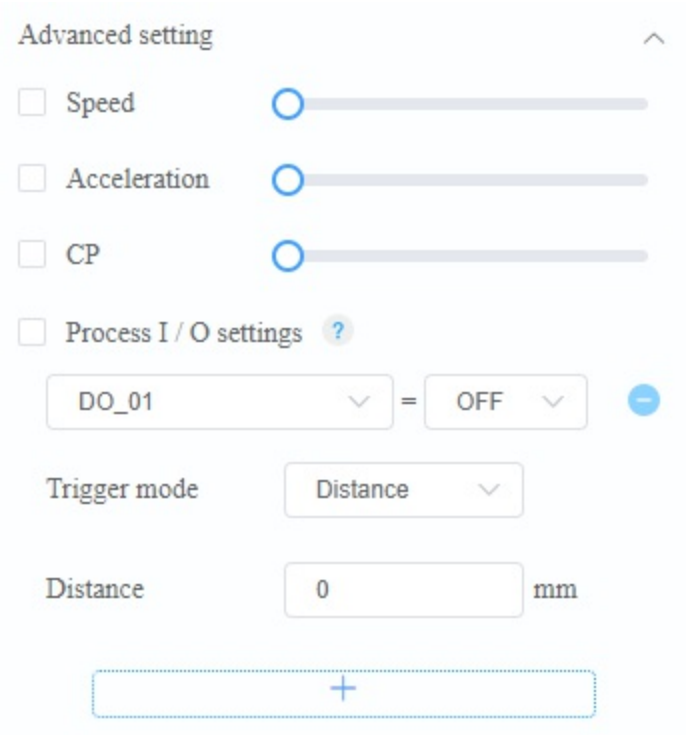
P: target point, which can be selected here after being added in the Point page, or defined in this page.



Advanced setting:

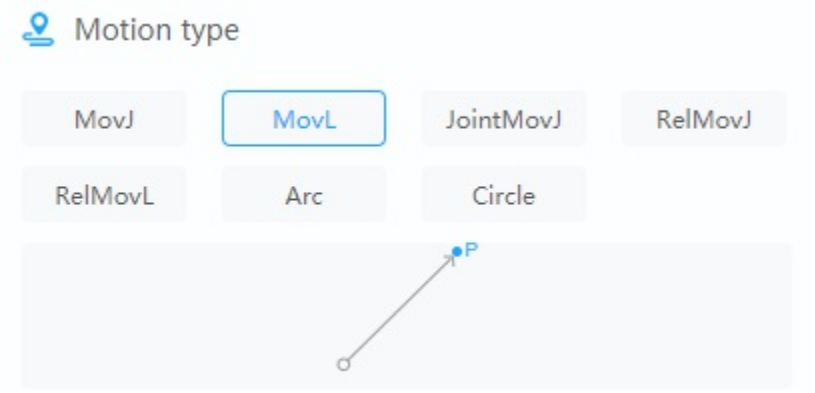
Select and configure the advanced parameters as required.

- 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.

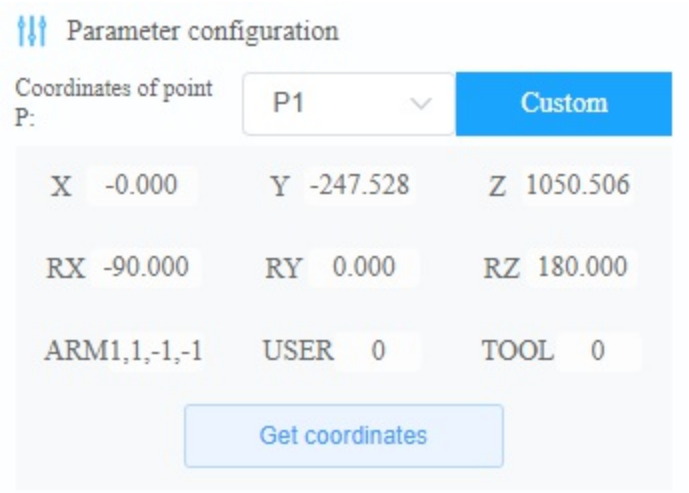


MovL

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



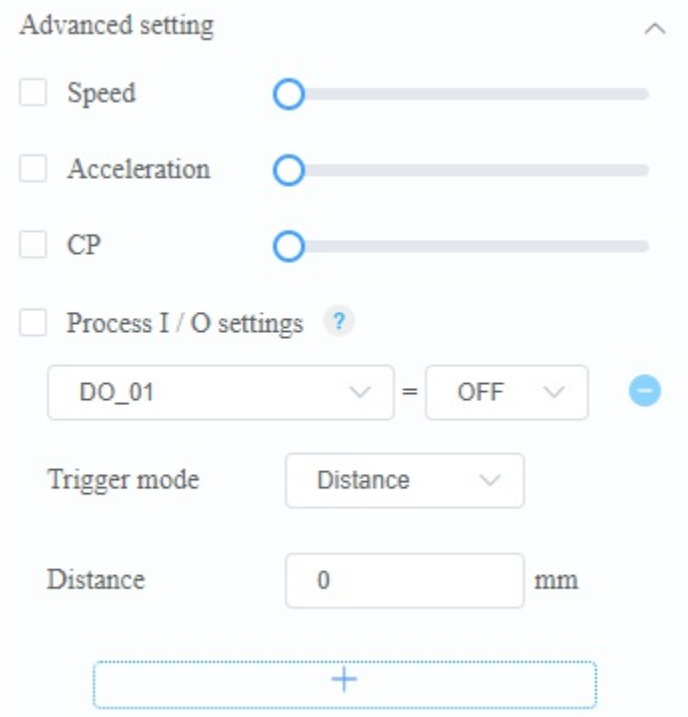
Basic setting: P: target point, which can be selected here after being added in the Point page, or defined in this page.



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.



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

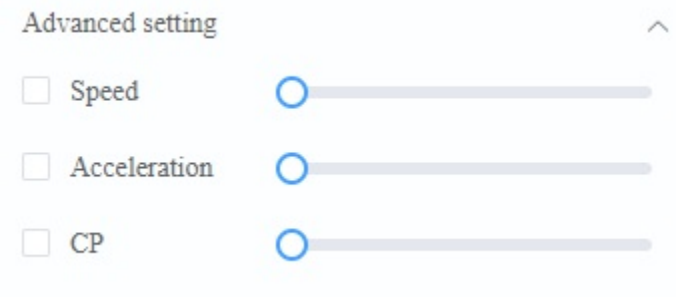
Motion mode: Move from the current position to the target offset position under the Cartesian coordinate system in a joint-interpolated mode.

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.

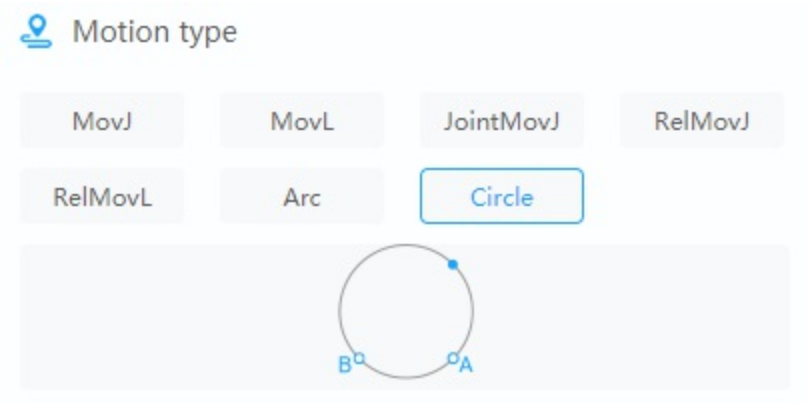
Advanced setting:

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.

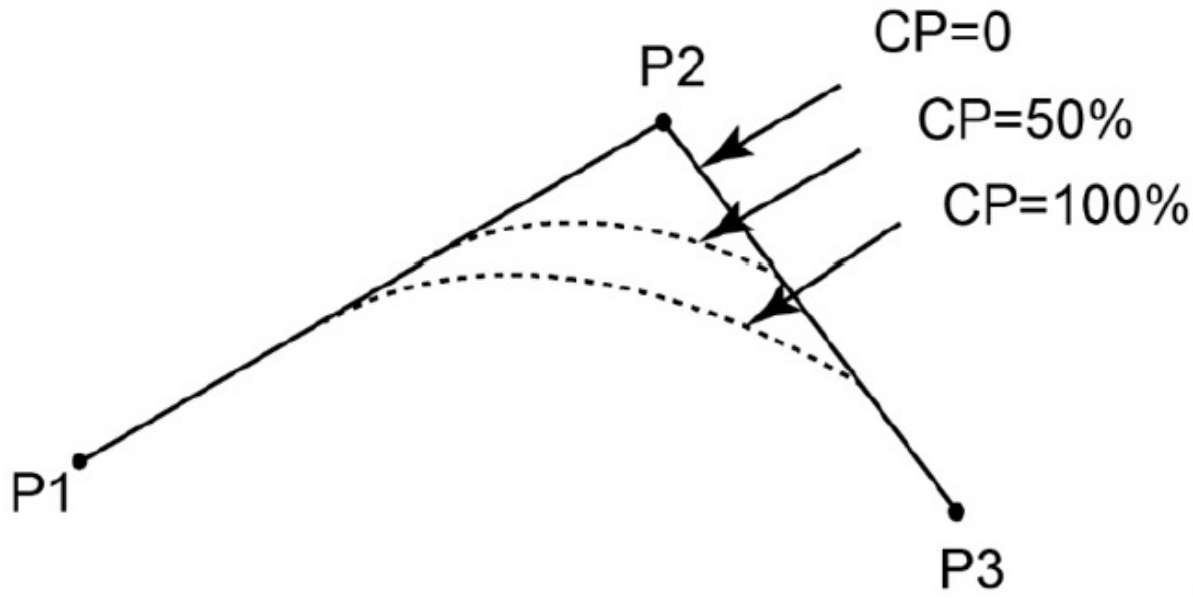
Advanced setting:

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.



Modbus commands

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

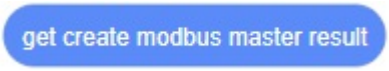
Create Modbus master

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:

- 0: Modbus master has been created successfully.
- 1: As there are 4 created master stations, a new one failed to be created.
- 2: Modbus master failed to be initialized. It is recommended to check whether the IP, port and network is normal.
- 3: Modbus slave failed to be connected. It is recommended to check whether the slave is established properly and whether the network is normal.

Wait for input register

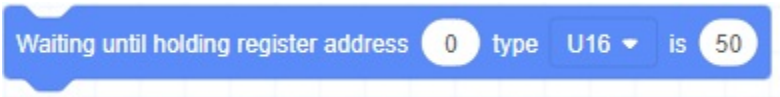
Description: Wait for the value of the specified address of input register to meet the condition before executing the next command.

Parameter:

- Address: Starting address of the input registers. Value range: 0~9998.

- 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

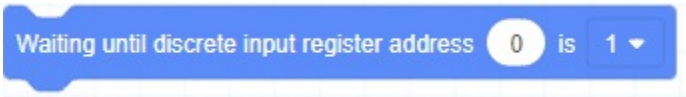


Description: Wait for the value of the specified address of holding register to meet the condition before executing the next command.

Parameter:

- Address: Starting address of the holding registers. Value range: 0~9998.
- 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

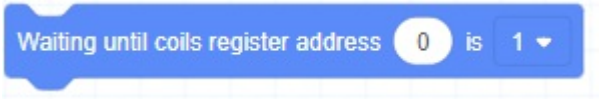


Description: Wait for the value of the specified address of discrete input register to meet the condition before executing the next command.

Parameter:

- Address: Starting address of the discrete input registers. Value range: 0~9999.
- 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: 0~9999.
- condition that the value is required to meet

Get input register

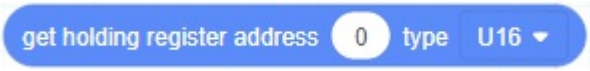
Description: Get the value of the specified address of input register.

Parameter:

- Address: Starting address of the input registers. Value range: 0~9998.
- 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



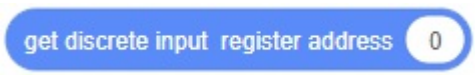
Description: Get the value of the specified address of holding register.

Parameter:

- Address: Starting address of the holding registers. Value range: 0~9998.
- 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

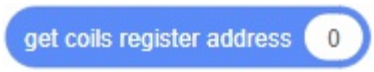


Description: Get the value of the specified address of discrete input register.

Parameter: Starting address of the discrete input register. Value range: 0~9999.

Return: discrete input value

Get coil register

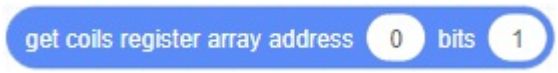


Description: Get the value of the specified address of coil register.

Parameter: Starting address of the coil register. Value range: 0~9999.

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: 0~9999.
- Number of register bits. The maximum value is 216 (CR series) or 984 (Nova series) when communicating with the slave via the tool I/O interface, and 2008 for the other scenarios.

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:

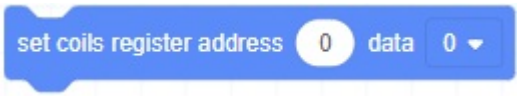
- Starting address of the input registers. Value range: 0~9998.
- Number of values to be read.
- Data type
  - U16: 16-bit unsigned integer (two bytes, occupy one register). Up to 13 values (CR series) or 61 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 125 values are read continuously for the other scenarios.
  - U32: 32-bit unsigned integer (four bytes, occupy two registers). Up to 6 values (CR series) or 30 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 62 values are read continuously for the other scenarios.
  - F32: 32-bit single-precision float number (four bytes, occupy two registers). Up to 6 values (CR

series) or 30 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 62 values are read continuously for the other scenarios.

- F64: 64-bit double-precision float number (eight bytes, occupy four registers). Up to 3 values (CR series) or 15 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 31 values are read continuously for the other scenarios.

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

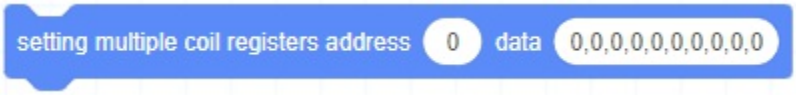


Description: Write the value to the specified address of coil register.

Parameter:

- Starting address of the coil register. Value range: 6~9999.
- 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: 0~9999.
- 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. Up to 440 values are written at a time when communicating with the slave via the tool I/O interface, and up to 1976 values are written at a time for the other scenarios.

Set holding register

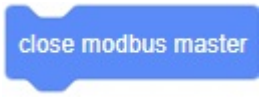
Description: Write the value to the specified address of holding register.

Parameter:

- Starting address of the input registers. Value range: 0~9998.
- Value to be written, which should correspond to the selected data type.

- Data type
  - U16: 16-bit unsigned integer (two bytes, occupy one register). Up to 27 values are written continuously when communicating with the slave via the tool I/O interface, and up to 123 values are written continuously for the other scenarios.
  - U32: 32-bit unsigned integer (four bytes, occupy two registers). Up to 13 values are written continuously when communicating with the slave via the tool I/O interface, and up to 61 values are written continuously for the other scenarios.
  - F32: 32-bit single-precision float number (four bytes, occupy two registers). Up to 13 values are written continuously when communicating with the slave via the tool I/O interface, and up to 61 values are written continuously for the other scenarios.
  - F64: 64-bit double-precision float number (eight bytes, occupy four registers). Up to 6 values are written continuously when communicating with the slave via the tool I/O interface, and up to 30 values are written continuously for the other scenarios.

Close Modbus master

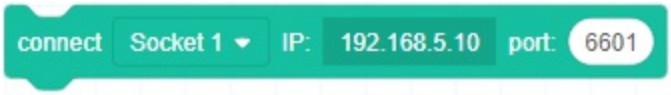


Description: Close the Modbus master, and disconnect from all slaves.

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.

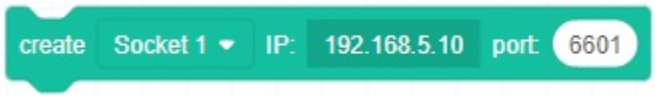
Get result of connecting SOCKET

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.

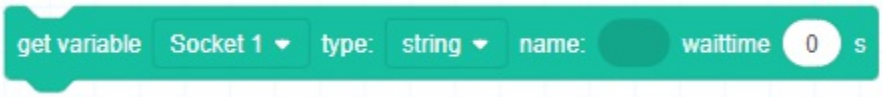
Return: It returns 0 for successful creation, and 1 for failing to be created.

Close SOCKET

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

Parameter: Select SOCKET index.

Get variables

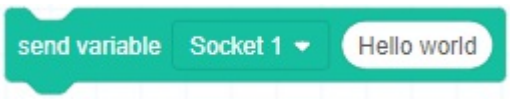


Description: Get variables through TCP communication and save it.

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

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 operator

- 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                --Create variable a
b=5                    --Create variable b
print(a==b)            --Determine whether a is equal to b: false
print(a~=b)            --Determine whether a is not equal to b: true
print(a<=b)            --Determine whether a is less than or equal to b: false
print(a>=b)            --Determine whether a is greater than or equal to b: true
print(a<b)            --Determine whether a is less than b: false
print(a>b)            --Determine whether a is greater than b: true
```

Logical 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;
--[ Check conditions --]
if(a == 100)
then
   --[Execute the following code if the condition is true--]
   if(b == 200)
   then
       --[Execute the following code if the condition is true--]
      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

General description

Motion mode

The motion modes of the robot include the following types.

Joint motion

The robot arm plans the motion of each joint according to the difference between the current joint angle and the joint angle of the target point, so that each joint completes the motion at the same time. The joint motion does not constrain the trajectory of TCP (Tool Center Point), which is generally not a straight line.

As the joint motion is not limited by the singularity position (see the corresponding hardware guide for details), if there is no requirement for the motion trajectory, or the target point is near the singularity position, it is recommended to use joint motion.

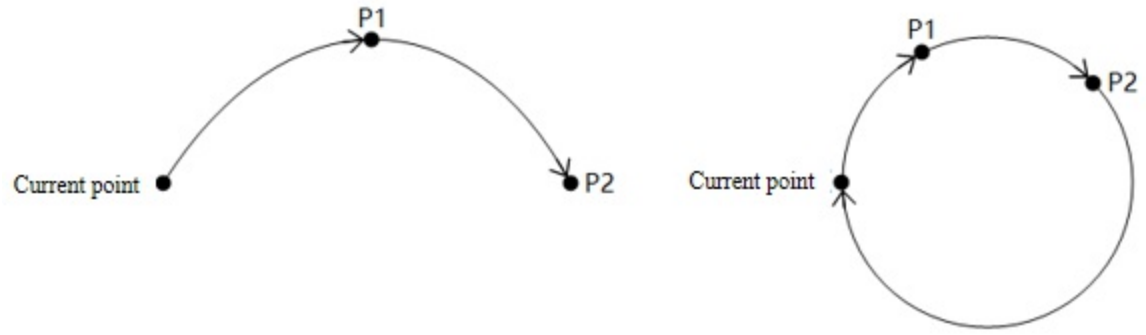
Linear motion

The robot arm plans the motion trajectory according to the current posture and the posture of the target point, so that the TCP motion trajectory is a straight line, and the posture of the end changes uniformly during the movement.

When the trajectory may pass through the singularity position, an error will be reported when the linear motion command is delivered to the robot arm. It is recommended to re-plan the point or use joint motion near the singularity position.

Arc motion

The robot arm determines an arc or a circle through three non-collinear points: the current position, P1 and P2. The posture of the end of the robot arm during the movement is calculated by the posture interpolation of the current point and P2, and the posture of P1 is not included in the operation (i.e, the posture of the robot arm when it reaches P1 during the movement may be different from the taught posture).



When the trajectory may pass through the singularity position, an error will be reported when the arc motion command is delivered to the robot arm. It is recommended to re-plan the point or use joint motion near the singularity position.

Point parameters

The point parameters in the commands generally use the taught point in the Points page, and it can be called directly using the name of taught point. The taught point is actually saved as a constant in the background in the following format.

```
--[[
    name: name of taught point
    joint: joint coordinates of taught point
    tool: tool coordinate system index in hand guiding
    user: user coordinate system index in hand guiding
    pose: posture variable value of taught point
--]]
{
    name = "name",
    joint = {j1, j2, j3, j4, j5, j6},
    tool = index,
    user = index,
    pose = {x, y, z, rx, ry, rz}
}
```

In addition to the taught point, users can also define the Cartesian coordinate points or joint coordinate points as the point parameters, see the parameter description of each command for details.

Cartesian coordinate points:

```
{pose = {x, y, z, rx, ry, rz} }
```

Joint coordinate points:

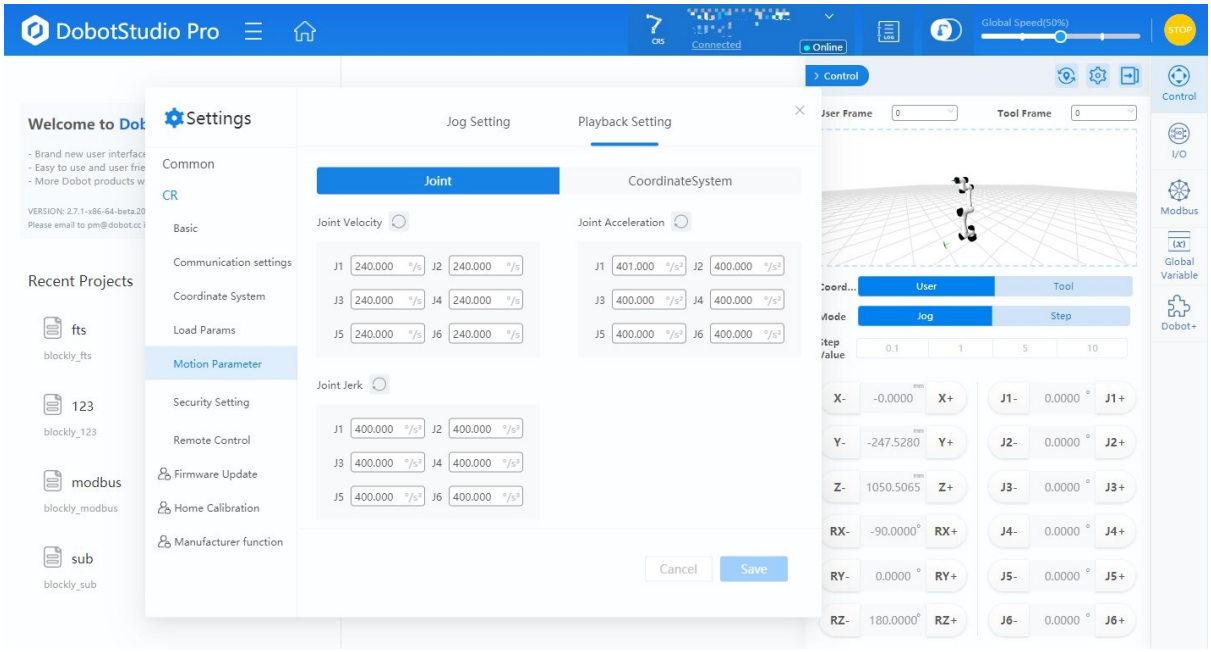
```
{joint = {j1, j2, j3, j4, j5, j6} }
```

Speed parameters

The Speed/SpeedS and Accel/AccelS in the optional parameters are used to specify the speed and acceleration rate when the robot arm executes the motion command.

```
Robot actual motion speed/acceleration = maximum speed/acceleration x global speed x command r
ate
```

The maximum acceleration/velocity is limited by **Playback Setting**, which can be viewed and modified in "Motion parameter" page in the software.



The global speed rate can be set through the control software (upper right corner of the figure above) or the SpeedFactor command.

The command rate is carried by the optional parameters of the motion commands. When the acceleration/velocity rate is not specified through the optional parameters, the value set in the motion parameters is used by default (see Speed, Accel, SpeedS, AccelS, SpeedR, AccelR commands for details, and the default value is 50 when the command setting is not called).

Example:

```
Accel(50) -- Set the default acceleration of joint motion to 50%
Speed(60) -- Set the default speed of joint motion to 60%
AccelS(70) -- Set the default acceleration of linear motion to 70%
SpeedS(80) -- Set the default speed of linear motion to 80%

-- Global speed rate: 20%

Go(P1) -- Move to P1 at the acceleration of (maximum joint acceleration x 20% x 50%) and speed
 of (maximum joint speed x 20% x 60%) through the joint motion
Go(P1,"Speed = 40, Accel = 90") -- Move to P1 at the acceleration of (maximum joint accelerati
```

```
on x 20% x 30%) and speed of (maximum joint speed x 20% x 80%) through the joint motion

Move(P1) -- Moves to P1 at the acceleration of (maximum Cartesian acceleration x 20% x 70%) an
d speed of (maximum Cartesian speed x 20% x 80%) in the linear mode
Move(P1,"SpeedS = 40, AccelS = 90")  -- Moves to P1 at the acceleration of (maximum Cartesian
acceleration x 20% x 40%) and speed of (maximum Cartesian speed x 20% x 90%) in the linear mode
```

Immediate command and Queue command

The commands provided by Dobot are divided into immediate command and queue command.

- Immediate command is executed immediately once it is issued.
- Queue command is not executed immediately after it is issued, but enters the background algorithmic queue and wait to be executed.

The background algorithm queue will not block the thread. If the immediate command is called after the queue command, the immediate command may be executed before the queue command is completed, such as the following example:

```
Go(P1)  -- Queue command
local currentPose = GetPose()  -- Immediate command
```

In this example, the point that GetPose() gets is not P1, but a process point in motion.

If you want to make sure that all the previous commands have been executed when executing the immediate command, you can call the Sync ( ) command before calling the immediate command, which will block the program from executing until all the previous commands are fully executed, such as the following example:

```
Go(P1)  -- Queue command
Sync()
local currentPose = GetPose()  -- Immediate command
```

In this example, the point that GetPose() gets is P1.

Motion

The motion commands is used to control the movement of the robot arm. The commands in this group are all queue commands.

Go

Command:

```
Go(P,"User=1 Tool=2 CP=1 Speed=50 Accel=20 SYNC=1")
```

Description:

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

- User: user coordinate system index, which can be used here after being added in Settings.
- Tool: tool coordinate system index, which can be used here after being added in Settings.
- 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:

```
Go(P1)
```

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

Move

Command:

```
Move(P,"User=1 Tool=2 CP=1 SpeedS=50 AccelS=20 SYNC=1 STOP=1")
```

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:

- User: user coordinate system index, which can be used here after being added in the Settings.
- Tool: tool coordinate system index, which can be used here after being added in the Settings.
- CP: set continuous path in motion (see CP command in Motion parameter), range: 0 - 100.
- SpeedS: velocity rate, range: 1 - 100.
- AccelS: 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.
- STOP: abort the motion. Specify the DI index. When the specified DI changes to ON during the execution of this motion command, the robot immediately aborts the current motion and directly executes the next command. When this parameter is set, the SYNC parameter setting is invalid and defaults to 1.

Example:

```
Move(P1)
```

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

```
Move(P1,"STOP=3")
Move(P2)
```

The robot arm moves to P1 in a linear mode. When DI3 changes into ON during the motion, it stops moving to P1 immediately and turns to move to P2 in a linear mode.

MoveJ

Command:

```
MoveJ(P,"CP=1 Speed=50 Accel=20 SYNC=1")
```

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.
- 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:

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

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

Circle3

Command:

```
Circle3(P1,P2,Count,"User=1 Tool=2 CP=1 SpeedS=50 AccelS=20 SYNC=1")
```

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:

- User: user coordinate system index, which can be used here after being added in Settings.
- Tool: tool coordinate system index, which can be used here after being added in Settings.
- CP: set continuous path in motion (see CP command in Motion parameter), range: 0 - 100.
- SpeedS: velocity rate, range: 1 - 100
- AccelS: vcceleration 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:

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

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

Arc3

Command:

```
Arc3(P1,P2,"User=1 Tool=2 CP=1 SpeedS=50 AccelS=20 SYNC=1")
```

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:

- User: user coordinate system index, which can be used here after being added in the Settings.
- Tool: tool coordinate system index, which can be used here after being added in the Settings.

- CP: set continuous path in motion (see CP command in Motion parameter), range: 0 - 100.
- SpeedS: velocity rate, range: 1 - 100
- AccelS: 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:

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

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

GoIO

Command:

```
GoIO(P,{ {Mode,Distance,Index,Status},{Mode,Distance,Index,Status}...}, "User=1 Tool=2 CP=1 Speed=50 Accel=20 SYNC=1")
```

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:

- User: user coordinate system index, which can be used here after being added in Settings.
- Tool: tool coordinate system index, which can be used here after being added in Settings.
- 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:

```
GoIO(P1, { {0, 10, 2, 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.

MoveIO

Command:

```
MoveIO(P,{ {Mode,Distance,Index,Status},{Mode,Distance,Index,Status}},"User=1 Tool=2 CP=1 Speed
dS=50 AccelS=20 SYNC=1")
```

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:

- User: user coordinate system index, which can be used here after being added in Settings.
- Tool: tool coordinate system index, which can be used here after being added in Settings.
- 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:

```
MoveIO(P1, { {0, 10, 2, 1} })
```

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

MoveJIO

Command:

```
MoveJIO(P,{ {Mode,Distance,Index,Status},{Mode,Distance,Index,Status}...}, "User=1 Tool=2 CP=1
 Speed=50 Accel=20 SYNC=1")
```

Description:

Move from the current position to the target joint angle through joint motion, and set the status of digital output port when the robot is moving.

Required parameter:

- P: target joint angle.
- 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 rate 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:

```
local P = {joint={0,-0.0674194,0,0,0,0}}
MoveJIO(P, { {0, 10, 2, 1} })
```

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

ServoJ

Command:

```
ServoJ(P,"t=0.1 lookahead_time=50 gain=500")
```

Description:

The dynamic following command based on joint space is generally used for the stepping function of online control to realize dynamic following by cyclic calling. The calling frequency is recommended to be set to 33 Hz, that is, the interval of cyclic calling is 30 ms.

NOTICE

- This command is not affected by global speed.
- When the joint angle of the target point is quite different from that of the current point, and the motion time t is small, the joint movement speed will be too fast, which may lead to servo error or even power outage of the robot.
- Before using the command, it is recommended to carry out velocity planning for the running points, and issue the planned points according to the fixed time interval t to ensure that the robot can track the target points smoothly.

Required parameter:

P: target joint angle.

Optional parameter:

- t: running time of the point, unit: s, range: [0.02,3600.0], default value: 0.1.
- lookahead_time: lookahead_time, similar to the D term in PID control. Scalar, no unit, range: [20.0,100.0], default value: 50.
- gain: proportional gain of target position, similar to the P term in PID control. Scalar, no unit, range: [200.0,1000.0], default value: 500.

The lookahead_time and gain parameters jointly determine the response time and trajectory smoothness of the robot arm. A smaller lookahead_time value or a larger gain value can make the robot arm respond quickly, but may cause instability and jitter.

Example:

```
ServoJ({joint={0,-0.0674194,0,0,0,0}},"t=0.1 lookahead_time=50 gain=500")
```

ServoP

Command:

```
ServoP(P)
```

Description:

The dynamic following command based on Cartesian space is generally used for the stepping function of online control to realize dynamic following by cyclic calling. The calling frequency is recommended to be set to 33 Hz, that is, the interval of cyclic calling is 30 ms.

Parameter:

P: target Cartesian point.

Example:

```
ServoP({pose={-500,100,200,150,0,90} })
```

Motion parameter

The motion parameters are used to set or obtain relevant motion parameters of the robot. The commands in this group are queue commands except GetPose, GetAngle, and SpeedFactor.

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:

```
Go(P1)
Go(P2)
Sync()
```

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

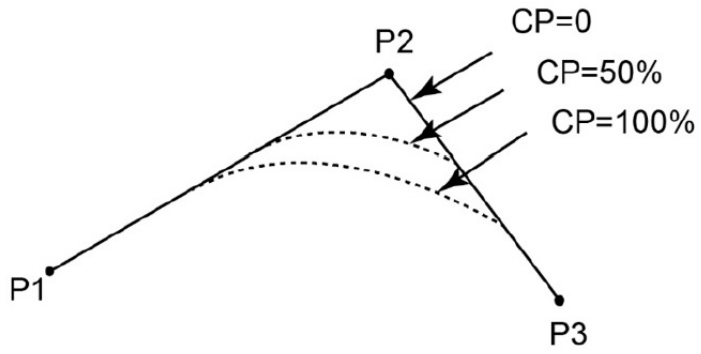
CP

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: 1 - 100

Example:

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

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

SpeedFactor

Command:

```
SpeedFactor(R)
```

Description:

Set global speed rate.

Required parameter:

R: velocity rate, range: 1 - 100.

Example:

```
SpeedFactor(20)
```

Set the global speed rate to 20%.

Speed

Command:

```
Speed(R)
```

Description:

Set the velocity rate of joint motion.

Required parameter:

R: velocity rate, range: 1 - 100.

Example:

```
Speed(20)
Go(P1)
```

The robot arm moves to P1 with 20% velocity rate.

Accel

Command:

```
Accel(R)
```

Description:

Set the acceleration rate of joint motion.

Required parameter:

R: acceleration rate, range: 1 - 100.

Example:

```
Accel(50)
Go(P1)
```

The robot arm moves to P1 with 50% acceleration rate.

SpeedS

Command:

```
SpeedS(R)
```

Description:

Set the space motion (X/Y/Z) velocity rate for linear and arc motion.

Required parameter:

R: velocity rate, range: 1 - 100.

Example:

```
SpeedS(20)
Move(P1)
```

The robot arm moves to P1 with 20% velocity rate.

AccelS

Command:

```
AccelS(R)
```

Description:

Set the space motion (X/Y/Z) acceleration rate for linear and arc motion.

Required parameter:

R: acceleration rate, range: 1 - 100.

Example:

```
AccelS(50)
Move(P1)
```

The robot arm moves to P1 with 50% acceleration rate.

SpeedR

Command:

```
SpeedR(R)
```

Description:

Set the pose motion (RX/RY/RZ) velocity rate for linear and arc motion.

Required parameter:

R: velocity rate, range: 1 - 100.

Example:

```
SpeedR(20)
Move(P1)
```

The robot arm moves to P1 with 20% velocity rate.

AccelR

Command:

```
AccelR(R)
```

Description:

Set the pose motion (RX/RY/RZ) acceleration rate for linear and arc motion.

Required parameter:

R: acceleration rate, range: 1 - 100.

Example:

```
AccelR(50)
Move(P1)
```

The robot arm moves to P1 with 50% acceleration rate.

GetPose

Command:

```
GetPose("user=0 tool=0")
```

Description:

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

Optional parameter:

- user: user coordinate system index corresponding to the posture, which needs to be added in the control software first. If not set, the global user coordinate system is used.
- tool: tool coordinate system index corresponding to the posture, which needs to be added in the control software first. If not set, the global tool coordinate system is used.

Return:

Cartesian coordinates of the current posture.

Example:

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

The robot arm 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()
Go(P1)
MoveJ(currentAngle)
```

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

CheckGo

Command:

```
CheckGo(P)
```

Description:

Check the operatibility of joint motion commands.

Required parameter:

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

Return:

Check result.

- 0: no error
- 16: End point closed to shoulder singularity point
- 17: End point inverse kinematics error with no solution
- 18: Inverse kinematics error with result out of working area
- 22: Arm orientation error
- 26: End point closed to wrist singularity point
- 27: End point closed to elbow singularity point
- 29: Speed parameter error
- 32: Shoulder singularity point in trajectory
- 33: Inverse kinematics error with no solution in trajectory
- 34: Inverse kinematics error with result out of working area in trajectory
- 35: Wrist singularity point in trajectory
- 36: Elbow singularity point in trajectory
- 37: Joint angle is changed over 180 degree

Example:

```
local status=CheckGo (P1)
if(status==0)
    then
        Go(P1)
    end
```

Check whether the robot arm can reach P1 through joint motion default setting. If it can, move to P1 through joint motion.

CheckMove

Command:

```
CheckMove(P)
```

Description:

Check the operatibility of linear motion commands.

Required parameter:

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

Return:

Check result.

- 0: no error
- 16: End point closed to shoulder singularity point
- 17: End point inverse kinematics error with no solution
- 18: Inverse kinematics error with result out of working area
- 22: Arm orientation error
- 26: End point closed to wrist singularity point
- 27: End point closed to elbow singularity point
- 29: Speed parameter error
- 32: Shoulder singularity point in trajectory
- 33: Inverse kinematics error with no solution in trajectory
- 34: Inverse kinematics error with result out of working area in trajectory
- 35: Wrist singularity point in trajectory
- 36: Elbow singularity point in trajectory
- 37: Joint angle is changed over 180 degree

Example:

```
local status=CheckMove (P1)
if(status==0)
    then
        Move(P1)
    end
```

Check whether the robot arm can reach P1 through linear motion default setting. If it can, move to P1 through linear motion.

TCPSpeed

Command:

```
TCPSpeed(vt)
```

Description:

Set the absolute speed. The motion commands of Cartesian coordinate system after this command will run at the set absolute speed, and the joint coordinate motion commands will not be affected. After setting TCPSpeed, SpeedS no longer takes effect, but the maximum speed is still limited by the global speed (including reduced mode).

When using the welding process package, if the command conflicts with the related commands of welding, the welding commands will take precedence.

Required parameter:

vt: absolute speed, unit: mm/s, range: (0,100000]

Example:

```
TCPSpeed(100)
Move(P1)
```

The robot arm moves to P1 at an absolute speed of 100mm/s through linear motion.

TCPSpeedEnd

Command:

```
TCPSpeedEnd()
```

Description:

The command is used with TCPSpeed command for switching off absolute speed settings.

Example:

```
TCPSpeed(100)
Move(P1)
TCPSpeedEnd()
Move(P2)
```

The robot arm moves to P1 at an absolute speed of 100mm/s through linear motion, and then moves to P2 at the global speed through linear motion.

InverseSolution

Command:

```
InverseSolution(P,User,Tool,isJointNear,JointNear)
```

Description:

Inverse solution: Calculate the joint coordinates of the robot arm, based on the given coordinates in the specified Cartesian coordinate system.

As Cartesian coordinates only define the spatial coordinates and tilt angle of the TCP, the robot arm can reach the same posture through different gestures, which means that one posture variable can correspond to multiple joint variables. To get a unique solution, the system requires a specified joint coordinate, and the solution closest to this joint coordinate is selected as the inverse solution. For the setting of this joint coordinate, see the isJointNear and JointNear parameters for details.

Required parameter:

- P: target Cartesian point.
- User: the calibrated user coordinate system index.
- Tool: the calibrated tool coordinate system index.

Optional parameter:

- isJointNear: used to set whether JointNear is effective. 0 or null: JointNear is ineffective, the algorithm selects the joint angles according to the current angle. 1: the algorithm selects the joint angles according to JointNear.
- JointNear: joint coordinates for selecting joint angles.

Return:

The joint coordinates of the points.

Example:

```
InverseSolution({473.000000,-141.000000,469.000000,-180.000000,0.000,-90.000},0,0)
```

The Cartesian coordinates of the end of the robot arm in the user coordinate system 0 and joint coordinate system 0 are {473,-141,469,-180,0,-90}. Calculate the joint coordinates and select the nearest solution to the current joint angle.

```
InverseSolution({473.000000,-141.000000,469.000000,-180.000000,0.000,-90.000},0,0,1,{0,0,-90,0,
90,0})
```

The Cartesian coordinates of the end of the robot arm in the user coordinate system 0 and joint coordinate system 0 are {473,-141,469,-180,0,-90}. Calculate the joint coordinates and select the nearest solution to {0,0,-90,0,90,0}.

SetBackDistance

Command:

```
SetBackDistance(distance)
```

Description:

Set the backoff distance after the robot detects collision. The value set through this command only takes effect when the current project is running, and will restore to the original value after the project stops.

Required parameter:

distance: collision backoff distance, range: [0,50], unit: mm.

Example:

```
SetBackDistance(20)
```

Set the collision backoff distance to 20mm.

Relative motion

The motion commands is used to control the movement of the robot arm. The commands in this group are queue commands except for RP and RJ.

RP

Command:

```
RP(P, {OffsetX, OffsetY, OffsetZ})
```

Description:

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

Required parameter:

- P1: Point before offset, which is user-defined or obtained from the TeachPoint page. Only Cartesian coordinate points are supported
- OffsetX, OffsetY, OffsetZ: X-axis, Y-axis, Z-axis offset in the Cartesian coordinate system; unit: mm

Return:

Cartesian coordinate point after offset

Example:

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

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

RJ

Command:

```
RJ(P1, {Offset1, Offset2, Offset3, Offset4, Offset5, Offset6})
```

Description:

Set the angle offset of J1~J6 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~Offset6: J1~J6 axes offset. unit: °.

Return:

Joint coordinate point after offset

Example:

```
MoveJ(RJ(P1, {60,50,32,30,25,30}))
```

Set the angle offset of J1~J6 axes, and move the robot arm to the target point after offset.

GoToolR

Command:

```
GoToolR({x, y, z, rx, ry, rz}, Tool, "User=0 CP=1 Speed=50 Accel=20 SYNC=1")
```

Description:

Move from the current position to the target position relatively through joint motion. The trajectory of joint motion is not linear, and all joints complete the motion at the same time.

Required parameter:

- {x, y, z, rx, ry, rz}: offset of the target point relative to the current position in the specified tool coordinate system. x, y, z represent the spatial offset, unit: mm; rx, ry, rz represent the angular offset, unit: °.
- Tool: tool coordinate system index, which can be used here after being added in Settings.

Optional parameter:

- User: user coordinate system index, which can be used here after being added in Settings.
- 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:

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

The robot arm moves to the specific offset point along the tool coordinate system 0 through joint motion with the default settings.

MoveToolR

Command:

```
MoveToolR({x, y, z, rx, ry, rz}, Tool, "User=0 CP=1 SpeedS=50 AccelS=20 SYNC=1")
```

Description:

Move from the current position to the target position relatively through linear motion.

Required parameter:

- {x, y, z, rx, ry, rz}: offset of the target point relative to the current position in the specified tool coordinate system. x, y, z represent the spatial offset, unit: mm; rx, ry, rz represent the angular offset, unit: °.
- Tool: tool coordinate system index, which can be used here after being added in Settings.

Optional parameter:

- User: user coordinate system index, which can be used here after being added in Settings.
- CP: set continuous path in motion (see CP command in Motion parameter), range: 0 - 100.
- SpeedS: velocity rate, range: 1 - 100.
- AccelS: 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:

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

The robot arm moves to the specific offset point along the tool coordinate system 0 through linear motion with the default settings.

GoUserR

Command:

```
GoUserR({x, y, z, rx, ry, rz}, User, "Tool=0 CP=1 Speed=50 Accel=20 SYNC=1")
```

Description:

Move from the current position to the target position relatively through joint motion. The trajectory of joint motion is not linear, and all joints complete the motion at the same time.

Required parameter:

- {x, y, z, rx, ry, rz}: offset of the target point relative to the current position in the specified user coordinate system. x, y, z represent the spatial offset, unit: mm; rx, ry, rz represent the angular offset, unit: °.
- User: user coordinate system index, which can be used here after being added in Settings.

Optional parameter:

- Tool: tool coordinate system index, which can be used here after being added in Settings.
- 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:

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

The robot arm moves to the specific offset point along the user coordinate system 0 through joint motion with the default settings.

MoveUserR

Command:

```
MoveUserR({x, y, z, rx, ry, rz}, User, "Tool=0 CP=1 SpeedS=50 AccelS=20 SYNC=1")
```

Description:

Move from the current position to the target position relatively through linear motion.

Required parameter:

- {x, y, z, rx, ry, rz}: offset of the target point relative to the current position in the specified user

coordinate system. x, y, z represent the spatial offset, unit: mm; rx, ry, rz represent the angular offset, unit: °.

- User: user coordinate system index, which can be used here after being added in Settings.

Optional parameter:

- Tool: tool coordinate system index, which can be used here after being added in Settings.
- CP: set continuous path in motion (see CP command in Motion parameter), range: 0 - 100.
- SpeedS: velocity rate, range: 1 - 100.
- AccelS: 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: synchronous execution. which means not returning after being called until the command is executed completely.

Example:

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

The robot arm moves to the specific offset point along the user coordinate system 0 through linear motion with the default settings.

MoveJR

Command:

```
MoveJR({Offset1, Offset2, Offset3, Offset4, Offset5, Offset6},"CP=1 Speed=50 Accel=20 SYNC=1")
```

Description:

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

Required parameter:

{Offset1, Offset2, Offset3, Offset4, Offset5, Offset6}: J1 - J6 axes offset under the Cartesian coordinate system, unit: °.

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:

```
MoveJR({20,20,10,0,10,0})
```

The robot arm moves to the offset angle through joint motion with the default setting.

GoR

Command:

```
GoR({OffsetX, OffsetY, OffsetZ},"User=1 Tool=2 CP=1 Speed=50 Accel=20 SYNC=1")
```

Description:

This command has been replaced by GoToolR and GoUserR.

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

Optional parameter:

- User: user coordinate system index, which can be used here after being added in the Settings.
- Tool: tool coordinate system index, which can be used here after being added in the Settings.
- 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:

```
GoR({10,10,10})
```

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

MoveR

Command:

```
MoveR({OffsetX, OffsetY, OffsetZ},"User=1 Tool=2 CP=1 SpeedS=50 AccelS=20 SYNC=1")
```

Description:

This command has been replaced by MoveToolR and MoveUserR.**

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

Required parameter:

OffsetX, OffsetY, OffsetZ: X-axis, Y-axis and Z-axis offset under the Cartesian coordinate system, unit: mm

Optional parameter:

- User: user coordinate system index, which can be used here after being added in the Settings.
- Tool: tool coordinate system index, which can be used here after being added in the Settings.
- CP: set continuous path in motion (see CP command in Motion parameter), range: 0 - 100.
- SpeedS: velocity rate, range: 1 - 100.
- AccelS: 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:

```
MoveR({10,10,10})
```

The robot arm moves to the target 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:

Immediate command. 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
Move(P1)
end
```

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

DIGroup

Command:

```
DIGroup(index1,...,indexN)
```

Description:

Immediate command. Get the status of multiple digital input ports.

Required parameter:

index: DI index, more than one can be entered, separated by commas.

Return:

Status (ON/OFF) of corresponding DI port, returned as an array.

Example:

```
local digroup = DIGroup(1,2)
if (digroup[1]&digroup[2]==ON)
then
    Move(P1)
end
```

The robot arm moves to P1 in a linear motion when both DI1 and DI2 are ON.

WaitDI

Command:

```
WaitDI(index,ON|OFF,period)
```

Description:

Immediate command. Get the status of the digital input port. If the status is consistent with the specified status, the program will continue to run. Otherwise, get the status of the digital input port at a specified interval.

Required parameter:

- index: DI index
- ON|OFF: status of DI port. ON: high level; OFF: low level
- period: period for getting DI status, unit: ms

Example:

```
WaitDI(1,ON,50)
Move(P1)
```

Get the status of DI1 every 50ms. If DI1 is high, the program continues to run, and the robot arm moves to P1 in a linear mode.

DO

Command:

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

Description:

Queue command. 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.

DOExecute

Command:

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

Description:

Immediate command. 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:

```
DOExecute(1,ON)
```

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

DOGroup

Command:

```
DOGroup({index1,ON|OFF},..,{indexN,ON|OFF})
```

Description:

Set the status of multiple digital output ports.

Required parameter:

- index: DO index.
- ON|OFF：status of the DO port.

Multiple groups can be set, with each group wrapped in curly brackets and separated by commas.

Example:

```
DOGroup({1,ON},{2,ON})
```

Set DO1 and DO2 to ON.

ToolDO

Command:

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

Description:

Queue command. Set the status of tool digital output port.

Required parameter:

- index: tool DO index
- ON|OFF: status of the DO port. ON: high level; OFF: low level

Example:

```
ToolDO(1,ON)
```

Set the status of tool DO1 to ON.

ToolDOExecute

Command:

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

Description:

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

Required parameter:

- index: tool DO index

- ON|OFF: status of the DO port. ON: high level; OFF: low level

Example:

```
ToolDOExecute(1,ON)
```

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

ToolDI

Command:

```
ToolDI(index)
```

Description:

Immediate command. Get the status of tool input port.

Required parameter:

index: tool DI index

Return:

Level (ON/OFF) of corresponding DI port

Example:

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

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

ToolAI

Command:

```
ToolAI(index)
```

Description:

Immediate command. Get the value of tool analog input port. You need to set the port to voltage-input mode through ToolAnalogMode before using.

Required parameter:

index: tool AI index

Return:

Value of corresponding AI index

Example:

```
test = ToolAI(1)
```

Get the value of tool AI1 and assign it to the variable "test".

ToolAnalogMode

Command:

```
ToolAnalogMode(mode)
```

Description:

Immediate command. Set the status of tool analog input port.

Required parameter:

mode: mode of analog input port

- 0: default, 485 mode
- 10: current acquisition mode
- 11: 0 - 3.3 V voltage input mode
- 12: 0 - 10 V voltage input mode

Example:

```
ToolAnalogMode(11)
```

Set the status of tool analog input port to 0 - 3.3 V voltage input mode .

AO

Command:

```
AO(index,value)
```

Description:

Queue command. Set the voltage of analog output port.

Required parameter:

- index: AO index
- value: voltage, range: 0 - 10

Example:

```
AO(1,2)
```

Set the voltage of AO1 to 2V.

AOExecute

Command:

```
AOExecute(index,value)
```

Description:

Immediate command. Set the voltage of analog output port immediately regardless of the current command queue.

Required parameter:

- index: AO index
- value: voltage, range: 0 - 10

Example:

```
AOExecute(1,2)
```

Set the voltage of AO1 to 2V immediately regardless of the current command queue.

AI

Command:

```
AI(index)
```

Description:

Immediate command. Get the voltage of analog input port.

Required parameter:

index: AI index

Return:

Value of corresponding AI index

Example:

```
test = AI(1)
```

Get the value of tool AI1 and assign it to the variable "test".

SetToolPower

Command:

```
SetToolPower(status)
```

Description:

Immediate command. Set the power status of end tool, generally used for restarting the end power, such as repowering and reinitializing the gripper. If you need to call the interface continuously, it is recommended to keep an interval of at least 4 ms.

Required parameter:

status: power status of end tool, 1: power off; 0: power on

Example:

```
SetToolPower(1)
```

Power off the tool.

SetABZPPC

Command:

```
SetABZPPC(value)
```

Description:

Immediate command. Set the current value of the encoder.

Required parameter:

value: position of encoder, unit: pulse.

Example:

```
SetABZPPC(1000)
```

Set the value of ABZ encoder to 1000 pulses.

GetABZ

Command:

```
GetABZ()
```

Description:

Immediate command. Get the resolution of configured ABZ encoder.

Return:

Resolution of encoder, unit: pulse/mm.

Example:

```
local abz = GetABZ()
```

Get the resolution of configured ABZ encoder, and assign the value to the variable "abz".

TCP/UDP

TCP/UDP commands are used for TCP/UDP communication. The commands in this group are all immediate commands.

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),

  30005, 30006, 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 IP address of 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 IP
 address of the server
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)
```

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:

```
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)
```

Description:

Send data to TCP peer.

Required parameter:

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

Return:

Result of sending data.

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

Example:

```
TCPWrite(socket, "test")
```

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)
```

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:

```
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)
```

Description:

Send data to UDP peer.

Required parameter:

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

Return:

Result of sending data.

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

Example:

```
UDPWrite(socket, "test")
```

Send UDP data "test".

Modbus

Modbus commands are used for Modbus communication. This group of commands are all immediate commands.

ModbusCreate

Command:

```
ModbusCreate()
```

Description:

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

Required parameter:

- IP: slave IP address.
- port: slave port.
- slave_id: slave ID. range: 0 - 4.
- isRTU: If it is null or 0, establish ModbusTCP communication. If it is 1, establish ModbusRTU communication.

> ⚠ **NOTICE**
> This parameter determines the protocol format used to transmit data after the connection is established, and does not affect the connection result. Therefore, if you set the parameter incorrectly when creating the master, the master can still be created successfully, but there may be exception in the subsequent communication.

Return:

- err:
  - 0: Modbus master station has been created successfully.
  - 1: As there are 5 created master stations, a new one failed to be created.
  - 2: Modbus master station failed to be initialized. It is recommended to check whether the IP, port and network is normal.
  - 3: Modbus slave station failed to be connected. It is recommended to check whether the slave is established properly and whether the network is normal.
- id: Master station index, range: 0 - 4.

Example 1:

```
local ip="192.168.5.123" -- slave IP address
local port=503 -- slave port
local err=0
local id=1
err, id = ModbusCreate(ip, port, 1)
```

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

Example 2:

When none of the parameters are specified or the IP is specified to 127.0.0.1 or 0.0.0.1, it indicates connecting to the local Modbus slave. Please refer to the following commands.

```
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
```

Example 3:

```
err,id=ModbusCreate("127.0.0.1",60000,1,1)
```

The example 3 is a special case, indicating communication with the external Modbus RTU slave via the 485 interface at the end of the robot arm.

GetInBits

Command:

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

Description:

Read the discrete input value from Modbus slave.

Required parameter:

- id: master ID.
- addr: starting address of the discrete inputs, range: 0 - 9999.
- count: number of the discrete inputs. The maximum value is 216 (CR series) or 984 (Nova series) when communicating with the slave via the tool I/O interface, and 2008 for the other scenarios.

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: master ID.
- addr: starting address of the input registers, range: 0 - 9998.
- count: number of input register values.

Optional parameter:

type: data type

- Empty: U16 by default
- U16: 16-bit unsigned integer (two bytes, occupy one register). Up to 13 values (CR series) or 61 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 125 values are read continuously for the other scenarios.
- U32: 32-bit unsigned integer (four bytes, occupy two registers). Up to 6 values (CR series) or 30 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 62 values are read continuously for the other scenarios.
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers). Up to 6 values (CR series) or 30 values (Nova series) are read continuously when communicating with the slave via

the tool I/O interface, and up to 62 values are read continuously for the other scenarios.

- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers). Up to 3 values (CR series) or 15 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 31 values are read continuously for the other scenarios.

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: master ID.
- addr: starting address of the coil register, range: 0 - 9999.
- count: number of coil register values. The maximum value is 216 (CR series) or 984 (Nova series) when communicating with the slave via the tool I/O interface, and 2008 for the other scenarios.

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: master ID.
- addr: starting address of the holding register, range: 0 - 9998.
- 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). Up to 13 values (CR series) or 61 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 125 values are read continuously for the other scenarios.
- U32: 32-bit unsigned integer (four bytes, occupy two registers). Up to 6 values (CR series) or 30 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 62 values are read continuously for the other scenarios.
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers). Up to 6 values (CR series) or 30 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 62 values are read continuously for the other scenarios.
- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers). Up to 3 values (CR series) or 15 values (Nova series) are read continuously when communicating with the slave via the tool I/O interface, and up to 31 values are read continuously for the other scenarios.

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: master ID.
- addr: starting address of the coil register, range: 6 - 9999.
- count: number of values to be written to the coil register. The maximum value is 440 when communicating with the slave via the tool I/O interface, and 1976 for the other scenarios.
- 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: master ID.
- addr: starting address of the holding register, range: 0 - 9998.
- 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). Up to 27 values are written

continuously when communicating with the slave via the tool I/O interface, and up to 123 values are written continuously for the other scenarios.

- U32: 32-bit unsigned integer (four bytes, occupy two registers). Up to 13 values are written continuously when communicating with the slave via the tool I/O interface, and up to 61 values are written continuously for the other scenarios.
- F32: 32-bit single-precision floating-point number (four bytes, occupy two registers). Up to 13 values are written continuously when communicating with the slave via the tool I/O interface, and up to 61 values are written continuously for the other scenarios.
- F64: 64-bit double-precision floating-point number (eight bytes, occupy four registers). Up to 6 values are written continuously when communicating with the slave via the tool I/O interface, and up to 30 values are written continuously for the other scenarios.

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: master ID.

Return:

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

Example:

```
ModbusClose(id)
```

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.

print

Command:

```
print(var)
```

Description:

Immediate command. Print specific information to the console.

Required parameter:

var: print information

Example:

```
local var = "Hello World"
print(var)
```

Print Hello World to the console.

Sleep

Command:

```
Sleep(time)
```

Description:

Immediate command. Delay the execution of the next command. This command is used to block the execution of the script.

Required parameter:

time: delay time, unit: ms. Decimal points are not allowed in the parameter, which will cause the alarm or command to be invalid, even in the form of 0.5 * 1000. Please use the integer as the parameter.

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:

Queue command. Deliver the motion command with a delay, or deliver the next command with a delay after the current motion is completed. This command only blocks the execution of background algorithm queue.

Required parameter:

time: delay time, unit: ms. Decimal points are not allowed in the parameter, which will cause the alarm or command to be invalid, even in the form of 0.5 * 1000. Please use the integer as the parameter.

Example:

```
DO(1,ON)
Wait(100)
Go(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:

```
Go(P1)
Pause()
Go(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:

Queue command. Set the level of collision detection. This settings only take effect when the current project is running. After the project stops running, the parameters restore to the original settings.

Required parameter:

level: collision detection level, range: 0 to 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:

Queue command. 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:

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

This interface can only count the time difference within 35 minutes. If the time difference exceeds 35 minutes, the interface will return an incorrect time difference. For longer time differences, it is recommended to use the Systime interface for calculation.

Return:

time between the start and the end of timing.

Example:

```
Go(P2, "Speed=100 Accel=100")
ResetElapsedTime()
for i=1,10 do
Move(P1)
Move(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:

Immediate command. Get the current system time.

Return:

Unix time stamp of the current time.

Example:

```
local time = Systime()
Move(P1)
```

```
local time2 = Systime()
print(time2-time1)
```

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

SetUser

Command:

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

Description:

Queue command. Modify the specified user coordinate system. This modification only takes effect while the current project is running. After the project is stopped, the coordinate system will be restored to the value before modification.

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, rx, ry, rz} format

Example:

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

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

CalcUser

Command:

```
CalcUser(index,matrix_direction,table)
```

Description:

Queue command. Calculate the user coordinate system.

Required parameter:

- index: index of user coordinate system, range: [0,9]. The initial value 0 refers to the default user coordinate system.
- matrix_direction: calculation method.
  - 1: left multiplication, indicating that the coordinate system specified by "index" deflects the value specified by "table" along the base coordinate system.

    - 0: right multiplication, indicating that the coordinate system specified by "index" deflects the value specified by "table" along itself.
- table: offset value of user coordinate system, in {x, y, z, rx, ry, rz} format.

Return:

The user coordinate system after calculation, in {x, y, z, rx, ry, rz} format.

Example:

```
-- The following calculation process can be equivalent to: A coordinate system with the same i
nitial posture as User coordinate system 1, moves {x=10, y=10, z=10} along the base coordinate
 system and rotates {rx=10, ry=10, rz=10}, and the new coordinate system is newUser.
newUser = CalcUser(1,1,{10,10,10,10,10,10})
```

```
-- The following calculation process can be equivalent to: A coordinate system with the same i
nitial posture as User coordinate system 1, moves {x=10, y=10, z=10} along User coordinate sys
tem 1 and rotates {rx=10, ry=10, rz=10}, and the new coordinate system is newUser.
newUser = CalcUser(1,0,{10,10,10,10,10,10})
```

SetTool

Command:

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

Description:

Queue command. Modify the specified tool coordinate system.

Required parameter:

- index: index of tool coordinate system, range: 0 - 9 (0 is default tool coordinate system).
- table: matrix for tool coordinate system, in {x, y, z, rx, ry, rz} format.

Example:

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

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

CalcTool

Command:

```
CalcTool(index,matrix_direction,table)
```

Description:

Queue command. Calculate the tool coordinate system.

Required parameter:

- index: index of tool coordinate system, range: [0,9]. The initial value 0 refers to the default flange coordinate system (TCP0).
- matrix_direction: calculation method.
  - 1: left multiplication, indicating that the coordinate system specified by "index" deflects the value specified by "table" along the flange coordinate system (TCP0).
  - 0: right multiplication, indicating that the coordinate system specified by "index" deflects the value specified by "table" along itself.
- table: tool coordinate system, in {x, y, z, rx, ry, rz} format.

Return:

The tool coordinate system after calculation, in {x, y, z, rx, ry, rz} format.

Example:

```
-- The following calculation process can be equivalent to: A coordinate system with the same i
nitial posture as Tool coordinate system 1, moves {x=10, y=10, z=10} along the flange coordina
te system (TCP0) and rotates {rx=10, ry=10, rz=10}, and the new coordinate system is newTool.
newTool = CalcTool(1,1,{10,10,10,10,10,10})
```

```
-- The following calculation process can be equivalent to: A coordinate system with the same i
nitial posture as Tool coordinate system 1, moves {x=10, y=10, z=10} along Tool coordinate sys
tem 1 and rotates {rx=10, ry=10, rz=10}, and the new coordinate system is newTool.
newTool = CalcTool(1,0,{10,10,10,10,10,10})
```

LoadSet

Command:

```
LoadSet(weight,inertia)
```

Description:

Queue command. Set the weight of the load. This settings only take effect when the current project is running. After the project stops running, the parameters restore to the original settings.

Required parameter:

- weight: weight of the load, the value range cannot exceed the load range of each robot model. Unit: kg.

- inertia: inertia of the load, unit: kgm².

Example:

```
LoadSet(3,0.4)
```

Set the weight of the load to 3 kg, and the inertia of the load to 0.4 kgm².

LoadSwitch

Command:

```
LoadSet(status)
```

Description:

Queue command. Switch on/off the load settings. Load setting is off by default, the collision detection sensitivity can be improved after it is switched on.

Required parameter:

status: switch of load settings. 0: OFF; 1: ON.

Example:

```
LoadSwitch(1)
```

Switch on the load settings.

Trajectory

GetTraceStartPose

Command:

```
GetTraceStartPose(track)
```

Description:

Get the first point of the trajectory for trajectory fitting (StartTrace).

Required parameter:

track: trajectory filename, with suffix.

Return:

Coordinate of the first point of the trajectory.

StartTrace

Command:

```
StartTrace(track, "User=1 Tool=2 CP=1 SpeedS=50 AccelS=20 SYNC=1")
```

Description:

Trajectory fitting. The motion trajectory is fitted and played back based on the points recorded in the trajectory file, and the motion speed is calculated in the same way as linear motion, which is not affected by the motion speed at the time of trajectory recording.

Required parameter:

track: trajectory filename, with suffix.

Optional parameter:

- User: user coordinate system index, which needs to be referenced here after being added in Settings.
- Tool: tool coordinate system index, which needs to be referenced here after being added in Settings.
- CP: set continuous path rate in motion (see CP command in Motion parameter), range: 0 - 100.
- SpeedS: motion velocity rate, range: 1 - 100.
- AccelS: motion 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 track = "track.json"
local P1 = GetTraceStartPose(track)
Go(P1)
StartTrace(track)
```

Fit and play back the trajectories recorded in the track.json file.

GetPathStartPose

Command:

```
GetPathStartPose(track)
```

Description:

Get the first point of the trajectory for trajectory playback (StartPath).

Required parameter:

track: trajectory filename, with suffix.

Return:

Coordinates of the first point of the trajectory.

StartPath

Command:

```
StartPath(track, "User=1 Tool=2 sample = 50 freq = 0.2 Multi=1 isConst=0 isCart=0 isRev=0 isRe
l=0")
```

Description:

Trajectory fitting. The motion trajectory is fitted and played back based on the points recorded in the trajectory file, and the motion speed is calculated in the same way as linear motion, which is not affected by the motion speed at the time of trajectory recording.

Required parameter:

track: trajectory filename, with suffix.

Optional parameter:

- User: user coordinate system index, which needs to be referenced here after being added in Settings.
- Tool: tool coordinate system index, which needs to be referenced here after being added in Settings.
- sample: the sampling interval of the trajectory points, i.e. the sampling time differences between two adjacent points when generating the trajectory file. Range: [8, 1000], unit: ms, 50 by default (the sample interval when the controller records the trajectory file).
- freq: filter coefficient. The smaller the value of this parameter, the smoother the playback trajectory curve is, but the more severe the distortion relative to the original trajectory. Please set the appropriate filter coefficient according to the smoothness of the original trajectory. Range: (0,1], 1 refers to turning off filtering, 0.2 by default.
- Multi: speed multiplier in playback, range: [0.1, 2], 1 by default.
- isConst: if or not to play back at a constant speed. 0 by default.
    - 0 means the trajectory will be played back at the same speed.
    - 1 means the trajectory will be played back at a constant speed, and the pauses and dead zones in the trajectory will be removed.
- isCart: if or not to use Cartesian points. 0 by default.
    - 0 means the trajectory is planned using the joint points in the trajectory file.
    - 1 means the trajectory is planned using the Cartesian points in the trajectory file. Only the joint positions are recorded in the trajectory files recorded with DobotStudio Pro or Dobot CRStudio, please do not set this parameter in playback.
- isRev: trajectory reversal. 0 by default.
    - 0 means the trajectory is not reversed.
    - 1 means the trajectory is reversed, taking the ending point in the trajectory file as the starting point and the starting point as the ending point.
- isRel: trajectory offset. 0 by default.
    - 0 means the trajectory is not offset.
    - 1 means the trajectory is offset, taking the current point of the robot arm as the starting point of the trajectory and offsetting the trajectory as a whole.

Example:

```
local track = "track.json"
local P1 = GetPathStartPose(track)
Go(P1)
StartPath(track)
```

Play back the trajectory recorded in the track.json file at the original speed.

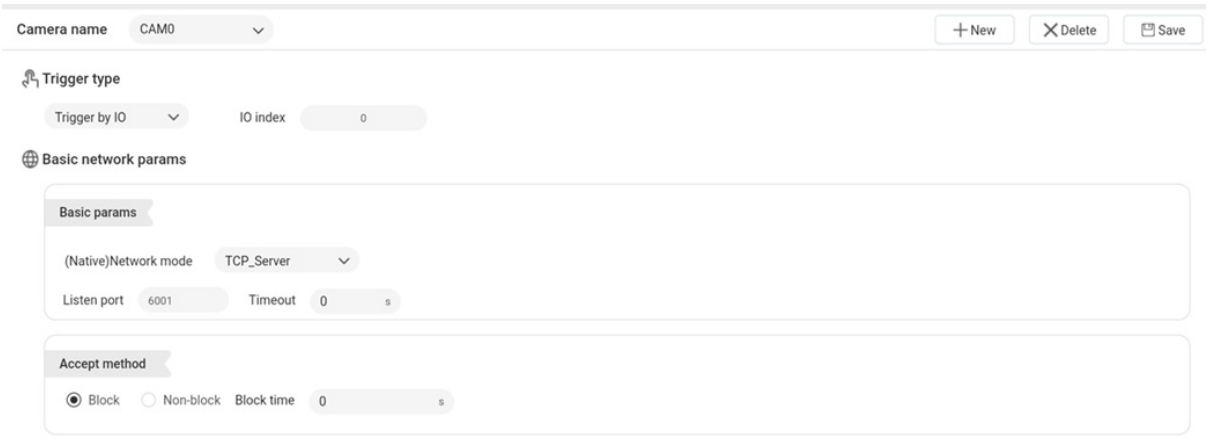
Vision

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.

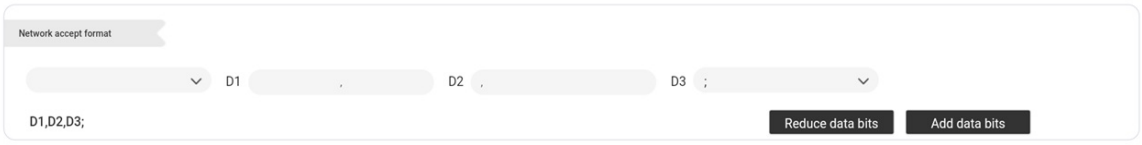
Basic network parameters

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.

Conveyor

The conveyor command must be used with the conveyor process package.

CnvVison

Command:

```
CnvVison(CnvID)
```

Description:

Initialize the specified conveyor.

Required parameter:

CnvID: conveyor ID

Return

The setting result.

- 0: It has been set successfully.
- 1: It fails to be set.

Example:

```
CnvVison(0)
```

Initialize the conveyor numbered 0.

GetCnvObject

Command:

```
GetCnvObject(CnvID, ObjID)
```

Description:

Get the data of specified object from the specified conveyor, and detects whether it enters the gripping area.

Required parameter:

- CnvID: conveyor ID.

- ObjID: object ID.

Return

- flag: indicates whether the object is detected. Value range: true or false
- typeObject: object type
- point: object coordinate {x,y,z}

Example:

```
flag,typeObject,point = GetCnvObject(0,0)
```

Get the data of object 0 on conveyor 0.

SetCnvPointOffset

Command:

```
SetCnvPointOffset(xOffset,yOffset)
```

Description:

Set the offset of the points in X and Y directions under the conveyor user coordinate system.

Required parameter:

- xOffset: X-axis offset, unit: mm
- yOffset: Y-axis offset, unit: mm

Return

The setting result.

- 0: It has been set successfully.
- 1: It fails to be set.

Example:

```
SetCnvPointOffset(10,10)
```

Set the conveyor point offset as X=10 mm and Y=10 mm.

SetCnvTimeCompensation

Command:

```
SetCnvTimeCompensation(time)
```

Description:

Set time compensation. When the gripping position of the robot arm falls behind the actual position of the object, it can be adjusted by setting time compensation.

Required parameter:

time: timing offset, unit: ms

Return

The setting result.

- 0: It has been set successfully.
- 1: It fails to be set.

Example:

```
SetCnvTimeCompensation(30)
```

Set the time compensation to 30 ms.

SyncCnv

Command:

```
SyncCnv(CnvID)
```

Description:

Synchronize the specified conveyor. After synchronization, points based on the conveyor user coordinate system move with the conveyor. The motion command between this command and the StopSyncCnv command only supports the Move command.

Required parameter:

CnvID: conveyor ID

Return

The synchronization result.

- 0: It has been synchronized successfully.
- 1: It fails to be synchronized.

Example:

```
SyncCnv(0)
```

Synchronize the conveyor numbered 0.

StopSyncCnv

Command:

```
StopSyncCnv(CnvID)
```

Description:

Stop synchronizing the conveyor, and only after running this command will it continue to issue other commands.

Required parameter:

CnvID: conveyor ID

Return

The execution result.

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

Example:

```
StopSyncCnv(0)
```

Stop synchronizing the conveyor numbered 0.

SafeSkin

SetSafeSkin

Command:

```
SetSafeSkin(status)
```

Description:

Enable or disable SafeSkin.

Required parameter:

status: 0 represents disabling SafeSkin, 1 represents enabling SafeSkin. If the Dobot+ inner SafeSkin is off, this command does not work.

Example:

```
EnableSafeSkin(1)
```

Enable SafeSkin.