Dobot M1 API Description

Issue: V1.0

Date: 2018-03-26

Shenzhen Yuejiang Technology Co., Ltd

Disclaimer

To the maximum extent permitted by applicable law, the products described (including its hardware, software and firmware, etc.) in this document are provided **AS IS**, which may have flaws, errors or faults. Yuejiang makes no warranties of any kind, express or implied, including but not limited to, merchantability, satisfaction of quality, fitness for a particular purpose and non-infringement of third party rights. In no event will Yuejiang be liable for any special, incidental, consequential or indirect damages resulting from the use of our products and documents.

Before using our product, please thoroughly read and understand the contents of this document and related technical documents that are published online, to ensure that the robotic arm is used on the premise of fully understanding the robotic arm and related knowledge. Please use this document with technical guidance from professionals. Even if follow this document or any other related instructions, Damages or losses will be happen in the using process, Dobot shall not be considered as a guarantee regarding to all security information contained in this document.

The user has the responsibility to make sure following the relevant practical laws and regulations of the country, in order that there is no significant danger in the use of the robotic arm.

Shenzhen Yuejiang Technology Co., Ltd

Address: 4F, A8, Tanglang Industrial Area, Taoyuan Street, Nanshan District, Shenzhen, PRC

Website: www.dobot.cc

Preface

Purpose

The document is aiming to have a detailed description of Dobot API and general process of Dobot API development program.

Intended Audience

This document is intended for:

- Secondary developer
- Technical Support Engineer

Change History

Date	Change Description
2018/03/17	The first release

Symbol Conventions

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

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

Contents

1. API Interface Description

1.1 Dobot Commands

Dobot controller supports two kind of commands: Immidiate command and queue command:

- Immidiate command: Dobot controller will process the command once received regardless of whether there is the rest commands processing or not in the current controller;
- Queue command: When Dobot controller receives a command, this command will be pressed into the controller internal command queue. Dobot controller will execute commands in the order in which the commands were pressed into the queue.

For more detailed information about Dobot commands, please refer to *Dobot protocol*.

1.2 Command Timeout

1.2.1 Setting Command Timeout

As described in *1.1 Dobot Commands*, all commands sent to Dobot controller have returns. When a command error occurs due to a communication link interference or any other factors, this command cannot be recognized by the controller and will have no return. Therefore, each command issued to the controller has a timeout period. The timeout period can be set by the following API.

Table 1.1 Set timeout

Prototype	void SetCmdTimeout(uint32_t cmdTimeout)
Description	Set command timeout. If a command is required to return data within a given time after issuing it, please call this API to set timeout to check whether the return of this command is overtime
Parameter	cmdTimeout: Command timeout. Unit: ms
Return	DobotCommunicate_NoError:There is no error

1.3 Connect/Disconnect

1.3.1 Searching for the Dobot

Table 1.2 Search for the Dobot

Prototype	int SearchDobot(char *dobotList, uint32_t maxLen)
Description	Search for Dobot, DLL will store the information of Dobot that has been searched for and use ConnectDobot to connect the searched Dobot
Parameter	dobotList: String pointer, DLL will write serial port/UDP searched into dobotList. For example, a specific dobotList is "**COM1 COM3 COM6 192.168.0.5**", different serial port or IP address should be separated by the space

	maxLen: Maximum String length, to avoid memory overflow
Return	The number of Dobot

1.3.2 Connecting to the Dobot

Table 1.3 Connect to the Dobot

Prototype	int ConnectDobot(const char *portName, uint32_t baudrate, char *fwType, char *version)
Description	Connecing to the Dobot. In this process, portName can be obtained from **dobotList** in the **SearchDobot(char *dobotList, uint32_t maxLen)** API. If portName is empty, and **ConnectDobot** is called directly, DLL will connect the random searched Dobot automatically
Parameter	portName: Dobot port. As for the serial port, portName is **COM3**; While for UDP, portName is **192.168.0.5** baudrate: Baud rate fwType: Firmware type. Dobot or Marlin version: Version
Return	DobotConnect_NoError： The connection is successful DobotConnect_NotFound： Dobot interface was not found DobotConnect_Occupied： Dobot interface is occupied or unavailable

⚠NOTICE

In order to make the API recognize the Dobot controller interface, please install the required driver in advance. For more details, please refer to *Dobot M1 User Guide*.

1.3.3 Disconnecting the Dobot

Table 1.4 Disconnect the Dobot

Prototype	void DisconnectDobot(void)
Description	Disconnect the Dobot
Parameter	None
Return	DobotConnect_NoError :There is no error

1.3.4 Demo: Connection Example

Program 1.1 Connection Example

```
#include "DobotDll.h"

int split(char **dst, char* str, const char* spl)
{
        int n = 0;
        char *result = NULL;
        result = strtok(str, spl);
        while( result != NULL )
        {
                strcpy(dst[n++], result);
                result = strtok(NULL, spl);
        }
        return n;
}

int main(void)
{
        int maxDevCount = 100;
        int maxDevLen = 20;

        char *devsChr = new char[maxDevCount * maxDevLen]();
        char **devsList = new char*[maxDevCount]();
        for(int i=0; i<maxDevCount; i++)
                devsList[i] = new char[maxDevLen]();

        SearchDobot(devsChr, 1024);
        split(devsList, devsChr, " ");
        ConnectDobot(devsList[0], 115200, NULL, NULL, NULL);

        // Control Dobot

        DisconnectDobot();

        delete[] devsChr;
        for(int i=0; i<maxDevCount; i++)
              delete[] devsList[i];
```

```
    delete[] devsList;
}
```

1.4 Command queue controlling

There is a queue in Dobot controller to store and execute commands in order. You can also start and stop a command in the command queue to realize asynchronous operations.

⚠NOTICE

Only the API where the **isQueued** parameter is set to **1** can be added to the command queue.

1.4.1 Starting Command in Command queue

Table 1.5 Start command in command queue

Prototype	int SetQueuedCmdStartExec(void)
Description	The Dobot controller starts to query command queue periodically. If there are commands in queue, Dobot controller will take them out and execute the commands in order, indicating that Dobot executes commands one after another
Parameter	None
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.4.2 Stopping Command in Command queue

Table 1.6 Stop command in command queue

Prototype	int SetQueuedCmdStopExec(void)
Description	The Dobot controller stops to query command queue and execute command. However, if one command is being executed when this API is called, this command will continue to be executed.
Parameter	None
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.4.3 Stopping Command in Command queue Forcedly

Table 1.7 Stop command in command queue forcedly

Prototype	int SetQueuedCmdForceStopExec(void)
Description	Dobot controller stops to query command queue and execute command. If one command is being executed when this API is called, this command will be stopped forcedly.
Parameter	None
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout (aka error)

1.4.4 Demo: Processing PTP Command and Control Queue Synchronously

For details about PTP, please refer to *1.11 PTP*.

Program 1.2 Process PTP command and control queue synchronously

```
#include "DobotDll.h"


int main(void)
{
    uint64_t queuedCmdIndex = 0;
    PTPCmd    cmd;


    cmd.ptpMode = 0;
    cmd.x        = 200;
    cmd.y        = 0;
    cmd.z        = 0;
    cmd.r        = 0;


    ConnectDobot(NULL, 115200, NULL, NULL, NULL);


    SetQueuedCmdStartExec();
    SetPTPCmd(&cmd, true, &queuedCmdIndex);
    SetQueuedCmdStopExec();


    DisconnectDobot();
}
```

1.4.5 Demo: Processing PTP Command and Controlling Queue Asynchronously

Program 1.3 Process PTP command and control queue asynchronously

```
#include "DobotDll.h"

// Main thread
int main(void)
{
    ConnectDobot(NULL, 115200, NULL, NULL, NULL);
}

int onButtonClick()
{
    static bool flag = True;
    if (flag)
        SetQueuedCmdStartExec();
    else
        SetQueuedCmdStopExec();
}

// Child thread
int thread(void)
{
    uint64_t queuedCmdIndex = 0;
    PTPCmd    cmd;

    cmd.ptpMode = 0;
    cmd.x        = 200;
    cmd.y        = 0;
    cmd.z        = 0;
    cmd.r        = 0;

    while(true)
        SetPTPCmd(&cmd, true, &queuedCmdIndex);
}
```

1.4.6 Clearing Command queue

This API can clear the command queue buffered in the Dobot controller.

Table 1.8 Clear command queue

Prototype	int SetQueuedCmdClear(void)
Description	Clear command queue
Parameter	None
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.4.7 Getting Command Index

In the Dobot controller, there is a 64-bit internal counter. When the controller executes a command, the counter will automatically increment. With this internal index, you can get how many commands the controller has executed.

Table 1.9 Get command index

Prototype	int GetQueuedCmdCurrentIndex(uint64_t *queuedCmdCurrentIndex)
Description	Get the index of the command the controller has executed currently
Parameter	queuedCmdCurrentIndex: Command index
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.4.8 Demo: Checking Whether the Commands Have Been Executed

Program 1.4 Check whether the commands have been executed by comparing the indexes

```
#include "DobotDll.h"


int main(void)
{
    uint64_t queuedCmdIndex     = 0;
    uint64_t executedCmdIndex = 0;
    PTPCmd      cmd;

    cmd.ptpMode = 0;
    cmd.x           = 200;
    cmd.y           = 0;
    cmd.z           = 0;
    cmd.r           = 0;

    ConnectDobot(NULL, 115200, NULL, NULL, NULL);

    SetQueuedCmdStartExec();
    SetPTPCmd(&cmd, true, &queuedCmdIndex);

    // Check whether the commands have been executed by comparing the indexes
    While(executedCmdIndex < queuedCmdIndex)
        GetQueuedCmdCurrentIndex(&executedCmdIndex);

    SetQueuedCmdStopExec();
    DisconnectDobot();
}
```

1.5 Device Information

1.5.1 Setting the Device Serial Number

Table 1.10 Set the device serial number

Prototype	int SetDeviceSN(const char *deviceSN)
Description	Set the device serial number. This API is valid only when shipped out (The special password is required)
Parameter	deviceSN: String pointer
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.5.2 Getting the Device Serial Number

Table 1.11 Get the device serial number

Prototype	int GetDeviceSN(char *deviceSN, uint32_t maxLen)
Description	Get the device serial number
Parameter	deviceSN: Strings of device serial number maxLen: Maximum string length, to avoid overflow
Return	DobotCommunicate_NoError: The command returns with no error

	DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.5.3 Setting the Device Name

Table 1.12 Set the device name

Prototype	int SetDeviceName(const char *deviceName)
Description	Set the device name. When there are multiple machines, you can use this API to set the device name for distinction
Parameter	deviceName: String pointer
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.5.4 Getting the Device Name

Table 1.13 Get the device name

Prototype	int GetDeviceName(char *deviceName, uint32_t maxLen)
Description	Get the device name. When there are multiple machines, you can use this API to get the device name for distinction.
Parameter	deviceName: String pointer maxLen: Maximum string length, to avoid overflow
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.5.5 Getting the Device Version

Table 1.14 Get the device version

Prototype	int GetDeviceVersion(uint8_t typeIndex, uint8_t *majorVersion, uint8_t *minorVersion, uint8_t *revision)
Description	Get the device version
Parameter	typeIndex: Version type enum FirmwareType{ NO_SWITCH, DOBOT_SWITCH,

	PRINTING_SWITCH, DRIVER1_SWITCH, DRIVER2_SWITCH, DRIVER3_SWITCH, DRIVER4_SWITCH, DRIVER5_SWITCH, FPGA_SWITCH, SWITCH_FM_MAX }; majorVersion: Main version minorVersion: Secondary version revision: Revised version
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.6 Real-time pose

In Dobot M1, the Dobot controller calculates the reference value of the real-time pose based on the value of each joint Encoder.

When controlling the Dobot, the Dobot controller will update the real-time pose based on the reference value and the real-time motion status.

1.6.1 Getting the Real-time Pose of the Dobot

Table 1.15 Get the real-time pose of Dobot

Prototype	int GetPose(Pose *pose)
Description	Get the real-time pose of the Dobot

Parameter	Pose:
	typedef struct tagPose {
	float x; //Cartesian coordinate system X-axis
	float y; //Cartesian coordinate system Y-axis
	float z; // Cartesian coordinate system Z-axis
	float r; //Cartesian coordinate system R-axis
	float jointAngle[4]; //Joints (including J1, J2, J3, and J4) angles
	}Pose;
	Pose: Pose pointer

Return	DobotCommunicate_NoError: The command returns with no error

	DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.7 ALARM

1.7.1 Getting the Alarm Status

Table 1.16 Get the alarm status

Prototype	int GetAlarmsState(uint8_t *alarmsState, uint32_t *len, uint32_t maxLen)
Description	Get the alarm status
Parameter	alarmsState: The first address of the array. Each byte in the array **alarmsState** identifies the alarms status of the eight alarm items, with the MSB (Most Significant Bit) at the top and LSB (Least Significant Bit) at the bottom. len: The byte occupied by the alarm. maxLen: Maximum array length, to avoid overflow
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.7.2 Clearing the Statuses of All Alarms

Table 1.17 Clear the statuses of all alarms

Prototype	int ClearAllAlarmsState(void)
Description	Clear the statuses of all alarms
Parameter	None
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.8 Homing Function

If your Dobot is running too fast or the load is too large for the dobot, the position precision can be reduced. You can execute the homing function to improve the precision.

1.8.1 Setting the Homing Position

Table 1.18 Set the homing position

Prototype	int SetHOMEParams(HOMEParams *homeParams, bool isQueued, uint64_t *queuedCmdIndex)

Description	Set the homing position

Parameter	HOMEParams:
	typedef struct tagHOMEParams {
	    float x; //Cartesian coordinate system X-axis
	    float y; //Cartesian coordinate system Y-axis
	    float z; // Cartesian coordinate system Z-axis
	    float r; //Cartesian coordinate system R-axis
	}HOMEParams;
	homeParams: HOMEParams pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.8.2 Getting the Homing Position

Table 1.19 Get the homing position

Prototype	int GetHOMEParams(HOMEParams *homeParams)
Description	Get the homing position

Parameter	HOMEParams:
	typedef struct tagHOMEParams {
	    float x; //Cartesian coordinate system X-axis
	    float y; //Cartesian coordinate system Y-axis
	    float z; // Cartesian coordinate system Z-axis
	    float r; //Cartesian coordinate system R-axis
	}HOMEParams;
	homeParams: HOMEParams pointer

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.8.3 Executing the Homing Function

Table 1.20 Execute the homing function

Prototype	int SetHOMECmd(HOMECmd *homeCmd, bool isQueued, uint64_t *queuedCmdIndex)
Description	Execute the homing function. If you call the **SetHOMEParams** API before calling this API, Dobot will move to the user-defined position. If not, Dobot will move to the default position directly.

Parameter	HOMECmd:
	typedef struct tagHOMECmd {
	uint32_t reserved; // Reserved for future use
	}HOMECmd;
	homeCmd: HOMECmd pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid.

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.9 Arm Orientation

1.9.1 Setting Arm Orientation

Table 1.21 Set arm orientation

Prototype	int SetArmOrientation(ArmOrientation armOrientation, bool isQueued, uint64_t *queuedCmdIndex)
Description	Set arm orientation
Parameter	ArmOrientation: typedef enum tagArmOrientation { LeftyArmOrientation, //Left hand RightyArmOrientation //Right hand }ArmOrientation; armOrientation： ArmOrientation enum isQueued: Whether to add this command to the queue queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid.
Return	DobotCommunicate_NoError: The command returns with no error

	DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.9.2 Getting Arm Orientation

Table 1.22 Get arm orientation

Prototype	int SetArmOrientation(ArmOrientation armOrientation, bool isQueued, uint64_t *queuedCmdIndex)
Description	Get arm orientation
Parameter	ArmOrientation: typedef enum tagArmOrientation { LeftyArmOrientation, //Left hand RightyArmOrientation //Right hand }ArmOrientation; armOrientation： ArmOrientation enum
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.10 JOG

1.10.1 Setting the Velocity Ratio and Acceleration Ratio when Jogging

Table 1.23 Set the velocity ratio and acceleration ratio when jogging

Prototype	int SetJOGCommonParams(JOGCommonParams *jogCommonParams, bool isQueued, uint64_t *queuedCmdIndex)
Description	Set the velocity ratio and acceleration ratio for each axis (in both Joint and Cartesian coordinate system) when jogging

Parameter	JOGCommonParams:
	typedef struct tagJOGCommonParams {
	float velocityRatio; //Velocity ratio
	float accelerationRatio; //Acceleration ratio
	}JOGCommonParams;
	jogCommonParams: JOGCommonParams pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex**

	indicates the index of this command in the queue. Otherwise, it is invalid
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.10.2 Getting the Velocity Ratio and Acceleration Ratio when Jogging

Table 1.24 Get the velocity ratio and acceleration ratio when jogging

Prototype	int GetJOGCommonParams(JOGCommonParams *jogCommonParams)
Description	Get the velocity ratio and acceleration ratio for each axis (in Joint and Cartesian coordinate system) when jogging

Parameter	JOGCommonParams:
	typedef struct tagJOGCommonParams {
	float velocityRatio; //Velocity ratio
	float accelerationRatio; //Acceleration ratio
	}JOGCommonParams;
	jogCommonParams: JOGCommonParams pointer

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.10.3 Executing the Jogging Command

Table 1.25 Execute the Jogging command

Prototype	int SetJOGCmd(JOGCmd *jogCmd, bool isQueued, uint64_t *queuedCmdIndex)
Description	Execute the Jogging command. Please call this API after setting the related parameters of jogging

Parameter	JOGCmd:
	typedef struct tagJOGCmd {
	uint8_t isJoint; //Jogging mode: 0, Jog in Cartesian coordinate system. 1, Jog in Joint coordinate system
	uint8_t cmd; //Jogging command: 0-10
	}JOGCmd;
	//Details for jogging commands

	enum {
	IDLE, // Idle
	AP_DOWN, // X+/Joint1+
	AN_DOWN, // X-/Joint1-
	BP_DOWN, // Y+/Joint2+
	BN_DOWN, // Y-/Joint2-
	CP_DOWN, // Z+/Joint3+
	CN_DOWN, // Z-/Joint3-
	DP_DOWN, // R+/Joint4+
	DN_DOWN, // R-/Joint4-
	LP_DOWN, // L+
	LN_DOWN // L-
	};
	jogCmd: JOGCmd pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.11 PTP

PTP mode supports MOVJ, MOVL, and JUMP, which is point-to-point movement. The trajectory of playback depends on the motion mode.

- MOVJ: Joint movement. From point A to point B, each joint will run from initial angle to its target angle, regardless of the trajectory, as shown in Figure 1.1.

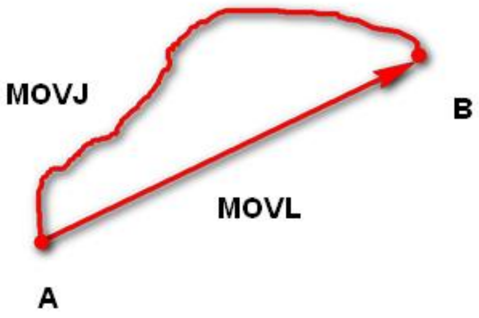


Figure 1.1 MOVL/MOVJ mode

- MOVL: Rectilinear movement. The joints will perform a straight line trajectory from point A to point B, as shown in Figure 1.1.
- JUMP: From point A to point B, The joints will move in MOVJ mode, of which the trajectory looks like a door, as shown in Figure 1.2.
    1. Move up to the lifting Height (**Height**) in MOVJ mode.
    2. Move up to the maximum lifting height (**Limit**).
    3. Move horizontally to a point that is above B by height.
    4. Move down to a point that is above B by height, which the height of the point is that of point B plus **Height**.
    5. Move down to Point B.

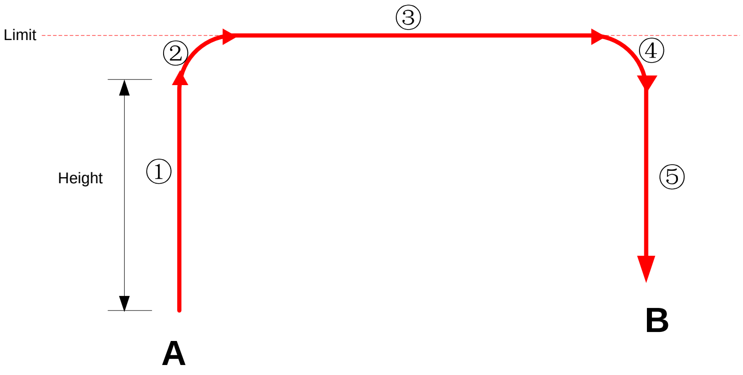
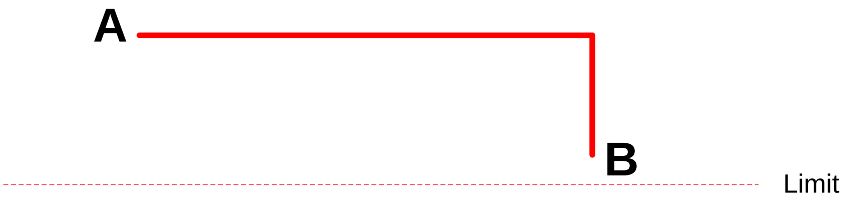


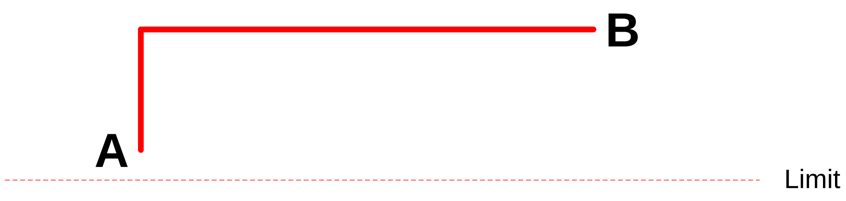
Figure 1.2 JUMP mode

In JUMP mode, if the starting point or the end point is higher than or equal to **Limit**, or the height that the end effector lifts upwards is higher than or equal to **Limit**, the trajectory is different to that of Figure 1.2. Assuming that point A is the starting point, point B is the end point, **Limit** is the maximum lifting height, and **Height** is the lifting height.

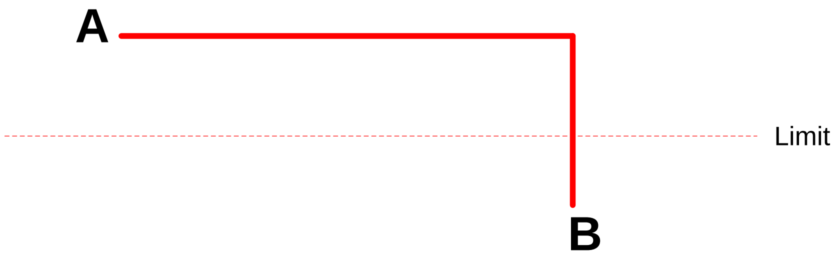
- Point A and point B are both higher than **Limit**, but point A is higher than point B.



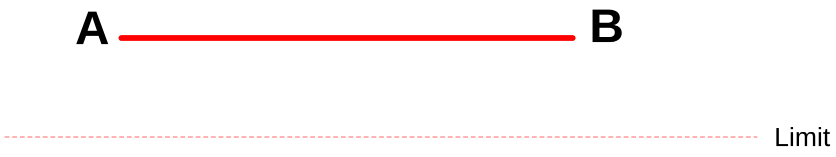
- Point A and point B are both higher than **Limit**, but point B is higher than point A.



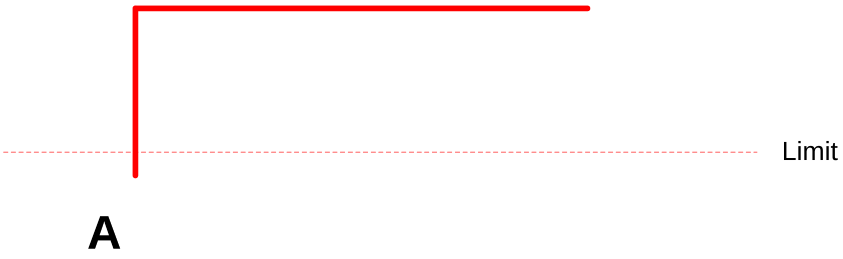
- Point A is higher than **Limit**, but point B is lower than **Limit**.



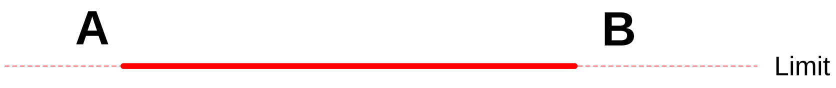
- The height of point A is the same as that of point B, but both are higher than **Limit**.



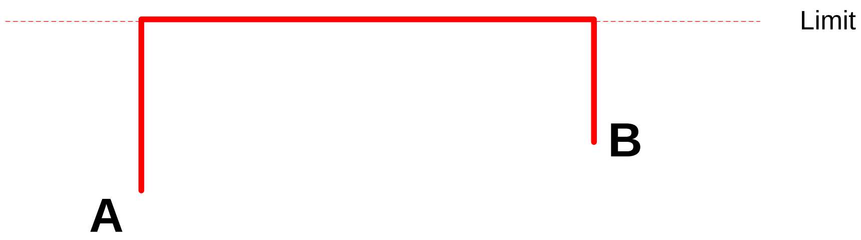
- Point A is lower than **Limit**, but point B is higher than **Limit**.



- The height of point A and point B are both the same as **Limit**.



- Point A and point B are both lower than **Limit**, but the height that the height of point A plus **Height** and that of point B plus **Height** is higher than **Limit**.



1.11.1 Setting the Lifting Height and the Maximum Lifting Height in JUMP mode

Table 1.26 Set the lifting height and the maximum lifting height in JUMP mode

Prototype	int SetPTPJumpParams(PTPJumpParams *ptpJumpParams, bool isQueued, uint64_t *queuedCmdIndex)
Description	Set the lifting height and the maximum height in JUMP mode

Parameter	PTPJumpParams:
	typedef struct tagPTPJumpParams {
	float jumpHeight; //Lifting height
	float zLimit; //Maximum lifting height
	}PTPJumpParams;
	ptpJumpParams: PTPJumpParams pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.11.2 Getting the Lifting Height and the Maximum Lifting Height in JUMP mode

Table 1.27 Get the lifting height and the maximum lifting height in JUMP mode

Prototype	int GetPTPJumpParams(PTPJumpParams *ptpJumpParams)
Description	Get the lifting height and the maximum lifting height in JUMP mode
Parameter	PTPJumpParams:

	typedef struct tagPTPJumpParams {
	float jumpHeight; //Lifting height
	float zLimit; //Maximum lifting height
	}PTPJumpParams;
	ptpJumpParams: PTPJumpParams pointer

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.11.3 Setting the Velocity Ratio and Acceleration Ratio in PTP Mode

Table 1.28 Set the velocity ratio and the acceleration ratio in PTP mode

Prototype	int SetPTPCommonParams(PTPCommonParams *ptpCommonParams, bool isQueued, uint64_t *queuedCmdIndex)
Description	Set the velocity ratio and acceleration ratio in PTP mode

Parameter	PTPCommonParams:
	typedef struct tagPTPCommonParams {
	float velocityRatio; //Velocity ratio
	float accelerationRatio; //Acceleration ratio
	}PTPCommonParams;
	ptpCommonParams: PTPCommonParams pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.11.4 Getting the Velocity Ratio and Acceleration Ratio in PTP Mode

Table 1.29 Get the velocity ratio and acceleration ratio in PTP mode

Prototype	int GetPTPCommonParams(PTPCommonParams *ptpCommonParams)
Description	Get the velocity ratio and acceleration ratio in PTP mode
Parameter	PTPCommonParams: typedef struct tagPTPCommonParams {

	float velocityRatio; //Velocity ratio
	float accelerationRatio; //Acceleration ratio
	}PTPCommonParams;
	ptpCommonParams: PTPCommonParams pointer

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.11.5 Executing a PTP Command

Table 1.30 Execute a PTP command

Prototype	int SetPTPCmd(PTPCmd *ptpCmd, bool isQueued, uint64_t *queuedCmdIndex)
Description	Execute a PTP command. Please call this API after setting the related parameters in PTP mode to make the Dobot move to the target point

Parameter	PTPCmd:
	typedef struct tagPTPCmd {
	uint8_t ptpMode; //PTP mode (0-9)
	float x; //Coordinate parameters in PTP mode. (x,y,z,r) can be set to Cartesian coordinate, joints angle, or increment of them
	float y;
	float z;
	float r;
	}PTPCmd;
	Details for ptpMode:
	enum {
	JUMP_XYZ, //JUMP mode, (x,y,z,r) is the target point in Cartesian coordinate system
	MOVJ_XYZ, //MOVJ mode, (x,y,z,r) is the target point in Cartesian coordinate system
	MOVL_XYZ, //MOVL mode, (x,y,z,r) is the target point in Cartesian coordinate system
	JUMP_ANGLE, //JUMP mode, (x,y,z,r) is the target point in Joint coordinate system
	MOVJ_ANGLE, //MOVJ mode, (x,y,z,r) is the target point in Joint coordinate system
	MOVL_ANGLE, //MOVL mode, (x,y,z,r) is the target point in

	Joint coordinate system MOVJ_INC, //MOVJ mode, (x,y,z,r) is the angle increment in Joint coordinate system MOVL_INC, //MOVL mode, (x,y,z,r) is the Cartesian coordinate increment in Joint coordinate system MOVJ_XYZ_INC, //MOVJ mode, (x,y,z,r) is the Cartesian coordinate increment in Cartesian coordinate system JUMP_MOVL_XYZ, //JUMP mode, (x,y,z,r) is the Cartesian coordinate increment in Cartesian coordinate system }; ptpCmd: PTPCmd pointer isQueued: Whether to add this command to the queue queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.12 CP

CP: Continuous Path.

1.12.1 Executing the CP Command

Table 1.31 Execute the CP command

Prototype	int SetCPCmd(CPCmd *cpCmd, bool isQueued, uint64_t *queuedCmdIndex)
Description	Execute the CP commands

Parameter	CPCmd
	typedef struct tagCPCmd {
	uint8_t cpMode; //CP mode. **0**: indicate that (x,y,z) is the Cartesian coordinate increment. **1**:indicate (x,y,z) is the target point in Cartesian coordinate system
	float x; //(x,y,z)can be set to Cartesian coordinate, or Cartesian coordinate increment
	float y;
	float z;

	union {
	float velocity; //Reserved
	float power; //Reserved
	}CPCmd;
	cpCmd: CPCmd pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

⚠NOTICE

When there are multiple CP commands in the command queue, the Dobot controller will look ahead automatically. The look-ahead condition is that there are no JOG, PTP, ARC, WAIT, and TRIG commands between the CP commands.

1.12.2 Executing the CP Command with the Laser Engraving

Table 1.32 Execute the CP command with laser engraving

Prototype	int SetCPCmd(CPCmd *cpCmd, bool isQueued, uint64_t *queuedCmdIndex)
Description	Execute the CP command with the laser engraving.

Parameter	typedef struct tagCPCmd {
	uint8_t cpMode; //CP mode. **0**: indicate that (x,y,z) is the Cartesian coordinate increment. **1**:indicate (x,y,z) is the target point in Cartesian coordinate system
	float x; //(x,y,z)can be set to Cartesian coordinate, or Cartesian coordinate increment
	float y;
	float z;
	union {
	float velocity; // Reserved
	float power; //Laser power 0-100
	}CPCmd;

	cpCmd: CPCmd pointer isQueued: Whether to add this command to the queue queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.13 ARC

The trajectory of the Dobot in ARC mode is an arc, which is determined by three points (the current point, any point and the end point on the arc), as shown in Figure 1.3.

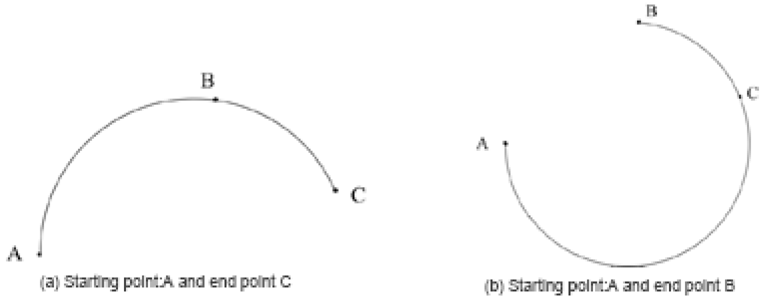

B
C
A
(a) Starting point:A and end point C
B
C
A
(b) Starting point:A and end point B

Figure 1.3 ARC mode

1.13.1 Executing the ARC Command

Table 1.33 Execute the ARC command

Prototype	int SetARCCmd(ARCCmd *arcCmd, bool isQueued, uint64_t *queuedCmdIndex)
Description	Execute the ARC command. Please call this API after setting the related parameters in ARC mode to make Dobot move to the target point. In ARC mode, it is necessary to confirm the three points with other motion modes.

Parameter	ARCCmd:
	typedef struct tagARCCmd {
	struct {
	float x;
	float y;
	float z;
	float r;

	}cirPoint ; //Middle point. (x,y,z,r) can be set to Cartesian coordinate
	struct {
	float x;
	float y;
	float z;
	float r;
	}toPoint; //End point. (x,y,z,r) can be set to Cartesian coordinate
	}ARCCmd;
	arcCmd: ARCCmd pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.13.2 Executing the CIRCLE Command

The CIRCLE mode is similar to the ARC mode, where the trajectory is a circle.

Table 1.34 Execute the CIRCLE command

Prototype	int SetCircleCmd(CircleCmd *circleCmd, bool isQueued, uint64_t *queuedCmdIndex)
Description	Execute the CIRCLE command. Please call this API after setting the related parameters of playback in CIRCLE mode to make Dobot move to the target point. In CIRCLE mode, it is necessary to confirm the three points with other motion modes.

Parameter	CircleCmd
	typedef struct tagCircleCmd {
	struct {
	float x;
	float y;
	float z;
	float r;
	}cirPoint ; //Middle point.(x,y,z,r) can be set to Cartesian

	coordinate
	struct {
	float x;
	float y;
	float z;
	float r;
	}toPoint; //End point. (x,y,z,r) can be set to Cartesian coordinate
	uint32_t count //Circle number
	}CircleCmd;
	circleCmd: CircleCmd pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.14 WAITING

1.14.1 Executing the Waiting Command

Table 1.35 Execute the Waiting command

Prototype	int SetWAITCmd(WAITCmd *waitCmd, bool isQueued, uint64_t *queuedCmdIndex)
Description	Execute the Waiting command. If you need to set the pause time between the two commands, please call this API This command must be added to the command queue, namely, **isQueued** must be set to **1**. If not, the parameter **timeout** of **Waiting** command in the command queue being executed may be changed because the **WAITCmd** memory is shared

Parameter	WAITCmd:
	typedef struct tagWAITCmd {
	uint32_t timeout; //Unit:ms
	}WAITCmd;
	waitCmd: WAITCmd pointer
	isQueued: Whether to add this command to the queue

	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.15 TRIGGERING

1.15.1 Executing the Triggering Command

Table 1.36 Execute the Triggering command

Prototype	int SetTRIGCmd(TRIGCmd *trigCmd, bool isQueued, uint64_t *queuedCmdIndex)
Description	Execute the triggering command. This command must be added to the command queue, namely, **isQueued** must be set to **1**. If not, the parameter **condition** of the **Triggering** command in the queue command being executed may be changed because the **TRIGCmd** memory is shared

Parameter	TRIGCmd:
	typedef struct tagTRIGCmd {
	uint8_t address; // EIO address: If **mode** is set to **0**, the value range is **1** to **24**. If mode is set to **1**, the value range is **1** to **6**
	uint8_t mode; //Triggering mode. **0**: Level trigger.**1**:A/D trigger
	uint8_t condition; //Triggering condition
	Level: **0**, equal. **1**, unequal
	A/D: **0**, less than. **1**,less than or equal
	2, greater than or equal. **3**, greater than
	uint16_t threshold; //Triggering threshold. Level : **0**,**1** .A/D: **0-4095**
	}TRIGCmd;
	trigCmd: TRIGCmd pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.16 EIO

In the Dobot controller, the addresses of the I/O interfaces are unified. Here, you can see as follows:

- High-low level output.
- Read High-low level output.
- Read analog-digital conversion value output.

For more details, please see *Dobot M1 User Guide.*

1.16.1 Setting the I/O Output

Table 1.37 Set the I/O output

Prototype	int SetIODO(IODO *ioDO, bool isQueued, uint64_t *queuedCmdIndex)
Description	Set the I/O output

Parameter	IODO:
	typedef struct tagIODO {
	uint8_t address; //I/O addres:**1-22**
	uint8_t level; //**0**: Low level.**1**: High level
	}IODO;
	ioDO: IODO pointer
	isQueued: Whether to add this command to the queue
	queuedCmdIndex: If this command is added to the queue, **queuedCmdIndex** indicates the index of this command in the queue. Otherwise, it is invalid

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_BufferFull: The command queue is full DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.16.2 Getting the I/O Output

Table 1.38 Get the I/O output

Prototype	int GetIODO(IODO *ioDO)
Description	Get the I/O output

Parameter	IODO:
	typedef struct tagIODO {
	uint8_t address; //I/O addres: **1-22**
	uint8_t level; //**0**: Low level.**1**: High level
	}IODO;
	ioDO: IODO pointer

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.16.3 Getting the I/O Input

Table 1.39 Get the I/O input

Prototype	int GetIODI(IODI *ioDI)
Description	Get the I/O input

Parameter	IODI:
	typedef struct tagIODI {
	uint8_t address; //I/O address: **1-24**
	uint8_t level; //**0**: Low level. **1**: High-level
	}IODI;
	ioDI: IODO pointer

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.16.4 Getting the A/D Input

Table 1.40 Get the A/D Input

Prototype	int GetIOADC(IOADC *ioADC)
Description	Get the A/D input

Parameter	IOADC:
	typedef struct tagIOADC {
	uint8_t address; //I/O address: **1-6**
	uint16_t value; //Input value: **0-4095**
	}IOADC;

	ioADC: IOADC pointer
Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.17 LAN

The Dobot can be connected to a Computer via LAN. After connecting the Dobot to the computer via network cable, you need to set the IP address, Sub netmask, Gateway to make the Dobot access LAN. After the access is successful, you can connect your Dobot to your Computer without using a USB cable.

1.17.1 Setting the LAN

Table 1.41 Set the LAN

Prototype	int SetFirmwareLanConfig(LanConfig * config)
Description	Set the LAN

Parameter	LanConfig:
	typedef struct tagLanConfig {
	uint8_t status; //Reserved
	bool dhcp; // Whether to enable DHCP. **0**: Disabled**1**:Enabled
	uint8_t addr[16]; //IP adress
	uint8_t mask[16]; //Sub netmask
	uint8_t gateway[16]; //Gateway
	uint8_t dns[16]; //DNS
	}LanConfig;
	Config: LanConfig pointer

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.17.2 Getting the LAN

Table 1.42 Get the LAN

Prototype	int GetFirmwareLanConfig (LanConfig * config)
Description	Get the LAN
Parameter	LanConfig: typedef struct tagLanConfig {

	uint8_t status; //Connection status
	0: Disconnected
	1: Connected
	2: Searching
	3: Connecting
	4: Disabled
	5: Error
	bool dhcp; // Whether to enable DHCP. **0**: Disabled**1**:Enabled
	uint8_t addr[16]; //IP adress
	uint8_t mask[16]; //Sub netmask
	uint8_t gateway[16]; //Gateway
	uint8_t dns[16]; //DNS
	}LanConfig;
	Config: LanConfig pointer

Return	DobotCommunicate_NoError: The command returns with no error DobotCommunicate_Timeout: The command does not return, resulting in a timeout

1.18 Other functions

1.18.1 Event Loop

In some languages, the application exits directly after calling an API because there is no event loop, resulting in the command unable to be issued to the Dobot controller. To avoid this, we provide an event loop API, which is called before the application exits (currently known, Python need to follow this).

Table 1.43 Event loop

Prototype	void DobotExec(void)
Description	Event loop
Parameter	None
Return	Void