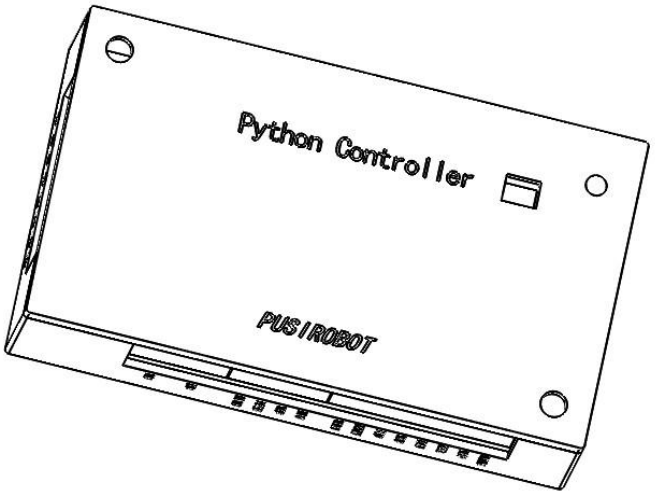
PUSIROBOT

CQPUSI ROBOT CONTROL SYSTEM

User Manual

Python Controller Series

Programmable controller



1. Version Control

1） Update Records

Date	Author	Version	Remark
2024-9-30	DengHE	V1.0.1	Initial

Catalog

1 Introduction

1.1 Disclaimer

The using method of the device and other content in the description of this manual is only used to provide convenience for you. To ensure the application conforms to the technical specifications is the responsibility of your own. CQPUSI does not make any form of statement or guarantee to the information, which include but not limited to usage, quality, performance, merchantability or applicability of specific purpose. CQPUSI is not responsible for these information and the consequences result caused by such information. If the CQPUSI device is used for life support and/or life safety applications, all risks are borne by the buyer. The buyer agrees to protect the CQPUSI from legal liability and compensation for any injury, claim, lawsuit or loss caused by the application.

2 Overview

2.1 General Description

The Python Controller is a programmable controller that integrates communication interfaces such as CANOpen, RS485, and SPI, along with analog input, isolated PWM output, and motor control interfaces. It can connect to various devices and achieve complex control logic through MicroPython programming.

2.2 Features

- ✓ 5V Type-C power supply
- ✓ Wide voltage supply range of 9-36V
- ✓ Programmable control via MicroPython
- ✓ One CAN channel
- ✓ Two RS485 channels
- ✓ Control channel for two DC motors or one stepper motor
- ✓ One analog input interface (4-20mA or 0-10V optional)
- ✓ One SPI interface
- ✓ Two GPIO interfaces
- ✓ Two isolated PWM output interfaces

3 Connector Description

3.1 Terminal port location

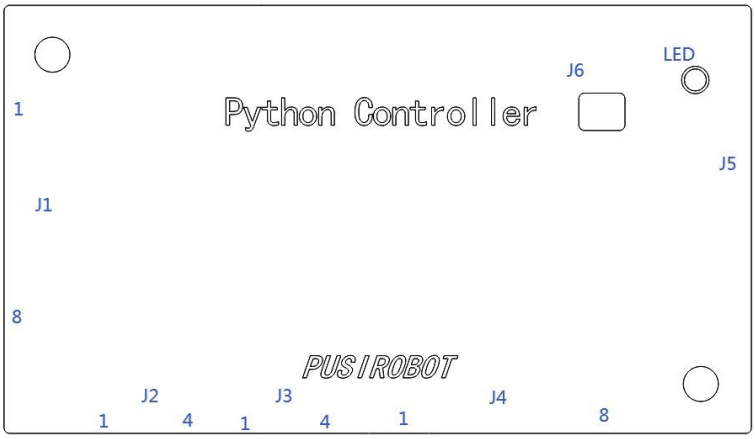


Figure 1

3.2 Connection J1

Pin No:	1	2	3	4	5	6	7	8
Designator	24+	GND	TX1	RX1	CANL	CANH	GPIO-1	GPIO-2

Description:

24+: Positive terminal for the controller's wide voltage power supply, 9-36V;

GND: Negative terminal for the controller's wide voltage power supply;

TX1: RS485 bus transmission signal 1, interface withstand voltage 5V;

RX1: RS485 bus receiving signal 1, interface withstand voltage 5V;

CANL: Connects to the CAN bus, interface withstand voltage 5V;

CANH: Connects to the CAN bus, interface withstand voltage 5V;

GPIO-1: General IO port 1, can be used for input or output, high level is 3.3V, interface withstand voltage 3.3V;

GPIO-2: General IO port 2, can be used for input or output, high level is 3.3V, interface withstand voltage 3.3V.

3.3 Motor connection J2

Pin No:	1	2	3	4
Designator	M1+	M1−	M2+	M2−

Description:

M1+: solenoid valve 1+ (DC brush motor 1+);

M1-: solenoid valve 1- (DC brush motor 1-);

M2+: solenoid valve 2+ (DC brush motor 2+);

M2-: solenoid valve 2- (DC brush motor 2-).

3.4 SPI connection J3

Pin No:	1	2	3	4
Designator	MISO	MOSI	SCK	CSN

Description:

MISO: Pin for the master device to send data, connected to the receiver of the slave device, interface withstand voltage 3.3V;

MOSI: Pin for the master device to receive data, connected to the transmitter of the slave device, interface withstand voltage 3.3V;

SCK: Clock signal, interface withstand voltage 3.3V;

CSN: Pin used to select the slave device, interface withstand voltage 3.3V.

3.5 Connection J4

Pin	1	2	3	4	5	6	7	8
Designator	PWM1	PWM2	AIN-	AIN+	RX2	TX2	GND	VDD5

Signal Description:

PWM1: PWM output 1;

PWM2: PWM output 2;

AIN-: Negative terminal for 4-20mA analog input;

AIN+: Positive terminal for 4-20mA / 0-10V analog input;

RX2: RS485 bus receiving signal 2, interface withstand voltage 5V;

TX2: RS485 bus transmission signal 2, interface withstand voltage 5V;

GND: Digital ground for the controller;

VDD5: 5V output from the controller.

3.6 Type-C connection J5

Connect to a computer via the Type-C interface for programming.

3.7 Reset button J6

Hold the button while connecting the device to the computer; the device will be recognized as a storage device.

4 Programming environment

4.1 Programming language

MicroPython is a complete software implementation of the Python 3 programming language, written in C and optimized to run on microcontrollers. It is a full Python compiler and runtime system that operates on microcontroller hardware. It provides an interactive prompt (REPL) for immediate command execution. In addition to the selected core Python libraries, MicroPython includes modules for accessing low-level hardware.

MicroPython strives to be as compatible as possible with standard Python (CPython), so if you know Python, you already know MicroPython. The more you understand MicroPython, the better you will perform in Python.

The examples presented here primarily rely on the 'machine' library in MicroPython, so for specific interpretations beyond the examples shown, please refer to the official documentation for function descriptions.

4.2 Programming software

MicroPython IDEs include Thonny, VSCode, and PyCharm, which can be downloaded and installed from their official websites. Thonny is specifically designed for MicroPython and can be used immediately after installation, while VSCode and PyCharm require additional plugins to provide MicroPython support.

4.3 Device connection

4.3.1 Connect PC

Connect the device to the computer using a Type-C data cable, then check the ports in Device Manager (see Figure 2).

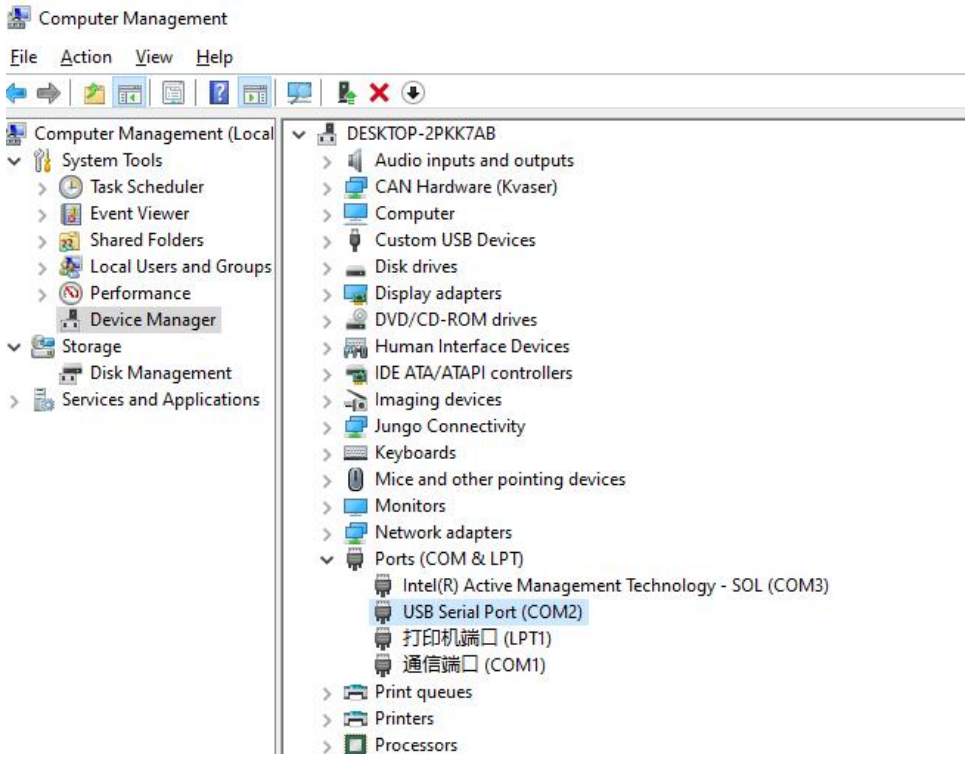


Figure 2

4.3.2 Thonny connection

Once the device is connected to the computer, open Thonny, click on the bottom right corner of the software, select the corresponding port for the device, and click to connect to COM18 (see Figure 3).

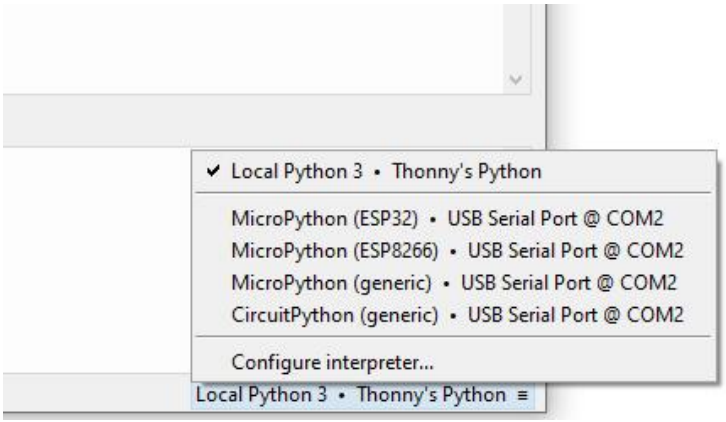


Figure 3

4.3.3 VSCode connection

Open the PyMakr plugin page, and in the DEVICES window, click to connect the device on the corresponding port: COM18 (see Figure 4).

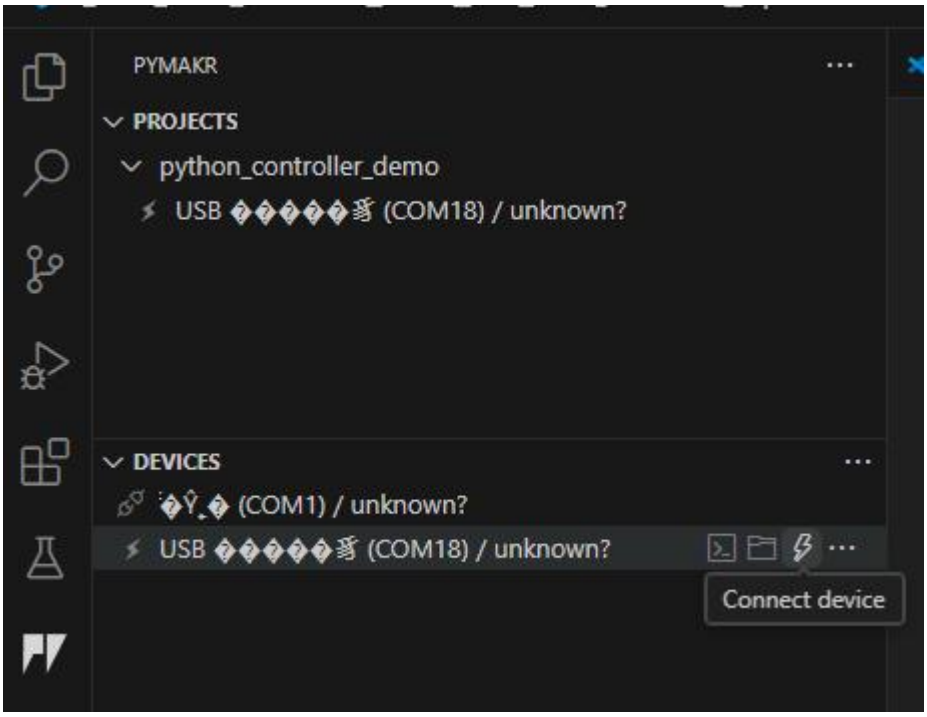


Figure 4

4.3.4 Pycharm connection

Install the MicroPython plugin and configure MicroPython settings (see Figure 5). PyCharm will automatically recognize the device and connect to it.

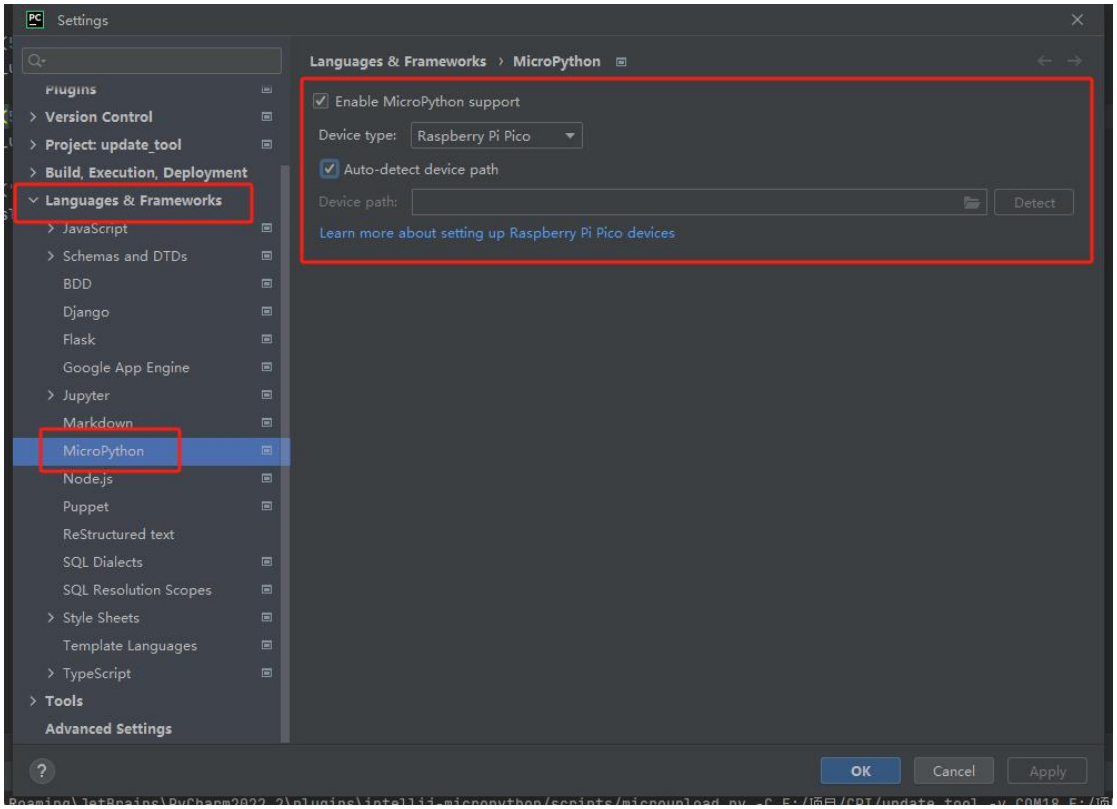


Figure 5

5 CANopen communication

The Python Controller has one CAN channel and includes a built-in CAN programming library, which makes it easy to handle CAN communication.

5.1 CANopen overview

CAL provides all the network management services and message transport protocols, but does not define the content of the objects or the types of objects being communicated (it only defines how, not what). This is the entry point for CANopen. CANopen is developed on the basis of CAL, utilizing a subset of CAL communication and service protocol, providing an implementation solution for distributed control systems. CANopen allows for functional expansion of nodes while ensuring interoperability of network nodes, whether simple or complex.

The core concept of CANopen is the Object Dictionary (OD), which is also used as a device description format in other fieldbus systems (such as Profibus and Interbus-S). CANopen communication can access all parameters of the drive through the Object Dictionary (OD). Note: The Object Dictionary is not part of CAL, but is implemented in CANopen. The Object Dictionary supported by the PMC006CX is shown in Appendix 1.

The CANopen communication model defines several types of messages (communication objects):

Abbreviation	Full Name	Description
SDO	Service Data Object	Used for non-time-critical data, such as parameters.
PDO	Process Data Object	Used for transmitting time-critical process data (e.g., set values, control words, status information).
SYNC	Synchronization Message	Used for synchronizing CAN nodes.
EMCY	Emergency Message	Used for transmitting emergency events from the drive.
NMT	Network Management	Used for managing the CANopen network.
Heartbeat	Error Control Protocol	Used to monitor the life status of all nodes.

5.2 CAN Frame Structure

CAN transmits data between the host (controller) and bus nodes through data frames. The table below shows the structure of the data frame.

Frame Header	Arbitration Field		Control Field	Data Field	Checksum Field	Acknowledgment Field	Frame Trailer
	COB-ID (Communication Object Identifier)	RTR (Remote Request)					
1 bit	11 or 29 bits	1 bit	6 bit	0~8bytes	16 bit	2bit	7bit

5.3 CAN connection

Using a CAN bus connection allows for a maximum transmission distance of 5000 meters. The diagram provides a network solution for connecting multiple CAN devices via the CAN bus, supporting up to 127 nodes.

Note: It is recommended to use shielded twisted pair cables specifically for CAN bus, with a 120-ohm terminating resistor connected at both ends of the twisted pair.

5.4 CAN Firmware Library

The Python Controller has a built-in CAN programming library called pycan, which enables easy implementation of CAN communication programming.

5.4.1 Function init

Function name	init
Function prototype	def __init__(self, bitrate)
Description	Initializes the CAN object
	Input parameter
bitrate	Baud rate in kbps.
	Return value
can	CAN object.

Example:

```
# Initialize a CAN object with a baud rate of 125 kbps
can = pylibrary.can(125)
```

5.4.2 Function send

Function name	send
Function prototype	def send(self, node_id, data_count, data)
Description	Sends data
	Input parameter
node_id	Frame ID
data_count	Length of data
data	Data to send
	Return value
-	-

Example:

```
can.send(0x0603, 8, [0x40, 0x0C, 0x60, 0x00, 0x01, 0x02, 0x03, 0x04])
```

5.4.3 Function check_message

Function name	Check_message
Function prototype	def check_message(self)
Description	Checks for received data
	Input parameter
-	-
	Return value
receive_data	Indicates if data is received
channel	Receive channel

Example:

```
recive_data, channel = can.check_message()
```

5.4.4 Function receive_interrupt_init

Function name	receive_interrupt_init
Function prototype	def receive_interrupt_init(self, handler)
Description	Configures reception interrupt
	Input parameter
handler	Callback function, which must have exactly one parameter pin
	Return value
-	-

Example:

```
def revice_handler(pin):
    while 1:
        recive_message, channel = can.check_message()
        if recive_message:
            frame_id, received_msg = can.receive(channel)
            print("frame_id: ", frame_id, " Received message:", received_msg)
        else:
            break
can.recive_interrupt_init(revice_handler)
```

5.4.5 Function receive

Function name	receive
Function prototype	def receive(self, channel)
Description	Retrieves received data
	Input parameter
channel	Data reception channel, obtained through the check_message function
	Return value
frame_data	Frame ID
received_msg	Receive data

Example:

```
frame_id, received_msg = can.receive(channel)
```

6 RS485 communication

The Python Controller provides two RS485 channels, and RS485 communication can be implemented through MicroPython programming. Initially, you can monitor the command sending and response situation using a serial assistant.

6.1 RS485 Firmware library

6.1.1 Function init

Function name	init
Function prototype	def __init__(self, rs485_number, baudrate)
Description	Initializes the RS485 object
	Input parameter
rs485_number	RS485 channel number (1 or 2) according to the connected channel
baudrate	Baud rate in bps
	Return value
rs485	rs485 object

Example:

```
# Initialize an RS485 object with a baud rate of 9600 bps
rs485 = pylibrary.rs485(1, 9600)
```

6.1.2 Function set_send_mode

Function name	set_send_mode
Function prototype	def set_send_mode(self)
Description	Sets the RS485 mode to send
	Input parameter
-	-
	Return value
-	-

Example:

```
rs485. set_send_mode()
```

6.1.3 Function set_receive_mode

Function name	set_receive_mode
Function prototype	def set_receive_mode(self)
Description	Sets the RS485 mode to receive
	Input parameter
-	-
	Return value
-	-

Example:

```
rs485. set_receive_mode()
```

6.1.4 Function write

Function name	write
Function prototype	def write(self, data)

Description	Sends data. Must be in send mode first
	Input parameter
data	Data to send, of type bytearray
	Return value
-	-

Example:

```
rs485.write(bytearray([0x01, 0x03, 0x60, 0x0C, 0x00, 0x02, 0x1A, 0x08]))
```

6.1.5 Function read

Function name	read
Function prototype	def read(self)
Description	Read data. Must be in receive mode first
	Input parameter
-	-
	Return value
received_msg	Received data

Example:

```
received_msg = rs485.read()
```

6.1.6 Function is_receive_data

Function name	is_receive_data
Function prototype	def is_receive_data(self)
Description	Checks if the receive buffer has data
	Input parameter
-	-
	Return value
is_receive_data	Boolean indicating if data is in the receive buffer (True for data present, False otherwise)

Example:

```
is_receive_data = rs485.is_receive_data()
```

6.1.7 Function is_send_success

Function name	is_send_success
Function prototype	def is_send_success(self)
Description	Checks if data sending is complete
	Input parameter
-	-
	Return value

is_send_success	Boolean indicating if data has been sent (True for complete, False for incomplete)

Example:

is_send_success = rs485.is_send_success()

6.1.8 Function write_read

Function name	write_read
Function prototype	def write_read(self, data)
Description	Sends and reads data
	Input parameter
data	Data to send, of type bytearray
	Return value
receive_data	Received data

Example:

receive_data = rs485.write_read(bytearray([0x01, 0x03, 0x60, 0x0C, 0x00, 0x02, 0x1A, 0x08]))

6.1.9 Function write_read_timeout

Function name	write_read
Function prototype	def write_read(self, data)
Description	Sends and reads data
	Input parameter
data	Data to send, of type bytearray
	Return value
receive_data	Received data

Example:

recive_data = rs485. write_read_timeout(bytearray([0x01, 0x03, 0x60, 0x0C, 0x00, 0x02, 0x1A, 0x08]), 1000)

7 SPI communication

7.1 Overview

SPI (Serial Peripheral Interface) is a full-duplex synchronous serial bus developed by Motorola for communication between microcontroller units (MCUs) and peripheral devices. It is primarily used to interface with EEPROMs, Flash memory, real-time clocks (RTCs), analog-to-digital converters (ADCs), network controllers, MCUs, digital signal processors (DSPs), and digital signal decoders. The SPI system can directly interface with a variety of standard peripheral devices from different manufacturers and generally uses four lines: serial clock line (SCK), master input/slave output data line (MISO), master output/slave input data line (MOSI),

and active-low slave select line (SSEL).

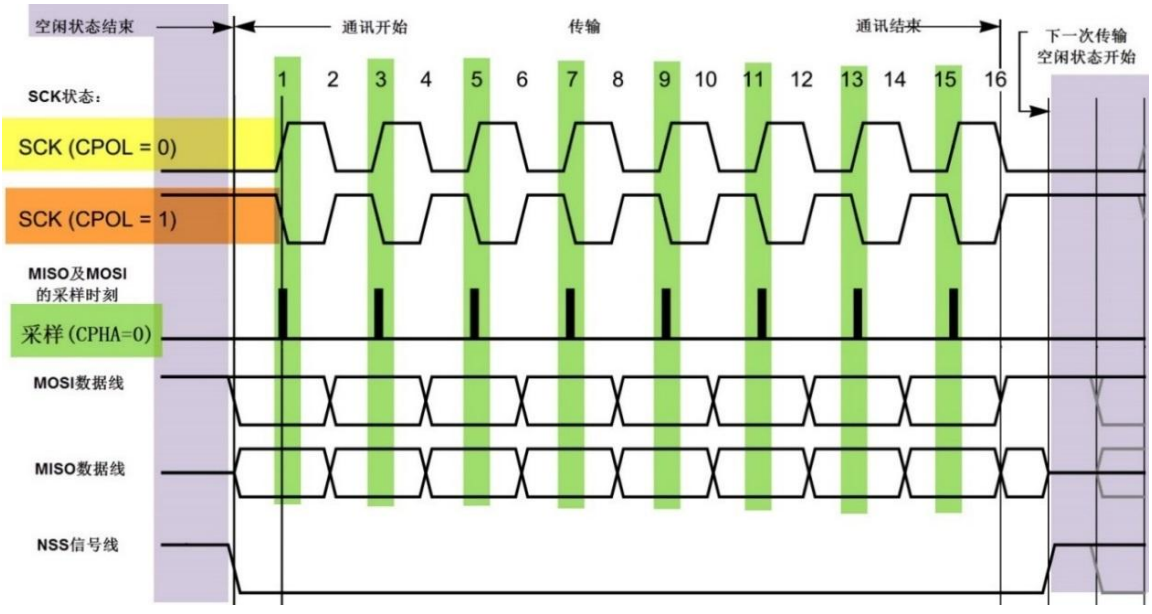
7.2 SPI Communication Lines

- SCK (Serial Clock): Provides the clock signal, controlled by the master. Data transmission relies on the SCK clock signal.
- MOSI (Master Out Slave In): Master output, slave input. Used for sending data from the master device to the slave device.
- MISO (Master In Slave Out): Slave output, master input. Used for sending data from the slave device to the master device.
- SS (Slave Select): Chip select signal used to select the slave device. Each slave device has an independent SS line, which is pulled low to select the corresponding slave device for communication.

7.3 Communication Process

This describes the communication timing for the master. NSS, SCK, and MOSI signals are all generated by the master, while the MISO signal is generated by the slave.

The high bit is sent first, followed by the low bit. After sending one byte, the next byte can be sent without requiring an acknowledgment. The NSS signal is used to select the slave device as a start/stop signal.



7.4 SPI Firmware Library

7.4.1 Function init

Function name	init
Function prototype	def __init__(self, baudrate, polarity, phase, bits)
Description	Initializes the SPI object with MSB bit order only
	Input parameter
baudrate	Baud rate in bps
polarity	Clock polarity
phase	Clock phase
bits	Number of bits (range: 4-8)
	Return value
spi	SPI object

Example:

```
spi = pylibrary.spi(1000000, 0, 0, 8)
```

7.4.2 Function start

Function name	start
Function prototype	Def start(self)
Description	Begins SPI communication by pulling CS low
	Input parameter
-	-
	Return value
-	-

Example:

```
spi.start()
```

7.4.3 Function stop

Function name	stop
Function prototype	Def stop(self)
Description	Ends SPI communication by releasing CS
	Input parameter
-	-
	Return value
-	-

Example:

```
spi.stop()
```

7.4.4 Function transmit

Function name	transmit
Function prototype	Def transmit(self, data_number, send_data)
Description	Sends and receives data
	Input parameter
data_number	Number of data elements to send
data	Data to send, in hex array format
	Return value
receive_data	Received data in hex array format

Example:

receive_data = spi.transmit(3, [0x01,0x02, 0x03])

8 Motor drive

8.1 Overview

The Python Controller provides control channels for two DC motors or solenoids, or one stepper motor. You can adjust the output levels of the two ports through code to change the direction, and adjust the output frequency to modify the running speed of the DC motors or the power supply ratio for the solenoids.

8.2 Motor drive Firmware Library

8.2.1 Function init

Function name	init
Function prototype	def __init__(self, motor_number, vref = 100)
Description	Initializes the motor driver object
	Input parameter
motor_number	Motor number (set to 1 or 2 based on wiring)
vref	Maximum voltage percentage (default 100%)
	Return value
motor	Motor driver object

Example:

motor = pylibrary.motor(1,80)

8.2.2 Function set_direction

Function name	set_direction
Function prototype	Def set_direction(self, dir)
Description	Sets the motor direction
	Input parameter

dir	Direction (1 for forward, 0 for reverse)
	Return value
-	-

Example:

motor.set_direction(1)

8.2.3 Function start

Function name	start
Function prototype	Def start(self, speed)
Description	Starts the motor at the specified speed percentage
	Input parameter
speed	Speed percentage
	Return value
-	-

Example:

Starts the motor at 50% speed

motor.start(50)

8.2.4 Function stop

Function name	stop
Function prototype	Def stop(self)
Description	Stops the motor
	Input parameter
-	-
	Return value
-	-

Example:

motor.stop()

8.2.5 Function set_verf

Function name	set_verf
Function prototype	Def set_verf(self, verf)
Description	Sets the maximum voltage percentage
	Input parameter
verf	Maximum voltage percentage
	Return value
-	-

Example:

```
# Sets the maximum voltage to 60%
motor.set_vref(60)
```

9 Analog Input

The Python Controller provides an analog input interface with an input range of 4-20mA / 0-10V. You can read the input values using MicroPython programming. In practical applications, this value is typically used in calculations to facilitate back-and-forth operations between two points or to set corresponding speed values.

Wiring: Connect the positive and negative outputs to AIN+ and AIN-, respectively.

9.1 ADC Firmware Library

9.1.1 Function init

Function name	init
Function prototype	def __init__(self)
Description	Initializes the ADC object
	Input parameter
-	-
	Return value
-	-

Example:

```
adc = pylibrary.adc()
```

9.1.2 Function read

Function name	read
Function prototype	def read(self)
Description	Reads the analog value
	Input parameter
-	-
	Return value
read_value	Analog value in the range 0-65535

Example:

```
read_value = adc.read()
```

10 PWM output

10.1 Overview

The Python Controller provides two PWM output interfaces. Connect the PWM+ of the

external device to the PWM1 or PWM2 pin on J4, and connect PWM- to the GND pin on J4. You can then achieve the desired PWM output control through MicroPython programming.

10.2 PWM Firmware Library

10.2.1 Function init

Function name	init
Function prototype	def _init_(self, pwm_number)
Description	Initializes the PWM object
	Input parameter
pwm_number	Set to 1 or 2 depending on wiring
	Return value
pwm	PWM object

Example:

pwm = pylibrary.pwm()

10.2.2 Function set_freq

Function name	set_freq
Function prototype	Def set_freq(self, freq)
Description	Sets the PWM frequency
	Input parameter
freq	Frequency in Hz
	Return value
-	-

Example:

pwm.set_freq(10000)

10.2.3 Function set_dutycycle

Function name	set_dutycycle
Function prototype	Def set_dutycycle(self, duty_cycle)
Description	Sets the duty cycle
	Input parameter
duty_cycle	Duty cycle percentage (0-100)
	Return value
-	-

Example:

pwm.set_dutycycle(50)

11 GPIO

The Python Controller provides two GPIO interfaces. You can control them as input or output interfaces through MicroPython programming. As an input, the voltage range is 0-3.3V; as an output, it can output either 3.3V or 0V. An output cannot be connected to GND or another output.

11.1 GPIO Firmware Library

11.1.1 Function init

Function name	init
Function prototype	def __init__(self, pin, mode, pull = None)
Description	Initializes the ADC object
	Input parameter
pin	GPIO channel, set to 1 or 2 based on wiring
mode	GPIO mode. Use IN for input or OUT for output
pull	Pull-up/pull-down setting (default is None). Use PULL_UP for pull-up or PULL_DOWN for pull-down
	Return value
gpio	GPIO object

Example:

```
/*initialize a GPIO object using channel 1, input mode, and the default pull-up*/
gpio = pylibrary.gpio(1, mode=pylibrary.gpio.IN, pull=pylibrary.gpio.PULL_UP)
```

11.1.2 Function bit_set

Function name	bit_set
Function prototype	def bit_set(self)
Description	Sets GPIO output to high
	Input parameter
-	-
	Return value
-	-

Example:

```
gpio.bit_set()
```

11.1.3 Function bit_reset

Function name	bit_reset
Function prototype	def bit_reset(self)
Description	Sets GPIO output to low
	Input parameter

-	-
	Return value
-	-

Example:

gpio.bit_reset()

11.1.4 Function bit_write

Function name	bit_write
Function prototype	def bit_write(self, value)
Description	Sets GPIO output level
	Input parameter
value	GPIO level (1 for high, 0 for low)
	Return value
-	-

Example:

gpio.bit_write(1)

11.1.5 Function bit_read

Function name	bit_read
Function prototype	def bit_read(self)
Description	Reads GPIO level
	Input parameter
-	-
	Return value
value	GPIO level (1 for high, 0 for low)

Example:

value = gpio.bit_read()

12 Electrical Characteristics

Parameter	Condition	Min	Typical	Max	Unit
Input voltage	25℃	9	24	36	V
Temperature	24V input voltage	-20		85	℃
IO max current	Sourcing/sinking	2	8	16	mA
Output current per phase	25℃	0	2	3	A

13 Dimensions (Unit: mm)