2.2 Low-level Parsing at the Slave End (MicroPython Version)

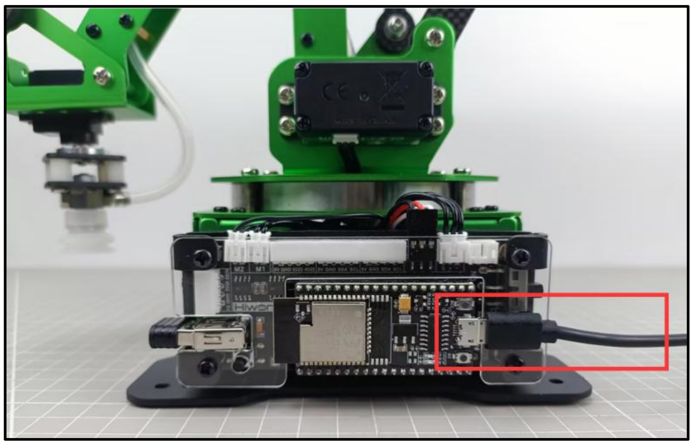
The lesson will explain the MicroPython version of low-level program of MaxArm's slave-side communication control functions, analyzing MaxArm's reception of data from other devices, parsing the data to control MaxArm, and implementing the functionality to send data to other devices.

1. Communication Connection

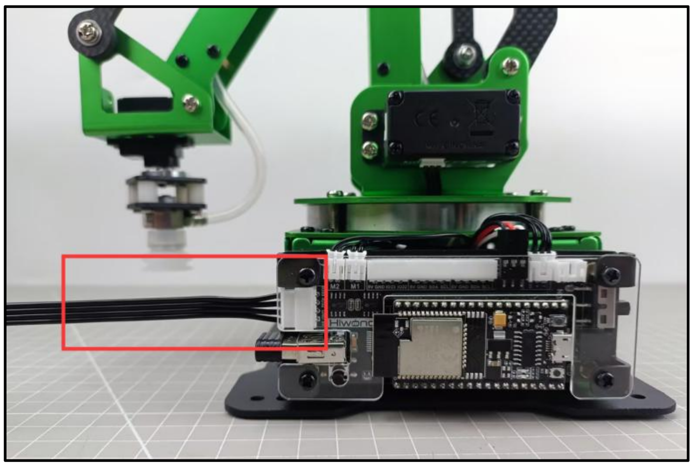
1.1 Hardware Connection

MaxArm's serial communication supports two interfaces, one is the 4-pin interface, suitable for devices with pin interfaces for UART communication, commonly used with Arduino development boards and STM32 development board. The other interface is the micro-USB interface, suitable for devices with USB master interfaces, such as Raspberry Pi, Jetson Nano and others.

1) The communication connection for the micro-USB interface is as follow:



2) The communication connection for 4Pin interfaces is as follow:



1.2 Communication Protocol

The communication protocol for both the master and slave devices in the MaxArm communication routines follows the following format:

Frame header	**Function code**	**Data length**	**Data information**	**Check bit**
0xAA 0x55	func	len	data	check

The annotations of each part of the protocol are as follows:

Frame header: if 0xAA and 0x55 are received sequentially, it indicates that there is data to be received, consisting of a fixed 2 bytes.

Function Code: Used to indicate the purpose of an information frame, consists of 1 byte.

Data Length: Indicates the number of data bits carried by the data frame.

Check Bit: Verifies the correctness of the data frame. If correct, the corresponding function is called; otherwise, the data frame is skipped.

The calculation method for the check bit is: calculate the sum of the function code, data length, and data, then take the complement, and finally, take the low byte, which serves as the checksum.

1.3 Functions and Corresponding Function Code Instructions

Function Name	Instruction	Function Code
FUNC_SET_ANGLE	Set servo angle	0x01
FUNC_SET_XYZ	Set the robotic arm coordinate position	0x03
FUNC_SET_PWMSERVO	Control PWM servo for nozzle	0x05
FUNC_SET_SUCTIONNOZZLE	Control the air pump state	0x07
FUNC_READ_ANGLE	Control servo angle	0x11
FUNC_READ_XYZ	Read robotic arm coordinate position	0x13

2. Program Interface Parsing

The routines analyzed in this document are located in **“Communication Routine Low-Level Files (MicroPython Version) File/ MaxArm_microPython_microUSB”** in the same directory. This lesson will primary analyze the “MaxArm_ctl.py” file, which is the program file implementing the underlying communication of the MaxArm robot's slave device.

2.1 Constructor

“**def _init_(self)**” is the constructor of MaxArm message receiving class, where a protocol parser, a buffer for receiving data, a bus servo control object, and a MaxArm robotic arm control object are created, and initializes the protocol parser to start parsing from the first byte of the data packet.

```
# 构造函数
def __init__(self):
  self.__pk_ctl = PacketController()
  self.data = []
  self.bus_servo = BusServo()
  self.arm = ESPMax(self.bus_servo)
  self.__pk_ctl.state = PacketControllerState.STARTBYTE1
```

2.1.1 Communication Protocol Format Enumeration

This class mainly defines 6 constants, corresponding to the frame header, function code, data length, data and check bit of the communication protocol respectively. These constants are sued to provide parsing status when parsing the data packets.

```
# 通信协议的格式枚举
class PacketControllerState:
  # 0xAA 0x55 Length Function Data Checksum
  STARTBYTE1 = const(0)
  STARTBYTE2 = const(1)
  FUNCTION = const(2)
  LENGTH = const(3)
  DATA = const(4)
  CHECKSUM = const(5)
```

2.1.2 Command Packet Construction

Encapsulate control commands and data into a data packet, and subsequently communicate with the master directly in the form of data packets. In the constructor of this class, all variables created in the format of the communication protocol are initialized to 0x00.

```
# 命令包结构
class PacketRawFrame:
  def __init__(self):
    self.start_byte1 = 0x00
    self.start_byte2 = 0x00
    self.function = 0x00
    self.data_length = 0x00
    # self.data = [0x00] * 256
    self.data = []
    self.checksum = 0x00
```

2.1.3 Protocol Parser

Construct variables related to protocol parsing and buffer:

state: Controls the current state of data packet parsing.

frame: An object representing a data packet, used for receiving or sending data packets.

data_index: Records the number of data processed.

len: Length of received data.

Data: buffer for receiving data.

```
# 协议解析器
class PacketController:
  def __init__(self):
    # 解析协议的相关变量
    self.state = PacketControllerState()
    self.frame = PacketRawFrame()
    self.data_index = 0
    self.index = 0
    # 缓冲区相关变量
    self.len = 0
    # data = [0x00] * 256
    self.data = []
```

2.2 Serial Port Activation Function

The begin () function identifies the serial port to be activated based on the parameter “port” passed to it.

If the value is PORT_FOR_USB, it opens the serial port for MicroUSB, where the corresponding pins are: TX->10, RX->9.

If the value is PORT_FOR_4Pin, it opens the serial port for the 4-pin interface, where the corresponding pins are: TX->33, RX->32.

Finally, it creates an object for the air pump and sets the servo controlling the nozzle to rotate to the position of 1500 (i.e., the central position).

```
# 串口开启函数，
# 若接USB口通讯，则用SELECT_PORT.PORT_FOR_USB；用4pin口通讯，则用SELECT_PORT.PORT_FOR_4Pin
def begin(self , port):
  if port == SELECT_PORT.PORT_FOR_USB:
    print("begin in USB")
    self.__uart = UART(1, 9600, tx=10, rx=9)
  else:
    print("begin in 4Pin")
    self.__uart = UART(1, 9600, tx=33, rx=32)
  self.nozzle = SuctionNozzle()
  self.arm.go_home(1500)
  time.sleep_ms(2000)
```

2.2.1 The enumeration for Selecting the Open Serial Port

The value of the parameter "**port**" can be used to select one of two serial ports for activation, with the corresponding parameter values being:

0x01: Activate the USB serial port

0x03: Activate the serial port for the 4-pin interface

```
class SELECT_PORT:
  PORT_FOR_USB = const(0x01)
  PORT_FOR_4Pin = const(0x03)
```

2.3 Reception and Parsing Function

First, use __uart.read() to read the data packet from the serial port. If a non-empty data packet is received, assign the length of the received data packet to the variable data_len. Then, create two intermediate variables:

index: Index for reading the data packet, used to read the data packet sequentially.

data_index: Index for reading the data part, used to read the data part of the data packet sequentially.

```
  # 接收解析函数
def rec_data(self):
  readbuffer = self.__uart.read()
  if readbuffer is not None:
    data_len = len(readbuffer)
    index = 0
    data_index = 0
```

By using a while loop, start processing the received data packet byte by byte. First, two frame headers are processed. The current part of the data frame being processed is updated based on the state of “self.__pk_ctl.state”.

If the first data frame being processed is CONST_STARTBYTE1 (that is, 0xAA), indicating it is identified successfully, and then the state is updated to STARTBYTE2 (that is, switching to identify the second frame header). If failing to identify, the first frame header will be identified again.

```
while (data_len > 0):
  # 处理帧头标记1
  if PacketControllerState.STARTBYTE1 == self.__pk_ctl.state:
    if CONST_STARTBYTE1 == readbuffer[index]:
      self.__pk_ctl.state = PacketControllerState.STARTBYTE2
    else:
      self.__pk_ctl.state = PacketControllerState.STARTBYTE1
```

When recognizing the second frame header, if it is recognized successfully, the state is set as FUNCTION (which is the function code recognition state). If the recognition fails, it goes back to recognize the first frame header.

```
# 处理帧头标记2
elif PacketControllerState.STARTBYTE2 == self.__pk_ctl.state:
  if CONST_STARTBYTE2 == readbuffer[index]:
    self.__pk_ctl.state = PacketControllerState.FUNCTION
  else:
    self.__pk_ctl.state = PacketControllerState.STARTBYTE1
```

If the current state is FUNCTION (e.i. function code recognition state), it enters the function code recognition state. Here, the state will be first updated to LENGTH (e.i. data length recognition state), and then through nested if statements, each function code is recognized one by one. If a valid function code is identified, it is temporarily stored in the variable “**frame.function**”. Otherwise, the state is updated back to recognizing the first frame header.

```
# 处理帧功能号
elif PacketControllerState.FUNCTION == self.__pk_ctl.state:
  self.__pk_ctl.state = PacketControllerState.LENGTH
  if PACKET_FUNCTION.FUNC_READ_ANGLE != readbuffer[index]:
    if PACKET_FUNCTION.FUNC_READ_XYZ != readbuffer[index]:
      if PACKET_FUNCTION.FUNC_SET_ANGLE != readbuffer[index]:
        if PACKET_FUNCTION.FUNC_SET_PWMSERVO != readbuffer[index]:
          if PACKET_FUNCTION.FUNC_SET_XYZ != readbuffer[index]:
            if PACKET_FUNCTION.FUNC_SET_SUCTIONNOZZLE != readbuffer[index]:
              self.__pk_ctl.state = PacketControllerState.STARTBYTE1
  if self.__pk_ctl.state == PacketControllerState.LENGTH:
    self.__pk_ctl.frame.function = readbuffer[index]
```

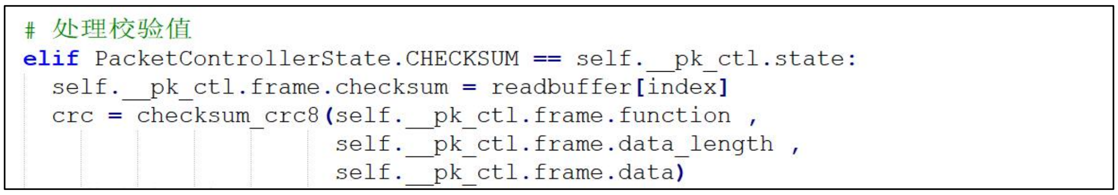
If the current state is LENGTH (i.e., processing the frame header data length state), the data length in data frame is temporarily stored in the variable “frame.data_length”, then check if the data length is 0. If it is 0, it indicates that the data packet is empty, so the state is directly set as CHEKSUM (processing checksum bit), and then it jumps directly to the part of handling the checksum. If the data length is not 0, then set the state to DATA (processing the frame data state), and proceed to handle the data part.

```
# 处理帧数据长度
elif PacketControllerState.LENGTH == self.__pk_ctl.state:
  self.__pk_ctl.frame.data_length = readbuffer[index]
  if 0 == self.__pk_ctl.frame.data_length:
    self.__pk_ctl.state = PacketControllerState.CHECKSUM
  else:
    self.__pk_ctl.state = PacketControllerState.DATA
  data_index = 0
```

If the current state is DATA, it successfully enters the part of processing the frame data. According to the index variable, start reading the data byte by byte from the first data and store it in the variable frame.data. When data_index (i.e., the actual received data length) is greater than or equal to the sent data length, it indicates that the reception is complete. Then, set the state to CHECKSUM (i.e., processing the checksum state).

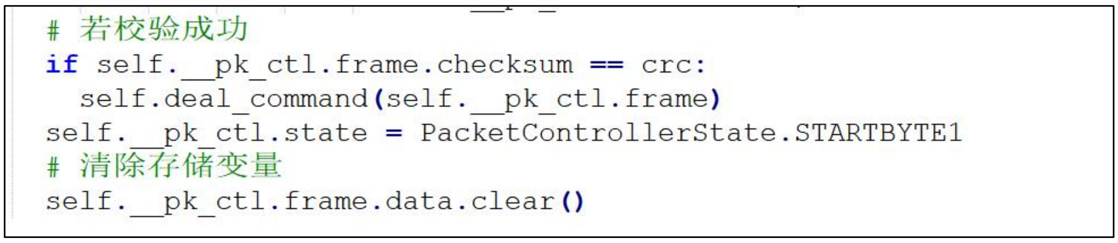
```
# 处理帧数据
elif PacketControllerState.DATA == self.__pk_ctl.state:
  self.__pk_ctl.frame.data.append(readbuffer[index])
  data_index += 1
  if data_index >= self.__pk_ctl.frame.data_length:
    self.__pk_ctl.state = PacketControllerState.CHECKSUM
```

When it comes to processing the checksum state, we first extract the checksum part from the data frame. Then, based on the previously read function code, data length, and the sum of all values in the data part, we calculate the checksum using the checksum_crc8 function in the format of a checksum and assign it to the variable crc.



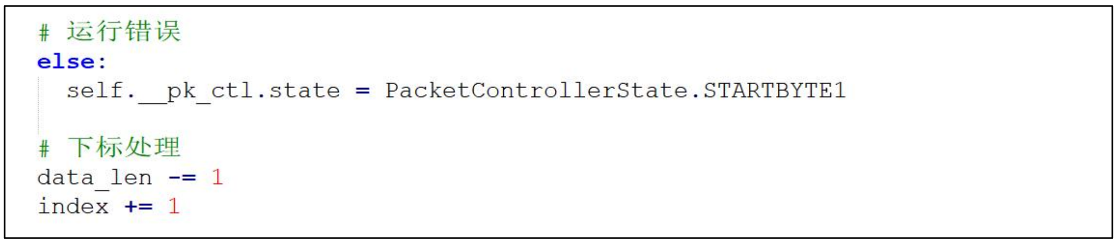
```
# 处理校验值
elif PacketControllerState.CHECKSUM == self.__pk_ctl.state:
    self.__pk_ctl.frame.checksum = readbuffer[index]
    crc = checksum_crc8(self.__pk_ctl.frame.function ,
                        self.__pk_ctl.frame.data_length ,
                        self.__pk_ctl.frame.data)
```

If the calculated checksum matches the checksum in the received data frame, then the checksum is successful. We return the frame object to the robotic arm control function for control. After completing these operations, reset the state to the recognition of the first frame header state, and clear all data in the frame object to wait for the next data reception or transmission.



```
# 若校验成功
if self.__pk_ctl.frame.checksum == crc:
  self.deal_command(self.__pk_ctl.frame)
self.__pk_ctl.state = PacketControllerState.STARTBYTE1
# 清除存储变量
self.__pk_ctl.frame.data.clear()
```

If an error occurs during execution, reset the state to recognize the first frame header. Finally, after processing the content of a data frame, decrement the data_len variable (remaining data frame length) by one, and increment the index variable (data frame index during processing) by one.



```
# 运行错误
else:
    self.__pk_ctl.state = PacketControllerState.STARTBYTE1

# 下标处理
data_len -= 1
index += 1
```

2.4 Robotic Arm Control Function

The deal_command() function has one parameter: ctl_com, which is an object representing a data packet in the same format as the communication protocol. It is used to control the robotic arm. At the beginning of the function, the length of the data in the packet is obtained to facilitate the subsequent data reading.

```
'''
机械臂控制函数
'''
def deal_command(self , ctl_com):
  len = ctl_com.data_length
```

Then, based on the function code in the data packet, the function determines the functionality that the current data packet implements.

```
    if ctl_com.function == PACKET_FUNCTION.FUNC_SET_ANGLE:

    elif ctl_com.function == PACKET_FUNCTION.FUNC_SET_XYZ:

    elif ctl_com.function == PACKET_FUNCTION.FUNC_SET_PWMSERVO:

    elif ctl_com.function == PACKET_FUNCTION.FUNC_SET_SUCTIONNOZZLE:

    elif ctl_com.function == PACKET_FUNCTION.FUNC_READ_ANGLE:

    elif ctl_com.function == PACKET_FUNCTION.FUNC_READ_XYZ:

    else:
      # print("ERROR")
      pass
```

If the recognized function code is invalid, the function passes and exits.

2.4.1 Set Servo Control Angle

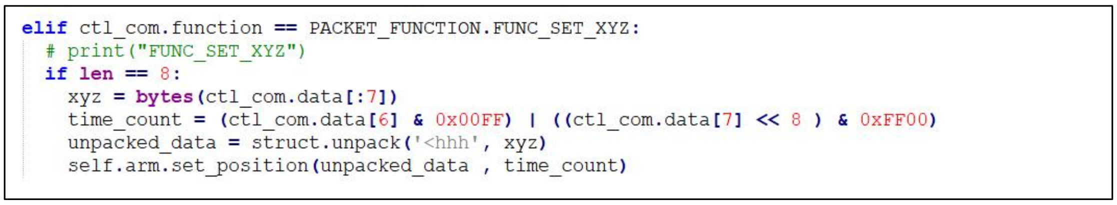
If the data length is 8, it is considered valid data. The function reads the angles of the three servos from the data packet using the angles array. In the ctl_com.data, indices 0, 2, 4 represent the low 8-bit values of the angles for each servo, while indices 1, 3, 5 represent the high 8-bit values of the angles for each servo. Indices 6 and 7 represent the low 8-bit and high 8-bit values of the servo rotation time, respectively.

Finally, the arm.set_servo_in_range() function is called to control the servos to rotate to the corresponding positions based on the read angles and time.

```
if ctl_com.function == PACKET_FUNCTION.FUNC_SET_ANGLE:
  # print("FUNC_SET_ANGLE")
  if len == 8:
    angles = []
    angles.append((ctl_com.data[0] & 0x00FF) | ((ctl_com.data[1] << 8 ) & 0xFF00))
    angles.append((ctl_com.data[2] & 0x00FF) | ((ctl_com.data[3] << 8 ) & 0xFF00))
    angles.append((ctl_com.data[4] & 0x00FF) | ((ctl_com.data[5] << 8 ) & 0xFF00))
    time_count = (ctl_com.data[6] & 0x00FF) | ((ctl_com.data[7] << 8 ) & 0xFF00)
    self.arm.set_servo_in_range(1 , angles[0] , time_count)
    time.sleep_ms(10)
    self.arm.set_servo_in_range(2 , angles[1] , time_count)
    time.sleep_ms(10)
    self.arm.set_servo_in_range(3 , angles[2] , time_count)
    time.sleep_ms(10)
```

2.4.2 Control Servo by Setting Coordinates

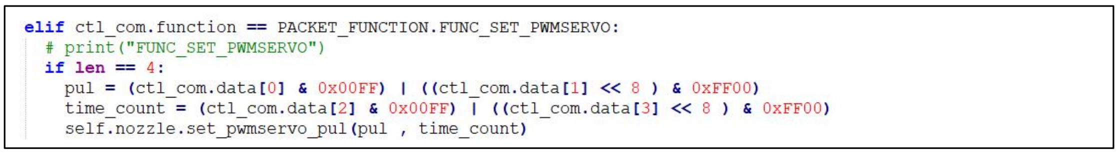
Convert the data at indices 0 to 6 of ctl_com.data to spatial coordinate values, then combine the data at indices 6 and 7 into a 16-bit runtime, and control the robotic arm to rotate in the spatial coordinate system using the set_position() function.



```
elif ctl_com.function == PACKET_FUNCTION.FUNC_SET_XYZ:
  # print("FUNC_SET_XYZ")
  if len == 8:
    xyz = bytes(ctl_com.data[:7])
    time_count = (ctl_com.data[6] & 0x00FF) | ((ctl_com.data[7] << 8 ) & 0xFF00)
    unpacked_data = struct.unpack('<hhh', xyz)
    self.arm.set_position(unpacked_data , time_count)
```

2.4.3 Control PWM Servo on Nozzle

The data length for controlling PWM servos in the data packet is only 4. In ctl_com.data, indices 0 and 1 contain the low 8-bit and high 8-bit values of the servo rotation angle, respectively, while indices 2 and 3 contain the low 8-bit and high 8-bit values of the timing time. Finally, the data is output to the PWM servo using the nozzle.set_pwmservo_pul() function.



```
elif ctl_com.function == PACKET_FUNCTION.FUNC_SET_PWMSERVO:
  # print("FUNC_SET_PWMSERVO")
  if len == 4:
    pul = (ctl_com.data[0] & 0x00FF) | ((ctl_com.data[1] << 8 ) & 0xFF00)
    time_count = (ctl_com.data[2] & 0x00FF) | ((ctl_com.data[3] << 8 ) & 0xFF00)
    self.nozzle.set_pwmservo_pul(pul , time_count)
```

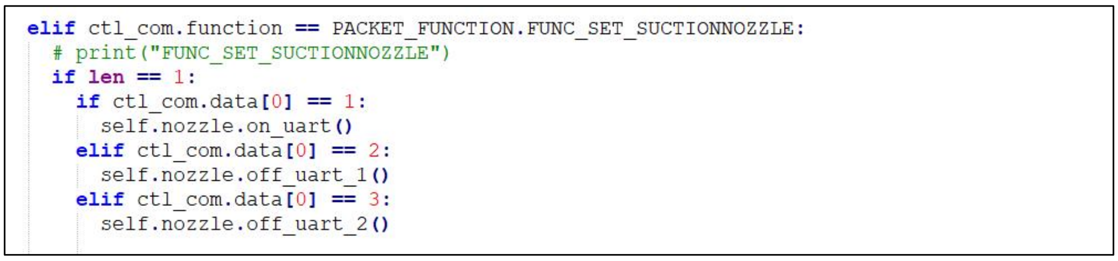
2.4.4 Control Air Pump

We can categorize the pump into three states:

State 1: Pump on, in this state, the pump starts suction. Corresponding control command: **nozzle.on_uart()**.

State 2: Pump off while opening the air valve to release air. In this state, the pump releases air. Corresponding control command: **nozzle.on_uart1()**.

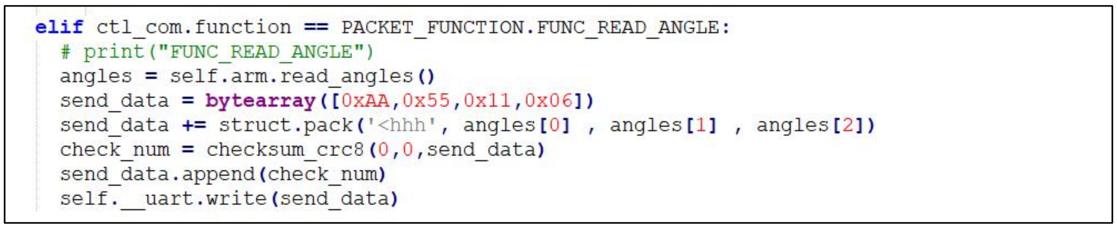
State 3: Close the air valve, and the pump is completely powered off. Corresponding control command: **nozzle.on_uart2()**.



```python
elif ctl_com.function == PACKET_FUNCTION.FUNC_SET_SUCTIONNOZZLE:
  # print("FUNC_SET_SUCTIONNOZZLE")
  if len == 1:
    if ctl_com.data[0] == 1:
      self.nozzle.on_uart()
    elif ctl_com.data[0] == 2:
      self.nozzle.off_uart_1()
    elif ctl_com.data[0] == 3:
      self.nozzle.off_uart_2()
```

2.4.5 Read Servo Angle

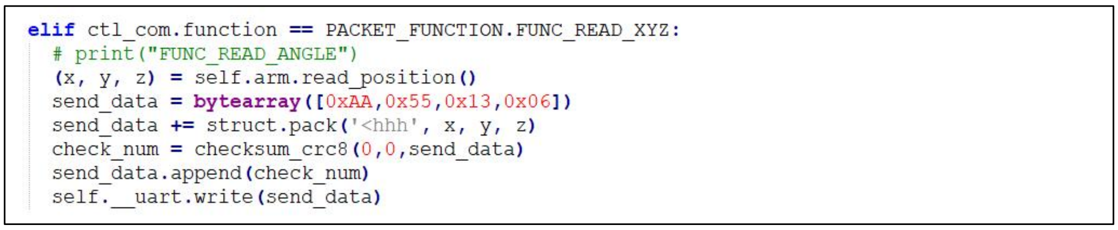
When the function code is for reading servo angles, the **arm.read_angle()** function is used to read the positions of the servos. Then, the data is encapsulated into a data packet format according to the communication protocol, and this data packet is returned to the host.



```python
elif ctl_com.function == PACKET_FUNCTION.FUNC_READ_ANGLE:
  # print("FUNC_READ_ANGLE")
  angles = self.arm.read_angles()
  send_data = bytearray([0xAA,0x55,0x11,0x06])
  send_data += struct.pack('<hhh', angles[0] , angles[1] , angles[2])
  check_num = checksum_crc8(0,0,send_data)
  send_data.append(check_num)
  self.__uart.write(send_data)
```

2.4.6 Read Robotic Arm Coordinate Position

When the recognized function code is for reading the position of the robotic arm in the coordinate system, we use the arm.read_position() function to read the position in the coordinate system. Then, we encapsulate it into a data packet format and send it back to the master.



```python
elif ctl_com.function == PACKET_FUNCTION.FUNC_READ_XYZ:
  # print("FUNC_READ_ANGLE")
  (x, y, z) = self.arm.read_position()
  send_data = bytearray([0xAA,0x55,0x13,0x06])
  send_data += struct.pack('<hhh', x, y, z)
  check_num = checksum_crc8(0,0,send_data)
  send_data.append(check_num)
  self.__uart.write(send_data)
```