Arduino Development Board Communication Routine Analysis

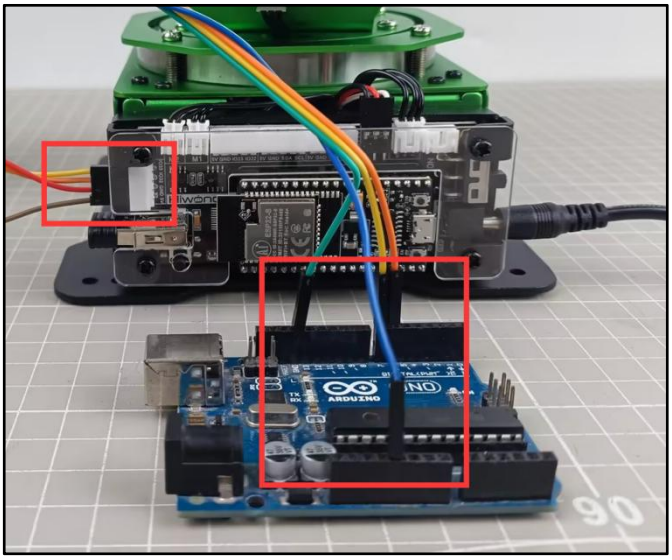
This lesson explains the Arduino-based low-level program for communication control as the host, analyzing the implementation of Arduino's encapsulation and transmission of data to MaxArm.

1. Communication Connection

1.1 Hardware Wiring

MaxArm has two types of serial communication interfaces: one is the 4-pin interface, which is suitable for devices with pin interfaces for UART communication, commonly used for Arduino development boards, STM32 development boards. The other is the microUSB interface, suitable for devices with USB host interfaces, such as Raspberry Pi, Jetson Nano, etc. Here, we need to connect to MaxArm's 4-pin interface for communication.

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



The specific pin connections are as follow:

MaxArm's IO32 pin serves as the RX signal pin, connected to pin 6 of the Arduino development board.

MaxArm's IO33 pin serves as the TX signal pin, connected to pin 7 of the Arduino development board.

MaxArm's GND pin is connected to the GND pin of the Arduino development board.

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 example routine analyzed in this document, demonstrated in the "**arduino_4Pin_MaxArm**" project in the same directory, primarily focuses on the "**MaxArm_ctl.h**" and "**MaxArm_ctl.c**" files. Their main purpose is to encapsulate and send control information to MaxArm.

2.1 Instantiate Serial Port

In MaxArm_ctl.h file, define rxPin and txPin intefaces as 7 and 6 respectively.

```
#define rxPin 7      //Arduino与MaxArm通讯串口
#define txPin 6
```

In the MaxArm_ctl.cpp file, instantiate a software serial port and set the communication baud rate to 9600.

```
static SoftwareSerial SerialM(rxPin, txPin);  //实例化软串口
```

```
void MaxArm_ctl::serial_begin(void)
{
  SerialM.begin(9600);
}
```

2.2 Calculate Checksum and Function

The function checksum_crc8() takes two parameters: *buf, which represents the data to be sent, and len, which indicates the length of the data. It calculates the checksum by summing up the values of the data bytes and adding the length of the data. Then, it takes the bitwise complement of the sum and extracts the lower 8 bits to obtain the checksum value.

```
/* CRC校验 */
static uint16_t checksum_crc8(const uint8_t *buf, uint16_t len)
{
    uint8_t check = 0;
    while (len--) {
        check = check + (*buf++);
    }
    check = ~check;
    return ((uint16_t) check) & 0x00FF;
}
```

2.3 Encapsulate Data Packet

2.3.1 Set Servo Angle

The variable msg[20] represents the encapsulated data packet. First, the values of CONST_STARTBYTE1 and CONST_STARTBYTE2 are added as the frame header to the first two positions of the msg variable. The value of CONST_STARTBYTE1 is 0xAA, and the value of CONST_STARTBYTE2 is 0x55. Then, the servo function code and data length are added to the data packet (the function code for this functionality is FUNC_SET_ANGLE).

```
uint16_t pul[3];
uint8_t msg[20] = {CONST_STARTBYTE1 , CONST_STARTBYTE2};
msg[2] = FUNC_SET_ANGLE;
msg[3] = 8;
```

Map the angles of each servo from the range of 0 to 240 to the range of 0 to 1000 using map() function, and add them to the data packet. Then, split the servo runtime into two bytes in low-high order and add them to the data packet. Finally, add the checksum to the data packet.

```
pul[0] = angles[0] > 240 ? 240 : angles[0];
pul[1] = angles[1] > 240 ? 240 : angles[1];
pul[2] = angles[2] > 240 ? 240 : angles[2];
pul[0] = map(pul[0] , 0 , 240 , 0 , 1000);
pul[1] = map(pul[1] , 0 , 240 , 0 , 1000);
pul[2] = map(pul[2] , 0 , 240 , 0 , 1000);
memcpy(&msg[4] , pul , 6);
msg[10] = time & 0x00ff;
msg[11] = (time>>8)&0x00FF;
msg[12] = checksum_crc8((uint8_t*)&msg[2] , 10);
```

The encapsulation of a data packet is now complete. By calling the write() function, the data packet can be sent via the serial port to the slave device.

```
SerialM.write((char*)&msg , 13);
```

2.3.2 Set Servo Angle

The set_xyz() function takes two parameters: *pos, which represents the data value to be sent, and time, which indicates the time for controlling the servo rotation.

```
//设置xyz
void MaxArm_ctl::set_xyz(int16_t* pos , uint16_t time)
{
```

In this function, first, the frame header, function code, and data length are stored in the data packet. Specifically, the function code is FUNC_SET_XYZ.

```
uint8_t msg[20] = {CONST_STARTBYTE1 , CONST_STARTBYTE2};
msg[2] = FUNC_SET_XYZ;
msg[3] = 8;
```

Copy the position of the coordinate system to the data packet, and simultaneously add the servo runtime (time) and the checksum_crc8 checksum to the data packet.

```
memcpy(&msg[4] , pos , 6);
msg[10] = time & 0x00ff;
msg[11] = (time>>8)&0x00FF;
msg[12] = checksum_crc8((uint8_t*)&msg[2] , 10);
```

Finally, use the write function to send the 13-byte data packet via the serial port to the slave device.

```
SerialM.write((char*)&msg , 13);
```

2.3.3 Set End-effector Servo Angle

The set_pwmservo() function has two parameters: angle for the servo rotation angle, and time for the servo rotation time.

```
//设置末端舵机角度 0~180度
void MaxArm_ctl::set_pwmservo(uint8_t angle , uint16_t time)
{
```

First, add the frame header, function code, and data length to the data packet.

```
uint8_t msg[10] = {CONST_STARTBYTE1 , CONST_STARTBYTE2};
msg[2] = FUNC_SET_PWMSERVO;
msg[3] = 4;
```

Limit the servo angle to a valid range, where if the servo angle is greater than 180 degrees, it is set to 180 degrees.

```
uint16_t pul = angle > 180 ? 180 : angle;
```

Map the servo angle from the range of 0 to 180 to the range of 500 to 2500. Then, store the servo angle and runtime in the data packet in the order of low 8 bits and high 8 bits. Finally, store the checksum in the data packet.

```
pul = map(pul , 0 , 180 , 500, 2500);
msg[4] = pul & 0x00ff;
msg[5] = (pul>>8)&0x00FF;
msg[6] = time & 0x00ff;
msg[7] = (time>>8)&0x00FF;
msg[8] = checksum_crc8((uint8_t*)&msg[2] , 6);
```

Finally, use the write function to send the 9-byte data packet via the serial port to the slave device.

```
SerialM.write((char*)&msg , 9);
```

2.3.4 Set Nozzle Function

The **set_SuctionNozzle()** function has one parameter, **func**, which represents the status of the suction nozzle. There are three states:

State 1: Turn on the air pump, it starts suction.

State 2: Turn off the air pump and open the air valve to release air, the pump is in the exhaust state.

State 3: Close the air valve, the pump is completely powered off.

The function encapsulates the air pump status into a data packet in the format of the communication protocol, including frame header, function code, data length, data, and checksum. The function code is FUNC_SET_SUCTIONNOZZLE.

```
//设置喷嘴功能
void MaxArm_ctl::set_SuctioNnozzle(int func)
{
  uint8_t msg[10] = {CONST_STARTBYTE1 , CONST_STARTBYTE2};
  msg[2] = FUNC_SET_SUCTIONNOZZLE;
  msg[3] = 1;
  // memcpy(&msg[4] , pul , 2);
  if(func < 0 && func > 3)
  {
    Serial.println("set_SuctioNnozzle ERROR");
    return;
  }
  msg[4] = func;
  msg[5] = checksum_crc8((uint8_t*)&msg[2] , 3);
  SerialM.write((char*)&msg , 6);
}
```

2.3.5 Read Angle

The read_angles() function takes one parameter, *angle, which stores the servo angles returned from the slave device. This function is mainly used to send a command to the slave device to read the angles, receive and parse the data from the slave device, and then store the parsed servo angle values in the angle variable.

```
//读取角度
bool MaxArm_ctl::read_angles(int* angles)
{
  uint8_t msg[10] = {CONST_STARTBYTE1 , CONST_STARTBYTE2};
  msg[2] = FUNC_READ_ANGLE;
  msg[3] = 0;
  msg[4] = checksum_crc8((uint8_t*)&msg[2] , 2);
  while(SerialM.available())
  {
    SerialM.read();
  }
  SerialM.write((char*)&msg , 5);
  uint16_t count = 3;
  delay(300);
  int16_t res[3];
  if(rec_handle((uint8_t*)res , 0x11))
  {
    angles[0] = map(res[0] , 0 , 1000 , 0 , 240);
    angles[1] = map(res[1] , 0 , 1000 , 0 , 240);
    angles[2] = map(res[2] , 0 , 1000 , 0 , 240);
```

Here, a data packet with a function code of FUNC_READ_ANGLE is first sent, with a data length of 0.

```
uint8_t msg[10] = {CONST_STARTBYTE1 , CONST_STARTBYTE2};
msg[2] = FUNC_READ_ANGLE;
msg[3] = 0;
msg[4] = checksum_crc8((uint8_t*)&msg[2] , 2);
while(SerialM.available())
{
  SerialM.read();
}
SerialM.write((char*)&msg , 5);
```

After sending the data packet, the function starts to receive the data packet from the slave device. It parses the data packet using the rec_handle() function and checks if the function code is 0x11 (0x11 is the function code for reading angles). If the read is successful and the function code is for reading angles, the data read is mapped from the range of 0 to 1000 to the range of 0 to 240 using the map() function, and then assigned to the angle array. Finally, if the read is successful, it returns true; otherwise, it returns false.

```
  if(rec_handle((uint8_t*)res , 0x11))
  {
    angles[0] = map(res[0] , 0 , 1000 , 0 , 240);
    angles[1] = map(res[1] , 0 , 1000 , 0 , 240);
    angles[2] = map(res[2] , 0 , 1000 , 0 , 240);
    return true;
  }
  return false;
}
```

2.3.6 Read xyz

The read_xyz() function takes one parameter, *pos, which stores the position of the spatial coordinate system returned from the slave device. This function is primarily used to send a command to the slave device to read the position of the spatial coordinate system, receive and parse the data returned from the slave device, and then store the parsed servo position values in the pos

variable.

```
//读取xyz
bool MaxArm_ctl::read_xyz(int* pos)
{
  uint8_t msg[10] = {CONST_STARTBYTE1 , CONST_STARTBYTE2};
  msg[2] = FUNC_READ_XYZ;
  msg[3] = 0;
  msg[4] = checksum_crc8((uint8_t*)&msg[2] , 2);
  while(SerialM.available())
  {
    SerialM.read();
  }
  SerialM.write((char*)&msg , 5);
```

Here, a data packet with a function code of FUNC_READ_XYZ is first sent, with a data length of 0.

```
uint8_t msg[10] = {CONST_STARTBYTE1 , CONST_STARTBYTE2};
msg[2] = FUNC_READ_XYZ;
msg[3] = 0;
msg[4] = checksum_crc8((uint8_t*)&msg[2] , 2);
while(SerialM.available())
{
  SerialM.read();
}
SerialM.write((char*)&msg , 5);
```

After sending the data packet, the function starts to receive the data packet returned from the slave device. It parses the data packet using the rec_handle() function and checks if the function code is 0x13 (0x13 is the function code for reading XYZ). If the read is successful and the function code is for reading XYZ, the data read is directly copied to the pos array. Finally, if the read is successful, it returns true; otherwise, it returns false.

```
  delay(300);
  int16_t res[3];
  if(rec_handle((uint8_t*)res , 0x13))
  {
    pos[0] = res[0];
    pos[1] = res[1];
    pos[2] = res[2];
    return true;
  }
  return false;
}
```

2.4 Parse Data Packet

The rec_handle() function primarily parses the data packets sent by the slave device. It takes two parameters: *res is used to store the received data portion, and func is used to identify the functionality of the received data packet.

```
bool MaxArm_ctl::rec_handle(uint8_t* res , uint8_t func)
{
  int len = SerialM.available();
  // 限制读取的数据长度
  len = len > 30 ? 30 : len;
  uint8_t step = 0;
  uint8_t data[8] , index = 0;
  while(len--)
```

First, the function retrieves the length of the data in the serial port and limits the read data length to 30 bytes. It creates a variable step as a parsing stage variable and also creates data[8] as a buffer for receiving data. index is used as a pointer to specify a byte of data in the data packet during parsing.

```
  int len = SerialM.available();
  // 限制读取的数据长度
  len = len > 30 ? 30 : len;
  uint8_t step = 0;
  uint8_t data[8] , index = 0;
```

Through a while loop, the function starts parsing based on the length of the received data packet. It reads one byte of data into the variable rd and begins parsing according to the step variable. Initially, when step = 0, it is the phase

for parsing the first frame header. If the data read is 0xAA, it is the first frame header, so step is incremented to move to the next phase, which is identifying the second frame header. If the identification fails, step is reset to 0 to restart the recognition process from the first phase.

```
while(len--)
{
  int rd = SerialM.read();
  switch(step)
  {
    case 0:
      if(rd == 0xAA)
      {
        step++;
      }
      break;
    case 1:
      if(rd == 0x55)
      {
        step++;
      }else{
        step = 0;
      }
      break;
```

Here's the parsing for the function code, data length, and data content:

case 2: This phase is for identifying the function code. Based on the func parameter passed, it checks if it matches the specified function code. If it does, the function code is stored in the data data buffer. If not, step is set to 0, returning to phase 1.

case 3: This phase is for identifying the data length. If the received data length is 6 bytes, the value of the data length is stored in the data data buffer. Otherwise, it returns to phase 1.

case 4: This phase is for parsing the data content. The function directly stores the read data into the data variable. When the number of stored data reaches or exceeds 8 bytes, it moves to the next phase: identifying the checksum.

```
        case 2:
          if(rd == func)
          {
            data[index++] = rd;
            step++;
          }else{
            step = 0;
          }
          break;
        case 3: //接收的数据长度必须为6个字节
          if(rd == 6)
          {
            data[index++] = rd;
            step++;
          }else{
            step = 0;
          }
          break;
        case 4:
          data[index++] = rd;
          if(index >= 8)
          {
            step++;
          }
          break;
```

In phase 5, the function calculates the checksum value for the previously obtained data length and data content using the checksum_crc8() checksum function. It then compares this checksum value with the checksum read from the received packet. If they are the same, the data parsing is successful; otherwise, it fails, and the function returns to phase 1 to restart the parsing process.

```
        case 5:
          if(checksum_crc8(data , 8) == rd)
          {
            memcpy(res , &data[2] , 6);
            return true;
          }else{
            return false;
          }
          break;
        default:
          step = 0;
          break;
      }
    }
    return false;
}
```