EMAC200 with touch screen communication

Key words: open motion controller; touch screen; configuration software; Modbus TCP communication protocol

A question

Touch screen configuration software to communicate with an external device, the usual solution is integrated in the touch screen and configuration software protocol of the external device that directly can use the protocol can develop touch-screen program. Universal motion controller protocol it is difficult to find in a touch screen, so we have no way to traditional solutions to establish communication.

Second, the solution concept

Beijing E-motion control equipment technology limited liability company EMAC200 motion controller is an open multi-axis motion controller. The provides PMDPeriphReceive, and PMDPeriphSend function in EMAC200 two functions of the user can receive the data and send the data to the physical channel physical channel. This function, we can choose a touch screen integration of the open protocol, use PMDPeriphReceive function receives touchscreen send data quoted text EMAC200 the CME processing and call PMDPeriphSend function of the needs of touch screen response quoted text returned to the touch screen.

EMAC200 multiple communication interfaces, all of these interfaces can send and receive data. Ethernet transmission rate, the agreement is relatively transparent, better support touch screen and configuration software, so our priority to choose EMAC200 Ethernet interface as a communication interface with touch screen.

Third, the choice of communication protocols

Modbus protocol is a common language used in electronic controller. Through this agreement, between the controller, the controller can communicate via the network (such as Ethernet) and other devices. It has become a common industry standard. With it, control devices of different manufacturers can be connected into industrial network, centralized monitoring.

Due to the openness, versatility and reliability of the Modbus protocol, we selected communication protocol Modbus protocol as EMAC200 touchscreen. Modbus Ethernet protocol called Modbus / TCP protocol.

Fourth, protocol-based

Modbus / TCP protocol uses a client / server model, shown in Figure 1:

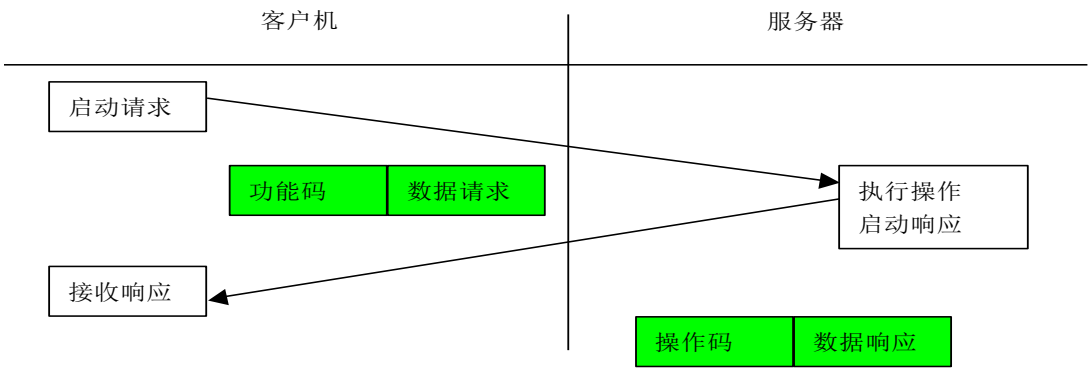

Figure 1

In Figure 1, we see the Modbus / TCP uses a request / response processing, data that the client sends a request, the server accepts the request, process the request and send the reply data to the client.

In the communication system of EMAC with the touch screen, the touch screen is equivalent to the function of the client, EMAC200 served the role of a server. Therefore, to achieve communication, we write a Modbus Server need in EMAC200 in the program.

The Server program to perform the following functions:

1, to request data packets sent by the client

2, from requests received data packets, the decomposition of the different functions of the client.

3, for different functions, call the EMAC200 function for processing.

4, send a reply message to the client.

Concrete realization of agreement

5.1 request packet classification

Request messages sent by the touch screen, we can be divided into four categories.

1, discrete variables (Boolean) read

2 discrete variables (Boolean) write

3, shaping variable read

4, shaping variable write

Modbus / TCP protocol provides many functions for reading and writing of parameters, one of the most commonly used function 3: read multiple registers and functions 16: Write Multiple Registers.

5.2 Add Modbus / TCP protocol

We use the Wei Lun touch the screen with EMAC200 communications. First, the to add Modbus protocol Wei Lun touch screen configuration software.

Our new add a PLC to PLC type select Modbus TCP / IP (Ethernet), Ethernet interface type select, set the IP address 192.168.10.40, the address for EMAC200 default IP address, port number is set to 502, the port is The default Modbus communication port number. PLC Setup Figure 5-1 shows:

5.3EMAC200 program development

We to in EMAC200 in running a Modbus Server. PMD provides a CME the demo program GenericUserPacket, this program can be read from the physical channel packet. Therefore, we are able to based on this program, to develop their own Modbus Server program.

5.4 Open EMAC200 Ethernet channels and ports

First of all, in the beginning of the program to define Ethernet channel

Define PMD_TCP_INTERFACE

Following open Ethernet connection and port

```
 unsigned int IPaddr = PMD_IP4_ADDR (0,0,0,0);

 int portnum = 502;
```

Because Ethernet is used for monitoring data, IP address 0.0.0.0

Modbus TCP default port 502, so we will port number is set to 502.

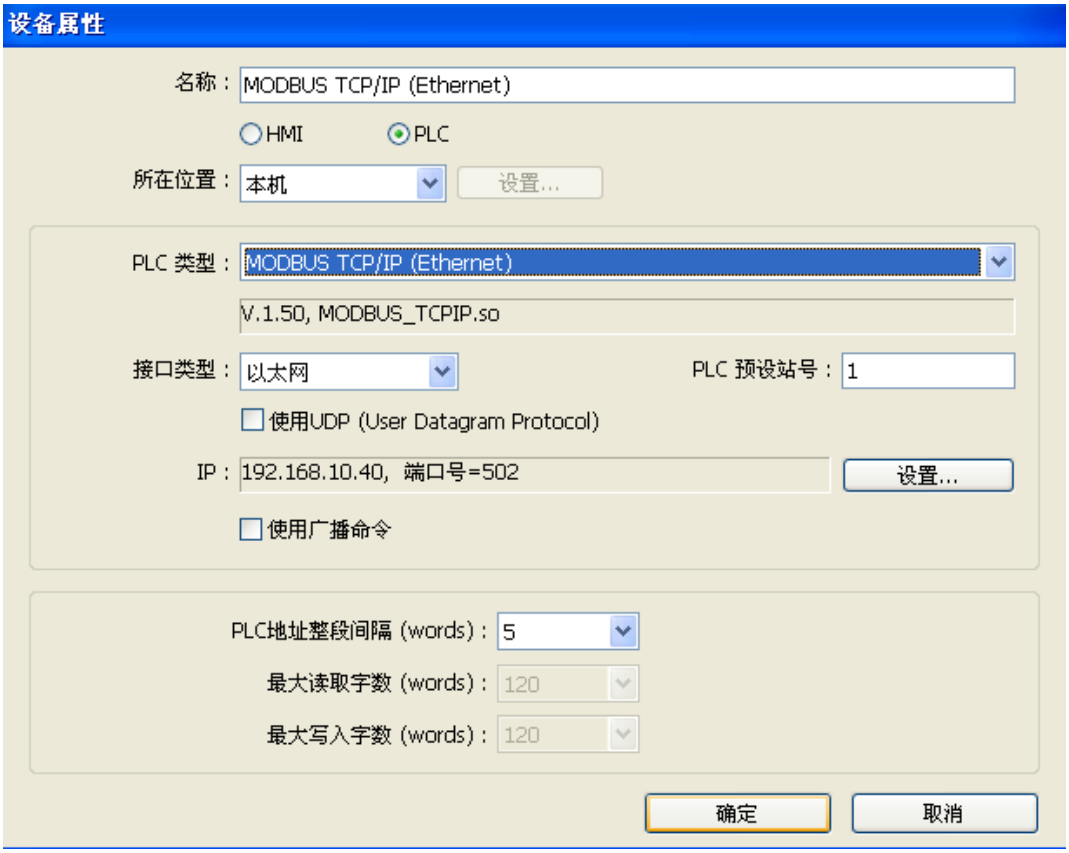

Figure 5-1

5.5-initialization motor and PID parameters

Motor PID parameters initialization process writes CME, and such control card electric motor and PID parameters can be done automatically after initialization.

result = PMDSetOutputMode (& hAxis1, 0); / / set control signal output mode, 0 for Parallel / / DAC the Offset Binary

result = PMDSetOutputMode (& hAxis2, 0);

result = PMDSetOperatingMode (& hAxis1, 0x33);

result = PMDSetOperatingMode (& hAxis2, 0x33);

result = PMDSetPositionLoop (& hAxis1, 0,12); / / proportional gain

result = PMDSetPositionLoop (& hAxis1, 1,40); / / integral gain

result = PMDSetPositionLoop (& hAxis1, 2,10000); / / integral limit

```
result = PMDSetPositionLoop (& hAxis1, 3,30); / / differential gain

result = PMDSetPositionLoop (& hAxis1, 4,1); / / derivative time

result = PMDSetPositionLoop (& hAxis1, 5,65535) ;/ / Kout

result = PMDSetPositionLoop (& hAxis1, 6,1000); / / speed feedforward

result = PMDSetPositionLoop (& hAxis1, 7,300); / / acceleration feedforward

result = PMDSetPositionLoop (& hAxis1, 8,0); / / do not use filtering

/ / Set axis 2 PID parameters

result = PMDSetPositionLoop (& hAxis2, 0,12); / / proportional gain

result = PMDSetPositionLoop (& hAxis2, 1,40); / / integral gain

result = PMDSetPositionLoop (& hAxis2, 2,10000); / / integral limit

result = PMDSetPositionLoop (& hAxis2, 3,30); / / differential gain

result = PMDSetPositionLoop (& hAxis2, 4,1); / / derivative time

result = PMDSetPositionLoop (& hAxis2, 5,65535) ;/ / Kout

result = PMDSetPositionLoop (& hAxis2, 6,1000); / / speed feedforward

result = PMDSetPositionLoop (& hAxis2, 7,300); / / acceleration feedforward

result = PMDSetPositionLoop (& hAxis2, 8,0); / / do not use filtering

/ / Position following error reset

PMDClearPositionError (& hAxis1);

PMDClearPositionError (& hAxis2);

PMDUpdate (& hAxis1);

PMDUpdate (& hAxis2);

/ / Set DAC enable output

result = PMDMBSetDACOutputEnable (& hAxis1, 1);

result = PMDMBSetDACOutputEnable (& hAxis2, 1);
```

/ / Set drive enable output

result = PMDMBSetAmplifierEnable (& hAxis1, 1,1);

result = PMDMBSetAmplifierEnable (& hAxis2, 2,2);

result = PMDResetEventStatus (& hAxis1, 0x0000);

result = PMDResetEventStatus (& hAxis2, 0x0000);

PMDSetAcceleration (& hAxis1, 429);

PMDSetAcceleration (& hAxis2, 429);

PMDSetDeceleration (& hAxis1, 429);

PMDSetDeceleration (& hAxis2, 429);

PMDSetVelocity (& hAxis1, 838861);

PMDSetVelocity (& hAxis2, 838861);

5.6 receiving the Modbus client data and monitor data packets

We realize the Modbus Server function on the loop while (1).

First, we use PMDPeriphReceive function receives Modbus TCP data packets. Code is as follows:

result = PMDPeriphReceive (& hPeriph, & data, & bytesReceived, BUFSIZE, 50);

Received packet is stored in the data array. Program start position of data definition, it is a PMDuint8 (char) array.

PMDuint8 data [BUFSIZE]; / / save the data received by the channel

Then we determine the value of the result, that received the correct packet only if the result no errors.

Now we need to monitor what data received. Fortunately EMAC200 control card provides a very useful function - CME console. It can monitor function outputs data PMDprintf. We can use the following statement to monitor the received Modbus packet.

for (i = 0; i <bytesReceived; i + +)

{

PMDprintf ("Data =% x \ n", data [i]);

}

5.7 send Modbus packet

Front we have learned, Modbus is a question-and-answer protocol. When our control card received Modbus data processing is complete, you must give the the Modbus client returns a set of data.

We use PMDPeriphSend functions to send Modbus data packet to the client. Here is the code:

result = PMDPeriphSend (& hPeriph, & sendData, 11,100);

The data packets to be transmitted is stored in the sendData array. Type consistent with data data.

5.8 Modbus processing function

When the the Modbus packet received, we need to determine the client issuing the request, and convert EMAC200 the implementation code to handle the request.

We mentioned earlier Modbus function 3 and function 16 are the two most important functions, the use of these two functions can be completed in the read and write operations of Boolean variables and integer variables. Modbus Server program in EMAC200 two Modbus functions.

Modbus TCP protocol, we learned that send 8 data packet Modbus function code. Therefore, we can very easily these two features distinguish the structure of the switch-case.

switch (data [7])

{

case 0x03:

/ / Function 3 processing

break;

case 0x10:

/ / Function 16 processing

break;

default:

break;

}

5.9 processing Modbus address assignment

We need to deal with data types include boolean and integer values. Modbus allows the use of the address space from 0 - 65535, so we can artificially separate address the Boolean variable and integer variables. Example: address 0 - 999 assigned to the Boolean variable address 1000 - 1999 assigned to the integer variable use.

Modbus protocol Modbus send packet data 9 and 10 indicate the address of the Modbus variables, use the following code to obtain the address of Modbus variables:

modbus_addr = (data [8] & 0X0F) * 256 + ((data [9] >> 4) & 0X0F) * 16 + (data [9] & 0x0F);

Here, we convert the Modbus address 10 hexadecimal form.

We distinguish between types of variables based on the value of modbus_addr.

The 5.10 read Boolean variables state

We read control cartoon DIO status example. The touch screen interface is shown in Figure 5-2:

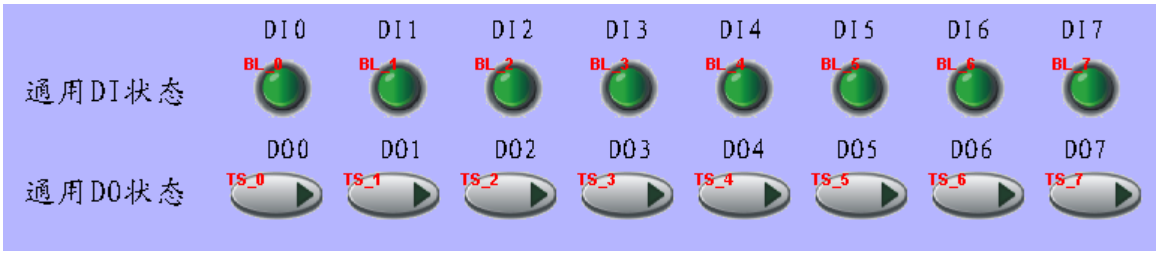

Figure 5-2

Modbus address is set to 16 LEDs 4x_bit 100 - 4x_bit 115. Figure 5-3 shows:

Figure 5-3

First, we use the the CME Console monitor touch screen to send EMAC200 packet. Figure 5-4 shows:

We see that the touch screen to continue to send data packets.

From the figure, we see, EMAC200 received 12 bytes of data, 12 the following meanings:

byte 0: transaction identifier - copied by the server

byte 1: transaction identifier - copied by the server

byte 2: protocol identifier = 0

byte 3: the agreement identifier = 0

Byte 4: the length field (upper half bytes) = 0 (all the length of the message is less than 256)

byte 5: the length field (lower byte) = subsequent byte quantity

byte 6: unit identifier (original "slave address")

byte 7: MODBUS function code

Byte 8-9: Reference values

Byte 10-11: instruction number (1-125)

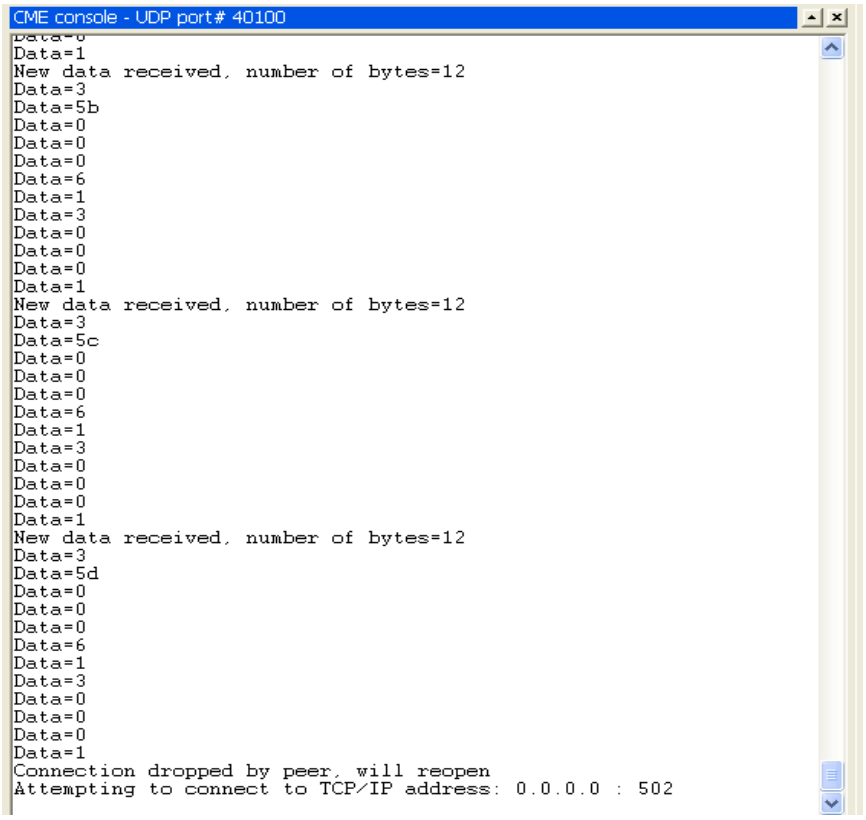

Figure 5-4

The response packet format:

byte0 - byte7: copy the received packet data

Byte 8: FC = 03

Byte 9: in response to the number of bytes (B = 2 x number of instructions)

Byte 10 - (B +1): Register values

We need to read the 16-bit data, i.e., two bytes, so in response the data packet byte9 for 2.

Here is the handler code:

PMDReadIO (& hAxis1, 0, & gio_status);

gio_di_status = (PMDuint8) ((gio_status & 0xff00) >> 8); / / Save universal di state

gio_do_status = (PMDuint8) (gio_status & 0x00ff); / / Save universal do state

```
for (j = 0; j <7; j + +)

{

sendData [j] = data [j];

}

sendData [7] = 0x3;

sendData [8] = 0x2;

sendData [9] = gio_do_status;

sendData [10] = gio_di_status;

result = PMDPeriphSend (& hPeriph, & sendData, 11,100)
```

5.11 write Boolean variable values

General DO, we can set its Boolean state. The same manner as the read Boolean value programming. Code is as follows:

Boolean variables we use function 16 to perform a write operation. The packet format:

byte 0: transaction identifier - copied by the server

byte 1: transaction identifier - copied by the server

byte 2: protocol identifier = 0

byte 3: the agreement identifier = 0

Byte 4: the length field (upper half bytes) = 0 (all the length of the message is less than 256)

byte 5: the length field (lower byte) = subsequent byte quantity

byte 6: unit identifier (original "slave address")

byte 7: MODBUS function code

Byte 8-9: Reference values

Byte 10-11: instruction number (1-100)

Byte 12: The number of bytes (B = 2 x word count)

Byte 13 - (B +5): register values

The response packet format:

byte0 - byte7: copy the received packet data

Byte 8: FC = 10 (hex)

Byte 9-10: Reference values

Byte 11-12: number of instructions

EMAC200 code as follows:

```
gio_do_write = ((PMDuint16) (data [13])) &0X00ff;

PMDWriteIO (& hAxis1, 0, gio_do_write);

for (j = 0; j <12; j + +)

{

sendData [j] = data [j];

}

sendData [5] = 0x6;

result = PMDPeriphSend (& hPeriph, & sendData, j, 100);
```

5.12 integer variable read

We write such a command position 5-5 shows the interface to read the two motors, the actual position and the position error.

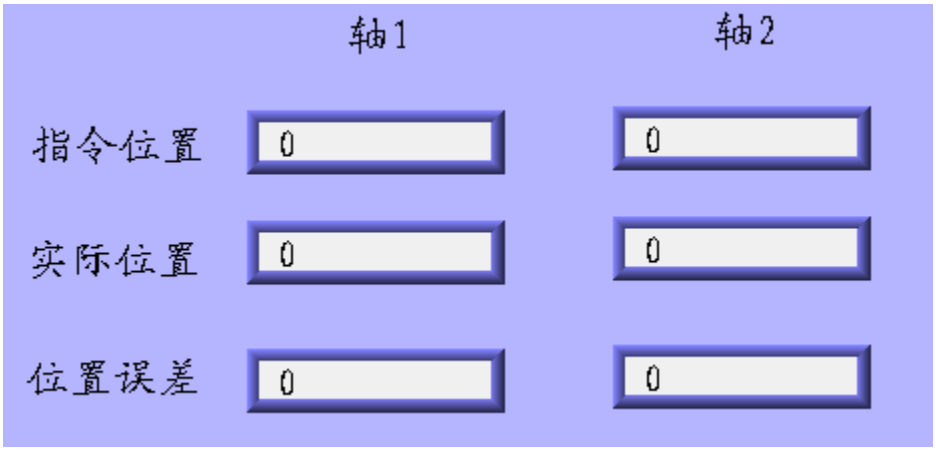

Figure 5-5

Use the same Modbus function read integer data.

Processing 32-bit integer data:

As a Modbus address corresponding data for the 16 bit (2 bytes), and here the position of the motor parameters are 32bit (4 bytes). Therefore, we need to use two address corresponds to a data.

Here, the shaft 1 instruction location address is 1000, the shaft 1, the actual location address is 1002, the shaft 1 position error address is 1004, and so forth. In this way, the two Modbus address combined into a 32bit data.

MODBUS "big-endian" to indicate the address and data objects, which means that when a digital representation of the number is greater than the transmission of a single byte, the maximum effective byte will be the first to be sent. For example:

Will be 16 - bits 0x1234 0x12 0x34

32 - bits 0x12345678L will be 0x12 0x34 0x56 0x78

When sending a 32bit data, we use the following code to handle the data:

```
sendData [9] = (PMDuint8) ((a1_comPos & 0Xff000000) >> 24);

sendData [10] = (PMDuint8) ((a1_comPos & 0X00ff0000) >> 16);

sendData [11] = (PMDuint8) ((a1_comPos & 0X0000ff00) >> 8);

sendData [12] = (PMDuint8) (a1_comPos & 0X000000ff);
```

First send 32bit data 8 high, re-transmission of 8-bit, and finally send a minimum of 8-bit data.

The read position EMAC200 in code as follows:

```
PMDGetActualPosition (& hAxis1, & a1_acrPos);

PMDGetCommandedPosition (& hAxis1, & a1_comPos);

PMDGetPositionError (& hAxis1, & a1_posErr);


PMDGetActualPosition (& hAxis2, & a2_acrPos);

PMDGetCommandedPosition (& hAxis2, & a2_comPos);

PMDGetPositionError (& hAxis2, & a2_posErr);

for (j = 0; j <7; j + +)

{

sendData [j] = data [j];

}

sendData [7] = 0x3;

sendData [8] = 0x24;

sendData [9] = (PMDuint8) ((a1_comPos & 0Xff000000) >> 24);

sendData [10] = (PMDuint8) ((a1_comPos & 0X00ff0000) >> 16);

sendData [11] = (PMDuint8) ((a1_comPos & 0X0000ff00) >> 8);

sendData [12] = (PMDuint8) (a1_comPos & 0X000000ff);

sendData [13] = (PMDuint8) ((a1_acrPos & 0Xff000000) >> 24);

sendData [14] = (PMDuint8) ((a1_acrPos & 0X00ff0000) >> 16);

sendData [15] = (PMDuint8) ((a1_acrPos & 0X0000ff00) >> 8);

sendData [16] = (PMDuint8) (a1_acrPos & 0X000000ff);

sendData [17] = (PMDuint8) ((a1_posErr & 0Xff000000) >> 24);
```

```
sendData [18] = (PMDuint8) ((a1_posErr & 0X00ff0000) >> 16);

sendData [19] = (PMDuint8) ((a1_posErr & 0X0000ff00) >> 8);

sendData [20] = (PMDuint8) (a1_posErr & 0X000000ff);


sendData [21] = (PMDuint8) ((a2_comPos & 0Xff000000) >> 24);

sendData [22] = (PMDuint8) ((a2_comPos & 0X00ff0000) >> 16);

sendData [23] = (PMDuint8) ((a2_comPos & 0X0000ff00) >> 8);

sendData [24] = (PMDuint8) (a2_comPos & 0X000000ff);

sendData [25] = (PMDuint8) ((a2_acrPos & 0Xff000000) >> 24);

sendData [26] = (PMDuint8) ((a2_acrPos & 0X00ff0000) >> 16);

sendData [27] = (PMDuint8) ((a2_acrPos & 0X0000ff00) >> 8);

sendData [28] = (PMDuint8) (a2_acrPos & 0X000000ff);

sendData [29] = (PMDuint8) ((a2_posErr & 0Xff000000) >> 24);

sendData [30] = (PMDuint8) ((a2_posErr & 0X00ff0000) >> 16);

sendData [31] = (PMDuint8) ((a2_posErr & 0X0000ff00) >> 8);

sendData [32] = (PMDuint8) (a2_posErr & 0X000000ff);


result = PMDPeriphSend (& hPeriph, & sendData, 33,100);
```

5.13 integer variable write

Figure 5-6 shows, we hope to set the shaft position, speed and other parameters.

Figure 5-6

The integer variable write using the same function 16.

Because written data is also of 32bit, so the method using the following conversion:

a1_setPos = ((((PMDuint32) data [13]) << 24) & 0xff000000) + ((((PMDuint32) data [14]) << 16) & 0x00ff0000)

+ ((((PMDuint32) data [15]) << 8) & 0x0000ff00) + (((PMDuint32) data [16]) & 0x000000ff);

PMDSetPosition (& hAxis1, a1_setPos);

Here is the specific code written to the integer data:

```
for (j = 0; j <12; j + +)

{

sendData [j] = data [j];

}

for (j = 0; j <= 3; j + +)

{

write_parm [j] = data [j +13];

}


switch (data [9])

{
```

```
case 0xcf: / / set the position of axis 1
a1_setPos = ((((PMDuint32) data [13]) << 24) & 0xff000000) + ((((PMDuint32) data [14]) << 16) & 0x00ff0000)
+ ((((PMDuint32) data [15]) << 8) & 0x0000ff00) + (((PMDuint32) data [16]) & 0x000000ff);
PMDSetPosition (& hAxis1, a1_setPos);
break;
case 0xd1: / / set the position of axis 2
a2_setPos = ((((PMDuint32) data [13]) << 24) & 0xff000000) + ((((PMDuint32) data [14]) << 16) & 0x00ff0000)
+ ((((PMDuint32) data [15]) << 8) & 0x0000ff00) + (((PMDuint32) data [16]) & 0x000000ff);
PMDSetPosition (& hAxis2, a2_setPos);
break;
case 0xd3: / / set axis 1
a1_setVel = ((((PMDuint32) data [13]) << 24) & 0xff000000) + ((((PMDuint32) data [14]) << 16) & 0x00ff0000)
+ ((((PMDuint32) data [15]) << 8) & 0x0000ff00) + (((PMDuint32) data [16]) & 0x000000ff);
PMDSetVelocity (& hAxis1, a1_setVel);
break;
case 0xd5: / / set axis 2
a2_setVel = ((((PMDuint32) data [13]) << 24) & 0xff000000) + ((((PMDuint32) data [14]) << 16) & 0x00ff0000)
+ ((((PMDuint32) data [15]) << 8) & 0x0000ff00) + (((PMDuint32) data [16]) & 0x000000ff);
PMDSetVelocity (& hAxis2, a2_setVel);
break;
case 0xd7: / / set axis acceleration
a1_setAcc = ((((PMDuint32) data [13]) << 24) & 0xff000000) + ((((PMDuint32) data [14]) << 16) & 0x00ff0000)
+ ((((PMDuint32) data [15]) << 8) & 0x0000ff00) + (((PMDuint32) data [16]) & 0x000000ff);
PMDSetAcceleration (& hAxis1, a1_setAcc);
```

```
break;

case 0xd9: / / set axis 2 acceleration

a2_setAcc = ((((PMDuint32) data [13]) << 24) & 0xff000000) + ((((PMDuint32) data [14]) << 16) & 0x00ff0000)

+ ((((PMDuint32) data [15]) << 8) & 0x0000ff00) + (((PMDuint32) data [16]) & 0x000000ff);

PMDSetAcceleration (& hAxis2, a2_setAcc);

break;

case 0xdb: / / set axis 1 deceleration

a1_setDec = ((((PMDuint32) data [13]) << 24) & 0xff000000) + ((((PMDuint32) data [14]) << 16) & 0x00ff0000)

+ ((((PMDuint32) data [15]) << 8) & 0x0000ff00) + (((PMDuint32) data [16]) & 0x000000ff);

PMDSetDeceleration (& hAxis1, a1_setDec);

break;

case 0xdd: / / set axis deceleration

a2_setDec = ((((PMDuint32) data [13]) << 24) & 0xff000000) + ((((PMDuint32) data [14]) << 16) & 0x00ff0000)

+ ((((PMDuint32) data [15]) << 8) & 0x0000ff00) + (((PMDuint32) data [16]) & 0x000000ff);

PMDSetDeceleration (& hAxis2, a2_setDec);
```

VI Summary

Modbus Server program written in EMAC200 communication based the touchscreen EMAC200 of the Modbus TCP protocol between. Modbus with an industrial general openness agreement, the touchscreen and configuration software are supported. Thus, as long as they support the Modbus protocol, we can achieve the communication with EMAC200.

Many PLC with Modbus interface, so that we can realize the communication between EMAC200 and PLC.

Modbus protocol in EMAC200 CEM in packet processing, and then for the rest of the data packet protocol, we should be able to use a similar method to deal with.

Hope the effect of forward play.