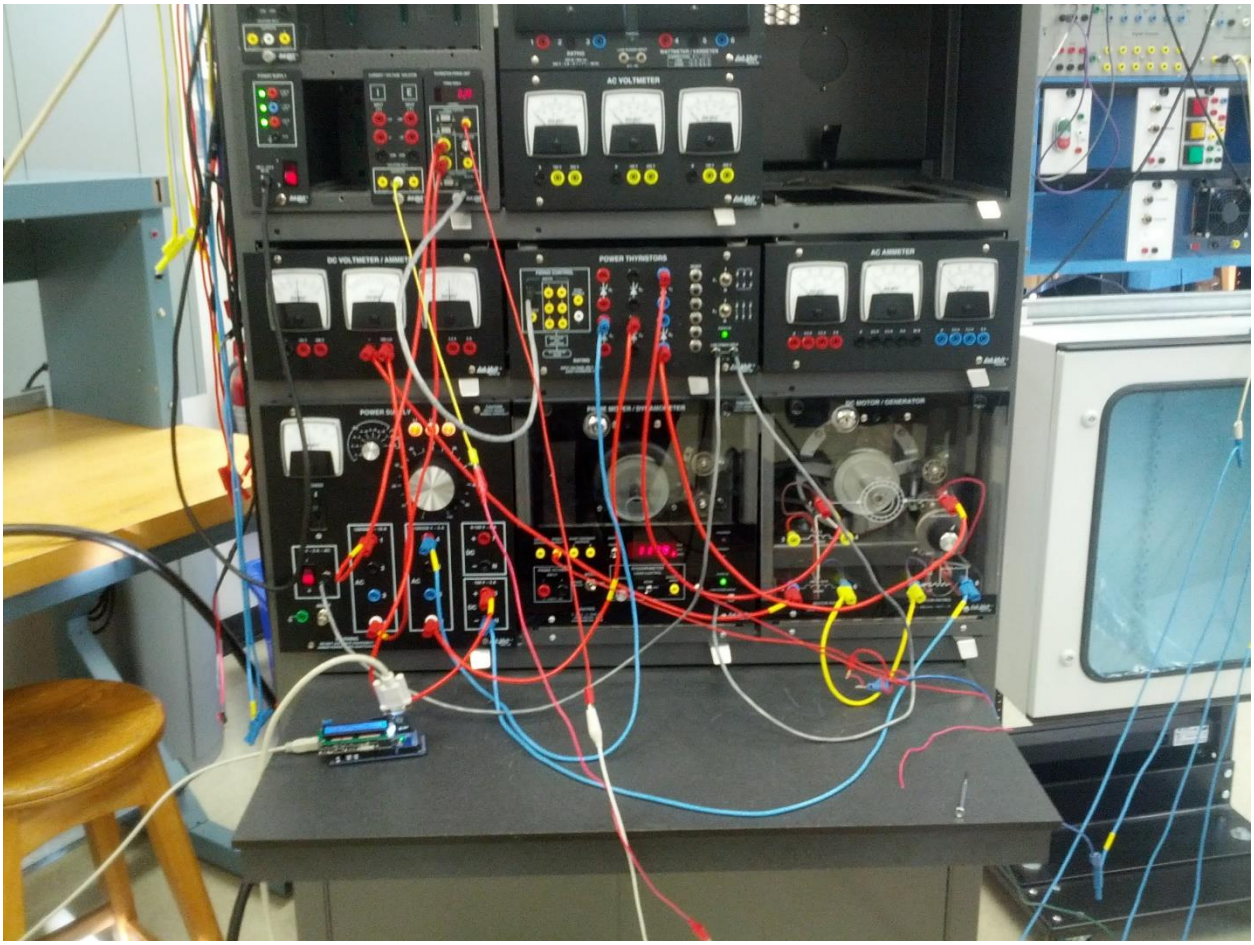
Arduino and PLC Communication



Designed and written by: Jason Eddrief

Table of Contents

System Introduction

In this system the Arduino functions as both the master controller and as the HMI (LCD shield) and the S7-200 is the slave. This system communicates on a basic command protocol when the Arduino sends a command the PLC will reply (regardless of what is sent but more on this later). Because this system is built to control a motor the plc sends back a speed in RPM that can be displayed on the LCD

The Arduino and the PLC have two ways of communication first is a simple I/O connection and second Serial. This manual will focus on the later. The S7-200 and the Arduino have a common way of communicating and that is through serial but requires signal conditioning the s7-200's single communication port operates on PPI or also known as RS-485 but they come with a cable (see fig 1.) to convert the signal to RS-232.

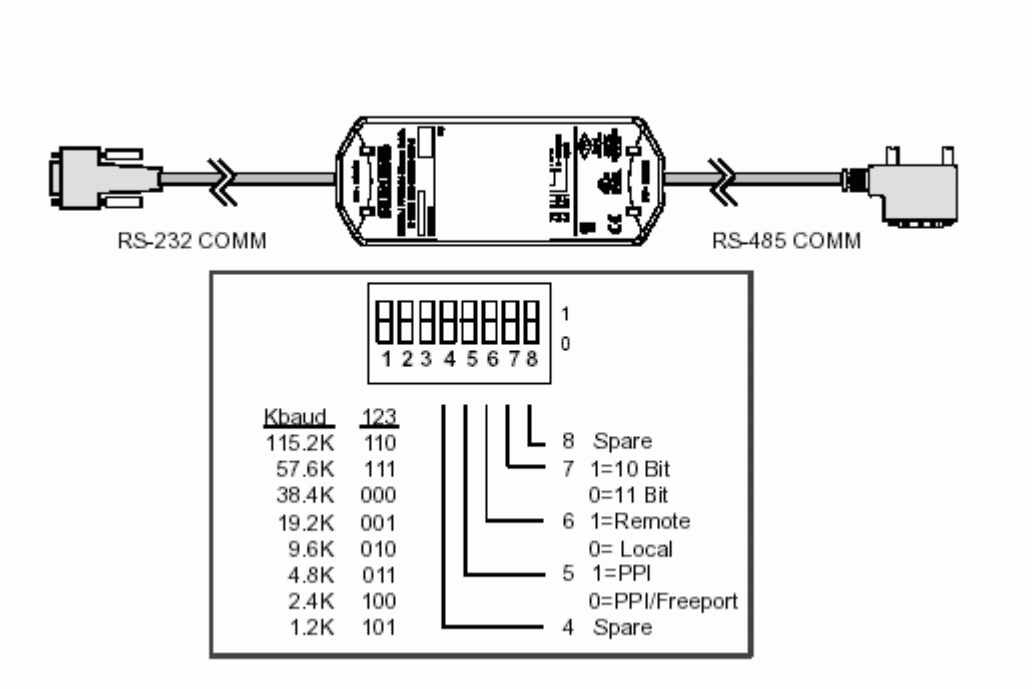


Figure 1

The Arduino communicates in serial but at TTL voltage levels so it also requires either a module (fig.3) (as will be used in this manual) or a full shield. Take note that a RS-485 can also be used just omitting the above cable and connecting the RS-485 RX+ to pin 3 and TX- to pin 8 and ground to pin 1 (see fig .2).

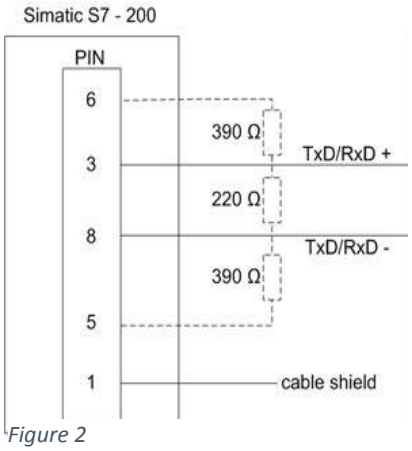


Figure 2

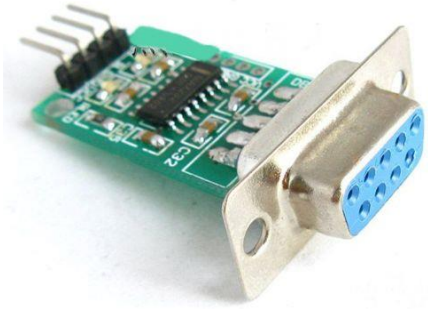


Figure 3

The preferred Arduino to use for this system is the Arduino Mega because it has 4 Serial communication headers. The Uno can be used but is not preferred because the RX0 and TX0 (digital pins 0 and 1 respectively) are hogged by the USB FTDI chip while connected to a computer.

For the s7-200 to communicate properly with the Arduino the PC/PPI cable (see fig. 1) to be set as follows:

1	2	3	4	5	6	7
0	1	0	0	0	1	0

Table 1

And to communicate with a pc, it needs to be in this configuration:

1	2	3	4	5	6	7
0	1	0	0	0	0	0

Table 2

Communication Protocol

The communication in this system is done with start and end character detection and commands.

In this system the “A” character (41 in hexadecimal) is used as a start character and the “B” character (42 in hexadecimal) these are used to send a string of data to the PLC only (however with adjustment to the code it can be used to send information back to the Arduino as well). The commands in this system are as follows:

u Increases the speed of the motor by 1000 bits.

d Decreases the speed of the motor by 1000 bits.

s Stops the motor.

i turns the test light on

o turns the test light off

The Arduino will send the PLC a command that looks like this: “AsB” the PLC will first see the “A” and start paying attention to the serial port, and begin recording what it receives in its buffer. Once it receives the “B” it will stop recording. The PLC can receive a maximum of 255 characters including the start and end character during any one communication once the end character is received, it will be loaded into a user specified table in the order it received. The first position in the table will be the number of characters received and the subsequent positions will be the message so a 4 byte string will look like this:

4	A	h	I	B
VB100	101	102	103	104

Table 3

Using this it can be assumed that VB102 will always be a command character more on this in the PLC section.

The Arduino works a little bit differently than the PLC in that It will always receive the serial string and it’s the users job to catch the string and manipulate it, So it can be useful the code that is in this manual uses the serialEvent() function to detect the start of a communication and will store each byte in an array until it receives its stop character which is “;” (like the PLC, this is user defined and can be changed) once that happens it will compile the array into a string.

Arduino

Sending Data:

Because of the protocol that was established sending data is fairly simple because the DFRobot LCD shield has 4 buttons on it the system can be controlled directly with the push buttons like so:

```
If ( button ==’s’)
{

Serial1.print(“AsB”);
//stop
}
```

Because this system is designed with the Mega it uses Serial1 which uses pins 18 and 19 (Tx and Rx respectively) and Serial is occupied by the USB the code is also written in such a way that it will retransmit an instruction sent from the PC to the PLC using the same method in which it receives data from the plc in a terminal you would type “s;” to send a stop command to the PLC.

This part of the code takes the compiled string received from the PC and adds the start and stop characters then transmits the data and clears the string for the next command:

```
if (pcStringComplete) {
        pcString ='A'+ pcString;
        pcString +='B';
        Serial1.println(pcString);
         pcString = "";
        pcStringComplete = false;
 }
```

Receiving Data:

When the Arduino receives data whether it be on Serial or Serial1 it will trigger a serial event. This happens between code scans and will receive data in a buffer during the scan. The code for when it receives a serial event works like this: When a serial event happens it will hold the program in a loop with Serial1.available() (which means the Rx pin is active). It will then add each character in an incrementing array. Once the ";" stop character is received it will take each element of the array and add it to the end of the plcString, it also stops 3 positions early because first there is 1 character extra that the plc sends. Second because of the stop character needs to be removed, and third there is an extra spot because the plcCnt++ happens before the for loop. It then trims any extra blank characters that may have been added before or after the string. (Note: the pcString is generated in the same way)

```
void serialEvent1() {
  while (Serial1.available()) {
    char plcInChar = (char)Serial1.read();
    //put the input bytes into an array
    plcArray[plcCnt] = plcInChar;
     plcCnt++;
   //check for end char
     if (plcInChar == ';') {
       //make a string with the array and remove the end char
        for (int i=0; i <= plcCnt-3; i++)
        {
          plcString += plcArray[i];
        }
        //trim off extra whitespace from the string
        plcString.trim();
        plcCnt=0;
        plcStringComplete = true;
    }
}
```

Communicating:

Above it was demonstrated how the Arduino sends and receives data. The system is designed in such a way that it increments and decrements speed by 1000 bits (on the PLC side) because of how the PLC is setup (this will be covered in the next section), the PLC will return the RPM information it has whenever it gets a full command string i.e. “AsB”. Once it gets the command it will drop everything and send “ 1234 ;” to the Arduino which converts it to an ASCII string ready to be displayed on the LCD (fig. 4).

```
if (plcStringComplete)
{
  lcd.setCursor(0,1);
   lcd.print("     ");
   lcd.print(plcString);
 lcd.print ("      ");
 plcString = "";
plcStringComplete = false;
 }
```

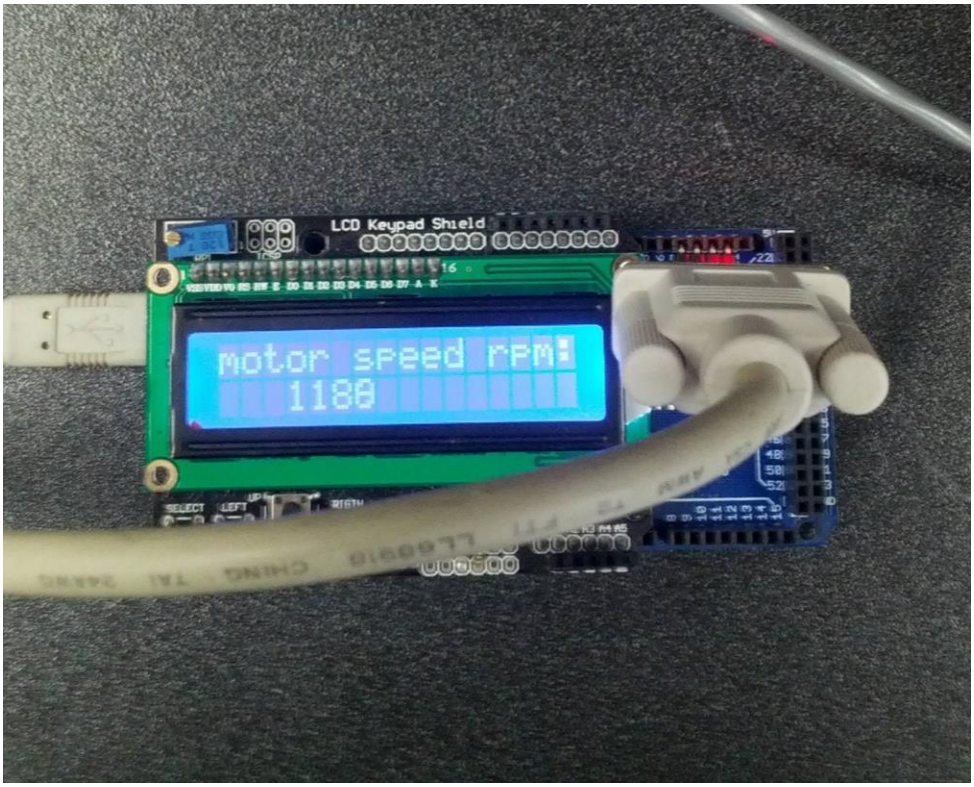


Figure 4

S7-200

The S7-200 communicates with the Arduino on Freeport mode at 9600 baud in this system. An important note: The PLC must be in stop mode to communicate with the PC when Freeport is enabled. The Arduino is connected to the PLC via the PPI/PC cable on Com0. The analog output (AQW0) is connected to the TFU on the washing machine and the analog input (AIW0) is connected to the speed output from the dynamo. The PLC will always transmit speed information when it receives any kind of data from the Arduino

Communications setup

The Freeport memory settings location is SMB30

The system uses Freeport, 9600 baud, 8bit per character and no parity

The s7-200 can accept either 2#01010000 or 16#50 for this configuration

0	1	0	1	0	0	0	0
Protocol bits		Baud rate bits			Data bit	Parity	

Table 4

Port 0	Port 1	Description
Format of SMB30	Format of SMB130	Freeport mode control byte MSB 7 … LSB 0 p p d b b b m m
SM30.0 and SM30.1	SM130.0 and SM130.1	mm: Protocol selection 00 =Point-to-Point Interface protocol (PPI/slave mode) 01 =Freeport protocol 10 =PPI/master mode 11 =Reserved (defaults to PPI/slave mode) Note: When you select code mm = 10 (PPI master), the S7-200 will become a master on the network and allow the NETR and NETW instructions to be executed. Bits 2 through 7 are ignored in PPI modes.
SM30.2 to SM30.4	SM130.2 to SM130.4	bbb: Freeport Baud rate 000 =38,400 baud 001 =19,200 baud 010 =9,600 baud 011 =4,800 baud 100 =2,400 baud 101 =1,200 baud 110 =115,200 baud 111 =57,600 baud
SM30.5	SM130.5	d: Data bits per character 0 =8 bits per character 1 =7 bits per character
SM30.6 and SM30.7	SM130.6 and SM130.7	pp: Parity select 00 =no parity 01 =even parity 10 =no parity 11 =odd parity

Figure 5

Receive Setup

For this system the receive settings are as follows:

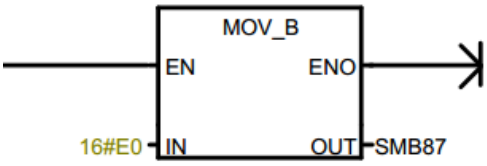


Figure 6

SMB87

1	1	1	0	0	0	0	0
Receive enabled	Detect start char	Detect end char	Detect line idle	Char timeout	Message timeout	Enable break condition	extra

Table 5

Start character:
SMB88, 16#41 (A)

Stop character:
SMB89, 16#42 (B)

Max characters that are expected to be received:
SMB 94, 100

SMB87	SMB187	Receive Message control byte MSB 7 … LSB 0 en | sc | ec | il | c/m | tmr | bk | 0 en: 0 =Receive Message function is disabled. 1 =Receive Message function is enabled. The enable/disable receive message bit is checked each time the RCV instruction is executed. sc: 0 =Ignore SMB88 or SMB188. 1 =Use the value of SMB88 or SMB188 to detect start of message. ec: 0 =Ignore SMB89 or SMB189. 1 =Use the value of SMB89 or SMB189 to detect end of message. il: 0 =Ignore SMW90 or SMW190. 1 =Use the value of SMW90 or SMW190 to detect an idle line condition. c/m: 0 =Timer is an inter-character timer. 1 =Timer is a message timer. tmr: 0 =Ignore SMW92 or SMW192. 1 =Terminate receive if the time period in SMW92 or SMW192 is exceeded. bk: 0 =Ignore break conditions. 1 =Use break condition as start of message detection.
SMB88	SMB188	Start of message character
SMB89	SMB189	End of message character
SMW90	SMW190	Idle line time period given in milliseconds. The first character received after idle line time has expired is the start of a new message.
SMW92	SMW192	Inter-character/message timer time-out value (in milliseconds). If the time period is exceeded, the receive message is terminated.
SMB94	SMB194	Maximum number of characters to be received (1 to 255 bytes). Note: This range must be set to the expected maximum buffer size, even if the character count message termination is not used.

Figure 7

Receiving (RCV) Data

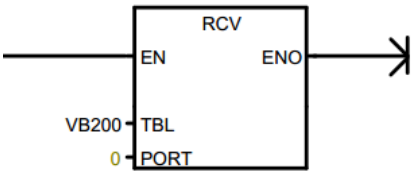


Figure 8

The PLC can receive a maximum of 255 characters including the start and end character during any one communication. Once the end character is received it will be loaded into a user specified table, in the order it was received. The first position in the table will be the number of characters received, and the subsequent positions will be the value so a 3 byte instruction will look like this:

3	A	S	B
VB200	201	202	203

Table 6

Transmitting (XMT) data

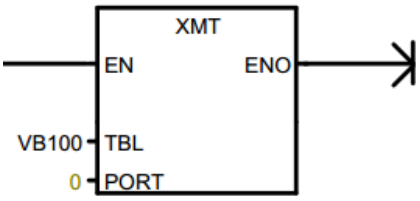


Figure 9

The XMT instruction is fairly simple. It only requires that the Freeport communication be setup properly, a table with the number of characters to send (first position of the table), and the message itself in the table. Once an enable signal is sent to the command, it will transmit the message in the table during the communications step of the cycle. However this is only true when the system is not receiving. So for this system to be able to transmit, the PLC detects that it has received data, then it will wait for the message to be complete, wait 10ms (this is to give time for the PPI/PC cables' buffer to empty), then it will transmit the data in VB100, and re-enable the RCV instruction.

i.e.:

4	1	2	3	4
Vb100	101	102	103	104

Table 7

The PLC transmits the contents of VB100 which was collected speed information from AIW0 or AIW2 (whichever is working on the PLC)

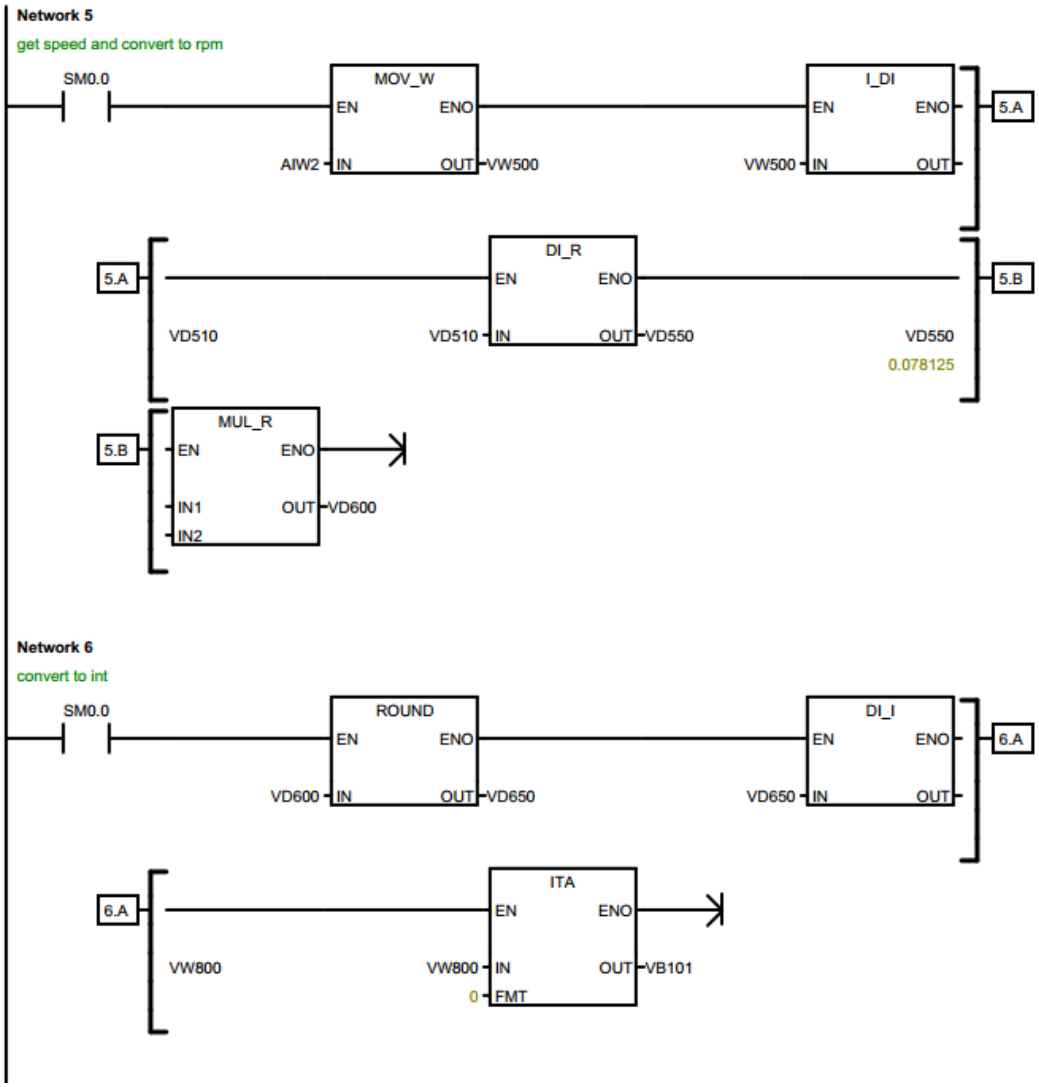


Figure 10

The table is created by predefining the number of characters to transmit (10), then filling the table using the ITA (integer to ASCII) command (see fig. 10) in Format '0'. Meaning it outputs whole numbers with no decimal or comma, it will always output 8 Characters. The Arduino stop character ";" is added to the last position of the table (this is done in the data block of the program).

10					1	2	3	4		;
VB100	101	102	103	104	105	106	107	108	109	110

Data Interpretation

When the PLC receives data on the serial port it stores it in a table specified by the programmer (as stated before). To interpret this command the system needs to know what to look for and where. Here the system uses ==B on VB202 because it is know that the command character will always be at this location. In fig. 11 the system is checks VB202 if it is an “I” then it enables Q0.1 (the test light). If VB202 is an “o” then it disables the test light.

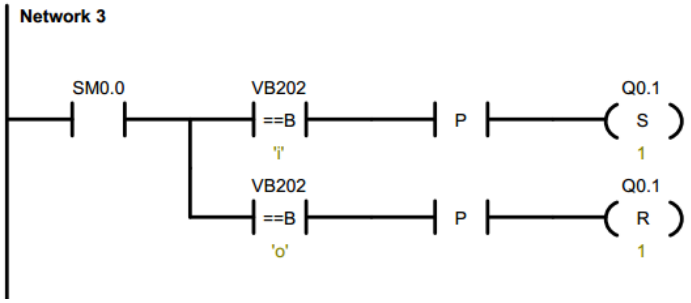


Figure 11

In fig. 12 the program uses an up/ down counter to set the speed of the motor. So if the PLC gets an “u” it will increase the count, if it gets a “d” it will decrease the count, and if it gets the “s” command it will set the counter to 0 which will stop the motor, VB202 is cleared later in the code (fig.13).

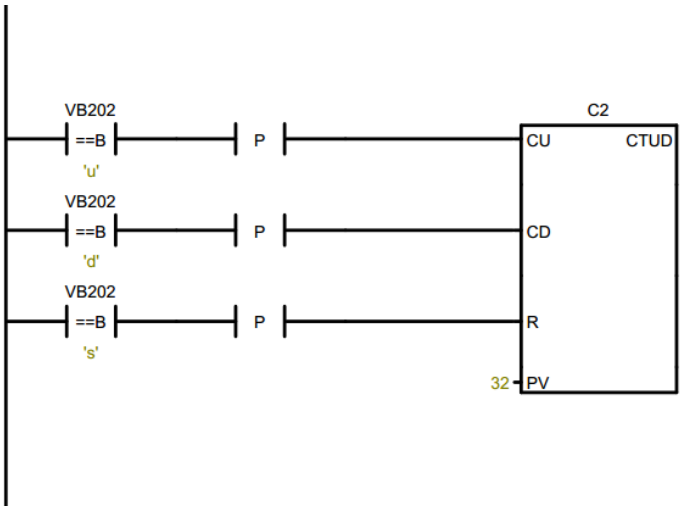


Figure 12

To create an appropriate output range (+32000 to -32000) the program uses the number from the counter. It subtracts the count by 32, inverts that number, then multiplies it by 1000 then sends it to AQW0 (not shown here). When it finishes that it clears VB202 with an unused command character (in this case ‘n’) so the system doesn’t repeatedly increment the counter on each scan.

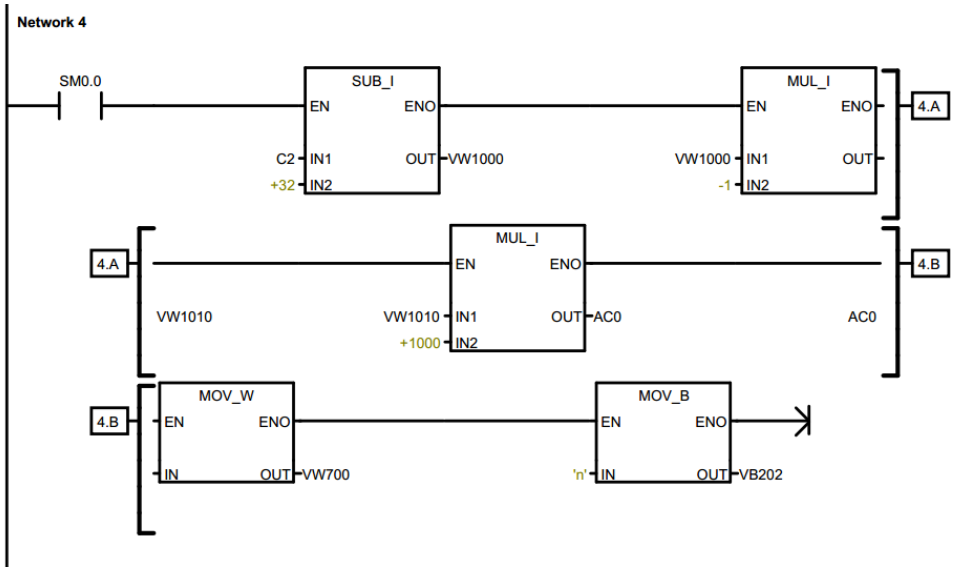


Figure 13

The "Washing Machine"

The washing machine is setup using the DC motor module (the SCIM can also be used), the prime mover/dynamo module, the thyristor module, and the thyristor firing unit and optionally a DC ammeter to set the current in the field windings.

The motor is setup as in a separately excited configuration with the field winding connected directly to 120V DC output and the armature is connected to the thyristors (see fig. 14) which is connected to the variable 120VAC.

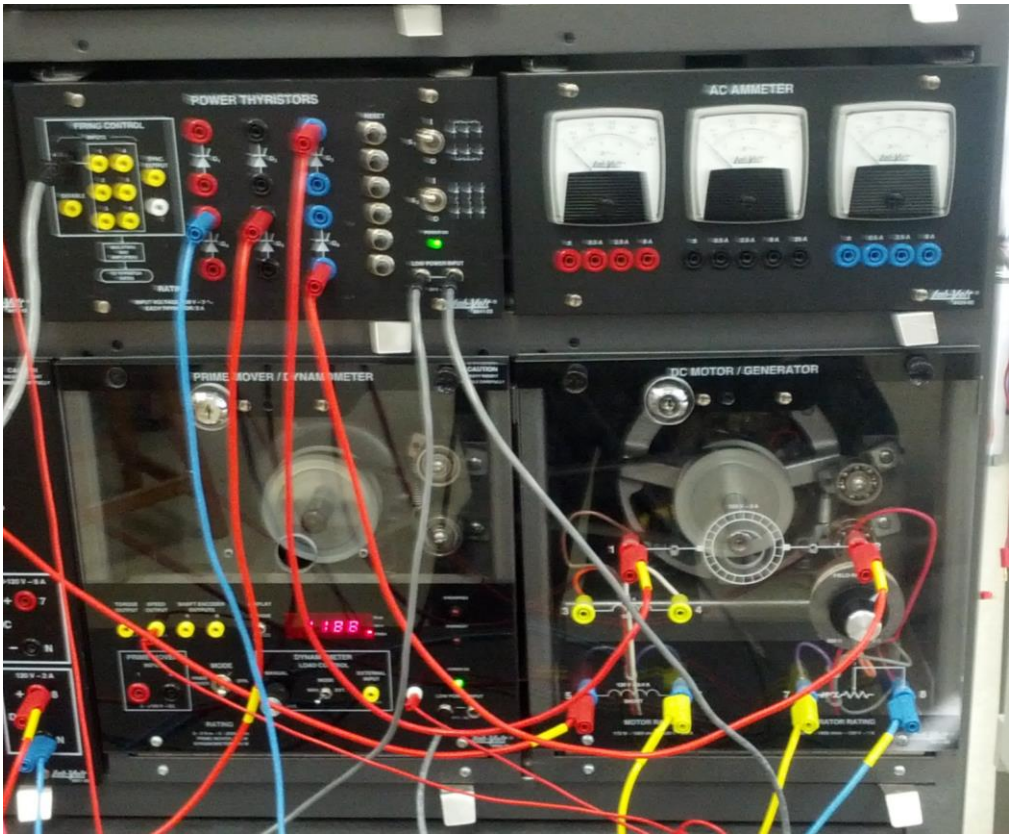


Figure 14