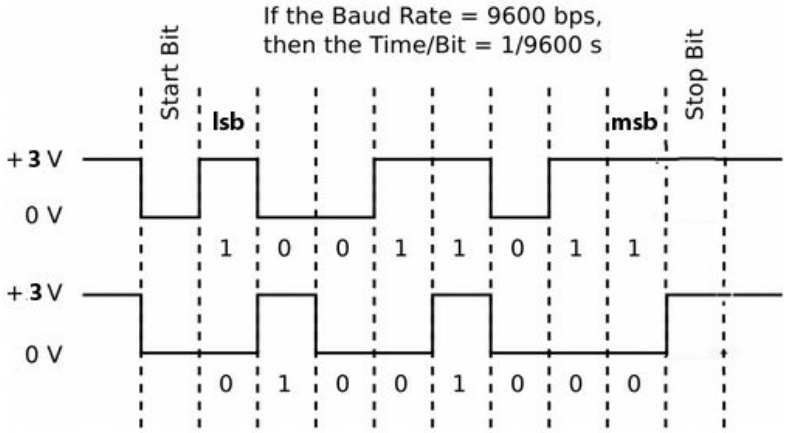
Controling *Tescom Solar-frequency inverter* over MODBUS-RTU protocol

Erdem Canaz

Communication needs two crucial components. One of them is the physical layer (i.e. RS485) which is responsible for transmitting the bits. And other one is the communication protocol (i.e. MODBUS) which is responsible for making sense out of those bytes.

RS485: Industrial enviroments are likely to be noisy since there are cables which carries high currents or motors switching periodically. In such environments, RS485 becomes favorable since it sends bits in differential form. The reciever do not compare the received signal to a threshold value but measures the differences of the two input signals. Since cables are close to each other, it is likely to see similar noises in both signals. Which is eliminated when two signals are subtracted from each other. Even though there are more advanced physical layers, RS485 is widely used since it is likely for older devices to run on RS485. In addition, with about 2km of range, 256 devices connected in series and moderete transmission speed, The protocol is sufficient for many case.

Each package sent over RS485 has UART format. The transmission line is kept HIGH when there is no data flow. And when there is data to be sent, line is pulled down for 1bit. Upcoming 5 to 11 (commonly 8) bit is the data. Then there is parity bit comes. If there is no parity check, the parity bit is still there but it is kept HIGH no matter what. Last 1 to 2 bits is STOP bits which is HIGH. 2 stop bit is used when there is older devices which may struggle to process data fast enough. By waiting for 2 bits, it is guaranteed that the slave device proceses data and there is no collision. Note that there is no limit for the number of stops bit. Since transmision line is pulled high either when there is no data is sent or the stop bit is sent.



There is abbreviations such as 8N1. It means Data consists of 8bits, there is none parity check and number of stop bits is 1. One can simply understand what 7N1 or such abbreviations means.

MODBUS: MODBUS is a communication protocol developed by Modicon around 1980s for their own PLCs which is then widely used. The important terms are MODBUS-RTU and MODBUS-ASCII. Consider a case where you want to send integer value 238 as data. in MODBUS-RTU it is sent in 8 bits (283_DEC= 11101110_BIN). Yet, in MODBUS_ASCII, same data requires atleast 24 bits to be send because each digit

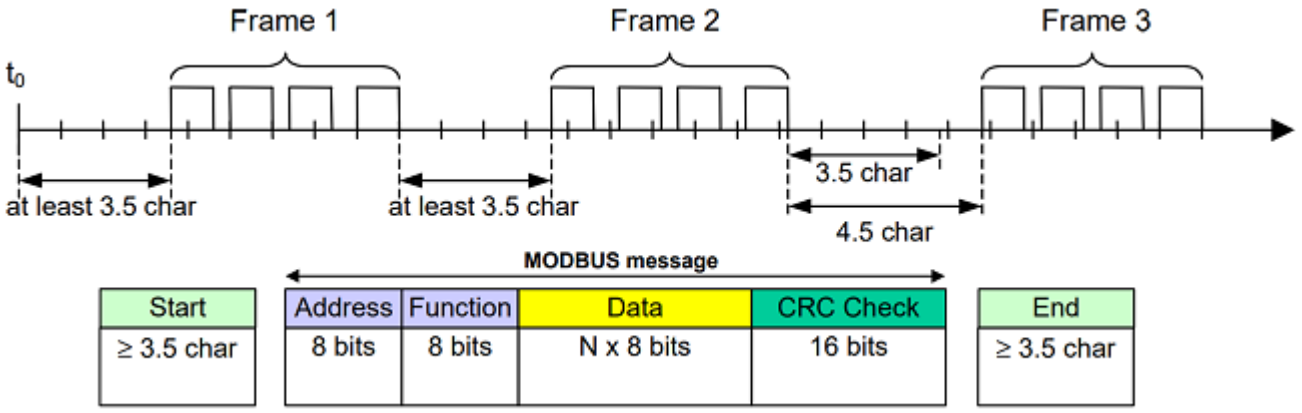
is sent as char. First 2, then 3 then 8 is sent each is 8 bits. MODBUS-RTU offers high speed whereas it is easy to debug MODBUS-ASCII.

`Slave Address (8bits)`: A number from 0 to 255. Should be unique for each slave in the same line.

`Function Code (8Bits):` Master indicates the operation that should be done by the slave. One may check web for detailed list of function codes but function code **3** is used for reading slave registers and function code **6** is used to write slave registers.

`Data(N*8Bits):` As its name suggest, information that frame contains. Number of registers to read, write. Which registers to write etc..

`CRC (Cyclic redundancy check)`: Last 8x2 bits are used to check whether the received data is corrupted or not. Different types of CRCs are widely used ranging from Ethernet to communication. I have no depth understanding of CRC and only applied the algorithm to generate one. But one may consider its working principle as follows. Say you want to send data 125. you shift it one digit and it becomes 1250. Chose lucky number as 3. When you divide 1250 to 3, the remainder is 2. If you subtract 2 from 1250, the result 1248 is divisible by 3. If slave recives it as 1249, recived message is not divisible by 3 and it is corrupted. CRC is more complicated and mathematically proven.



Useful Notes for our application.

Meaning of the function codes given in datasheet.

There are function codes written in the datasheet such as P07.12. If allowed, one may manually set the corresponding parameter. To set it over MODBUS, we need register adress. We may calculate register addreses corresponding to a spesific function code.

P07.12

Hex value of 07 = 07

Hex value of 12= 0C

Then the register address of P07.12 is 0x070C

MODBUS function code 6 (write single register):

Request by master format:

Slave ID | Function Code | Register Address | Data | CRC

1 Byte | 1 Byte | 2 Byte | 2 Byte | 2Byte

0F 06 20 00 00 05 43 27

Slave Address: 15 (0F)

Function Code: 6 (06)

Register address: 8192 (2000)

Data: 05 (05)

CRC: 17191 (4327)

Reply by slave format:

Slave ID | Function Code | Register Address | Data | CRC

1 Byte | 1 Byte | 2 Byte | 2 Byte | 2Byte

0F 06 20 00 00 05 43 27

Slave Address: 15 (0F)

Function Code: 6 (06)

Register address: 8192 (2000)

Data: 05 (05)

CRC: 17191 (4327)

if error occurs, function code replied by slave is neither 3 or 6.

MODBUS function code 3 (read holding registers):

Request by master format:

Slave ID | Function Code | Starting register's address | Quantitiy of registers | CRC

1 Byte | 1 Byte | 2 Byte | 2 Byte | 2Byte

0F 03 07 0B 00 01 F5 92

Slave addres: 15 (0F)

Function code: 3 (03)

Starting Adress: 1803 (070B)

Quantitiy of registers: 1 (0001)

CRC: 62866 (F592)

Reply by slave format:

Slave ID	Function Code	Data byte count	Data	CRC
1 Byte	1 Byte	2 Byte	(Data byte count) Byte	2Byte

0F 03 02 00 00 D1 85

Slave addres: 15 (0F)

Function code: 3 (03)

Byte count:2 (02)

Register values: 0 (0000)

CRC: 53637 (D185)

if error occurs, function code replied by slave is neither 3 or 6.

Configuring solar inverter for MODBUS operation:

1) To be sure, factory reset the device. Set P00.18 to 2;

2) Set P00.01 as 2. (run commands are given over MODBUS, remote control)

3) Set P00.06 as 8. (Do not know whether it actually important or not. Yet it works)

4) Set P00.09 as 0. (Do not know whether it actually important or not. Yet it works)

5) Set P14.00 as slave address (1-247)

6) Set P14.01 as 3 (BAUDRATE = 9600BPS)

7) Set P14.02 as 0 (8N1- 8bits data, none parity check, 1 stop bit)

8) May set P14.03 (read its description)

External sources.

- I learned how to use MATLAB MODBUS-RTU library from here:https://www.mathworks.com/matlabcentral/answers/1917-modbus-rtu-communication
- I found an excell that generates CRC for given data. I mimiced its algorithm in my code. Excell can be found here https://www.simplymodbus.ca/crc.xls

1.Setup (open serial port)

First, disconnect all the devices (i.e. PORTs), clean command window etc.

```
instrreset;
clear all;
close all;
clc;
```

```
% remove any remaining serial objects to prevent serial issues
```

Then continue with configuring serial object. You may want to some parameters according to the device settings. Possible values to be set is listed below (Ones that Tescom driver is set to is underlined).

- COM={**COM1, COM2, ..., COMX**}
- BaudRate={**110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200**}
- DataBits ={**7, 8, 9, 10, 11**}
- StopBits ={**1, 2**}
- Parity = {**'Odd', 'Even', 'None'**}

```
display('Communication parameters are;')
COM      ="COM5"
BAUD_RATE=9600
DATA_BITS=8
STOP_BITS=1
PARITY   ="None"
TIME_OUT =0.1
```

```
%SET which port you have connected device to
ser = serial(COM);
%SETconnection parameters
set(ser,'BaudRate',BAUD_RATE,'DataBits',DATA_BITS,'StopBits',STOP_BITS,'Parity',PARITY,'Timeout
%Open serial connection
fopen(ser);
% Specify Terminator - not used for binary mode (RTU) writing
ser.terminator = 'CR/LF';
% Set read mode
set(ser,'readasyncmode','continuous');
%Check Open serial connection
SERIAL_PORT_STATUS=ser.Status;

if(SERIAL_PORT_STATUS)
    display('Serial port is succesfully connected')
    %else is not needed, since it will throw a warning when running
    %fopen(s) if something goes wrong
end
```

2.Setting motor parameters

```
DRIVER_MODBUS_ADDRESS =    15;

NOMINAL_POWER_P02_01 =     25;% /10 kW
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '01'; '00'; dec2hex(NOMINAL_POWER_P
data_to_send= append_CRC_to(data_to_send);
```

```
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);

NOMINAL_FREQUENCY_P02_02 = 3500;% /100 Hz, should be less then maximum freq. otherwise gives er
significant_hex= bitshift(NOMINAL_FREQUENCY_P02_02,-8);
least_hex= mod(NOMINAL_FREQUENCY_P02_02,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '02'; dec2hex(significant_hex,2); c
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);

NOMINAL_RPM_P02_03 =     1200;% /1 rpm
significant_hex= bitshift(NOMINAL_RPM_P02_03,-8);
least_hex= mod(NOMINAL_RPM_P02_03,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '03'; dec2hex(significant_hex,2); c
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);

NOMINAL_VOLTAGE_P02_04=  225;% /1 Volt_RMS
significant_hex= bitshift(NOMINAL_VOLTAGE_P02_04,-8);
least_hex= mod(NOMINAL_VOLTAGE_P02_04,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '04'; dec2hex(significant_hex,2); c
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);

NOMINAL_CURRENT_P02_05=  15;% /10 Amps_RMS
significant_hex= bitshift(NOMINAL_CURRENT_P02_05,-8);
least_hex= mod(NOMINAL_CURRENT_P02_05,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '05'; dec2hex(significant_hex,2); c
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);

STATOR_RESISTANCE_P02_06=  878;% /100 Ohms
```

```
significant_hex= bitshift(STATOR_RESISTANCE_P02_06,-8);
least_hex= mod(STATOR_RESISTANCE_P02_06,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '06'; dec2hex(significant_hex,2); d
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);

ROTOR_RESISTANCE_P02_07=  750;% /100 Ohms
significant_hex= bitshift(ROTOR_RESISTANCE_P02_07,-8);
least_hex= mod(ROTOR_RESISTANCE_P02_07,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '07'; dec2hex(significant_hex,2); d
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);

LEAKAGE_INDUCTANCE_P02_08=  750;% /10 mH
significant_hex= bitshift(LEAKAGE_INDUCTANCE_P02_08,-8);
least_hex= mod(LEAKAGE_INDUCTANCE_P02_08,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '08'; dec2hex(significant_hex,2); d
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);

INDUCTANCE_P02_09=  750;% /10 mH
significant_hex= bitshift(INDUCTANCE_P02_09,-8);
least_hex= mod(INDUCTANCE_P02_09,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '09'; dec2hex(significant_hex,2); d
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);

FREE_WHEELLING_CURRENT_P02_10=  750;% /10 A
significant_hex= bitshift(FREE_WHEELLING_CURRENT_P02_10,-8);
least_hex= mod(FREE_WHEELLING_CURRENT_P02_10,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '02'; '09'; dec2hex(significant_hex,2); d
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
```

```
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);
```

3. Max output frequency

Max frequency can not be set if motor is not stopped. Be carefull.

```
MAX_OUTPUT_FREQUENCY_P00_03=  2550;% /100 Hz
significant_hex= bitshift(MAX_OUTPUT_FREQUENCY_P00_03,-8);
least_hex= mod(MAX_OUTPUT_FREQUENCY_P00_03,256);

data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '00'; '03'; dec2hex(significant_hex,2); d
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);
```

4.Turning-on and turning-off the motor

forward ve backwards run için 5 kullan, jog için 8 kullan durdururken. sebebini bilmiyorum

```
MOTOR_MODE=5;%STOP
if     (true)%STOP
 MOTOR_MODE=5;
elseif (false)%RUN FORWARD
 MOTOR_MODE=1;
elseif (false)%RUN BACKWARDS
 MOTOR_MODE=2;
elseif (false)%FORWARD JOG
 MOTOR_MODE=3;
elseif (false)%REVERSE JOG
 MOTOR_MODE=4;
elseif (false)%FREE-STOP
 MOTOR_MODE=6;
elseif (false)%ERROR-RESET
 MOTOR_MODE=7;
elseif (false)%STOP JOG
 MOTOR_MODE=8;
elseif (false)%PRE-MAGNETIZATION
 MOTOR_MODE=9;
end
```

```
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '06'; '20'; '00'; '00'; dec2hex(MOTOR_MODE,2)];
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
analyze_reply(outdec,true);
```

5. Operation parameters

```
%%INVERTER_TEMPERATURE P07_12 (C*10)
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '03'; '07'; '0C'; '00'; '01'];
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
INVERTER_TEMPERATURE_P07_12 = analyze_reply(outdec,true)
%%REFERENCE FREQUENCY P17_00 (Hz*100)
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '03'; '11'; '00'; '00'; '01'];
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
REFERENCE_FREQUENCY_P17_00 = analyze_reply(outdec,true)
%%OUTPUT FREQUENCY P17_01 (Hz*100)
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '03'; '11'; '01'; '00'; '01'];
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
OUTPUT_FREQUENCY_P17_01 = analyze_reply(outdec,true)
%%OUTPUT VOLTAGE P17_03 (V)
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '03'; '11'; '03'; '00'; '01'];
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
OUTPUT_VOLTAGE_P17_03 = analyze_reply(outdec,true)
%%OUTPUT CURRENT P17_04 (A/10)
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '03'; '11'; '04'; '00'; '01'];
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
OUTPUT_CURRENT_P17_04 = analyze_reply(outdec,true)
%%MOTOR RPM P17_05 (RPM)
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '03'; '11'; '05'; '00'; '01'];
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
```

```
outdec = fread(ser);
MOTOR_RPM_P17_05 = analyze_reply(outdec,true)
%%DC_BAR_VOLTAGE P17_11 (V)
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '03'; '11'; '05'; '00'; '01'];
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
DC_BAR_VOLTAGE_P17_11 = analyze_reply(outdec,true)
%%INPUT GRID CURRENT P17_35 (V)
data_to_send=[dec2hex(DRIVER_MODBUS_ADDRESS,2); '03'; '11'; '05'; '00'; '01'];
data_to_send= append_CRC_to(data_to_send);
request = uint8(hex2dec(data_to_send));
fwrite(ser, request);
outdec = fread(ser);
INPUT_GRID_CURRENT_P17_35 = analyze_reply(outdec,true)
```

6.Closing serial

```
fclose(ser);
delete(ser);
clear ser
disp('Serial port object s, is disconnected')
```

Appendix I: Slave reply decoding

This functions takes reply and decodes it. If it is succes, the returned value will be the numerical data of the readed register, otherwise it will be false. If should_print is set true, it will print the content of the reply if it is succes. Otherwise it will not print the content but will say ERROR.

```
function returned_data = analyze_reply(r_dec, should_print)

    isSucces= false;
 if r_dec(2) ==3 | r_dec(2)==6 %read->3, write->6,
    isSucces = true;
 end

 if(should_print)
  if r_dec(2)==6 %%reply for write request
      if(isSucces)
        fprintf('WRITE-SUCCES|%s| ID:%d REQUEST:%d ADDRESS_DEC:%d ADDRESS_HEX:%s%s DATA:%d\n',
        returned_data=r_dec(5)*256+r_dec(6);
      else
        fprintf("WRITE-ERROR|%s| ID:%d\n", datestr(now,'HH:MM:SS'),r_dec(1) );
        returned_data=false;
      end
  elseif r_dec(2)==3 %%reply for read request
```

```
      if(isSucces)
        fprintf("READ-SUCCES|%s| ID:%d VALUE:%d" , datestr(now,'HH:MM:SS.FFF'),r_dec(1),r_dec(4
        returned_data=r_dec(4)*256+r_dec(5);
      else
        fprintf("WRITE-ERROR|%s| ID:%d\n", datestr(now,'HH:MM:SS'),r_dec(1));
        returned_data=false;
      end
  end

 end
end
```

Appendix II: CRC-generator

```
%data_to_send = ['0F'; '06'; '20'; '00'; '00';'01' ];
%append_CRC_to(data_to_send)
function CRC_ADDED_DATA = append_CRC_to(DATA_TO_SEND)
    last= 65535;
    for i=1:length(DATA_TO_SEND)
    last = CRC_16(hex2dec(DATA_TO_SEND(i,1:2)),last);
    end

    significant_hex= dec2hex(bitshift(last,-8),2);
    least_hex = dec2hex(mod(last,256),2);

    CRC_ADDED_DATA = [DATA_TO_SEND;least_hex; significant_hex];

end
function CRC_16_bit = CRC_16(DATA, LAST_DATA)
 KEY = 40961; %1010 0000 0000 0001
 %if this is first data (i.e LAST_DATA==null), LAST_DATA= 65535 = FFFF
 data = bitxor(DATA, LAST_DATA);

for c = 1:8
   should_xor= false;
   if (mod(data,2)==1)
           should_xor=true;
   end
   data=bitshift(data,-1);
   if(should_xor)
   data=bitxor(data, KEY);
   end
end

CRC_16_bit = data;
end
```