Control of a Solar Panel with Modbus

Julien Chevalley and Nils Ritler

Systèmes d'Information - Microcontrôleur

S3fb

Sion, le 3/17/2023

V1.0

TABLE DES MATIÈRES

1. INTRODUCTION

The objective of this laboratory is to drive the load on a small solar panel in order to achieve the MPP (Maximum Power Point) of this solar panel. The solar panel is connected to a PIC18. This system allows the PIC to measure the current and voltage delivered by the solar panel as well as driving the load using PWM on the solar panel. This system which represents the server needs then to be driven by a client using the Modbus protocol communication via UART. This project is separated in two parts:

a) Measure, display and save in registers of the solar panel parameters (Current, Voltage and PWM values)
b) Communication with the client using Modbus protocol.

The control and search of the MPPT is done by the client (PC connected by USB to the microcontroller). The PWM value is then feed back to the microcontroller to reach the MPP. The program used on the client was provided by the school.

2. DESCRIPTION AND DOCUMENTATION OF THE CODE

2.1 Measure of Voltage, Current and setup of the PWM on the PIC

In order to perform the measures, we have been given a Solar Panel with an integrated analog measurement circuit. To interpret the output voltage from the OpAmps we must use the Analog-to-Digital converter of the PIC. Indeed, the OpAmps can output a voltage between 3.3V and 0V that represent the physical value to be measured.

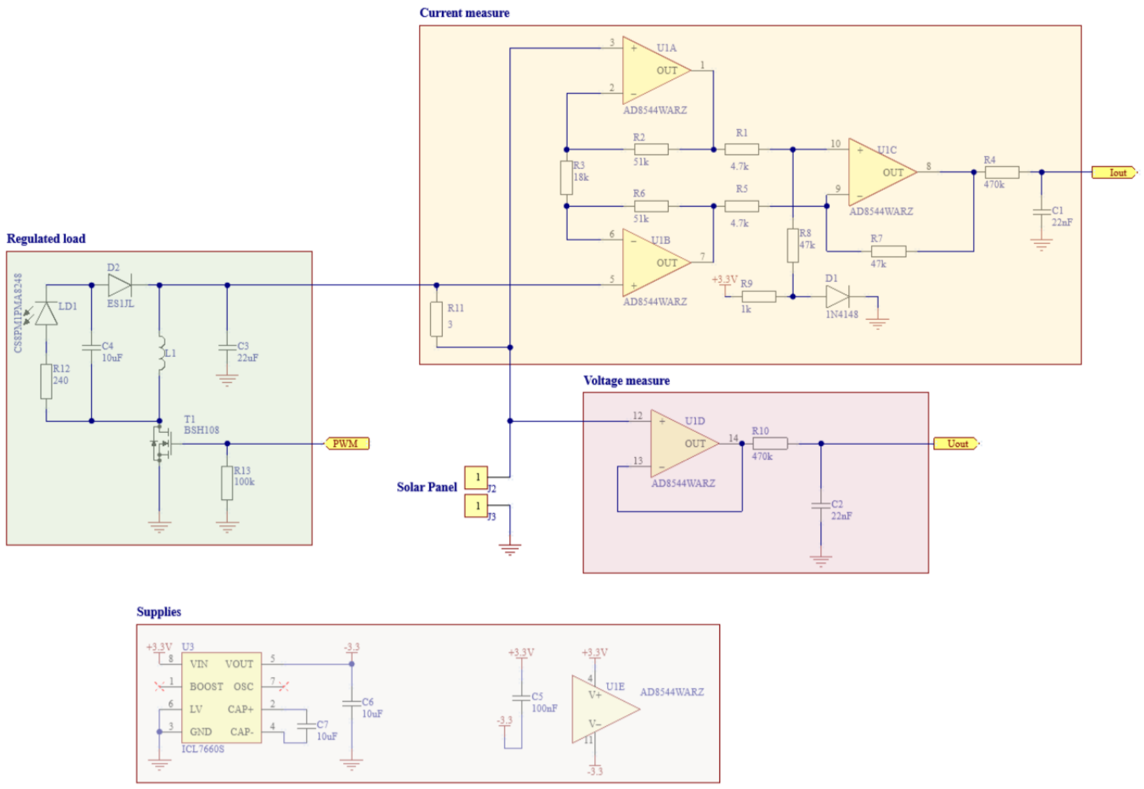
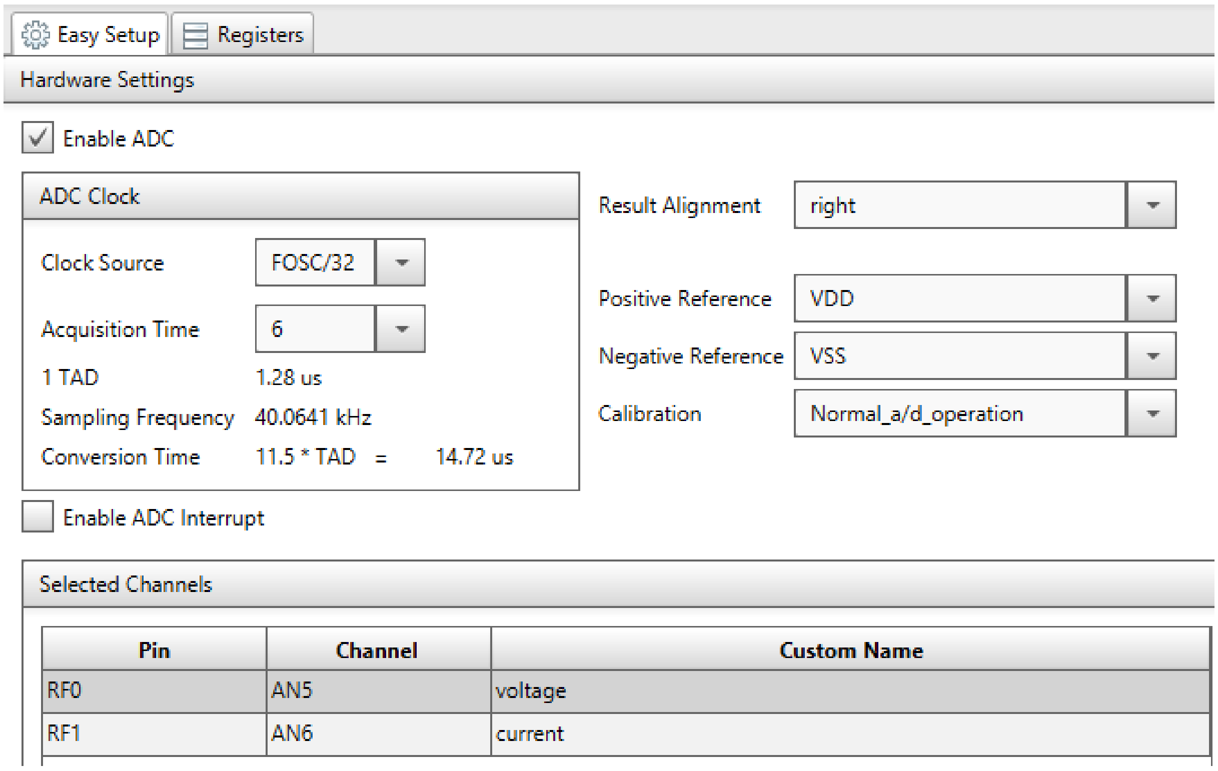


Figure 1 - Analog measurement circuit of the Solar Panel

The ADC was configured using the MCC module of MPLAB. You can see in figure 2 below how the ADC was configured. We made an additional change by naming channel AN6 and AN5 as current and voltage for ease-of-use later on in the code.



✓ Enable ADC

ADC Clock

Clock Source	FOSC/32		
Acquisition Time	6		
1 TAD	1.28 us		
Sampling Frequency	40.0641 kHz		
Conversion Time	11.5 * TAD	=	14.72 us

Result Alignment	right
Positive Reference	VDD
Negative Reference	VSS
Calibration	Normal_a/d_operation

Enable ADC Interrupt

Selected Channels

Pin	Channel	Custom Name
RF0	AN5	voltage
RF1	AN6	current

Figure 2 - Setup of the Analog-To-Digital Converter

In order to get a coherent value, we needed to do a sampling of the measurement using the equation below :

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad (1)$$

Where n represent the sampling rate defined as AVERAGE_SAMPLES in the measure.c file. In this case, we then take the average of 32 measures as our value. This step is encapsulated in the measure_adc() function.

We then need to convert the 10bit sampled value in an integer that we can later on display on an LCD Screen and/or transmit via Modbus.

a. Voltage

The measure of the voltage is made by the function measure_voltage(). In there, we take the value of the ADC measure of the input "voltage" and save it in the variable "result". The result value is on 10-bit. This means that for 3.3V (biggest output value of the OpAmp) we will have a value of 1024 and for 0V, we will have a value of 0. In order to output a value which is human readable, we simply implemented the following function in our code :

$$V_{meas} = \frac{Result * MaxOpAmpOutput}{ADC\_Resolution} \; [mV]$$

We had to take extra care that in that calculus, we did not overflow the value of V_meas. That is why we had to use 32-bit variables. That value is then returned by the function. Now we have the correct value of the voltage from the panel and can use it for later.

b. Current

The measure of the current is like the measure of the voltage. It is also made by a function called measure_current(uint16_t offset). Like in voltage, we take the measure_adc() of the current, multiply it with the ADC_REFH and divide it by ADC_RESOLUTION. The value that we get here is millivolts as ADC_REFH value is 3300 (mV).

This means that the value we have now is the voltage at the output of the current measurement circuit. In order to have a value for the current we need to take into account the input resistor (R3) as well as the gain of that circuit (A=66).

As the current that our solar panel is extremely small, we need to measure µA. Therefore, using Ohm's law, the gain of the circuit and the units in presence, we had to make the following calculation in our code:

$$I_{meas} = \frac{1000 * mV_{OpAmpCurrent}}{R3 * A} [\mu A]$$

We then need to subtract to that value the offset if its smaller than the value measured.

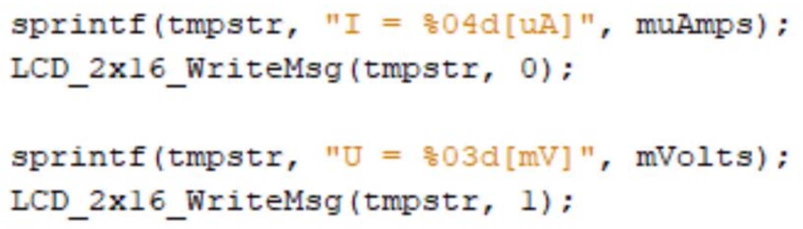
To measure that offset at startup, we measure the current using an offset of 0 and a PWM of 0.

2.1.1 Main

At this point both functions for the measures were made and now we could store this values in the correct input registers. For that we made two variables in the main with the names mVolts an muAmps. In the variable mVolts we stored the value of measure_voltage() and put that variable in the input_registers[0]. The same we did with measure_current(firstMuAmps), which we stored in the input_register[1]. The parameter firstMuAmps we got from a first measurement with a PWM of 0.

2.1.2 LCD

To control our measurement, we displayed the values of both measures of an LCD-Display. For that we had to include the library lcd/lcd.h. First, we had to initialize the LCD with the function Lcd_Init(). With the function sprintf() we could show our values on the LCD.



```
sprintf(tmpstr, "I = %04d[uA]", muAmps);
LCD_2x16_WriteMsg(tmpstr, 0);

sprintf(tmpstr, "U = %03d[mV]", mVolts);
LCD_2x16_WriteMsg(tmpstr, 1);
```

Figure 3: Display of the measurements

This was only used for us to have a better oversight of the values.

2.1.3 PWM

In order to be able to control the load on the solar panel and reach the MPP, we need to enable the PWM. As the PWM is using Timer 2, we also had to activate the Timer 2. These two steps were done using MCC. The code generated in epwm1.c provided us with a method to load a duty value. As PWM on the PIC uses 10-bits words, we can load a value between 0 and 1024. The ideal PWM value is calculated on the client using the software solar_controller.jar.

So once the PWM and Timer 2 were activated using MCC, all we had to do was to use the method EPWM1_LoadDutyValue(uint16_t dutyValue) in the loop. For the client to correctly tune the solar panel, we had to use the value stored in the holding_register[0] in that function. Hence, at each loop, we get the value stored in the holding_register[0] and then take that value (ZControl in the main) to the EPWM1_LoadDutyValue() method. The holding_register[0] value will be updated by the client (computer) through Modbus using the write register function.

2.2 Setup of Modbus server and communication via UART

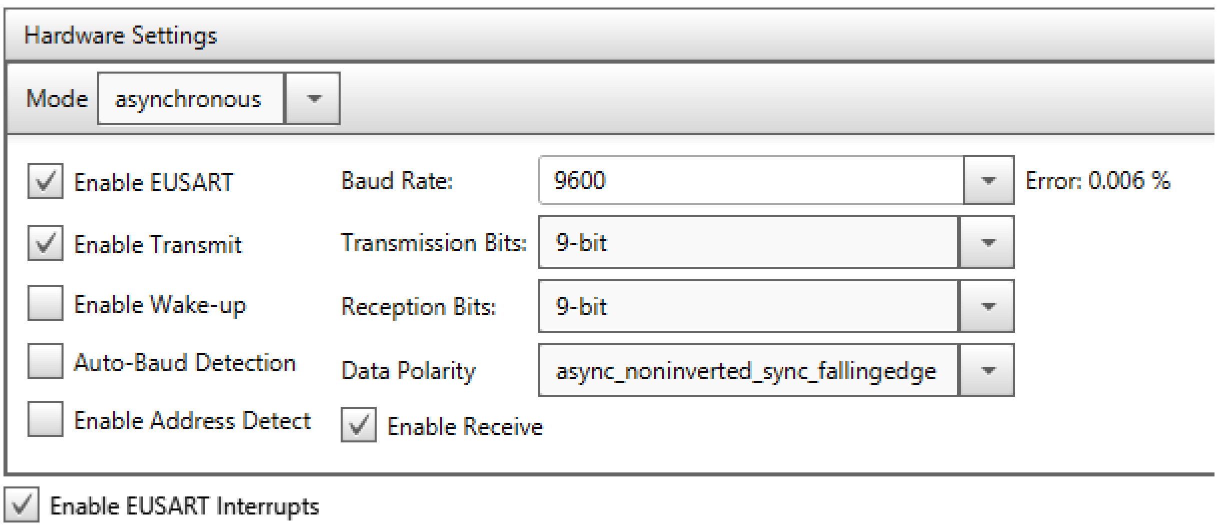
At this point we have our measures of voltage and current stored in the input registers as well as a mean to regulate the load using a PWM value between 0 and 1024 stored in the holding register. We now need to setup the uart communication on the client and write some code to setup the handling of the Modbus protocol.

2.2.1 UART And Timer0 Setup

In order to setup our server (PIC) so that it can respond via Modbus to the client, we need to ensure we have a reception container. That container is the rx buffer of the UART. We also need to check that we have finished to receive a frame. The communication between the server and the client is regulated by the baud rate which must be set at 9'600 in this application. We also must use 2 stop bits. One uart communication consist of:

- 1x 8-bit (=1 byte) character,
- 1x start bit.
- 2x stop bit.

We will then setup the UART in MCC as described : baud rate of 9600, 9-bits transmission (8 bits + 1 additional stop bit) and interrupt enabled. This is shown in figure 4 below.


Hardware Settings

Mode: asynchronous

- [✓] Enable EUSART
- [✓] Enable Transmit
- [ ] Enable Wake-up
- [ ] Auto-Baud Detection
- [ ] Enable Address Detect
- [✓] Enable Receive

Baud Rate: 9600 Error: 0.006 %
Transmission Bits: 9-bit
Reception Bits: 9-bit
Data Polarity: async_noninverted_sync_fallingedge

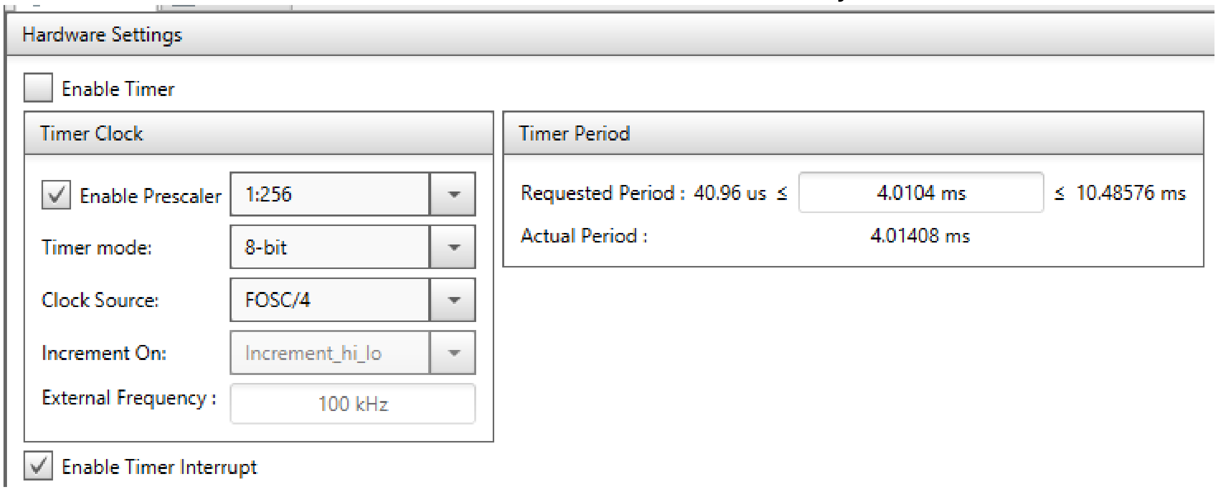
- [✓] Enable EUSART Interrupts

Figure 4 - Setup of UART in MCC

In Modbus, the transmission of each character will last: 11*1/9600 = 1.145 ms. The duration of 3.5 character is then 3.5*1.145 = 4.0104 ms. We then need to configure 2 more things before starting the Modbus analysis : Timer0 and Interrupts.

The Timer 0 must need to complete a loop (from load value to FFFF) in 4.0104 ms. That is the time where we are sure the transmission of a Modbus frame is over. When that timer overflows, it raises an interrupt in which we can process the Modbus request stored in the RX buffer. That setup was also done in MCC as shown in figure 5 below. When the Timer 0 overflows, it raises an interrupt flag and the function modbus_timer() is executed (in the interrupt manager). This function initiates the processing of a Modbus frame.

Finally, we need to start the Timer 0 at the right time. That means, each time a new character is received, the Timer 0 must restart at its load value. In order to detect the reception of a new character, we need to use the interrupt of the UART. That is why interrupts were enabled at the setup of UART (see figure 4 above). So with this setup, each time a new character is received through UART, an interrupt is raised and we need at this moment to reload the starting value and start / restart the timer 0. That is, the timer 0 starts only once the first character has


Hardware Settings

☐ Enable Timer

Timer Clock

✓ Enable Prescaler	1:256
Timer mode:	8-bit
Clock Source:	FOSC/4
Increment On:	Increment_hi_lo
External Frequency :	100 kHz

Timer Period

Requested Period : 40.96 us ≤ 4.0104 ms ≤ 10.48576 ms
Actual Period : 4.01408 ms

✓ Enable Timer Interrupt

Figure 5 - Timer 0 setup in MCC

been received. That is, the timer 0 starts only once the first character has been received. This part (reload and start timer) was implemented in the modbus_char_recvd(c) which also stores the character received in a buffer called rx_buf. The call of modbus_char_recvd(c) is done from the interrupt_manager.c file when a EUASART1 interrupt occurs (new char received).

2.2.2 Modbus Frame Analysis

In this part, we will briefly describe how we coded the handling of the Modbus on the client's side. We will start with the main.c file and then describe the functions in the modbus.c file.

In the main.c file, we only do a few initialization steps for the Modbus protocol to be analyzed correctly:

1. Activate the Global Interrupts (For Timer 0 and UART)
2. Activate the Peripheral Interrupts (For UART specifically)

3. Give the address of the "Modbus server" (modbus_init(0x80)). Variable modbusAddress is then initialized.

The rest of the coding was done within the interrupts.

As seen previously, whenever a frame is complete the modbus_timer() function is called. Whenever a character has been received, the function modbus_char_recvd(c) is called and the character is stored in a table and the Timer 0 is reset.

When the frame is complete, the Timer 0 must be stopped (until a new character is received) and the Modbus frame must be analyzed in order to prepare the correct response to the client. This is done in the modbus_analyse_and_answer() function.

Requests are as follows:

a. Read one or several values from the input registers.
b. Read the value from the holding register.
c. Write a value to the holding register.

The three requests above are part of a switch case and are then each treated differently. In order to address these cases we must first ensure that:

a. The client is talking to that server. If the address specified in the Modbus frame is the one of the server (and not another server). This is simply done by checking if the modbusAddress (in the PIC) correspond to the first byte of the rx_buf.
b. The CRC is correct.

To check if the CRC is correct, we used the function CRC16(..) which was provided and returns a 16 bits word. We then compared that value to the last two bytes of the rx_buf put together with the line of code:

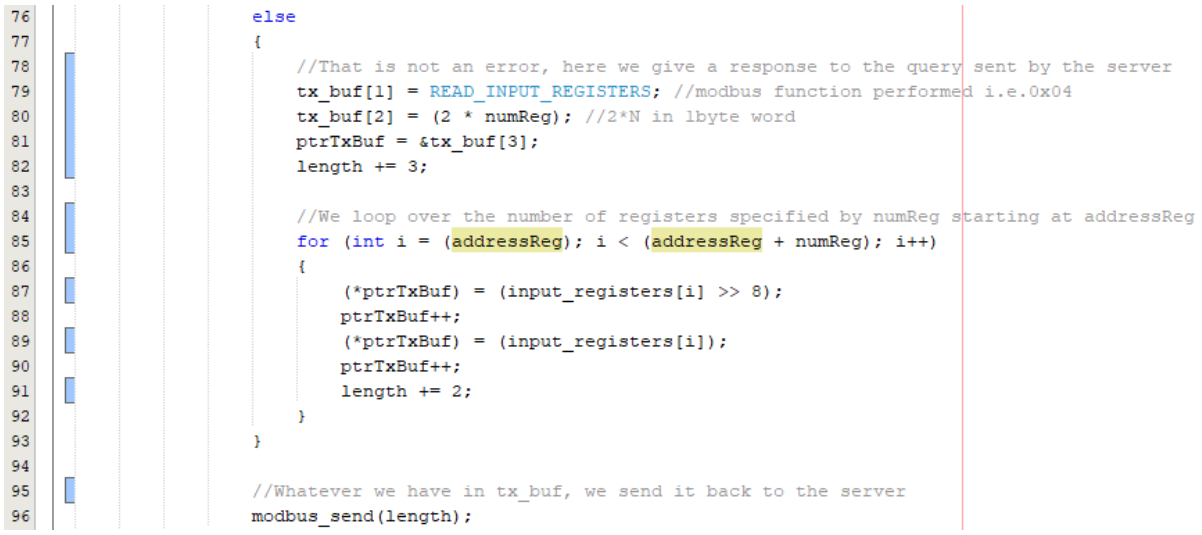
```
(rx_buf[index-1] << 8 | rx_buf[index - 2]) == CRC16(rx_buf, (index - 2))
```

Regarding the different requests, those are analyzed in a switch case which is done on the second value of the rx_buf table. The possible values are: 0x04, 0x03, 0x06 which in the code are defined as READ_INPUT_REGISTERS, READ_HOLDING_REGITERS, WRITE_SINGLE_REGISTER. For each case we had to use an if…else if…else to handle errors. We will go briefly through the first case. We used A similar approach for the other cases.

READ_INPUT_REGISTERS

To this point we ensured by reading the rx buffer that the request was for that server, that the Modbus frame was not corrupted (CRC) and the request was to read one or more registers. The next two bytes (rx_buf[2] and [3]) contain the starting address and are then stored in a variable. And finally, the bytes 4 and 5 contain the number of registers to read (1 or 2).

We then fill the tx_buf with the right values which in the by byte order are: ModbusAddress, function value (0x04), 2*N (N=number of registers to read), 1 or 2 values from the input_registers (voltage, current or voltage and current) each stored in two bytes. Finally, we update the length of the tx_buf depending on the number of registers that was assigned to the tx_buf and execute the modbus_send() function as can be seen in the figure 6 below.



```
else
{
    //That is not an error, here we give a response to the query sent by the server
    tx_buf[1] = READ_INPUT_REGISTERS; //modbus function performed i.e.0x04
    tx_buf[2] = (2 * numReg); //2*N in 1byte word
    ptrTxBuf = &tx_buf[3];
    length += 3;

    //We loop over the number of registers specified by numReg starting at addressReg
    for (int i = (addressReg); i < (addressReg + numReg); i++)
    {
        (*ptrTxBuf) = (input_registers[i] >> 8);
        ptrTxBuf++;
        (*ptrTxBuf) = (input_registers[i]);
        ptrTxBuf++;
        length += 2;
    }
}

//Whatever we have in tx_buf, we send it back to the server
modbus_send(length);
```

Figure 6 - code extract of tx_buf building.

The modbus_send() function takes one argument which is the length of the tx_buf table. In this function, we compute the CRC for the transmit frame and add it to the tx_buf table. We also update the length of that table. Finally, we loop over each byte in the tx_buf and send it to the uart using the method EUSART1_Write().

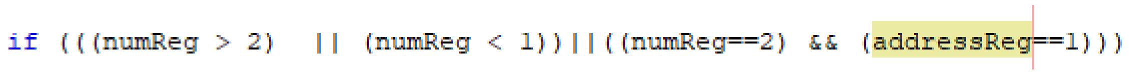
At this point, we have managed to read a request from the client and send it back. The rest of the project remained in testing the Modbus frames using the modbus_analyzer program as well as using the solar_controller.jar which sends read requests constantly to get the last values of current and voltage and sends back write requests to update the PWM holding registers in order to find the MPP.

2.2.3 Modbus Errors Handling

To deal with wrong requests from the client we have upgraded our code so that the client could "understand" why a request would not work and get an error code that could be interpreted. We implemented three error codes:

1. ILLEGAL DATA VALUE – 03
2. ILLEGAL DATA ADDRESS – 02
3. ILLEGAL FUNCTION – 04

The illegal data value error code was raised when the number of registers requested was incorrect. In the case of a read input register, that condition was a bit more complicated to find as there is one or two registers that could be read but depending on the starting address, probably only one register could be read. We then used the line of code below to address that problem:



```
if (((numReg > 2)  || (numReg < 1))||((numReg==2) && (addressReg==1)))
```

The illegal data address error code is raised only when the address requested is out of range. For instance, requesting to write on address 1 of the holding registers is inconsistent as there is only one address in that register that can be written, and its value is 0.

For the illegal function error code, this was simply put in the default case of our switch case, as this would mean the function code was not recognized.

The routine below was then elaborated to test each of our cases and using the modbus_analyzer we could confirm that our code was running as expected. In the figure 7, we can see that each frame was provided with a corresponding error code. The error displayed in that figure is related to the last test which is the ILLEGAL FUNCTION error code.

a. Modbus frame to read input register with a wrong number of registers :
80 04 00 01 00 02
We try to read 2 registers starting at address 1. As we start at address 1, we can only read 1 register.
Expected response code: 84 – 03 – ILLEGAL DATA Value

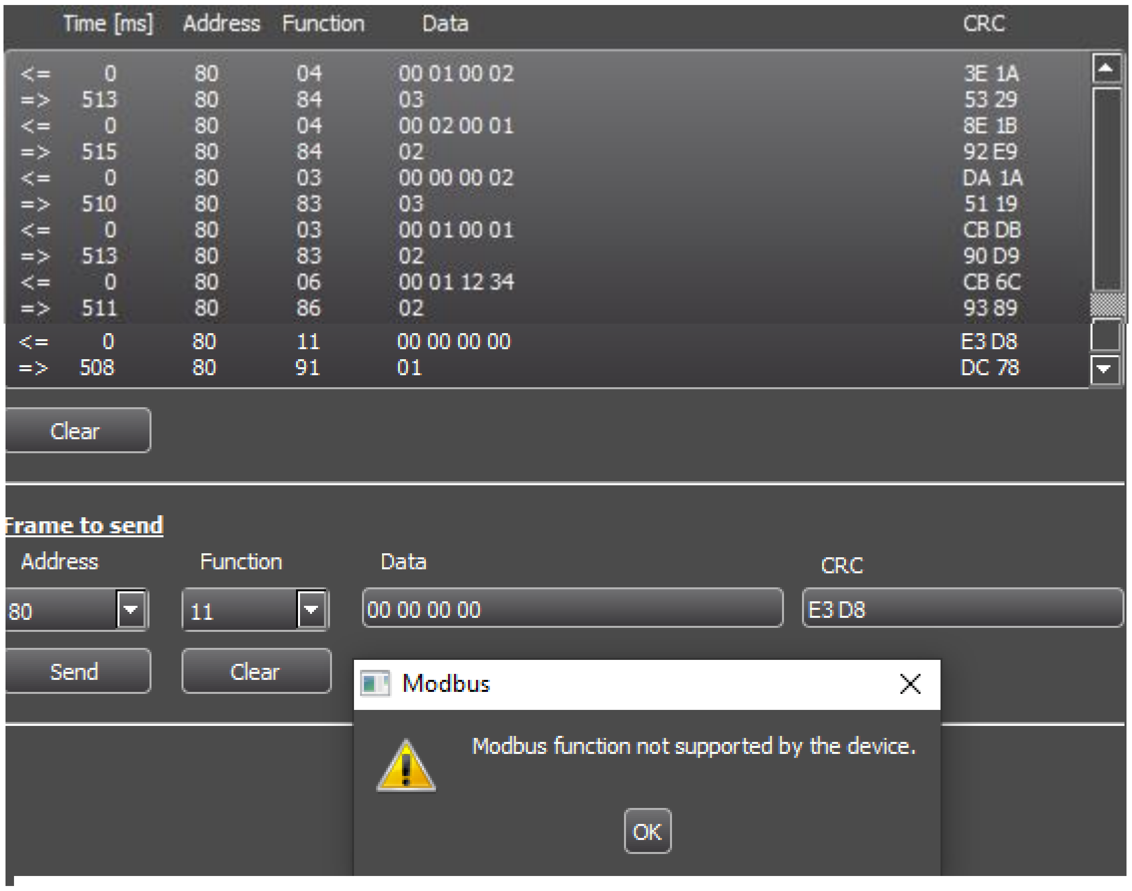
b. Modbus frame to read an input register with a wrong address:
80 04 00 02 00 01
We try to read 1 register starting at address 2 although the last readable address is 1.
Expected response code: 84 - 02 - ILLEGAL DATA Address

c. Modbus frame to read holding register with a wrong number of registers:
80 03 00 00 00 02
We try to read 2 registers starting at address 0 and we can only read 1 holding register.
Expected response: 83 - 03 - ILLEGAL Data Value

d. Modbus frame to read holding register with a wrong address:
80 03 00 01 00 01
We want to read 1 register starting at address 1 which is out of the accessible address range.
Expected response: 83 - 02 - ILLEGAL DATA Address

e. Modbus frame to write in the holding register using a wrong address.
80 06 00 01 12 34
We want to write the value 0x1234 to address 1 which is out of range.
Expected response: 86 - 02 - ILLEGAL DATA Address

f. Modbus frame using a wrong function code:
80 11 00 00 00 00
Function 0x11=17 does not exist.
Expected response: 91 - 01 ILLEGAL FUNCTION



	Time [ms]	Address	Function	Data	CRC
<=	0	80	04	00 01 00 02	3E 1A
=>	513	80	84	03	53 29
<=	0	80	04	00 02 00 01	8E 1B
=>	515	80	84	02	92 E9
<=	0	80	03	00 00 00 02	DA 1A
=>	510	80	83	03	51 19
<=	0	80	03	00 01 00 01	CB DB
=>	513	80	83	02	90 D9
<=	0	80	06	00 01 12 34	CB 6C
=>	511	80	86	02	93 89
<=	0	80	11	00 00 00 00	E3 D8
=>	508	80	91	01	DC 78

Clear

Frame to send

Address	Function	Data	CRC
80	11	00 00 00 00	E3 D8

Send Clear

Modbus

Modbus function not supported by the device.

OK

Figure 7 - Modbus analyzer : responses to wrong Modbus frames

3. DESCRIPTION OF PROBLEMS / DIFFICULTIES AND CORRESPONDING SOLUTIONS

3.1 Measures of Voltage and Current

First we had to understand where we measure voltage and where current. For that we looked at the Analog measurement of the solar panel (Figure 1), which we had in the project description. There we saw that for the current measure, we had to divide by the gain of the OpAmp and the Resistor R11, because there we had the correct values of the current. For the voltage measure we didn't have to do that.

The measurements of U and I also contained noise. To erase that noise we had to do use a 32-sample of the measurements. This was easily done by adding every measure till we reached a number of measurements of 32. This sum we then divide by 32 and so we had an average result and filtered the noise.

After the calculation of the correct value of the two measurements, we had to put it in the correct unit. The predefined units were µA and mV. The ADC measure of the voltage was already in mV and so was the current in mA. We then had to multiply the current measure by 1000 in order to have the correct unit.

When we then did our first tests with the measurements, we saw that the current measure was not changing. That was because we had no correct PMW, because we didn't configure it at that time. That problem we solved by putting the input of the PMW to GND. This solution also solved another problem of the current measure. Every time the offset of the measure changes at the beginning. With a PMW of 0 we saw the actual value of the offset. Now we saw a correct value of both measurements and how they change.

The problem with the offset we then corrected, by using a PMW of 0 at every initialization of the program. So we had the offset at every start and could subtract it from the current measure.

The PWM was also made by the program MCC. We did a little change in the generated code for our program. Later we saw that the PWM was not correct defined in the MCC and we changed that, so that we have a PWM of about 24 KHz. As soon as we generated the new PWM, the program asked us, if we wanted to delete our changed code. We didn't understand how it was correctly done and so it created not the correct code for the right PWM. With the help of our teacher, we could solve that problem and we had an optimal PWM.

3.2 Modbus, UART and Timer0

This part of the project was done relatively flawlessly except probably for the beginning and the end. Indeed, it was very difficult to understand how to correctly setup the UART, the interrupt and the timer 0. Indeed, we needed some time to adjust to the use of a GUI instead of assembler to configure those elements. The fact that a lot of code was generated automatically and had to be used to go further in the rest of the project made the project look harder than it actually was.

The part where we also struggled was during the use of the modbus_analyzer when testing our code. As we had to build Modbus frames "by hand", it was particularly difficult to remember what each byte meant how and it should be treated later in the code.

It was also pretty difficult not to lose itself while testing and coding. Indeed, between the coding and the decryption / encryption of Modbus frames it was very easy to lose sight of the objective pursued.

The handling of errors was particularly challenging as we had to test each error case individually and build dedicated Modbus frame to test those errors. We therefore have no certainty that we have treated all the errors we wanted.

Finally, we were lucky because we benefitted from the bad experience of other groups with the solar_controller.jar which runs on a very peculiar version of Java which of course was not installed on the computer we used for the project.

4. CONCLUSION

During this project, we achieved the driving of a load on a solar panel using the Modbus protocol. As said before, the objective was not to write a driving algorithm for tuning the panel, but rather allowing the microcontroller to interpret the commands sent by the client and respond adequately. Hence, the communication between the microcontroller was implemented through the Modbus protocol. In order for the server (microcontroller) to respond correctly to requests from the client, we firstly had to program the microcontroller to harvest the data from a measurement circuit as well as setup a control function of the solar panel (PWM). In the second part of the project, we had to program our microcontroller to store the received Modbus frames (through UART) in a register and then cut it in bytes in order to decode the request. Depending on the type of request, we had to form a Modbus style answer with the correct data and send it back to the client. Finally, we also implemented the handling of errors and managed to form error Modbus coded answers. So if in any case, the client would send a "bad" request, the server would be able to tell what went wrong. The only part we did not implement was the case when the CRC did not check. That is if data would get corrupted while "traveling" through the UART. If this were to happen, the server would simply not answer to the client. The client would then need to send another request.

5. SIGNATURES

Julien Chevalley

[illegible]

Nils Ritler

[illegible]

Sion, 17th of March 2023