|  |
| --- |
| Control of a Solar Panel with ModBus  Julien Chevalley et Nils Ritler  Systèmes d’Information - MicroControleur  S3fb  Sion, le 3/14/2023  V1.0 |



Table des matières

[1. Introduction 3](#_Toc129726259)

[2. Description and Documentation of The Code 3](#_Toc129726260)

[2.1 Measure of Voltage, Current and setup of the PWM on the PIC 3](#_Toc129726261)

[2.2 Setup of ModBus server and communication via UART 4](#_Toc129726262)

[2.2.1 UART And Timer0 Setup 5](#_Toc129726263)

[2.2.2 Modbus Frame Analysis 6](#_Toc129726264)

[2.2.3 Modbus Errors Handling 7](#_Toc129726265)

[9](#_Toc129726266)

[3. Description of problems / difficulties and corresponding solutions 9](#_Toc129726267)

[3.1 Measures of Voltage and Current 9](#_Toc129726268)

[3.2 Modbus, UART and Timer0 9](#_Toc129726269)

[4. Conclusion 10](#_Toc129726270)

[5. Annexe 10](#_Toc129726271)

[6. Signatures 10](#_Toc129726272)

# 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 represent the server needs then to be driven by a client using the modbus protocol communication via UART. This project is separated in two parts :

1. Measure, display and save in registers of the solar panel parameters (Current, Voltage and PWM values)
2. 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 in order to reach the MPP. The program used on the client was provided by the school.

# Description and Documentation of The Code

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

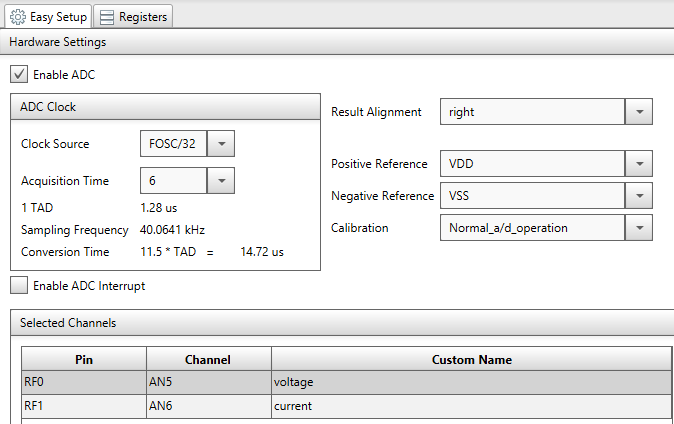
In order to perform the measures we have been given a Solar Panel with an integrated analog measurement circuit. In order to interpret the output voltage from the OpAmps we must use the Analog-to-Digital converter oft he PIC. Indeed, the OpAmps can output a voltage between 3.3V and 0V that represent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

Une image contenant diagramme, schématique

Description générée automatiquement

*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 additionnal change by naming channel AN6 and AN5 as current and voltage for ease-of-use later on in the code.



*Figure 2 - Setup of the Analog-To-Digittal Converter*

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

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.

1. Voltage

The output of the voltage opAmp which is feed into the ADC of the PIC (Pin RF0) and sampled .

1. Current

## 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.

### UART And Timer0 Setup

In order to setup our client (PIC) so that it can respond via modbus to the server, 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 recieve a frame. The communication between the server and the client is regulated by the baud rate which has to be set at 9'600 in this application. We also have to use 2 stop bit. 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 XX below.

Une image contenant texte

Description générée automatiquement

Figure 3 - 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 raise 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 XX below. When the Timer 0 overflows, it raises an interrupt flag and the function modbus\_timer() is executed (in the interrupt manager). This function initiate the processing of a modbus frame.

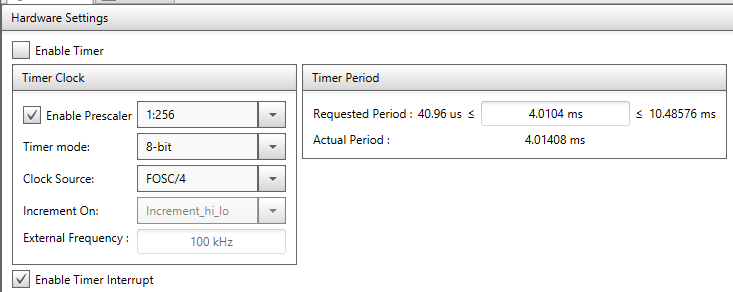
Finally we need to start the Timer 0 at the right time. That means, each time a new character is received, the Timer 0 must restart at its load value. In order to detect the reception of a new character, we need to use the interrupt of the UART. That is why interrupts were enabled at the setup of UART (see figure XX 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 

Figure 4 - 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).

### 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 client” (modbus\_init(0x80)). Variable modbusAddress is then initialized.

The rest of the coding was done within the interupts

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 server. This is done in the modbus\_analyse\_and\_answer() function.

Requests are as follows :

1. Read one or several values from the input registers
2. Read the value from the holding register
3. 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 theses case we must first ensure that :

1. The server is talking to the client. If the address specified in the modbus frame is the one of the client (and not another client). This simply done by checking if the modbusAddress correspond to the first byte of the rx\_buf.
2. The CRC is correct

In order 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 :



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 in order to handle errors. We will go briefly through the 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 client, 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 : ModbusAdress, 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 XX below.

Une image contenant texte

Description générée automatiquement

Figure 5 - 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 server and send it back to the server. The rest of the project remained in testing the modbus frames using the modbus\_analyzer 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.

### 

### Modbus Errors Handling

In order to deal with wrong requests from the server we have upgraded our code so that the server 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 :



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 XX, 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.

* 1. 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**

* 1. 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**

* 1. 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**

* 1. 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 oft he accessible address range.

**Expected response : 83 - 02 - ILLEGAL DATA Address**

* 1. 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**

* 1. Modbus frame using a wrong function code :

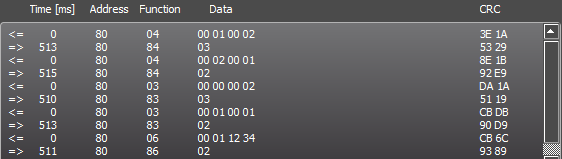
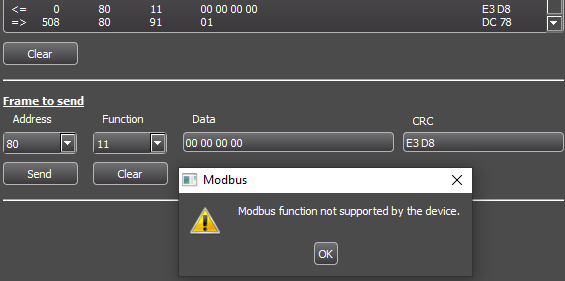
80 11 00 00 00 00

Function 0x11=17 does not exist.

**Expected response: 91 - 01 ILLEGAL FUNCTION**

## 

Figure 6 - modbus analyzer : responses to wrong modbus frames



# Description of problems / difficulties and corresponding solutions

## Measures of Voltage and Current

## 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.

# Conclusion

# Annexe

# Signatures



Julien Chevalley Nils Ritler

Une image contenant texte

Description générée automatiquement

Sion, le 14 avril 2023