An Environment Light Intensity Regulator

Pedro Silva[1] and Pedro Almeida[2]

July 29, 2019

***Abstract*— A PmodALS (ambient light sensor) from Digilent, was used through SPI communication to create an Environment Light Intensity Regulator. The value of the light intensity was thus given to a ATmega328P, our microcontroller unit (MCU), and with it the light intensity of a Mini 8x8 LED Matrix, from Adafruit, was regulated via I2C. The code used to program the MCU was in C and we followed the Misra-C (2004) guidelines, to the best of our ability. Memory tests to the FLASH, SRAM and EEPROM of our MCU and to the HT16K33 of our LED matrix backpack were made. A schematic and a PCB of our Industrial Prototype was done having in mind its size and the width of the channels.**

We were fruitful in making An Environment Light Intensity Regulator with the above specifications, and our work was presented using an ATmega328P, in circuit programmed using an Arduino UNO, via SPI.

I. INTRODUCTION

Our ultimate goal was not only to make a simple Environment Light Intensity Regulator, but to design it while following the bellow mentioned requirements:

- Use C with a given set of guidelines that ensure defensive programming.
- Implement 2 communication protocols.
- Implement software to make memory tests.
- Choose the components and project a PCB for mass production
- Plan out the work on a GANTT chart and properly maintain your code using a version control.

We choose as our source of light a mini 8x8 LED Matrix Backpack, that uses a HT16K33, which is controlled using the I2C protocol. As for the light sensor, the choice fell on the PmodALS from Digilent, based on the Texas Instrument's ADC081S021 and on the Vishay Semiconductor's TEMT60000X01 light sensor. Being that the ALS stands for Ambient Light Sensor and that it communicates with SPI it is perfect for the job at hand.

For the MCU we chose the ATmega328P, being that not only it is one the most well documented microcontrollers, it is also the microcontroller used on the Arduino UNO, thus creating a smooth transition from the Lab Prototype, made using .ino files to the Industrial Prototype made using C. For version control we opted to use Git and as a host for our repository GitLab. As for the guidelines we used Misra-C 2004. We made memory tests to the FLASH, to the SRAM and the EEPROM of the ATmega328p and to the HT16K33 of the LED backpack.

An unrealistic GANTT chart was made, and for the most part it was not followed, because of time conflicts with other subjects as well as logistic problems. Both charts will be present in the annexes.

II. BACKGROUND

Before getting into the specifics of our work, it's important to understand the operation of the communication protocols we decided to use and how their respective devices interact with the ATmega328P. It is also relevant to discuss the different types of memory on the ATmega328P and how they can be tested.

A. I2C & LED Matrix

Invented by Phillips Semiconductors, I2C is a synchronous, multi-master, multi-slave serial computer bus. Being appropriate for peripherals where simplicity and low manufacturing costs are more important then speed, it uses only two bidirectional open drain lines: Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with one resistor which. Typically the voltages +5V and +3.3V are used as V_{dd}. I2C uses a 7-bit address space, plus a read/write bit and an acknowledge from the receiving end. The condition to start the transmission is a fall from the SDA line when the clock is up. Meaning that when the SCL is up, the SDA value will be read and it should remain the same as long as the SCL is up, this if you desire to read data.

The LED Matrix in itself is irrelevant for what the ATmega328P is concerned, the microcontroller will only talk with the HT16K33: RAM mapping 16*8 LED Controller Driver with a keyscan. Right from the get go we will set aside half of the RAM, being that we will only use an eight by eight LED Matrix. The keyscan will not be used as well. The working principle is quite simple:

1) start condition and send an address with a final bit for r/w
2) wait for the acknowledge (ack) from the slave
3) send data
4) wait for the acknowledge (ack) from the slave
5) stop condition

The address is 0x70 for writing or 0x71 for reading. As for data we start to have more options, for example in 0xEX, where E defines the command for dimming control and X represents the duty cicle. The 0x0X data, has a 0 meaning that we are now pointing to one the sixteen rows of the HT16K33 of one by eight. Given that we only have a 8x8

Material and guidance provided by the Faculty of Engineering from the University of Porto

[1]Pedro Silva is with the Faculty of Sciences and Faculty of Engineering from the University of Porto `up201405288@fc.up.pt`

[2]Pedro Almeida is with the Faculty of Engineering from the University of Porto `ee12052@fe.up.pt`

matrix, we can see that we have a pattern of USE—UNUSED for the rows on the RAM. If we use the 0x0X, the next byte we send will set the LEDS that are lit in row X, and if we keep sending data, it will set the LEDS in the next row, and the one after that and so on. Keep in mind that using this method of sending data after the 0x0X, half of the bytes sent will not be used, since they do not connect to any LEDS.

B. SPI & PmodALS

Serial peripheral interface is an interface that enables the serial exchange of data between a master and one or multiple slaves. The term was originally coined by Motorola. SPI is synchronous and operates in full duplex mode. The data from the master or the slave is synchronized on the rising or falling clock edge and both master and slave can transmit data at the same time. The SPI is most often employed in systems for communication between the central control unit and peripheral devices (which is our case). The connection can be either 3-wire or 4-wire and this interface supports much higher clock frequencies compared to I2C.

4-wire SPI devices have four signals:

1) Clock (SPI CLK)
2) Chip select (CS)
3) Master out, slave in (MOSI)
4) Master in, slave out (MISO)

3-wire SPI devices either have MOSI or MISO. The device that generates the clock signal is called the master. The chip select signal from the master is used to select the slave and it is normally an active low signal. MOSI and MISO are the data lines. While MOSI transmits data from the master to the slave, MISO transmits data from the slave to the master.

The working principle of this interface is also quite simple. To begin SPI communication, the master must send the clock signal and select the slave by enabling the CS signal. After this signal is sent, the data lines begin to send their respective bits simultaneously. The serial clock edge synchronizes the shifting and sampling of the data.

More specifically the PmodALS is a read-only module, so the only wires in the SPI protocol that are required are the Chip Select, Master-In-Slave-Out, and Serial Clock. The PmodALS reports to the host board when the analog to digital converter is placed in normal mode by bringing the CS pin low, and delivers a single reading in 16 SCLK clock cycles. The PmodALS requires the frequency of the SCLK to be between 1 MHz and 4 MHz. The bits of information, placed on the falling edge of the SCLK and valid on the subsequent rising edge of SCLK, consist of three leading zeroes, the eight bits of information with the MSB first, and four trailing zeroes. Any external power applied to the PmodALS must be within 2.7V and 5.25V.

C. ATmega328P and It's Memories

The ATmega328P is a single-chip, 8-bit AVR RISC-based microcontroller created by Atmel. It combines 32 kB ISP flash memory, 2 kB of SRAM, 23 general purpose I/O lines and 32 general purpose working registers.

The flash memory is where the code is stored in the form of a .hex files. SRAM is where the program creates and manipulates variables when it runs. The EEPROM is a memory space that programmers can use to store long-term information. Both flash and EEPROM are invariable memories, this meaning that the number of writes are quite limited. EEPROM is limited to 100.000 writes while the flash is only limited to 10.000 writes, and this must be taken into account when making the memory tests. Both of them store their value when the microcontroller is turned off, something that does not happen to the SRAM, a variable memory, which in turn is not limited by the number of writes.

There are 2 types of memory tests that will be done in this work. The first one, and the one that will be used on the SRAM, on the EEPROM and on the HT16K33 is the March X algorithm. The HT16K33 is not a memory of the ATmega328P, but since it is a memory that we will be using, it will also be tested. The March X is quite straightforward algorithm:

1) set all bits to zero
2) from the first bit till the last, read if it is zero and set it to one.
3) from the last bit till the first, read if it is one and set it to zero.
4) read if all bits are zero.

And it is best suited for the variable memory since it involves extensive writing. We applied it to the EPPROM anyway, for educational purposes, thought we advice not to do it, since it is a standard case of ”the medicine being worse then the cure”. As for the Flash test, we will compare the hash of the .hex file that is sent to the microcontroller with the hash of the program on the Flash. The hash is calculated making an XOR operation of all the words on the .hex and on the words on the Flash.

III. PROCEDURE

Our focus will now turn to how we did our Environment Light Intensity Regulator, and in this section we will discuss our Montage, our SPI & I2C setup code, LED intensity control and ISP of the ATmega328P.

A. Assemblies

The montage used for both the Lab prototype and for the Industrial prototype code is the same if we do not plan on making use of a single ATmega328P, programmed using ISP. The first montage can be seen on figure 1, the latter can be seen in figure 2.

As stated above the Adafruit Mini 8x8 LED Backpack, composed by both the LED matrix and the HT16K33, communicates with the Arduino/ATmega328P via I2C. Since it needs 5V to work, the pinout in figure 1 is quite straight-forward and without the need for extra components that are not wires: We connect the C to the SCL (PC5), D to the SDA (PC4), $+$ to 5V and the $-$ to GND. For the Pmod ALS from Digilent, and since this sensor can only communicate with the Master and not vice-versa, we will only need to connect the MISO pin (PB4) to the SDO, the SCK pin (PB5) to the

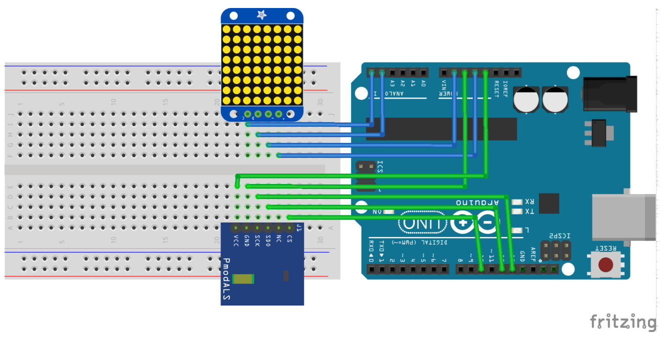


Fig. 1. Montage of the circuit used to make the laboratory prototype and the C code for the industrial prototype, where both the LED matrix and the Ambient Light Sensor are displayed with the Arduino UNO, as well as the connections that exist between them.

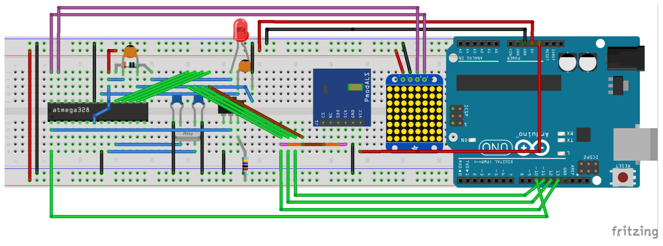


Fig. 2. Montage of the industrial prototype with the circuit of a standalone ATmega328P, and it's connections with the LED matrix, the Pmod ALS and the Arduino.

SCK and SS pin (PB2) to the CS. Keep in mind that the Pmod ALS should be supplied with 3.3V. All this is visible on figure 1.

As for the use of a single ATmega328P we do the typical standalone montage, but now we connect almost all of the SPI ports of the ATmega to the ones on the Arduino in order to use the latter to program the first via SPI. The only exception if the SS pin from the Arduino, that is connected to the first pin of the microcontroller. Both the LED matrix and the Ambient Light Sensor are connected to the ATmega328P.

B. I2C, SPI & setup code

The code that enables the I2C to be used can be divided in three different functions for briefness: one for starting, one for writing and one for stopping. This, because our aim here is to mainly write to the LED matrix and not read from it. Our code is heavily based on [1][2].

```
unsigned char i2c_start(unsigned char address)
{
    static uint8_t twst;
    /*We send the start*/
    TWCR=(1<<TWINT)|(1<<TWSTA)|(1<<TWEN);

    /*We wait for the transmission to complete*/
    while(!(TWCR & (1<<TWINT)))
    {
        ;
    }
    /*We want to check TWSR while masking the prescaler*/
    twst = TW_STATUS & 0xF8;
    if((twst!=TW_START)&&(twst!=TW_REP_START))
    {
        return 1;
    }
    /*We send the address*/
    TWDR = address;
    TWCR = (1<<TWINT) | (1<<TWEN);

    while (!(TWCR & (1<<TWINT)))
    {
        ;
    }

    twst = TW_STATUS & 0xF8;
    if((twst!=TW_MT_SLA_ACK)&&(twst!=TW_MR_SLA_ACK))
    {
            return 1;
    }

    return 0;
}
```

In order to start the TWI (2-wire Serial Interface) which is compatible with Phillips' IC protocol, he have to set the bits `TWINT`, `TWSTA` and `TWEN` of the TWI control register (`TWCR`) to 1. `TWEN` must be set to enable TWI. `TWSTA` must be written to one to transmit a start condition and `TWINT` must also be written to clear the `TWINT` flag. Once the start condition has been transmitted, the `TWINT` flag is set by the hardware, condition that we wait for on the `while`. The status on the TWI status register (`TWSR`) should be 0x08, which we try to compare on the following `if`. The registers have different names in the code from the ones being used here, because we are using the `<compat/twi.h>` library in the code. We save the address of HT16K33 (0x70) with an ending 0 for writing on the TWI data register (`TWDR`). We set `TWINT` and `TWEN` to 1 on `TWCR` and once again we wait for the flag to change, meaning that we have sent the address. We save the `TWSR` on `twst` and we have a final `if` to check if we don't have the status of an acknowledge of on either master receiver or master transmitter mode, something that we should have.

```
unsigned char i2c_write(unsigned char data)
{
    static uint8_t twst;
    TWDR = data;
    TWCR = (1<<TWINT) | (1<<TWEN);

    while (!(TWCR & (1<<TWINT)))
    {
        ; /*We wait for the transmission to complete*/
    }
    twst = TW_STATUS & 0xF8;
    if ( twst != TW_MT_DATA_ACK)
    {
        return 1;
    }

    return 0;

}

void i2c_stop(void)
{
    TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);

    while (TWCR & (1<<TWSTO))
    {
        ; /*waiting for the stop bit clear*/
    }
}
```

The `i2c_write()` function in similar to the last part of function `i2c_start()`, but this time we a have a token `if` that returns 1 if we don't have a status expected for a master transmitter acknowledge from the slave, 0x28. `i2c_stop()` simply sets the `TWINT`, `TWEN` and `TWSTO` to 1 and we wait for the `TWSTO` to be cleared, meaning that the stop condition as been sent. If we have an hypothetical function `sending()`, then to setup code for the LEDs should look something like this:

```
void setup(void)
{
    uint8_t LED = 0x70U;
    sending(0x21U,LED);
    sending(0xA0U,LED);
    sending(0xEFU,LED);
    sending(0x82U,LED);
    i2c_start(LED);
    i2c_write(0x00U);
    i2c_write(0xFFU);
    for (e = 0; e<15 ; e++)
    {
        i2c_write(0x00U);
        i2c_write(0xFFU);
    }
    i2c_stop();
    sending(0x81U,LED);
}

void sending(uint8_t message, uint8_t address)
{
    i2c_start(address);
    i2c_write(message);
    i2c_stop();
}
```

The five `sending()`, by order: internal system clock is enabled, the INT/ROW output pin is set to ROW driver output, the dimming is set to the biggest duty-cycle, we set the display to off and we set the display to on. The `for` cycle, as explained before, sets the rows that are being used to fully lit, 0xFF, and the rows that are not used to 0x00.

```
    /*SETUP DE SPI*/
    DDRB |= (1<<CS)|(1<<MOSI)|(1<<SCK);
    DDRB &= ~(1<<MISO);
    PORTB |= (1<<CS);

    DDRB |= (1<<0);

    SPCR |= (1<<MSTR);

    SPCR |= (1<<SPR0);
    SPCR &= ~(1<<SPR1);

    SPCR &= ~(1<<CPOL);
    SPCR &= ~(1<<CPHA);

    SPCR &= ~(1<<DORD);
    SPCR |= (1<<SPE);
    /*END OF SPI SETUP*/
```

This code refers to the initialization and setup of the ATmega328P for SPI. First we start defining CS, MOSI(which we're not using) and SCK pins as outputs and MSIO as an input and pull the CS pin to high, later we will do the opposite (pull the pin to low) in order to start the communication. The MSTR bit selects master SPI mode when written to one so we did that accordingly. The next two bits we manipulate are SPR0 and SPR1. These two bits control the SCK rate of the device configured as a master. A different combination of these bits provides different frequencies. We opted for 1MHz so we needed to divide for 16 and the combination that provides that is to set SPR0 to 1 and SPR1 to 0. Next up we manipulated the Clock Polarity and Clock Phase bits. CPHA determines if data is sampled on the leading (first) or trailing (last) edge of the clock and CP0L determines if the data is sampled on the rising or falling edge of the clock. We chose MODE 0 which means the data is sampled on the leading and rising edge of the clock so we needed to set CPOL and CPHA both to 0. When the DORD bit is written to one, the LSB of the data word is transmitted first and when the DORD bit is written to zero, the MSB of the data word is sent first so we set it to 0. Finally there is the SPE, when written to one, the SPI is enabled. This bit must be set to enable any SPI operations.

C. *LED Intensity Control*

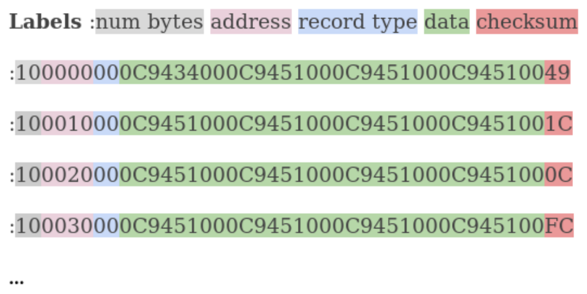
Now that the setup code is done, all that is left is creating something that reads the information from the sensor and changes the LED's brightness. The code bellow does exactly that. The missing code that is commented out includes both the communication protocol setup as well as memory tests.

```
int main(void)
{
    static uint8_t dataIn[2];
    static uint8_t lumiere;
    static uint8_t brilho=0xEFU;
    static uint8_t lumiere2;
    uint16_t e = 0;

    /*
   MEMORY TESTS & COMMUNICATION SETUP CODE
    */

    /*Environment Light Intensity Regulator*/

     while(1)
     {
      PORTB &= ~(1<<CS); /*set CS low*/

      for (e=0;e<2;e=e+1)
      {
        SPDR = 0x00; /*send trash*/
        while (!(SPSR & (1<<SPIF)))
            {
                ; /*wait for last bit sent*/
            }
        dataIn[e] = SPDR; /*save data*/
      }

      PORTB |= (1<<CS); /*set CS high*/

        /*Reconstitution of the 8-bit data*/
        lumiere=(dataIn[0]<<4)|(dataIn[1]>>4);
        /*Parameterization*/
        lumiere2=(255U-lumiere)/16U;
        brilho=0xE0U;
        brilho=brilho+lumiere2;
        i2c_start(LED);
        i2c_write(brilho);
        i2c_stop();
        _delay_ms(400);
}
```

Firstly the CS is set to low in order to make the PmodAls send us the light intensity, but since it sends the information broken between 2 bytes, we create a `for(e=0;e<2;e++)`, where we set `SPDR` to 0, hence we simulate that the data to be sent is 0. We make a `while(!(SPSR &(1<<SPIF)));` where we wait for the last bit of a given byte to be sent, and once it is we store it on an array. Once the `for` is over we set the CS to high, completing the SPI data transfer. We join the data that was separated in 2 bytes into a single `uint8_t`, and we parameterized the values of ambient light intensity, that spawn from 0 to 255, into values that go from 16 to 0. Thus, the brighter the ambient light, the weaker the intensity from the LED's. Since the last four bits of `brilho` represent one of the sixteen values of duty-cycle available, and the first four the command to control the dimming, we simply add them and send `brilho` to the LED matrix. The delay is just a value that we thought appropriate for our project, though we have shown that we can go as low as 5 ms or lower.

D. Memory Tests

As stated above, there will be 3 memory tests, one for the EEPROM, SRAM and HT16K33 that apply the March X algorithm, also explained above, with an extra intrabyte test to the SRAM, since it's unit of memory is the byte. As for the flash, it will have it's contents compered with what we expect to have in there.



```
Labels :num bytes address record type data checksum

:100000000C9434000C9451000C9451000C94510049

:100010000C9451000C9451000C9451000C9451001C

:100020000C9451000C9451000C9451000C9451000C

:100030000C9451000C9451000C9451000C945100FC

...
```

Fig. 3. Format of the Intel HEX protocol according to [3], with HEX data from one of our .hex files.

Flash We have a .hex file that is sent to the flash and we have to make sure that what is inside the flash is equal to said .hex. In figure 3 we have the structure of the .hex file, and from it we calculate an hash that goes through every word on the .hex and does an XOR operation: `hash = word ^ hash ;`, with `hash` starting at 0. We also record the number of bytes that were used to calculate the hash. All of this is calculated externally, not on the microcontroller, and the final value of the `hash` is recorded on the EEPROM on the bytes 0x45 and 0x46.

On the microcontroller, the function `pgm_read_word_near()` from the `avr/pgmspace.h` is used to go through every word on the Flash memory. The bytes analyzed on the Flash start at 0 and end at the number of bytes used to calculate the correct `hash` externally. Now we use the above mentioned function to give us the words needed to make the same XOR operation as before.The hash of the flash is thus calculated, and will ultimately be compared with the correct hash. If both are equal, then we pass one of the memory tests and go on to the next one.

SRAM The March X algorithm was used to test this memory, but since the March X is destructive to the memory, we need to divide it, in our case a division in 8 sections was selected. The first is the buffer, where we store the values of the next 7 sections, thus the buffer, itself, when tested has it's contents destroyed. For a test of this kind to work, we need to define all the variables to be used as `register`'s, by doing so we erase the need for allocation in the stack. There is an intrabyte test, where the values: 0x55, 0xAA, 0x33, 0xCC, 0xF0 and 0x0F, are written in every byte and a `if`, tests if they have indeed the correct values. The test functions for this part are heavily based on [4] and a code that is made available online by Atmel, meaning that we tried our best to not change it, and the only changes made were so that the test would work as expected in our code.

EEPROM The functions `eeprom_write_byte()` and `eeprom_read_byte()` from the `avr/eeprom.h` library were used to implement the March X algorithm. Every time we use the read function, if an unexpected value is read a variable that stores the number of errors goes up by one. Before we start the March algorithm, we store the value of the hash that is needed to make the flash test on a regular variable, in other words, we store it in the SRAM.

HT16K33 During this memory test we will rely on the TWI communication protocol. This time, we want to write and read values of one of the eight rows of LEDS that are used. The test is based on the March X:

1) write all the bits on the HT16K33 to 0
2) `reading()` and `writing()` are used to see if the first bit on the first row is 0 and set it to 1.
3) we do 2) to the second bit, and the third, and the fourth... till we get to the last one.
4) we change to the next row and do 2) and 3), and then to the row after that... till the last one.
5) `reading()` and `writing()` are used to do the same as before, but now on the opposite direction and we check if the bits are 1 and set them to 0.
6) read if all the bits are 0

```
uint8_t reading (uint8_t row, uint8_t value)
{
  static uint8_t ca;
  i2c_start(LED);
  i2c_write(row);
  i2c_rep_start(LED+1U);
  ca=i2c_readNak();
  i2c_stop();
  if(ca==value)
  {
    return 0U;
  }
  else
  {
    return 1U;
  }
}

void writing (uint8_t row, uint8_t value)
{
  i2c_start(LED);
  i2c_write(row);
  i2c_write(value);
  i2c_stop();
}
```

Above we have the code for the `writing()` and `reading()` functions. The only thing new on this functions are the repeated start, that is used to send the address of the HT16K33 with a reading bit, followed by a function that returns the data from a given row and send an acknowledge.

E. Autodesk Eagle

The elaborated schematic can be found within the appendix section and is coherent with everything described above.

The two-layer PCB layout was designed with the goal of occupying the least possible amount of area. It is also important to point that once the PCB is assembled there is no way to re-program the MCU due to the lack of external connections which was one of the goals of the project. The PCB has a built-in SMD photo-transistor that captures the light. It also has a surface mount ATmega328P and a surface mount crystal in order to save space and to achieve less and easier connections.

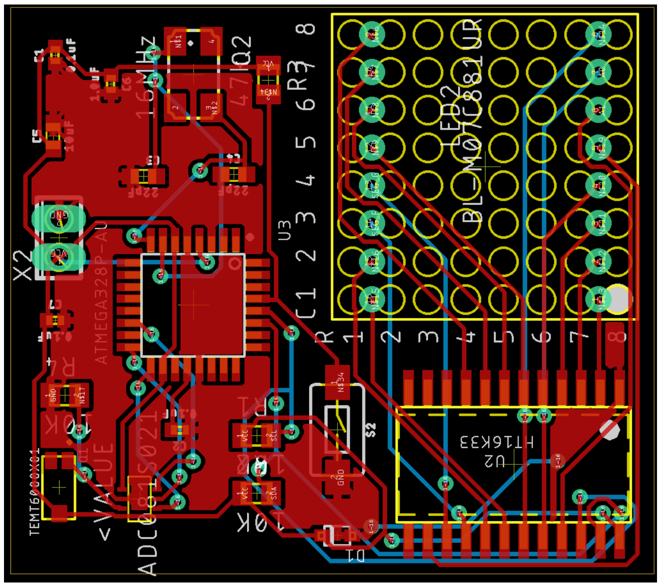


Fig. 4. Our final printed circuit board

IV. DISCUSSION

A. Communication Protocol Analysis

We have used an oscilloscope to analyze the information that is sent to the microcontroller from the PmodALS, vias SPI, and what the microcontroller sends to LED backpack via I2C. The code from which both snapshots of the oscilloscope were taken comes from the code in `int main()`, in particular the one that is bellow the Environment Light Intensity Regulator, that is commented out.

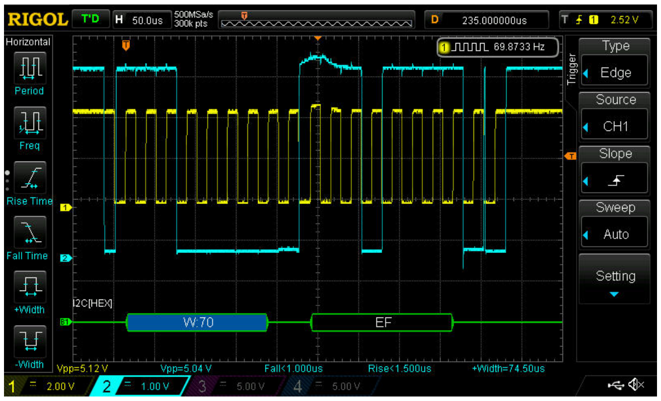


Fig. 5. Dimming command being sent to the LED backpack, via I2C with the clock and the data lines.

From figure 4 we can see three things that were expected:

1) We have two bytes being sent, one with the correct address (0x07) for writing and one with the expected data for dimming control (0xEX). The last four bits being sent are 0xF, meaning that we are setting the our LEDs to full brightness.
2) There are two ground's present that are not fully equal: one from the microcontroller, the lower of the two, and one from the slave.
3) We have a small peak close to the end, meaning that the slave let go from the line at the end of the ACK and the microcontroller was not quick enough to control the line in time and set it to ground creating a peak that goes all the way to the VCC.

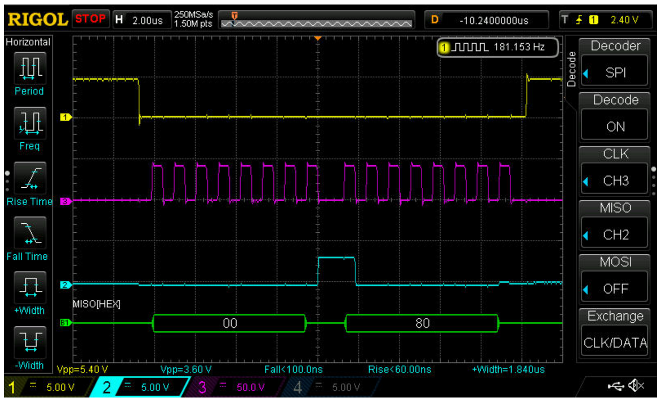


Fig. 6. Ambient light intensity value being sent over 2 bytes, via SPI, with the MISO, the Clock and the SS lines

Figures 5 shows the regular plot for the SPI protocol. We see that the communication starts by setting the SS line to ground, and with it we have the start of the Clock and of the MISO line. We can also see that we are sending two bytes of information and the first and the last four bits are zero as stated by the datasheet[5].

B. EEPROM test solution

The March X algorithm is not suited for the EEPROM, since we are talking about an invariable memory. A solution for this would be to write the appropriate value of the hash on the chosen bytes of the EEPROM (something that we are already doing) and to make sure that all the other bytes are zero. Now the test of the EEPROM would be a lot like the one of the flash, we make an XOR of all the bytes to make an hash and compare it the XOR of the bytes sent to 0x45 and 0x46. This way we reduce the excessive number of writes in our EEPROM test.

C. HT16K33 test is not a March X

As the title says we did not really implement a March X algorithm even though it sure looks like it during the test. As stated before for a given row we set the LEDs that are on by sending 8 bits, where 1 is lit and 0 is unlit, e.g. 0xFF means that all the LEDs are lit. This means that if we want to light the first LED we send 0x01, if we want to light the second LED without closing the light of the first, we need to send 0x03. This means that the first LED of every row, will be lit up 8 times and will see it's light read 8 times. And this will happen on the way back. On the March X algorithm each bit is only tested and written once both times.

D. Misra-C 2004 compliance

We tried to follow the Misra-C guidelines in order to protect our code as well as some defensive programming tricks that were taught in class. We were cable of enforcing many of the mandatory rules and some of the advisory, e.g.:

- 5.1 "Identifiers shall not rely on more than 31 character"
- 5.7 "No identifier name should be reused"
- 8.7 "Objects shall be defined at block scope if they are only accessed from within a single function."

etc

But more important then showing what we did, is to tell what we did not as we are required to do so when following Misra-C:

- 3.6 "All libraries used in production code shall be written to comply with the provisions of this document" - this was already mentioned before, we tried not to change some of the code that controlled I2C communication and the SRAM, in both cases we tried to follow as closely as possible what was said by Atmel.
- 10.1-5 We tried to follow the Arithmetic type conversions, but we recognize that we might have some errors.
- 14.5 "The continue statement shall not be used." - we have a `continue` on our I2C code.

For disclosure we might also have failed to comply with some other rules.

E. Work Planning and Division

Pedro Silva did the Lab Prototype, the I2C code in C and the memory tests and the Industrial Prototype using the ATmega328P with ISP. Pedro Almeida did the SPI code in C and the schematic and the PCB. We worked together to build the Environment Light Intensity Regulator code for the Industrial Prototype, and both tried to make our code Misra-C compliant. We tried to follow the initial GANTT, but were not able. There is a real GANTT chart, that lays out our work load in the appendix. All together we spent 84 hours which in billable ours would account for 3360 in expenses, this without tacking into account the time spent doing this report and the final tweaks to the PCB. The hardware expenses don't come near the billable ours so we don't consider them relevant.

V. CONCLUSION

We were fruitful in making an Environment Light Intensity Regulator using Misra-C, two different communication protocols, four memory tests (Flash, SRAM, EERPOM, HT16K33) and on doing an Industrial Prototype, programming the ATmega328P in circuit via SPI, using an Arduino UNO. Our major flaw came in our difficulty to plan our work, in our aggressive memory test to the EEPROM, for which we found a solution, and in some rules that we were not capable to follow of Misra.

REFERENCES

[1] AVR-GCC Source Examples: http://homepage.hispeed.ch/peterfleury/avr-software.html?fbclid=IwAR19aLNgDJ8xrqxXsCwFIsa05KUyX-1ue1m0OiZoWhzkyQ4lB5RWM8LNpYc

[2] ATmega328P datasheet: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

[3] Intel Hex: https://en.wikipedia.org/wiki/Intel_HEX

[4] Memory Tests: http://www.atmel.com/images/doc42008.pdf

[5] PmodALS datasheet: https://reference.digilentinc.com/reference/pmod/pmodals/reference-manual

[6] HT16K33 datasheet: https://cdn-shop.adafruit.com/datasheets/ht16K33v110.pdf

VI. APPENDIX