Crystal Clear Electronics

The development of the Crystal Clear Electronics curriculum was supported by the European Commission in the framework of the Erasmus + programme in connection with the “Developing an innovative electronics curriculum for school education” project under “2018-1-HU01-KA201-047718” project number.

Erasmus+

The project was implemented by an international partnership of the following 5 institutions:

- Xtalin Engineering Ltd. – Budapest
- ELTE Bolyai János Practice Primary and Secondary Grammar School – Szombathely
- Bolyai Farkas High School – Târgu Mureș
- Selye János High School – Komárno
- Pro Ratio Foundation working in cooperation with Madách Imre High School – Šamorín

XTALIN

Contact http://crystalclearelectronics.eu/en/
info@kristalytisztaelektronika.hu

19 - EEPROM, Non-Volatile Memory

Written by Szabolcs Veréb

English translation by Xtalin Engineering Ltd.

Revised by Ádám Szabó, Máté Trádler

Types of Non-Volatile Memory

In the previous chapters you could already read about some of the volatile, and non-volatile memories found in microcontrollers. An example for volatile memories was RAM (Random Access Memory), which has very fast access speeds and as such perfect for storing variables, and results of operations.

However, when the supply voltage is turned off the RAM loses all information stored inside, so if we were to store our code in RAM, we would have to reprogram the microcontroller after each restart. This is unfeasible of course, imagine how absurd it would be if every time after turning the ignition in our car we had to call someone from the manufacturer to reprogram all microcontrollers inside it.

So, there is an obvious need for memories which retain their content, such as the program code, even after the device is powered off. For this purpose, ROM (Read Only Memory) was first developed. As you can guess from its name, this memory type was only readable, it was written once during manufacturing with the program code and constants, and you had no way to modify the contents afterwards.

You can obviously feel that that was a huge restriction on the users' side, since the memory chip was only usable for one task. PROMs (Programmable ROM) gave a little more leeway, as they were programmable by the user to suit the task at hand, albeit only once. It was an OTP (One Time Programmable) memory, so if you made a mistake, the chip had to be discarded.

Imagine if you have written `a-b` into your program instead of `a+b` by giving the wrong instruction to the processor, you had to throw away the whole chip. Or, if you wanted to add some functionality later, you also had to buy a new chip. This is like if we had to buy a new HDD to install some new software on our PC.

PROM was followed by EPROM (Erasable PROM) where you had the ability to erase the chip's contents by UV light. This gave users a much greater freedom, but to erase the chip you needed some special equipment (UV light source) which was still a restriction. Finally, EEPROMs (Electronically Erasable PROM) brought the real breakthrough, as they could be programmed and erased by electricity. Its name still has the "read-only" part in it, but today this only reflects on its origin, as we can read, write, and erase it just fine.

Writing and erasing

Writing is a process where we set the groups of bytes in memory to an arbitrary value. Erasing is a special write, where the bytes are filled with "1"-s, which is 0b11111111 or 0xFF.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

With EEPROM we have a memory that we can use to store data or program code that we will need after reset. Unfortunately, this property has a negative impact on performance, the speed of EEPROMs is much lower than RAMs. In an ATmega16A, reading and writing RAM takes only two clock cycles, which is 0.25 µs with an 8 MHz clock, whereas doing the same with an EEPROM takes 8.5 ms, which is 34000 times slower. You can imagine this as short-term and long-term memory in humans: we can remember things fast, but to retain them for a long time we need a longer time to process it.

FLASH AND EEPROM

Another non-volatile memory type is called Flash memory, which is an improvement over the EEPROM. It has an increased memory capacity while having the same physical size, so it has an increased data-density. This is achieved by cutting back on the internal eraser, writer, and reader circuits for every byte, and grouping them into blocks instead.

This way a single byte can be deleted only if the whole block is deleted too, so to delete one byte you have to copy the whole block to somewhere else (usually RAM), erase it, and then copy everything back leaving the deleted byte out. This is obviously slower than deleting one byte on an EEPROM, but when reading and writing we have access to a whole block instead of a byte, so in that case it is faster than an EEPROM.

You can probably see from this that Flash memory was not designed to access or clear single bytes, but to access big chunks of data, so it is perfect to store the whole program code during programming the microcontroller, and when starting it up, copy everything to the much faster RAM. EEPROM on the other hand makes it easy to access a single value, making it perfect for storing parameters, calibration constants needed by our program.

Both Flash memories and EEPROMs have a finite lifetime unlike RAM, which is usually given by the guaranteed number of writes and erases. This write-erase cycle count is usually higher in EEPROMs compared to Flash memory. In the ATmega16A the Flash can withstand 10 000 cycles, while the EEPROM guaranteed 100 000 cycles. This means that we can modify each bit of the memory region that many times.

USE CASES OF EEPROM

During certain task we need a parameter in memory that retains its value after turning off the controller. If we want to able to modify this value at runtime as well that it is not enough to store it as a constant in program code. A constant does retain its value, it cannot be change. A variable can be changed, but it doesn't retain its value after reset.

A great solution would be if we could modify the code itself at runtime, and thus the value of the constants inside it. This, however, causes some problems as the deterministic behaviour of the program could be compromised, we could accidentally modify something that could change the behaviour of our program.

We could define a region in memory instead that is exclusively used to store parameters. This can be done in two ways: one is doing so in hardware and the other one is doing so in software. The former means that some part of the Flash memory is accessible at runtime, but the rest of it is not. It can also happen

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

that the microcontroller allows access to the whole Flash region, in this case it is the job of the software developer to separate a region in flash to be used.

In the ATmega16A the latter is true, the whole Flash memory can be modified at runtime, and it is extended with a 512-byte long EEPROM. If the microcontroller has a dedicated EEPROM, such as the ATmega16A, it is usually better to use that instead of the Flash, as its lifetime is an order of magnitude longer.

Application Areas of EEPROM

EEROM is usually used to store configuration parameters that are required for correct operation (calibration), or the end-user can modify freely, maybe it records the activity of the device (logging). For example, a remote control of an A/C unit can store the last temperature set in EEPROM, which would be frustrating to set again every time you turn on the A/C.

Or, if we are writing a software where the speed setting of the serial port can be changed, and we can't or don't want to change it at every start-up. In this case we can simply store the setting in EEPROM. In industrial applications where it is important to record any abnormal behaviour the device can store an error code in EEPROM, so the reason of failure can be identified when servicing, similarly to the black box of an airplane.

EEPROMs can also be used to retain calibration information of sensors. A good example would be a joystick, or steering wheel controller connected to the PC, because when installing it the first time we have to calibrate the maximum range in each direction, and at later the device remembers them until recalibrated with new values.

In a weather station it could be important to save every day the lowest and highest temperature of the day, or the highest wind speed, and we don't want to lose the data if the power goes out. We can store user settings, such as the parameters of the motor in a motor controller: coin resistance, inductance, maximum current, speed, power, control parameters.

This way we don't have to input these values over and over again after each start-up, but when connecting the controller to another motor, we can modify the parameters quickly even with the values measured with our own motor controller. An access control system could store the identifier in the EEPROM of the user's device, and the wall unit can store the entries made by each user, and the date of the latest entry, maybe even the number of failed authentication.

Lifespan of the Memory

Hopefully I've managed to illustrate how many things an EEPROM can be used for, the limit is really only our imagination. But when using an EEPROM it is important to keep some stuff in mind. Most importantly our memory has a finite erase-write cycles, so if we want to get the most out of it, make sure to only write if it is necessary. What constitutes as necessary depends on the memory and the application. With the EEPROM in the ATmega16A we have a guaranteed 100 000 cycles. If we would store the value of a variable every second, we would use up our memory in little more than a day, as there are 86 400 seconds in a day. If we only write a value once every 315 seconds, our EEPROM is guaranteed to last more than a year!

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

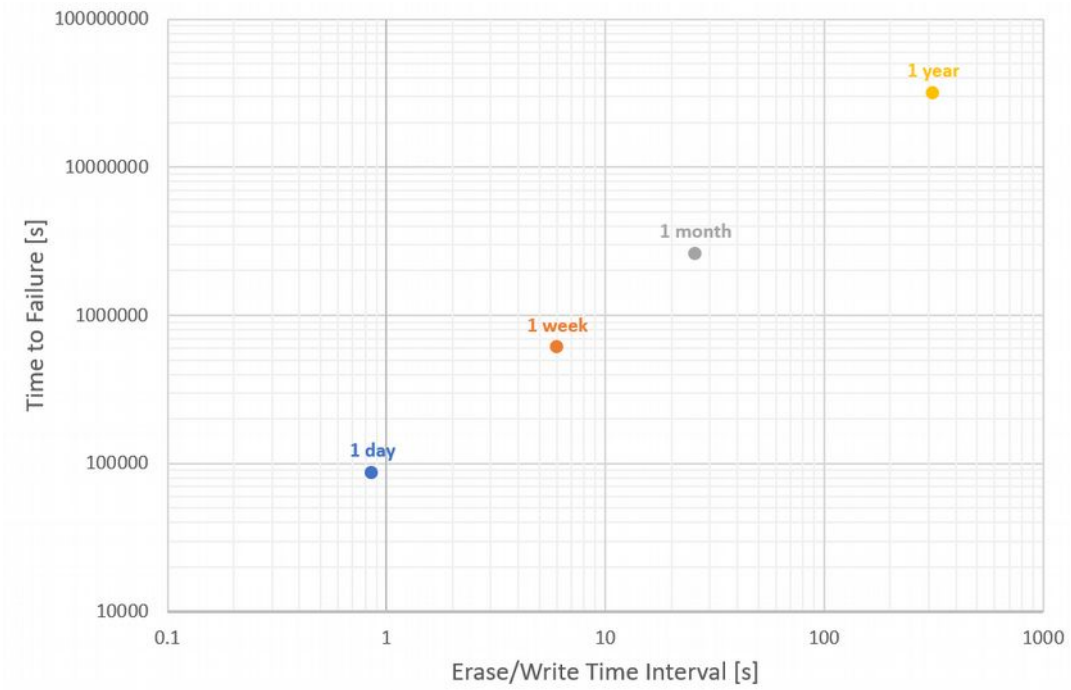

Figure 1 – Failure of ATmega16A EEPROM as a function of writing frequency

METHODS TO INCREASE LIFESPAN

So, we already know that EEPROMs are not ideal for storing fast-changing values, and it doesn't make sense to erase and write them frequently since their main purpose is retaining values when the device is powered off. To increase the lifespan of the memory we have several techniques. The most obvious is avoiding unnecessary writes by only writing a value if it differs from the currently stored value.

This means that before writing anything we read the contents of the memory, compare it to the new value, and we only write if they don't match. This method increases write time in cases where it is actually required but decreases it where no real write operation is necessary. Therefore, this is not the method to use where the values are guaranteed to be different every time (for example counting).

Another technique is if we know when our device will switch off, we can write every important value that we were storing in RAM to EEPROM before switching off, and on the next power-up we load them back to RAM. We can build a circuit that detects power supply failure, but we have to be careful that during the whole EEPROM write process the supply voltage of the microcontroller and the EEPROM must stay above the level specified in the datasheet. If the supply voltage is lower than the limit, the behaviour of the microcontroller is undefined and we may write incorrect data to the EEPROM, or the whole write process may fail.

A third option can be used if we only need a small portion of the available memory, which is usually true. Suppose we have an EEPROM with 512 bytes, but we only need 30 bytes to store our data. The EEPROM guarantees 100 000 erase-write cycles, after which we would have 30 bytes nearing the end of its lifespan, and 482 bytes that have never been used and still has at least 100 000 cycles left. It would be foolish not to use this memory and replace the controller.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

In practice we try to evenly distribute the wear of bytes across the whole area of the memory, so at every write we are using a different address. We segment the memory to 30-byte blocks (17 of them in total), and we always write to the next block, looping back to the beginning when reaching the end. We are not done yet, because we need to know when turning on, which block contains the most recent data, and what address we need to write next.

We could store this in a pointer stored in EEPROM. The naïve solution is to store it at a fixed address, and at power on read the address of the most recent block from there. But, this would get us nowhere, as this pointer has to be updated every time we do an EEPROM write, so that memory area would wear out after the guaranteed 100 000 writes, not giving us any benefit from moving the data around in memory.

The solution is not to use a pointer but mark the next block to be used for writing with a special pattern. Because reading the EEPROM does not cause any kind of wear, we can create an algorithm which reads the entire memory region after powering on the device looking for the pattern. If found the data should be written there next time, if not found this is probably the first time using the device, and we should use the beginning of the memory.

This special pattern can be the 0xFF byte, which signifies an empty byte in EEPROM. We only have to make sure to store data in such a way that 0xFF does not occur in it. A write process then looks like this: Fill the current block with useful data and erase the next block (fill it with 0xFF). This way at every write we are writing two blocks, so we “only” get 8.5 times as much lifetime from our 512-byte memory (if talking about 30-byte blocks), instead of 17 times as much.

If we use both the wear-levelling and the power supply detection method, we can increase the lifespan of our EEPROM considerably. Of course, if we are storing values that rarely change (1-2 times a day) there is probably no need to use these methods and complicate the code, the default guaranteed lifespan of the EEPROM is probably more than the devices lifetime already.

Theoretical example of a pedometer

The job of a pedometer is to count the steps of the person wearing it and store the value until the user resets it to 0. A pedometer also has to store the total steps counted since manufacturing. The device works from battery power and has a 256-byte EEPROM with 100 000 guaranteed write cycles. The current step count is stored as a 24-bit integer, and the total number of steps is stored as a 32-bit integer. We want to count steps during the day when the user is awake which is 14 hours. Calculate the guaranteed lifespan of the EEPROM in the simple case, then with only storing data at power off, then with added wear-levelling as well! For simplicity assume that there is at least half a second between two steps.

Solution

When pushing the reset button, we have to add the current steps to the total number of steps, and then set the current step counter to zero. The average user is interested in how many steps he or she had taken during the day, so we can assume the device will be reset once every day. This means we will write the total step count in EEPROM once a day, which is such a low frequency we can simply allocate 4 bytes to it. The lifespan of those 4 bytes is:

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

$$T = \frac{24\ [\mathrm{h}]}{1} \cdot 100\ 000 = \ 2\ 400\ 000\ [\mathrm{h}] = 274\ [\mathrm{years}]$$

The current number of steps changes a lot each day, so it makes sense to use the discussed techniques.

The simple case:

We store every change in EEPROM. This means that after every step, i.e. every 0.5 seconds we increase the value in EEPROM. The lifetime of that memory region is:

$$T = 0.5[\mathrm{s}] \bullet 100\ 000 \ = 50\ 000[\mathrm{s}] = 13.89\ [\mathrm{h}]$$

As you can see the device would not even last the first 14 hours of usage, which is at most 7 days if we assume only two hours every day. I'm not sure many people would buy it with that written on the box.

Only saving data at power off:

The pedometer senses the steps after idling and turns on, and after a long time without any steps (the user is sitting perhaps) it switches off to save battery power. Every time it turns on it reads the current steps from EEPROM and increases this counter in RAM during operation. Before turning off, it updates the value in EEPROM. Suppose that the user is health-conscious and moves for 5 minutes every 15 minutes. This results in the device turning on 56 times during the 14-hour awake period, which means 56 EEPROM writes each day.

$$T = \frac{24\ [\mathrm{h}]}{56} \bullet 100\ 000 \ = \ 42\ 857[\mathrm{h}] = 4.89\ [\mathrm{years}]$$

With this solution our device can work for almost five years, which is a lot better than 1 week.

Saving at power-off, with wear-levelling:

We are storing the step count on 3 bytes, so we can partition the remaining 252 bytes (remember we have allocated 4 bytes to the total steps) into 252/3=84 blocks. As we have discussed we are writing two blocks each time, so the lifetime should be 42 times bigger than in the previous case, which is about 205 years.

You can see from these examples that you have to be careful and think about the application when using EEPROMs if you want a good lifetime. And you can also see that with a few tricks the lifetime can be greatly increased.

USING THE EEPROM OF THE ATMEGA16A

In the following section we will show how to use the embedded EEPROM in the ATmega16A microcontroller. You can find detailed information on which registers to use, and how in order to read and write the EEPROM.

There are three registers controlling the EEPROM. The EECR (EEPROM Control Register), the EEAR (EEPROM Address Register), and the EEDR (EEPROM Data Register). The first one stores the settings for

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

the EEPROM, and the current operation (read or write). The second stores the memory address used, and the third is the data itself being written or read.

I won't go into the detailed usage of these here, because Atmel Studio provides an EEPROM library that does all the hard work for us, and we can control the EEPROM through a few simple functions. The `eeprom_read_byte()` and `eeprom_write_byte()` functions are used to read or write a single byte, but there are variants ending in `_word()` (16-bit), `_dword()` (32-bit), `_float()`, `_block()` (block of bytes) for the different data types as well. In addition, there is an `eeprom_update_byte()` (with the same variants as before) that only writes data if it differs from the stored data.

The `eeprom_update_block()` function has three arguments: the memory address of the byte array in RAM which we want to save, the starting address in EEPROM where we want to save the data, and the length of the block. The method copies the first n bytes (specified in the length parameter) of the byte array to the EEPROM region starting from the given start address, while only writing the memory if the value in the byte array is different from the value stored in EEPROM.

PRACTICAL EXAMPLE

Let's demonstrate through a simple example how to store a few configuration parameters (three in this example) using the EEPROM. The status of the three parameters will be the displayed by three LEDs and it will be adjustable by three buttons. We won't give them names, but they could be for example settings for serial communication over UART, 1: enable parity bit, 2: toggle between even and odd parity, 3: number of stop bits (one or two). Of course, you can imagine anything else behind them.

Load the *18_1_EEPROM* project into Atmel Studio (you can find the project files on the website of the curriculum), open *main.c* and look at the code. Build the circuit according the instructions given before the code, referring to the datasheet if necessary.

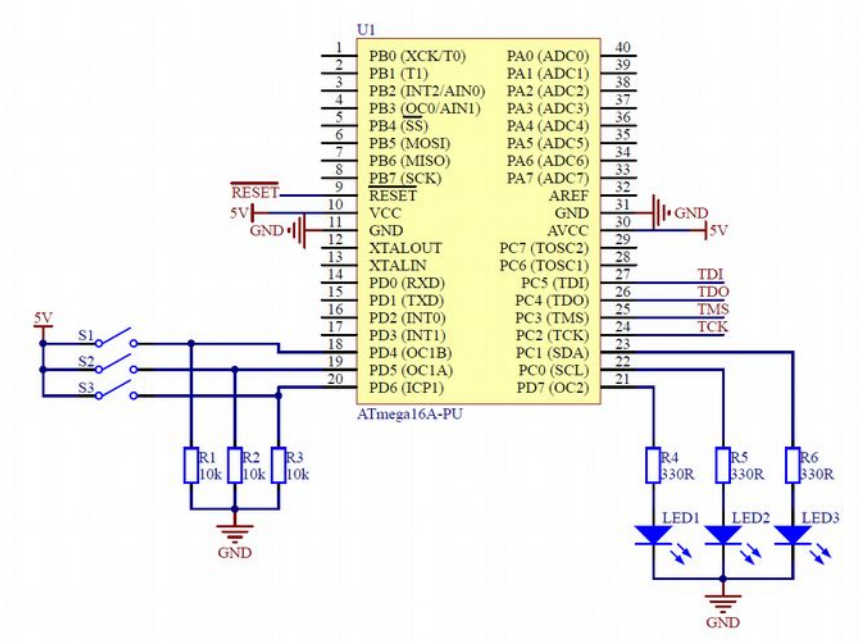

Figure 2 – Schematic for the EEPROM example code

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

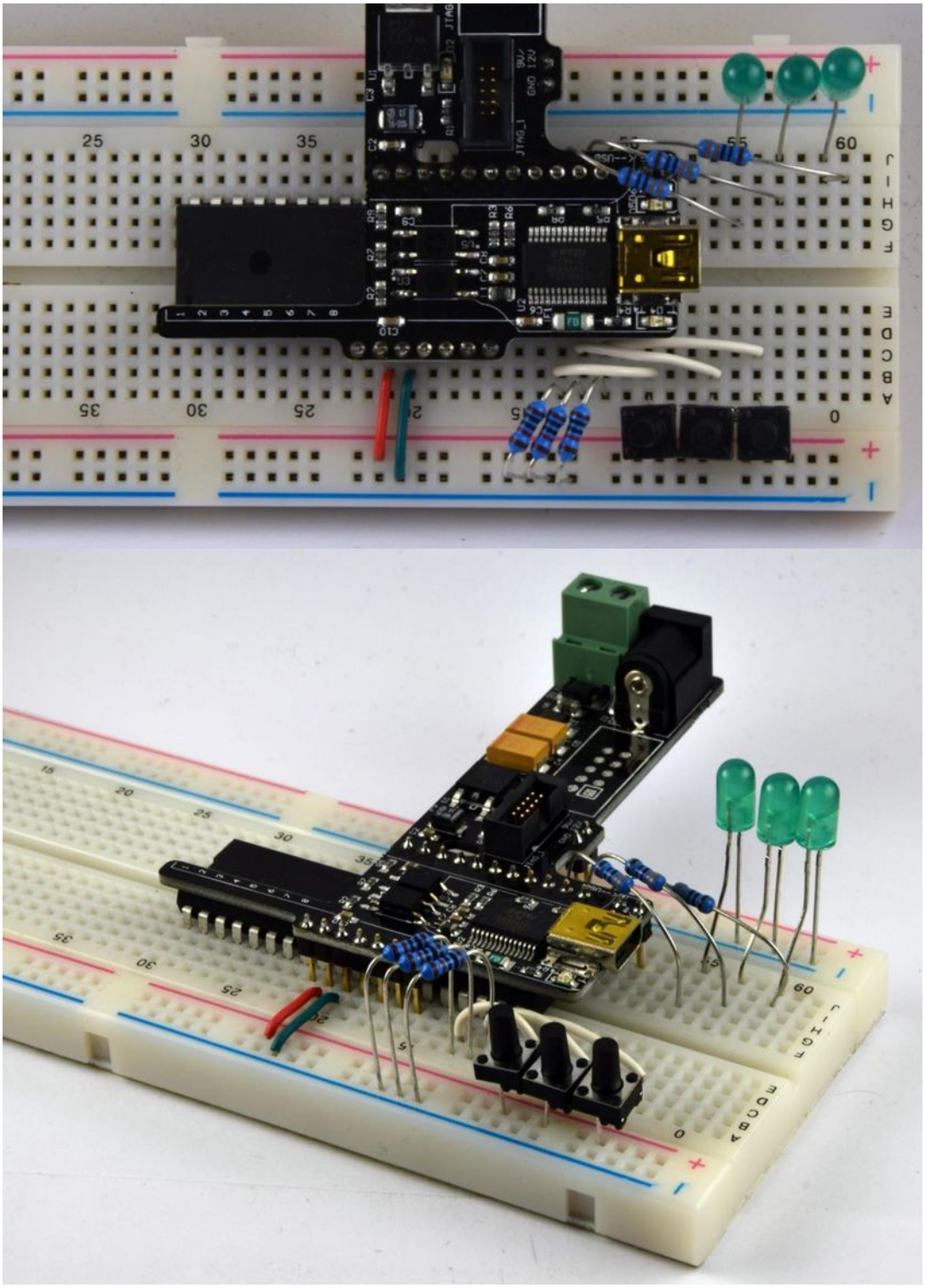

Figure 3 – Assembled example circuit for the EEPROM project

IMPORTANT: Before uploading the code to the microcontroller put a comment before every line with `eeprom_write_byte()` in it. If there is a mistake in the wiring one of the pins may float, which results in rapidly blinking LEDs, and rapid writing of the EEPROM, which must be avoided. If the LEDs are working properly, then you can remove the comments and write the EEPROM safely.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Floating

Floating means that the pin does not have anything connecting to it to either a logic high or low level (supply voltage or ground). In this case the logic state read by the microcontroller can change very fast as the electromagnetic interference from the environment causes the voltage of the pin to fluctuate.

```
/*
 * Chapter 19
 *
 * The buttons connected to PD4,5,6 pins affect the state of PD7,PC0, PC1 pins, where
 * LEDs are connected.
 * These states are stored in the EEPROM, the states can be restored after power cycle.
 */

#include "../Headers/main.h"

//EEPROM memory addresses for the states of each LED
const uint8_t* led1_ptr = 0x00;
const uint8_t* led2_ptr = 0x01;
const uint8_t* led3_ptr = 0x02;
```

The final goal is that pushing a button changes the state of LEDs, so if they are lit, switch them off and vice versa. As a first step we declare constants for memory addresses storing the state of each LED, or in other words pointers. the memory pointed by`led1_ptr` will store the status of the first LED (on or off), `led2_ptr` points to the status of the second LED, and `led3_ptr` points to the third. The EEPROM of ATmega16A is 512 bytes long, so memory addresses can be between 0 and 511 (0x000 and 0x1FF). In the example we addressed the first three memory addresses for simplicity.

```
int main(void)
{
    //Initializing PORTs
    IOInit();

    //Check stored state at led1_ptr address
    if (eeprom_read_byte(led1_ptr) > 0)
    {
        sbi(PORTC, 1);
    }

    //Check stored state at led2_ptr address
    if (eeprom_read_byte(led2_ptr) > 0)
    {
        sbi(PORTC, 0);
    }

    //Check stored state at led3_ptr address
    if (eeprom_read_byte(led3_ptr) > 0)
    {
        sbi(PORTD, 7);
    }
```

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

First the program completes the initialization of I/O with the `IOInit()` function. If the value in the EEPROM is 0, the LED must be switched off, else it must be switched on. When running for the first time, the EEPROM usually contains erased state (filled with 0xFF), so all three LEDs will be on. After initialization we read the three numbers stored in the EEPROM and set the LEDs accordingly. To read a byte, we use the `eeprom_read_byte()` function, which has a single argument that is the memory address we want to read, and returns the value of the byte. With this the initialization part of our program is done, and the next part is the infinite loop where we read the state of the buttons. The `tbi()` function returns the value of the specified input, while `sbi()` and `cbi()` are used to set the outputs.

```
    //Infinite loop
    while (1)
    {
        //Handling the first button
       //If button is pressed
        if (tbi(PIND, 4) != 0)
        {
            //Waiting for transients to decay, software debouncing
            _delay_ms(BTN_DELAY);

            //Wait for button release
            while (tbi(PIND, 4) != 0);

            //Waiting for transients to decay, software debouncing
            _delay_ms(BTN_DELAY);

            //If LED on PC1 is "ON"
            if (tbi(PINC, 1))
            {
                //New state to be stored in "OFF"
                eeprom_write_byte(led1_ptr, 0x00);
                //Turn LED off
                cbi(PORTC, 1);
            }

            //If LED on PC1 is "OFF"
            else
            {
                //New state to be stored in "ON"
                eeprom_write_byte(led1_ptr, 0xFF);
                //Turn LED on
                sbi(PORTC, 1);
            }
        }
      ...
```

The system reacts to the three buttons separately, but in a similar way. The only difference in the rest of the code compared to the section above is the EEPROM address, and the pins. We enter the first if, when the button on PD4 gets pressed (pulled up to logic high).

Debouncing

We did not implement hardware debouncing for the push buttons, but we still want to avoid the negative effects of bouncing (for example unnecessarily writing the EEPROM at every edge, wearing it out). Instead

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

we have to implement debouncing in software. One way of doing this is to measure a time after which the bouncing is sure to be stopped, and then after detecting the first edge, make the microcontroller wait for that measured amount of time before continuing. This is done by the inserted `delay_ms()` functions which wait for the amount of milliseconds passed to it as an argument.

After pressing the button, we must wait for it to be released, because that is when a complete button press is completed. Then, we read back the value of our digital output, and change it to its opposite while also writing to the EEPROM. The `eeprom_write_byte()` function takes two arguments and has no return value, the first one being the memory address where we want to write, the second being the actual value.

Homework

This example can only handle one button press at a time. Try to implement the code that is capable of detecting all three buttons being pressed at the same time. Keep in mind, that writing to the EEPROM is a relatively slow operation, and you can only access one memory address at once.

Useful tips

Until you have made sure that your circuit and your code is working as expected it is a good idea to comment out all EEPROM operations and replace them with reading and writing regular variables.

If writing to EEPROM does not need to happen immediately, or the software does not guarantee that the data to be saved will be different from the data already in the EEPROM it is good idea to use `eeprom_update_xxxx()` functions instead of `eeprom_write_xxxx()`.

Switches, opposed to push buttons, retain their state mechanically, so when using them there is no need to store the state electronically. Their drawback is, however, that we cannot modify their state electronically, for example by a message sent over a communication channel, while we can do so with values stored in EEPROM. On top of that, storing numbers with switches is cumbersome as you can only assign one bit to each switch, meaning that you need 16 switches to store a 16-bit number.

SUMMARY

In this section we have learned the basic usage of EEPROM and explored its possible application through a number of examples. You can experiment further with EEPROMs based on what you have learned here.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).