Experiment 5

Name: syed mustafa mohiuddin

Student id : 26293

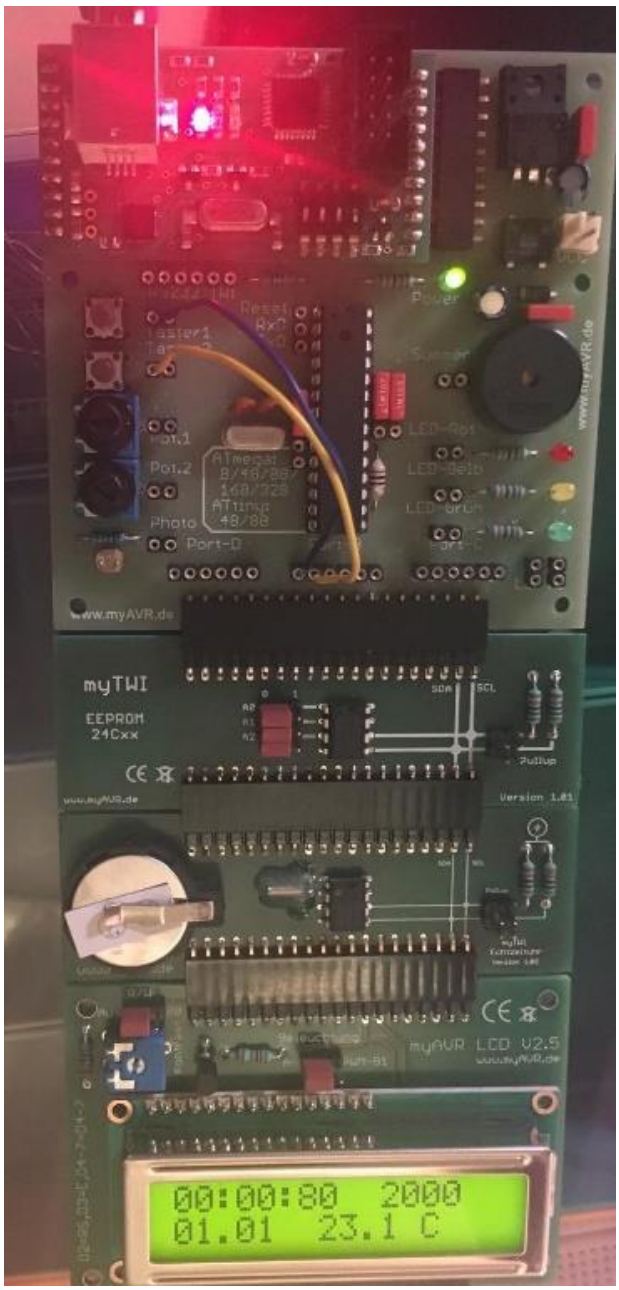
Study course: mechatronics systems engineering

Slot: pc13slot4

Submission date: 25/01/21

Task1

Circuit description



- Key 1 (button read) is connected to pin B0
- Key 2 (button wire) is connected to pin B1
- LCD display add-on
- EEPROM add-on
- RTC add-on

Program description

In this experiment, a simple temperature data logger was implemented by use of DS1307 RTC Board for time data, built-in temperature sensor for temperature data, ST 24C02 EEPROM Board for storage and LCD Board for display.

User interface for both tasks in this experiment are the same as follows:Data logger starts with the stand-by mode with current date, time and temperature value on LCD.Pressing Key 1 checks if any data was stored in memory. If there is no data, it goes back to standby mode after displaying “No Data to Show”.

If there is data, it goes to Read Mode after displaying “Going Read Mode / In memory: [count of stored temperature values].In Read Mode, the first saved value is displayed with associated time and date.

Each subsequent pressing of Key 1 displays the next saved value, count and the associated time and date. If Key 1 is pressed after the last saved value was displayed, LCD shows “No More Data”, and then shows the last saved value, count, and the associated time and date. If both keys are pressed at any point in Read Mode, user prompt “Reset Memory ? / Key1: Y. Key2: N” is displayed. Consequently, if Key 1 is pressed, 0’s are written to all bytes staring from 0th byte while on LCD “Clearing Memory.” is displayed; if Key 2 is pressed, standby mode is activated after displaying “ Read Mode Off” on the LCD.

After memory reset, " Memory Cleared" / "Exitng Read Mode" is displayed on LCD, and then standby mode is activated again. Pressing Key 2 in standby mode checks if any data was stored in memory before. If there is no data, "Ready for Data" is displayed on LCD, and the device goes to Log Mode. First data is written at the beginning of the next minute. While waiting for the next minute, the regular standby screen is active. First logging writes count, year, month, day, hour, minute, and the temperature value in memory. Each subsequent logging writes only the temperature value. During the moment of logging, LCD shows “Recording Data” / “No.: [count] [temperature value] C”. If both keys are pressed at any point in Log Mode or if count (number of written values) reaches 125, device reverts back to standby mode after displaying "Exiting Log Mode" / "In memory: [count of stored temperature values]".

Detailed description (Updated)

the first thing define the value of DS1307 to be 0XD0 and we define the temperature offset value . The main part of the temperature offset lies within the oversampling of the temperature readings obtained by the ADC sensor.

The design of our code should be such with that it puts sheer significance onto the importance of time. The productivity of the entirety of our readings were made on one free factor which is time.

On our MCU, we utilize the RTC, which is the "Real Time Clock". The amounts of time estimated by this important part are seconds, minutes, hours, days, months and years . Thus we will undoubtedly make variables for each ideal amount.

After getting done with our time based boundaries, we are just left with around 250 bytes. This is because since the all out bytes were around 256 and 6 bytes had been involved by the time based variables.

An another additional problem is that the bytes are really coordinated to accept 8 cycle readings, anyway we have expanded even our 10 bit goal to around 12 bits. For this situation a solitary byte would now possess the memory areas committed to two 8 byte values. Since (250/2=125). We can get the storage for 125, 12 bit temperature values.

Further in the code we will read directly from the EEPROM. The variables will perform bitwise multiplication shifting to obtain the desired data.

Measurement that are done, time was an important factor that was considered with each and every reading. With each measurement the time had to be constantly updated.

Later in the code, we might reach a point where time values are being directly from the EEPROM. Memory locations within EEPROM might, in that case, contain information for more than one time variable.

Starting Temperature sensor and ADC

```
// Analog Input
ADMUX = 0;//TODO
ADMUX |= (1 << REFS1) | (1 << REFS0);// 1.1V as reference
ADMUX |= (1 << MUX3);
ADCSRA = (1 << ADPS2)| (1 << ADPS1); // ADC Prescale by 64
ADCSRA |= (1 << ADSC)| (1 << ADEN); // Start first conversion (dummy read)
```

The code shows how we can enable ADC, set the prescaler, enable the temperature sensor and set the reference voltage. MUX bit enables the temperature sensor.

Sensor to sense any change in temperature, need to obtain a voltage measurement and then gauge it to some set standard in order to give a sense of the measurements and their magnitude. We set the internal reference voltage is the as standard here. The value of this standard is around 1.1V. This is also as called “Analog Reference voltage” which is the also called V(AREF) .In order to enable this we would set a value of 1 to the REFS1 and REFS0 bits in the ADMUX register, we can check this in the table.

MUX3 is a special bit in the ADMUX register, its unique feature is to initialize the temperature sensor. As soon as 1 is written to MUX3, the temperature sensor is enabled and the ADC is then used in the single conversion mode instead of the free running mode. Single conversion mode is preferred here,as it is also to prevent writing 1 to ADATE

Within the main function our variables and functions are initialized. Here We have set the temperature_count to zero

```
int main(void){
    uint16_t nowtemp;
    temperature_count= 0;

    init();      // Function to initialise I/Os
    lcd_init(); // Function to initialise LCD display
    i2c_master_init(1, 10); // Init TWI
    ds1307_rtc(1);
```

In the Initialization we had also considered the LCD, TWI and ADC. The respective code and header files were written to initialize them.

In the loop we see the following code executed.

Which is simple to understand.

The following will happen when we press buttons.

When button 1 is pressed: If we press button one the program will increment the No. variable by one and show the stored values of date, time and temperature variables.

When button 2 is pressed: If you run the program and press button 2 the program enters record mode and then the log_data() function is called.

When no buttons are pressed: If we flash the program and don't press any button. Then initially it will show the current temperature by calling the function display_standby().

pressing both buttons: together If we press both the buttons together then it will clear the memory and take you back to the current time and show the current temperature.

Displaying Standby Screen

Now we want to set The standby screen display. In order to set this we will use the display_standby() function:

```
void display_standby(uint16_t t){
    char str[16];

    // Time and Year
    snprintf(str, 16, "%02d:%02d:%02d  20%02d", hour, minute,
            second, year);

    lcd_clear();
    lcd_string(str);


    // Date and Temperature
    snprintf(str, 16, "%02d.%02d  %d.%d C", day, month, t/10, t%10);

    lcd_setcursor(0,2);
    lcd_string(str);

    return;
}
```

The temperature is read as an input into the function. Now we would need some mathematical operations to be performed for this input so that it can fulfill our needs. Furthermore the divide operation is performed followed on by the modulus operation. These procedures are done to entice separation within the input, and create a readable temperature value.

The code below shows how the functions display the case of a leap year.

```
// Check for gap year
if ((((year % 4) == 0) && ((year % 100) != 0)) || ((year % 400) == 0)){
    days[2] = 29;
} else {
    return;
}
```

OVERSAMPLING OF ADC FOR MORE ACCURATE TEMPERATURE

Oversampling of ADC for is done for more accurate temperature and higher resolution. First, the ADC is initialized, and the value from ADCW is written to the variable sum 128 times. By dividing by 32 this has now increased the precision by 2 bits. Then, by subtracting the OFFSET and converting to 0.27 C per step the ADCW value can now be read as the room temperature. Our Microcontroller Unit has a resolution of about 10 bits. In the majority of the cases, this should be fine. However whilst dealing with temperature, we require a higher accuracy therefore a higher precision. One of the methods to achieve this would be to use special signal processing techniques. So we use an even better alternative the technique of oversampling, which increases the resolution without the need of an external ADC. All of our work here simply involves taking more values and then average them. This decreases overall noise, increases the signal to noise ratio, hence adds a couple more bits of resolution. In our cases the process involves initializing ADC, and then the value from ADCW is written to the variable sum 128 times. Furthermore we incur a division by 32.

```
uint16_t adc_temperature_oversample(void){
    uint8_t i;
    uint32_t sum = 0;

    for (i = 0; i < 128; i++){
        ADCSRA |= (1 << ADSC)| (1 << ADEN); // Start ADC

        while( ADCSRA & (1 << ADSC) ) // wait for ADC complete
            ;

        sum += ADCW;
    }


    sum /= 32;

    // substract offset
    sum -= TEMPSENOR_OFFSET;

    // 0.27 deg. Celsius per step
    sum *= 27;
    sum /= 10;

    return sum;
}
```

In the sum variable we can see that the TEMPSENSOR_OFFSET is being subtracted. After the subtraction of the TEMPSENSOR_OFFSET. We require the Sum value to display the temperature. For this we would need a two step mathematical procedure. Firstly we would multiply the sum variable by 27, and then we would divide it by 10. By this two-step procedure what we achieve is that it causes the temperature displayed to be increased by 0.27 at every step. To mathematically elaborate this, the following range is an example of how the temperature would increase step by step:20.4 to 20.7 to 21.0 and it goes on.

For saving temperature and time values to EEPROM we made use of the save_value8bit() and save_value16bit(). The function save_value8bit() was used to save the time values as they only occupied the space of around 8 bits. However for the temperature values we required a 16 bit memory, so we used the save_value16bit().

To save the hour part of the address 0x02, the byte has to be multiplied bitwise by 127. 127 in binary also happens to be 11111111. Now we want to save the day part of 0x02. To do this we use bit 7 of 0x02 by shifting the whole

number to the left by 7 bits, followed on by bitwise multiplication by 1 to give only the LSB bit of the day. This efficiently also prevents overflow. We will approach all the other time variables with relevantly similar concepts and operations.

We also need to account for the stored temperature values. For this Address 0x05 can be used as the storage. Nonetheless we settled with an alternate approach by handling this role to a for loop. What this does is count the number of stored temperature values and store the count in the (temperature_count) in the volatile memory.

Display Functions

The following are the display functions in conflation with a short description.

- display_standby() – Standby display screen, shows time, date, year and current temperature.
- display_EndOfStorage() – This is displayed when the recording has reached the end and the storage is FULL

In the log data function bits are changed as per required for the task.

save_value8bit(second,0x00); // save all 6 time data to EEPROM, 0x00 - 0x05

save_value8bit(minute,0x01);

save_value8bit((hour & 127) | (day & 1)<<7,0x02); // save the hour and first bit of day

save_value8bit(((day >>1) & 15) | (weekday << 4),0x03) ; // next 4 bits of day to first half,

and in the second half weekday.

save_value8bit((month & 15)|((year & 15)<<4),0x04); //first 4 bits month, 2nd 4 bit year

When push button 1 is pressed the show_data function is called values are assigned to every time value that need to be modified

```
second= load_value8bit(0x00);
minute = load_value8bit(0x01);
hour = (load_value8bit(0x02)) & 127;
weekday = ( (load_value8bit(0x03)) >> 4) & 15;
day = ((load_value8bit(0x02)) >> 7) + (((load_value8bit(0x03)) & 15) << 1);
month = (load_value8bit(0x04)) & 15;
year = ((load_value8bit(0x04)) >> 4) & 15;
temperature = load_value16bit(0x06);
```

In order for appropriate values to be read there is a need for the bits to be masked and shifted. So when the code wants to load seconds and minutes, there would be no problem and loading would be normal as the addresses are not mutual and are not shared. But for Hours it has a slightly different approach. We need to bitwise multiply in order to mask out the bit 7.

For the days, LSB bit 7 would be taken from the address 0x02 and added to the first 4 bits from the address 0x03, this indicates a shift from the left by 1 bit. Apart from this, all the other time variables follow operations with major similarities.

Rework questions

1.The address of second variable is 0x00 of memory and it is using 1 byte.

2.At the address/location 5 temperature count is stored

```
second= load_value8bit(0x00);
minute = load_value8bit(0x01);
hour = (load_value8bit(0x02)) & 127;
weekday = ( (load_value8bit(0x03)) >> 4) & 15;
day = ((load_value8bit(0x02)) >> 7) + (((load_value8bit(0x03)) & 15) << 1);
month = (load_value8bit(0x04)) & 15;
year = ((load_value8bit(0x04)) >> 4) & 15;
temperature = load_value16bit(0x05);
counter = 1;
```

The function that is save_value16bit() works in a similar fashion. However it separates the 16 bit integer into two constituent parts, these would be the two 8 bit integers. The address start is written at 0x05, which is the next available address on the EEPROM. To put it into an idea, if the temperature value XY.Z is logged on to the address at 0x08, the save bit function breaks it into two 8 bit integers and then saves the highbyte to 0x08 and the lowbyte to 0x09.

Rework

I have edited the code properly and everything runs good and have also added start time function

It displays the date of 29th of February, since this is the more unique feature, as February 29th is a leap year. To ensure this, we set the clock date not too far behind, on February 28, 2020 at 23:59:50, so that immediately the date unique to leap year could be shown. This is done with the ds1307_SetToFeb28() function. In addition to that the ds1307 real time clock is also started

```
void ds1307_SetToFeb28(void){

    // Defined start time
    second = 50;
    minute = 59;
    hour = 23;
    day = 28;
    month = 2;
    year = 20;
    weekday = 1;

    //TODO: Write this time to the DS1307
    ds1307_setTime(); //write to set time
}
```

Dear professor, its working perfectly fine with me I have worked really hard this semester to complete all the labs and lab reports.