**Introduction to Microprocessors | Embedded Systems Development**

EEE 347 | CNG 336

# *LAB MODULE #4*:

***WORKING WITH STANDARD USER INTERFACE DEVICES, A/D CONVERTERS, AND MOTORS***


**Submitted by:**

Talal Shafei 2542371

Noor Ul Zain 2528644

# Declaration

***"The content of the report represents the work completed by the submitting team only,***

***and no material has been borrowed in any form."***

# Introduction

In this module, we have integrated more functionality and built a separate sensor and user side instead of using virtual terminals and manually inputting data packets, as done in module 3. In the final system, we have 3 AVRs, user interface devices, A/D converter and motor. The objective was to have a smart logging system and integrate all these devices together and we have successfully achieved that, and, in this report, we demonstrate our results.

*Division of work:*

With all the labs, we relied on the power of teamwork and made sure efficient division of work was done to increase productivity and to make the project a success. We divided the coding and the testing part wherever possible. For example, for this module, we first worked separately on the user and sensor node. Then we integrated it together and made sure our individual schematics are working in harmony.

# Preliminary Work

a) Explain the purpose of each part of the block diagram shown below. Do you think this A/D converter has parallel or serial digital interface? Explain.

   i. Sample and Hold: responsible to sample the analogue input at a certain frequency and then hold the sampled values. It should hold this sampled value until another sampling occurs.

   ii. D/A Converter: converts the digital values to analogue and provides the comparator the Vref needed to do the comparison.

   iii. Shift Register: gives a voltage value after comparisons have been made by the comparator (final output signal).

   iv. Control Logic: generates a value that is used by the comparator.

v. Comparator: it is responsible for comparing the held sample value and the input signal and the value generated by the control logic.

The interface this A/D converter has is parallel. This can be understood by how the data is being transmitted between the components. Let's outline the whole process: the device firstly samples the input signal and then compares this sampled value with the approximate digital value generated by the control logic. If sampled is greater than the generated, the comparator outputs one otherwise it outputs zero. This continues until the value generated is ideal. In the end, the value is sent to the shift register, which outputs it parallelly.

b) *If ATmega128 operates with an MCU clock frequency of 16-MHz, estimate the minimum possible single-ended A/D conversion time, showing corresponding MCU configuration requirements, and calculation. You may ignore the time it takes to initialize the analog circuitry, and may assume free running mode.*

To have the minimum possible conversion time we should have the highest frequency with making sure that its less than 200KHZ so the ADC circuit function correctly
To achieve this we must use 128 for the pre-scaling which will result in

$$\frac{16MHZ}{128} = 125KHZ$$

ADC require 13 cycles thus,

$$Total\ time = 13 * \frac{1}{125KHZ} = 104\mu s$$

c) *Calculate and fill in the table with the effective resolution of the system in terms of voltage. Also, describe how you may use the existing 10-bit A/D in ATmega128 in obtaining the provided 5-bit adjusted values.*

Eff. Resolution calculations:
Equation: $\frac{Vspan}{2^5}$ because we're using 5 bits only.

Temperature: $(4-2)/2\text{^}5 = 0.0625\ (V)$
Moisture: $(4.2-1.8)/2\text{^}5 = 0.075\ (V)$
Water: $(2.8-2)/2\text{^}5 = 0.025\ (V)$
Battery: $(5-3)/2\text{^}5 = 0.0625\ (V)$

| Parameter | Min. (V) | Max. (V) | Min. (digital) | Max. (digital) | Eff. Resolution |
|---|---|---|---|---|---|
| T | 2.0 | 4.0 | 0x03 | 0x1B | 0.0625 |
| M | 1.8 | 4.2 | 0x02 | 0x1E | 0.075 |
| W | 2.0 | 2.8 | 0x04 | 0x1A | 0.025 |
| B | 3.0 | 5.0 | 0x01 | 0x1F | 0.0625 |

We wish to achieve these Eff. Resolutions for these spans using 5 volts external on the AVCC of the AVR. Therefore, we must convert the 10 bits digital output to match the above assumed limits using linear transformations.

For 5 Volts and 10 bits the AVR step size (resolution) is $\frac{5}{2^{10}}$ which is equal to 4.88mv

Now to find the linear transformation all we need to do is to convert the limits using the 4.88mv step size then find a linear relationship (slope and offset) between the 10bits output and the 5bits limits.

This will result in two equations after solving them we will have:

$Slope\ (m) = \frac{max_{5bits} - min_{5bits}}{max_{10bits} - min_{10bits}}$
$Offset\ (c) = min_{5bits} - m * min_{10bits}$ (we can use the max equation too)

Note we are not removing the fractions here to be as precise as possible.

**Temperature:**
Min: 2V -> $2/4.88mv$ -> 409.6 (digital in 10 bits)
Max: 4V -> $4/4.88mv$ -> 819.2 (digital in 10bits)

Slope = 15/256
C= -21 V

**Moisture:**
Min: 1.8V -> $1.8/4.88mv$ -> 368.64 (digital in 10 bits)
Max: 4.2V -> $4.2/4.88mv$ -> 860.16 (digital in 10bits)

Slope = 175/3072
C= -19 V

**Water:**
Min: 2V -> $2/4.88mv$ -> 409.6 (digital in 10 bits)
Max: 2.8V -> $2.8/4.88mv$ -> 573.44 (digital in 10bits)

Slope = 275/2048
C= -51 V

**Battery:**
Min: 3V -> $3/4.88mv$ -> 614.4 (digital in 10 bits)
Max: 5V -> $5/4.88mv$ -> 1024 (digital in 10bits)

Slope = 75/1024
C= -44 V

by this less than 3.2V must be less than 0x04 and we will check this value when we want to display "change the battery immediately" message

also please note when we apply these linear equations on the 10bits of the ADC we make sure to multiply first then divide so we don't lose all the information because of the integer division (since AVR doesn't have instructions for float numbers)

finally, outputs might be shifted by +1 or -1 because of the integer division so we added 1 for the Battery and moisture outputs since we saw that they usually result in fractions > 0.5 so to be more precise we added the one.

d) *Given the rotational speed of the water pump (motor) will vary between 20% to 80%, depending on the moisture (M) level at the remote node, calculate and indicate relevant PWM generation settings you plan to program in the motor control section of your remote node solution. What will be the default motor speed before any data has been received from moisture sensor? Why?*

The motor will activate each 10 seconds for 5 seconds so there will be for sure data of the moisture arrived but in case there is no data arrived or its interrupt activated before the sensors interrupt the speed of the motor will be 20% duty cycle since the default values at the beginning for the sensor buffer is 0's

For the motor we used two PWM:

1. Fast PWM on Timer1 that will count until compare match interrupt to start the motor but because we are using Fast PWM timer will keep counting until overflow which will result in another interrupt that will stop the motor after 5s from the first compare match interrupt, we were able to achieve this by using 4MHZ with 1024 pre-scaling and counting until 40000 cycles(rounded) for the 10s and the rest which is nearly 20000 for the 5s, also we didn't configure the type of PWM (inverted or not) because we are not using its wave and not using the port.
2. Phase correct PWM on Timer 0 that will generate the wave on OC0 port in inverted mode that will keep toggling the EN in the motor diver which will result in rotating the motor, we used for that 1024 pre-scaling and another equation to make sure to make the duty cycle proportional to the value of the moisture.

   The motor equation:

$$\frac{Moisture * 0x5FA - 0xBF4}{0x118} + 0x33$$

   Explained:

   First we are doing Min max normalization of moisture value* 0x100(100% duty cycle) for the Duty cycle corresponding to the moisture so 0 moisture will result in 0 duty cycle and 0x1E will result in 100% then we multiply by 60% so 0 moisture will result in 0 duty cycle and 0x1E will result in 60% duty cycle then we shifted by 20% of Timer 0 cycles which

corresponds to 0x33(0.2 * 0xFF), so finally 0 moisture will correspond to 20% duty cycle and 0x1E moisture will correspond to 80% duty cycle.

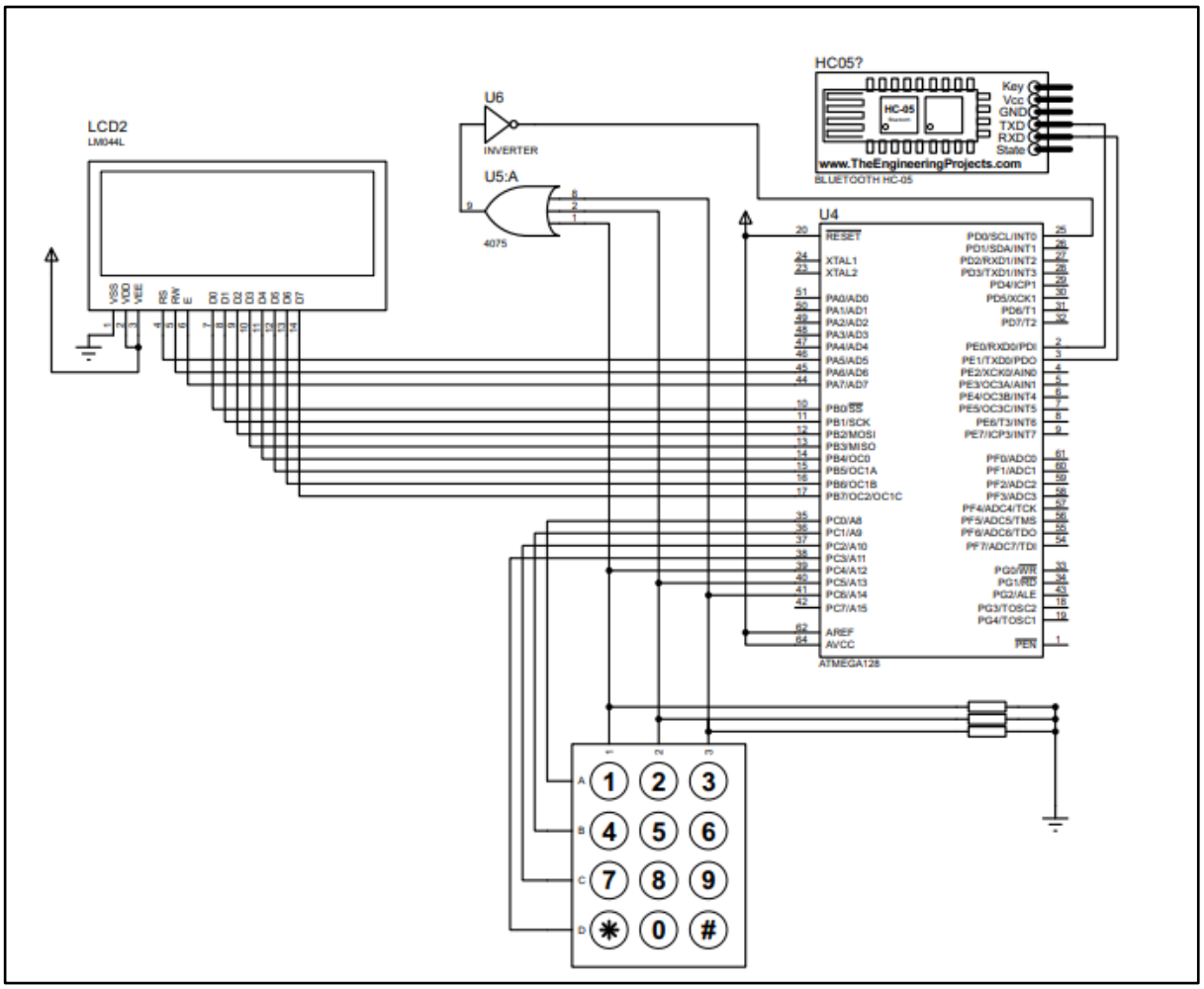
e) *Outline the main differences between 16x2 LCD discussed in lectures, and 20x4 LCD to be utilized at the user node*.

- Total character capacity for the 16x2 LCD is : 16x2 = 32.
- Total character capacity for the 20x4 LCD is: 20x4 = 80.
- This indicates the difference between the sizes of the DDRAM matrix and the mapping to the screen.
- The larger display area of the 20x4 LCD generally improves readability, especially when displaying longer messages or complex data.
- The electrical pin configuration for connecting the LCD module to the user node may vary between the 16x2 and 20x4 LCDs.

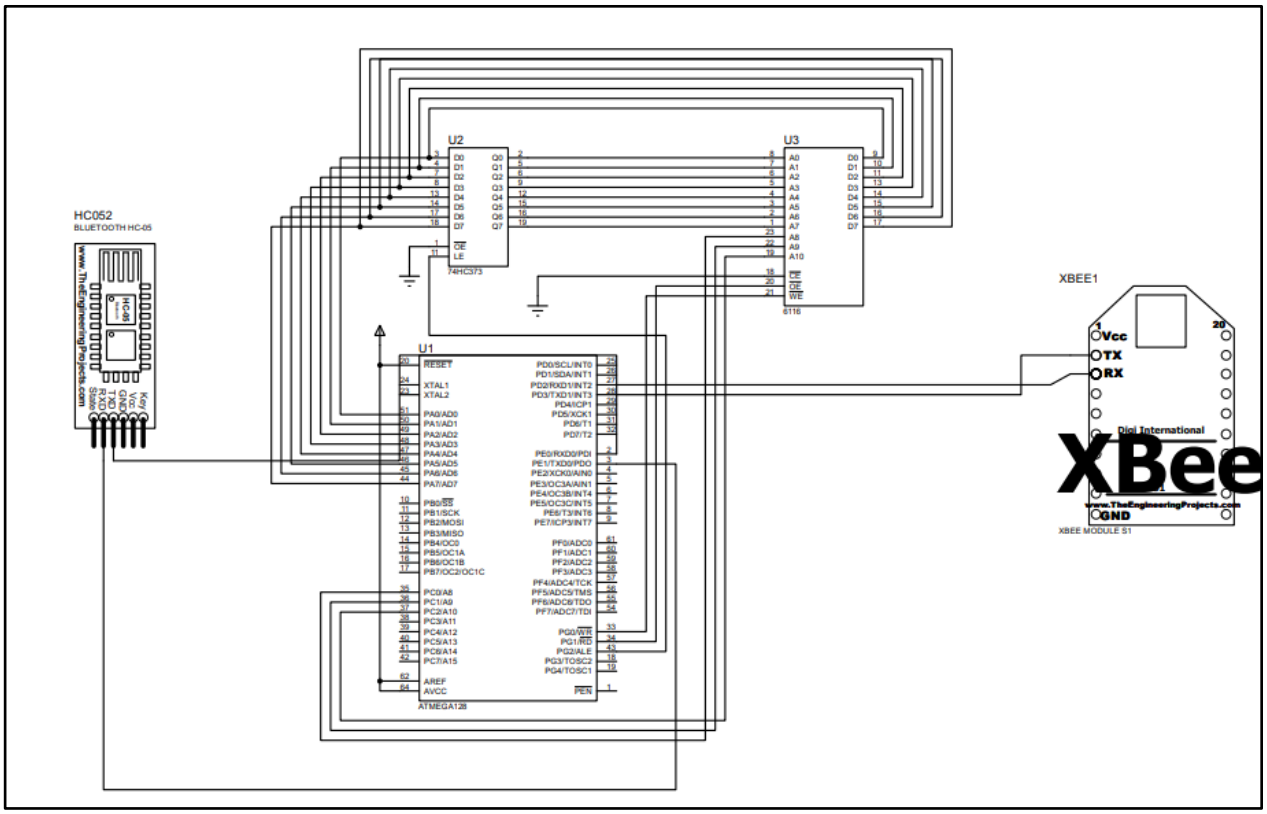
*f) Sketch a system schematic diagram that has the full smart farming system, including 3 ATmega128 MCUs and their connectivity to the peripheral components. Your sketch should be organized and readable, preferably using a drawing application such as Visio. Pin level connectivity should be clear for each pin of each component.*

Please note that we used schematic export feature of Proteus.

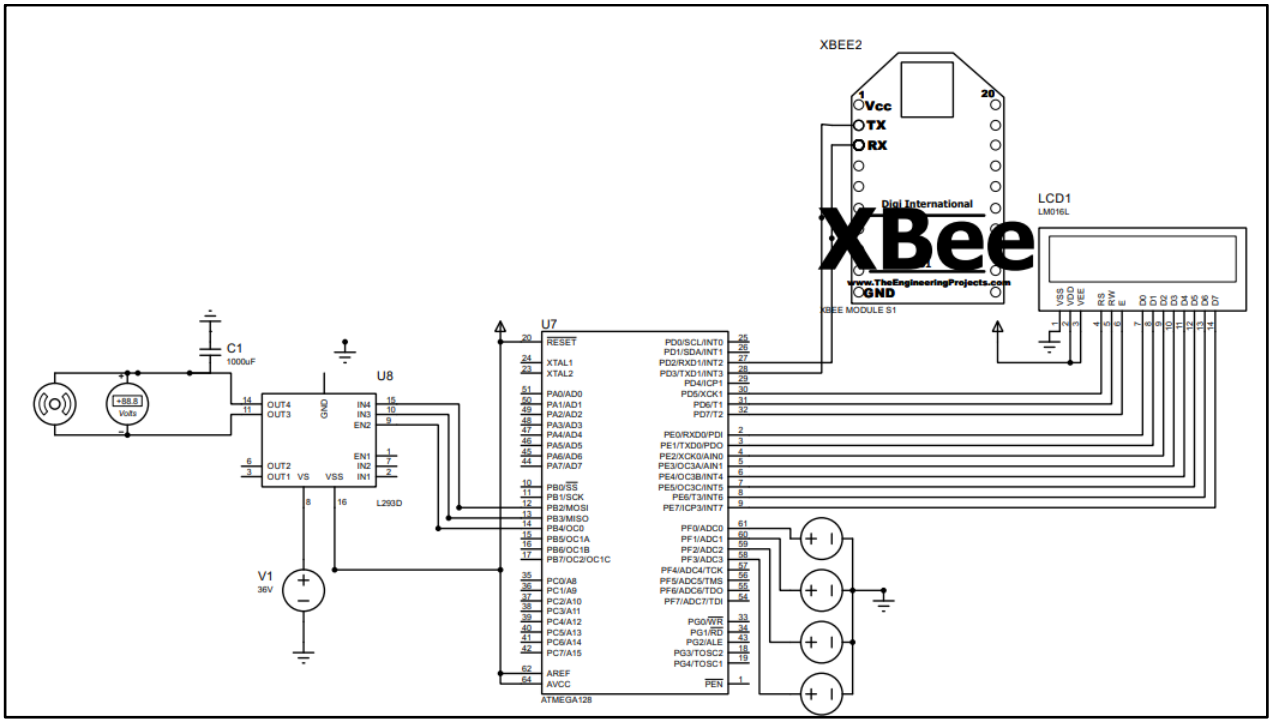
## User Node



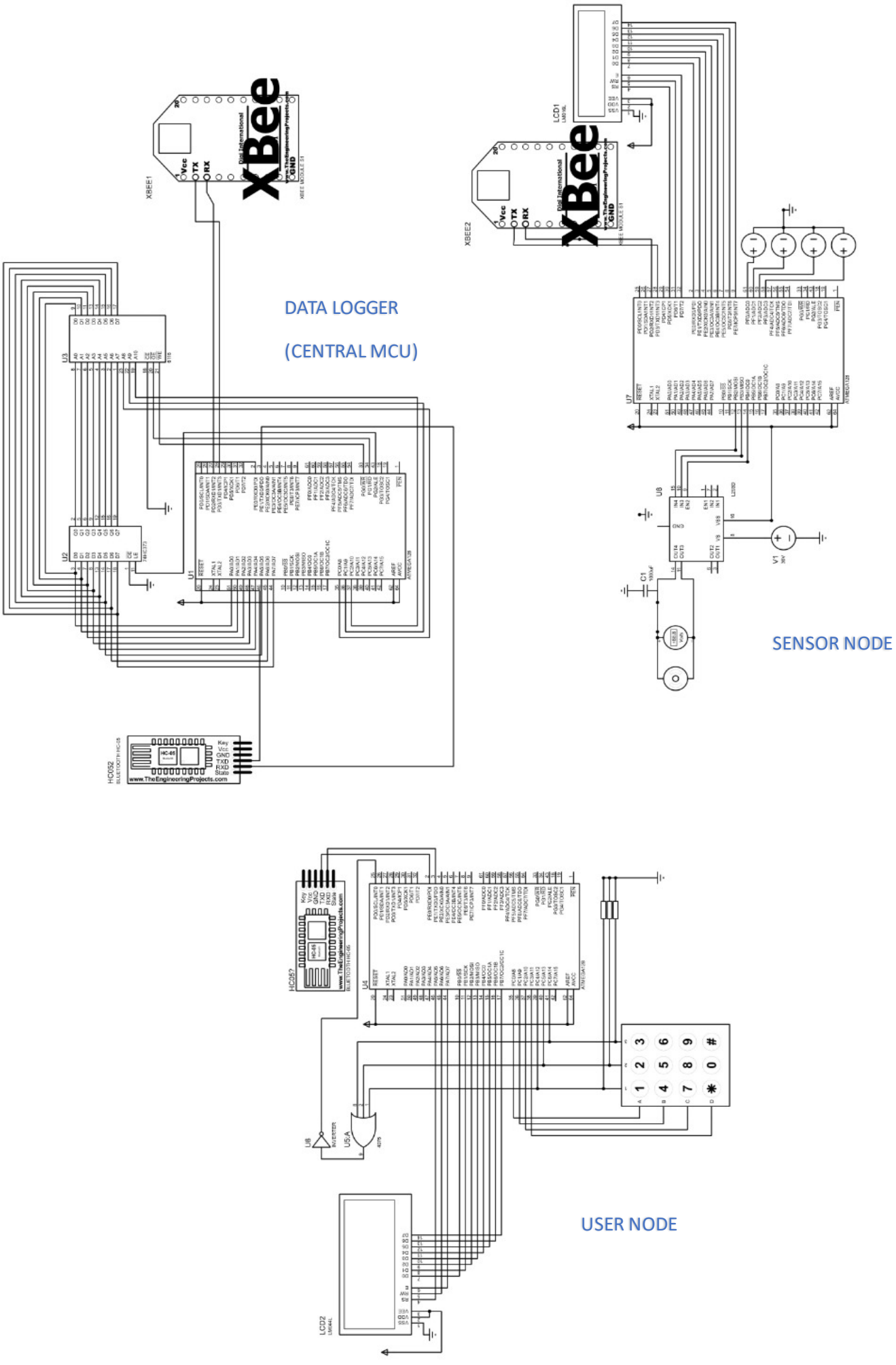
## Central MCU/ Data logger



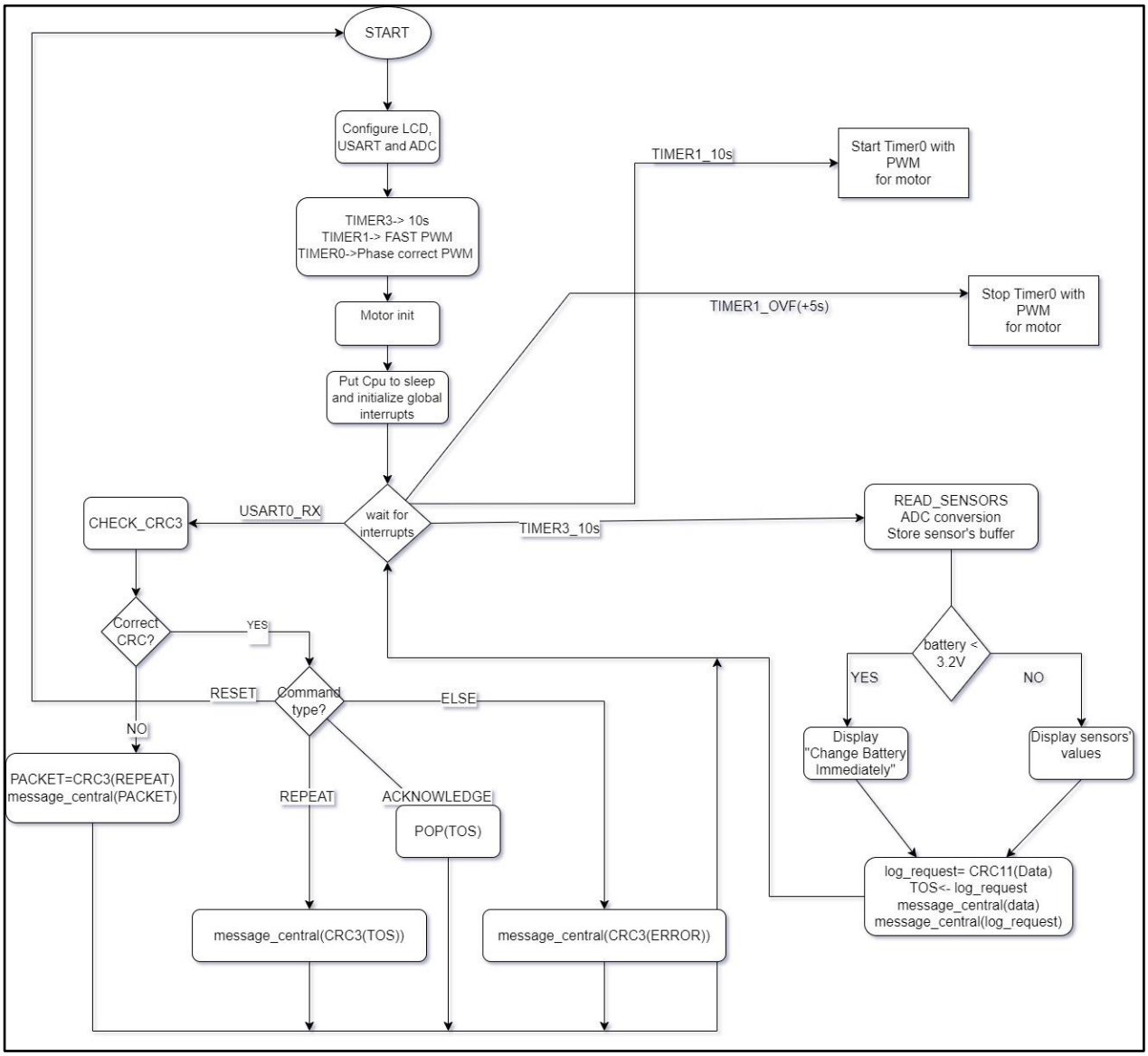

## Sensor Node



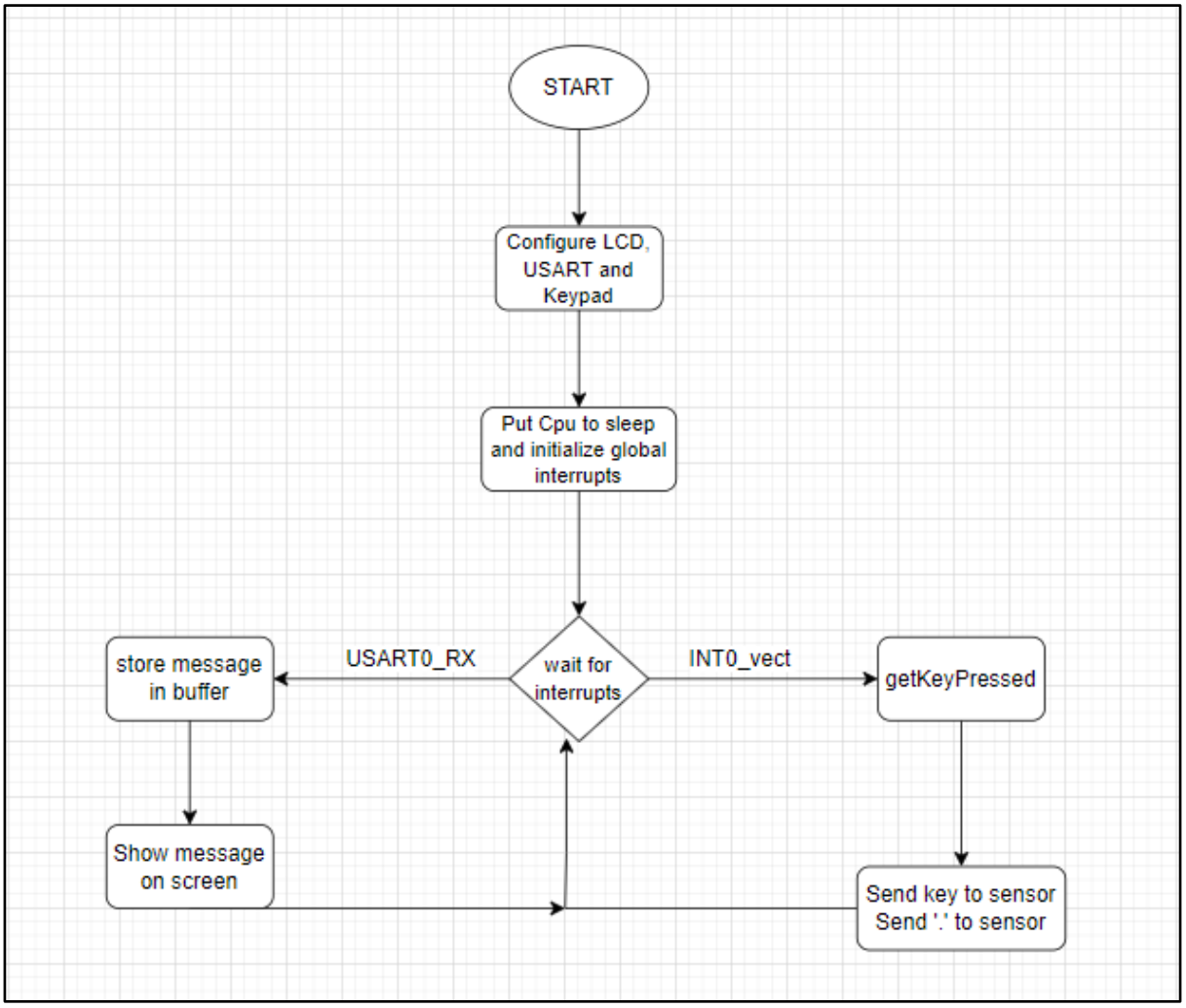
# SYSTEM



*g) Sketch an algorithmic flowchart to accurately show the program executed in the Remote Sensor Node MCU.*



*h) Sketch an algorithmic flowchart to accurately show the program executed in the User Node MCU.*



*i. Considering your answers to (f-h), and farming system representation in Figure 4.1, use component datasheets to investigate estimated minimum (IDLE) and maximum (ACTIVE) power dissipation for components in your system, including times when both wireless transmission interfaces are active into your worst-case power scenario. Complete the blanks in Table 4.1.*

| Component Power (mW) | Approx. best-Case (IDLE) | Approx. worst-Case (ACTIVE) |
|---|---|---|
| MCU (Central) | 4mA* 5V = 20mW | 9mA*5V = 45mW |
| MCU (User-node) | 4mA* 5V = 20mW | 9mA*5V = 45mW |
| MCU (Remote-node) | 4mA* 5V = 20mW | 9mA*5V = 45mW |
| Bluetooth Interface | 9uA* 3.3V = 29.7uW | 40mA*3.3V=132mW |
| Xbee Interface | 12uA* 3.3V=39.6uW | 45mA*3.3V=148.5mW |
| 16x2 LCD display | 1.0mA * 5V = 5mW | 3.0mA * 6.5=19.5mW |
| 20x4 LCD display | 0.4mA * 5V = 2mW | 2.4mA *7=16.8mW |
| Motor driver | 0 | 200 |
| Waterpump | 0 | 1000 |

Explanation:

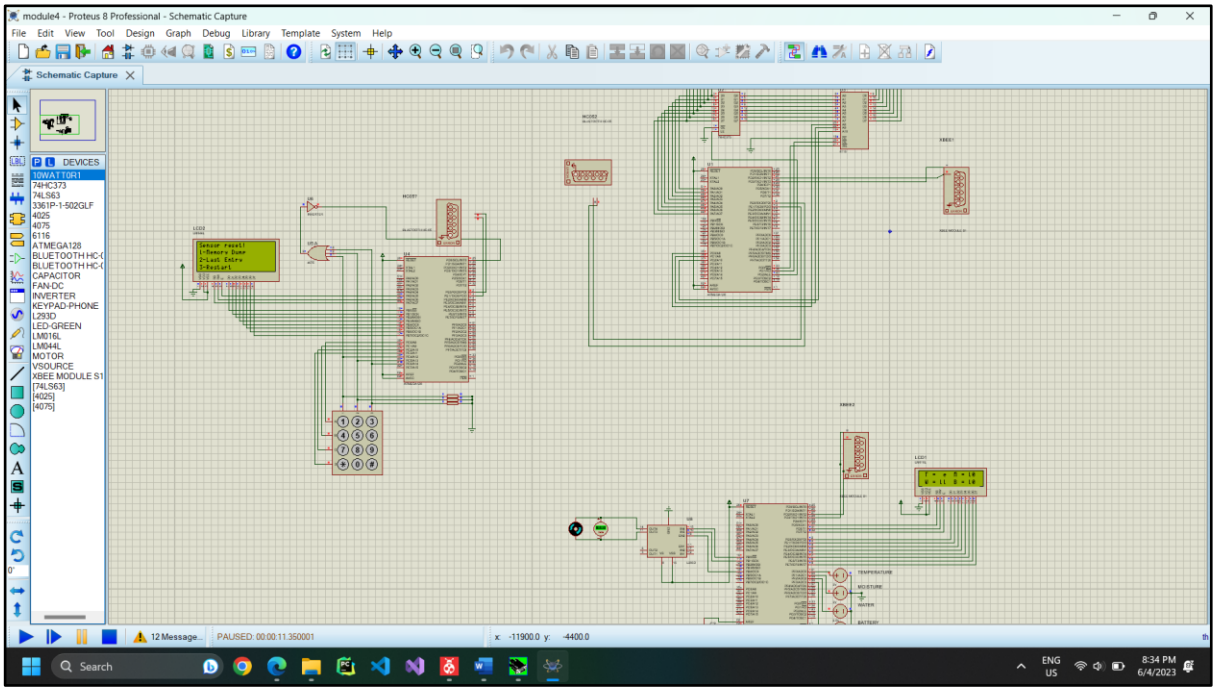
Please note that for MCUs, our reference is the datasheet value for 4Mhz operating frequency.

We used the component specific datasheets for others. For the worst case of Xbee and Bluetooth we will consider the transmit current since it is higher and will give a clearer worst-case idea.

# Testing

(all codes at the end)

## Schematic of the system



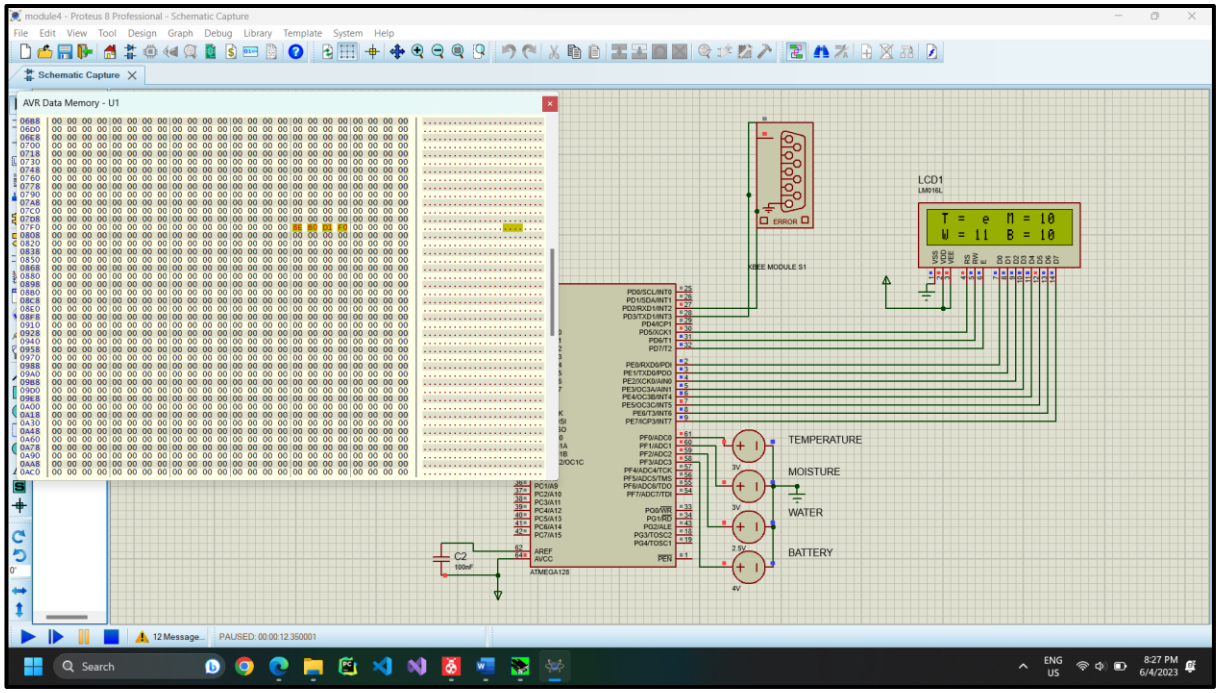

## SENSOR NODE

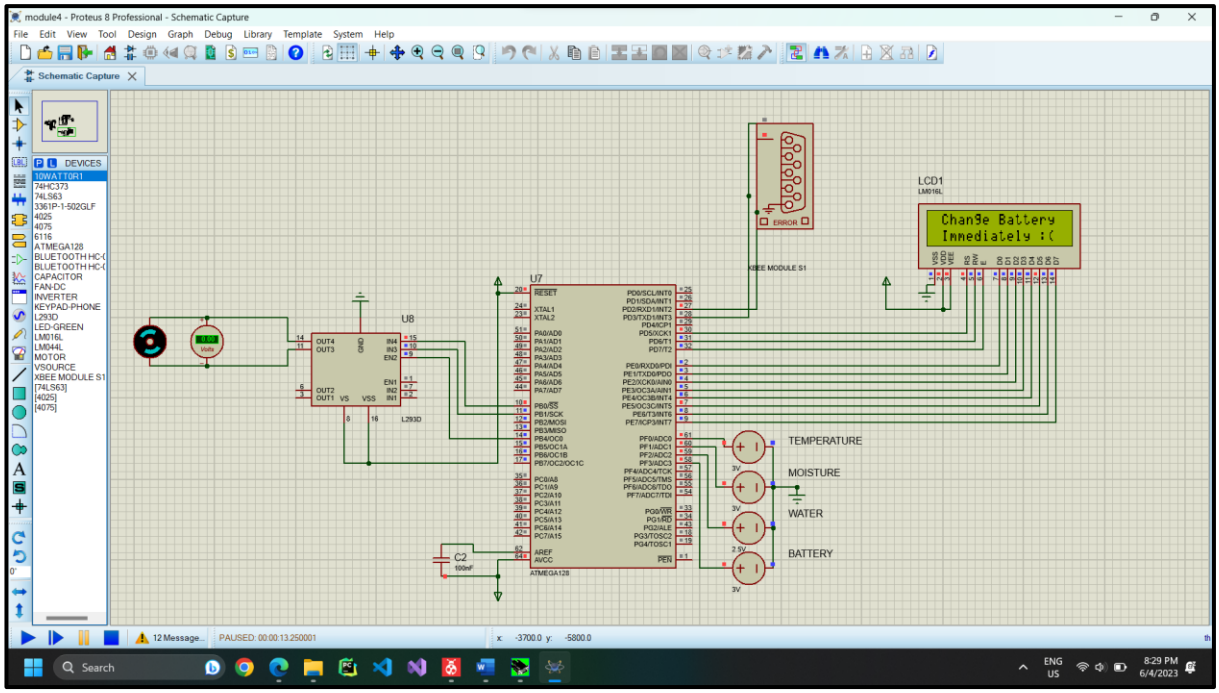
Here the Temperature is 3V which is 614.4 in 10 bits then after applying the Temperature linear transformation (614.4 * slope + offset) it will become 14 (0xe) in 5bits.

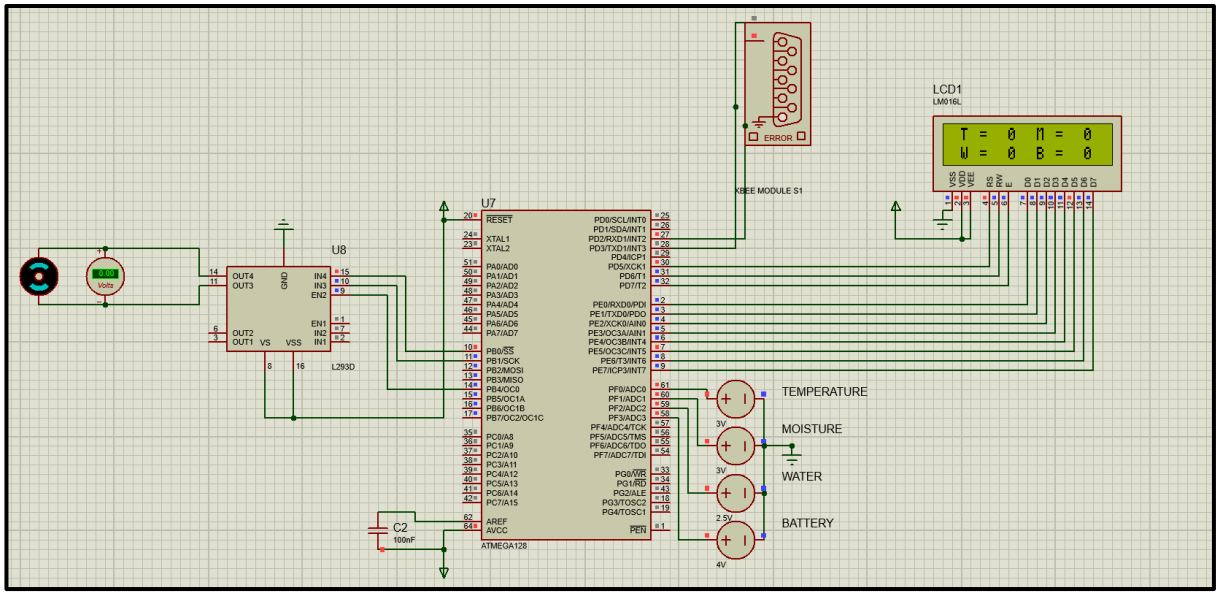
Moisture is also 3V -> 614.4 -> applying Moisture equation will result in 16 (0x10)

Water 3V -> 614.4 -> applying Water equation will result in 17 (0x11)

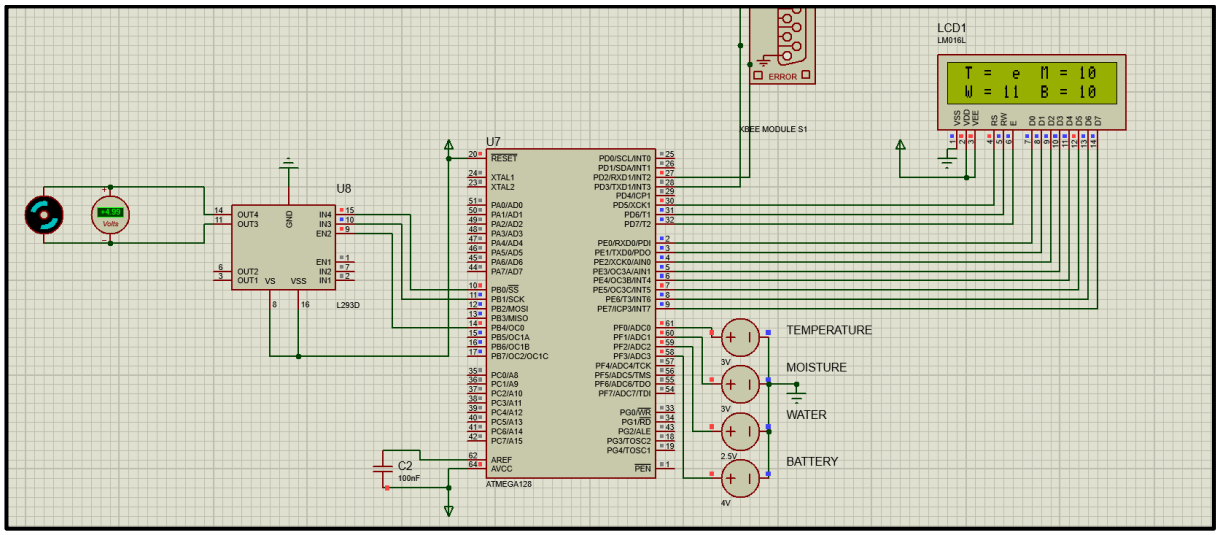
Battery 4V -> 819.2 -> applying Battery equation will result in 16 (0x10)


module4 - Proteus 8 Professional - Schematic Capture
Schematic Capture
AVR Data Memory - U1
LCD1
LM016L
T = e M = 10
W = 11 B = 10
TEMPERATURE
MOISTURE
WATER
BATTERY
ATMEGA128
C2
100nF
12 Message...
PAUSED: 00:00:12.350001
Search
ENG
US
8:27 PM
6/4/2023

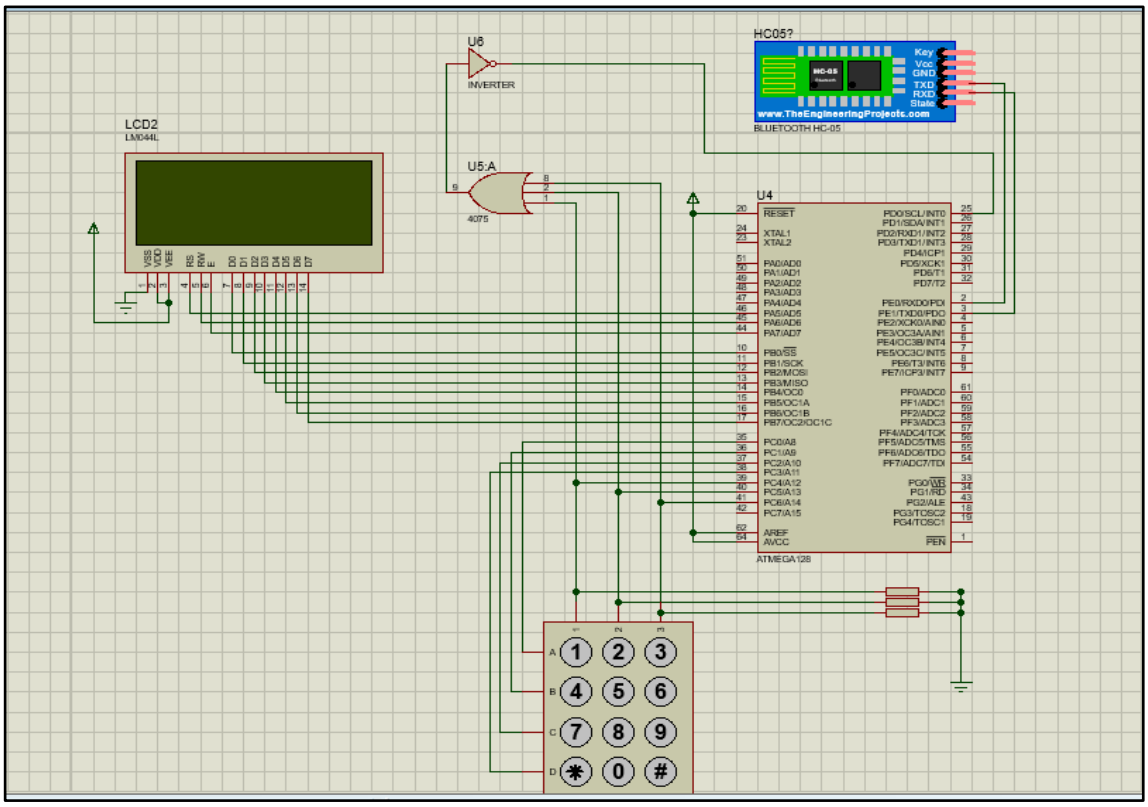

module4 - Proteus 8 Professional - Schematic Capture
Schematic Capture
DEVICES
LCD1
LM016L
Chan9e Battery
Immediately :(
U7
U8
L293D
TEMPERATURE
MOISTURE
WATER
BATTERY
ATMEGA128
C2
100nF
12 Message...
PAUSED: 00:00:13.250001
Search
ENG
US
8:29 PM
6/4/2023




As you can see in the figure below, the motor is rotating clockwise we initialize IN4 to be 1 and IN3 to be 0 also the speed for Moisture 0x10 based on the equation explained above would be 49.8% duty cycle (because we would have stored 7F in the Timer0 compare register)

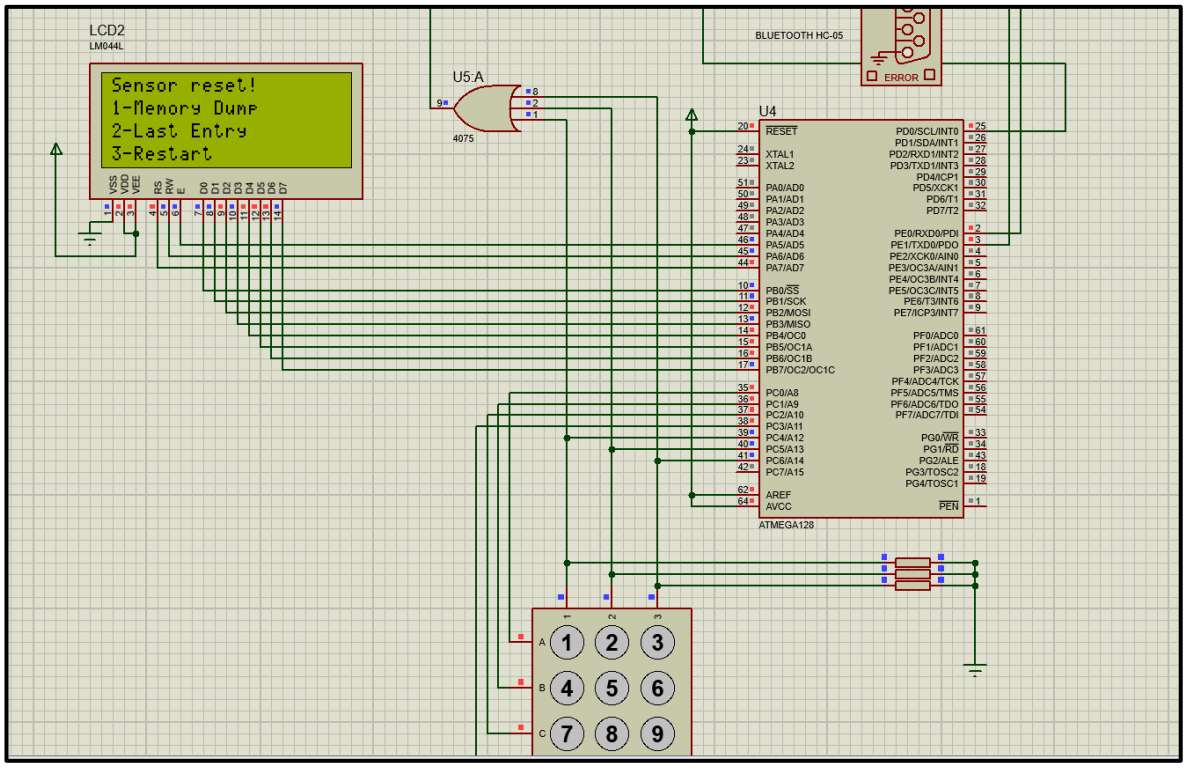


# USER NODE

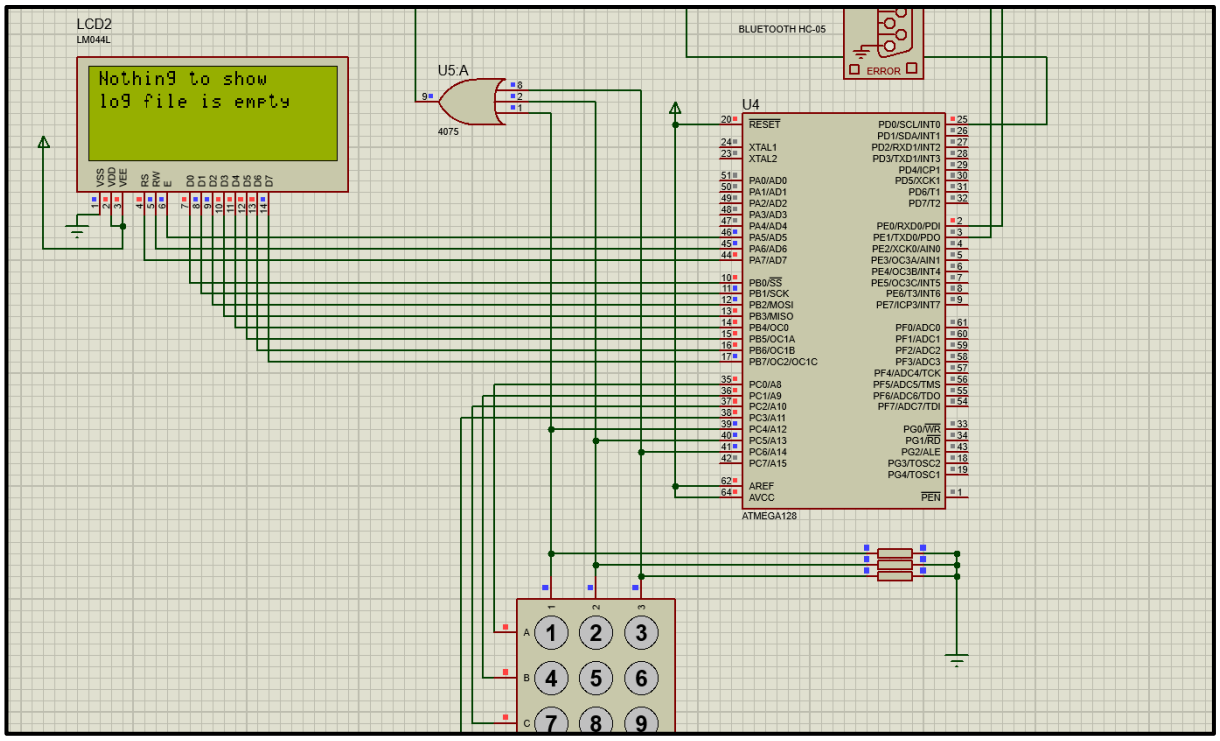



## Testing

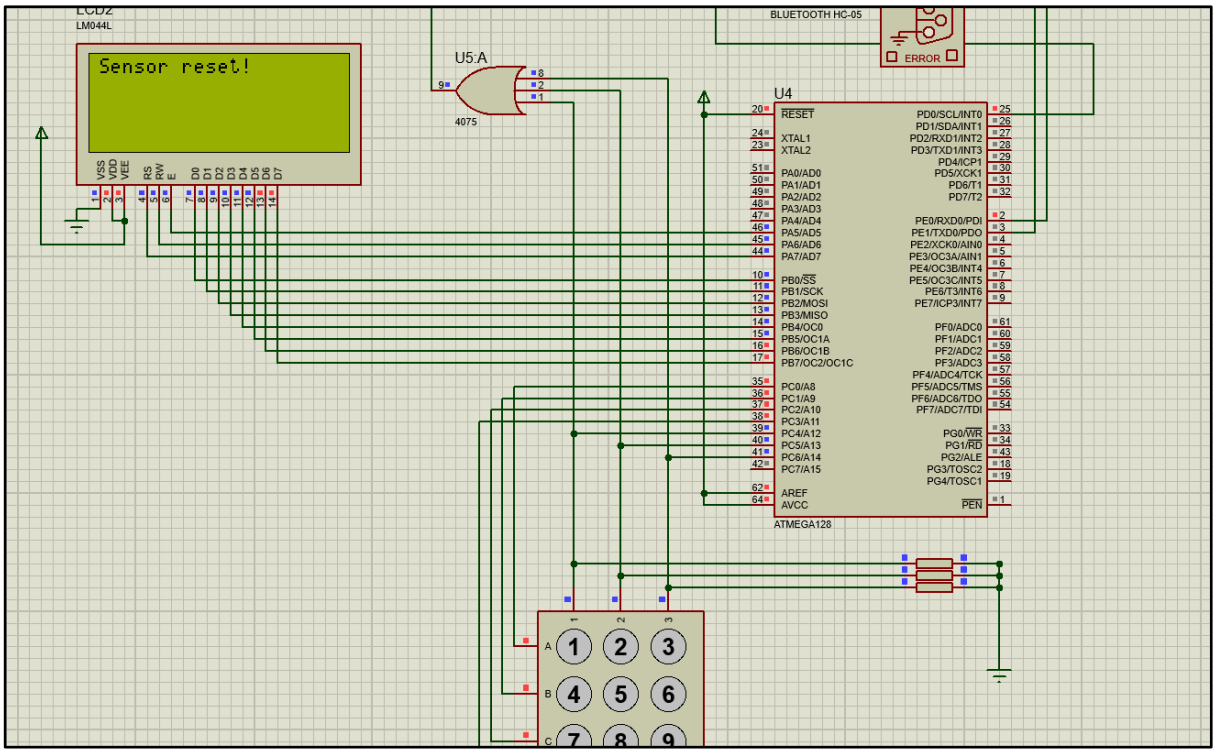
1) Showing the USER MENU



2) Pressed 1 or 2 and nothing was logged because it has not been 10 seconds yet.

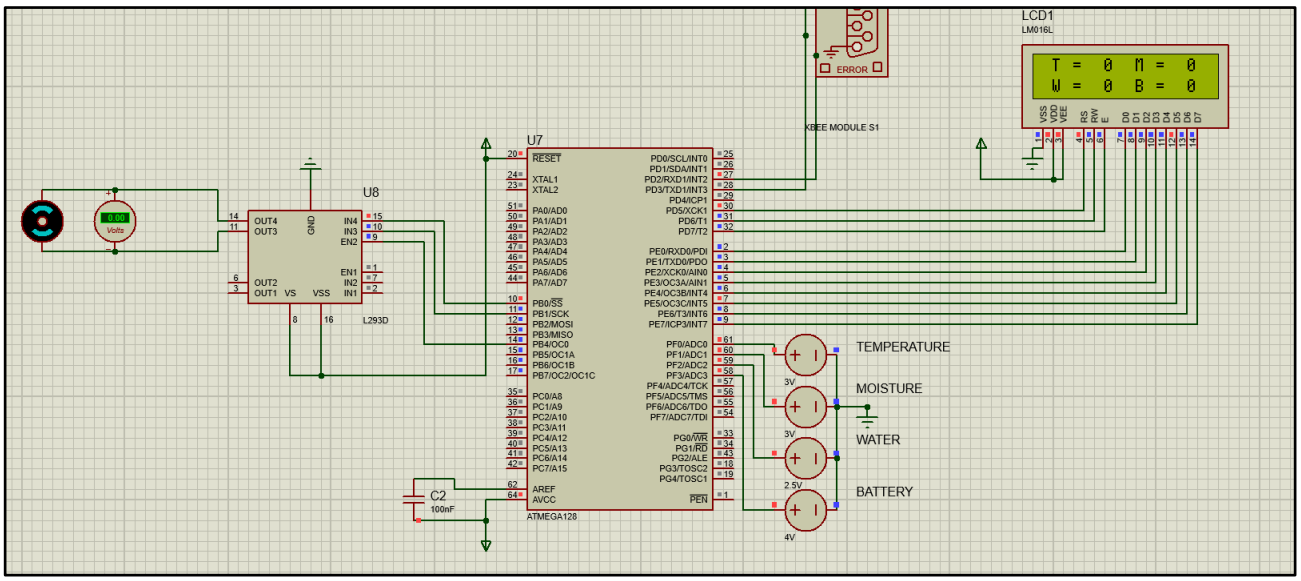



3) User pressed 3 and Sensor has been resetted.

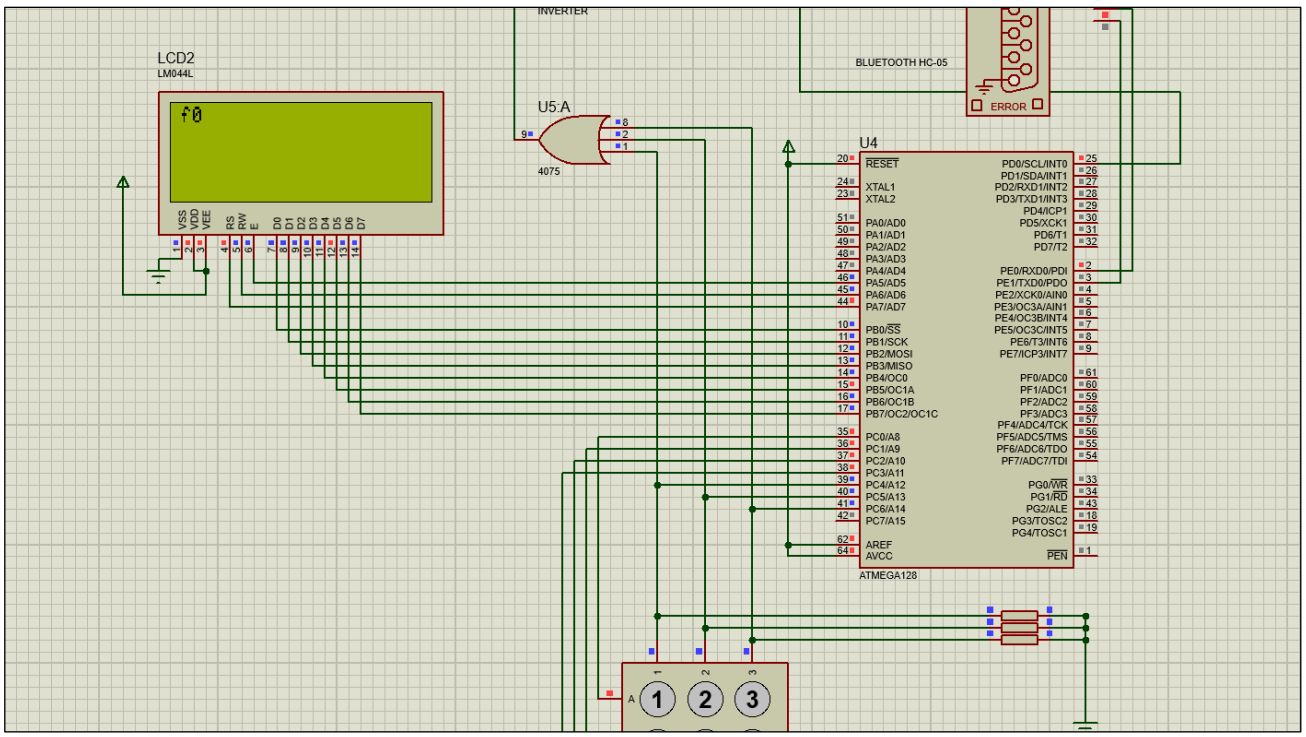



Please note that this screenshot follows the one above and as seen all the values are resetted and have become zero.

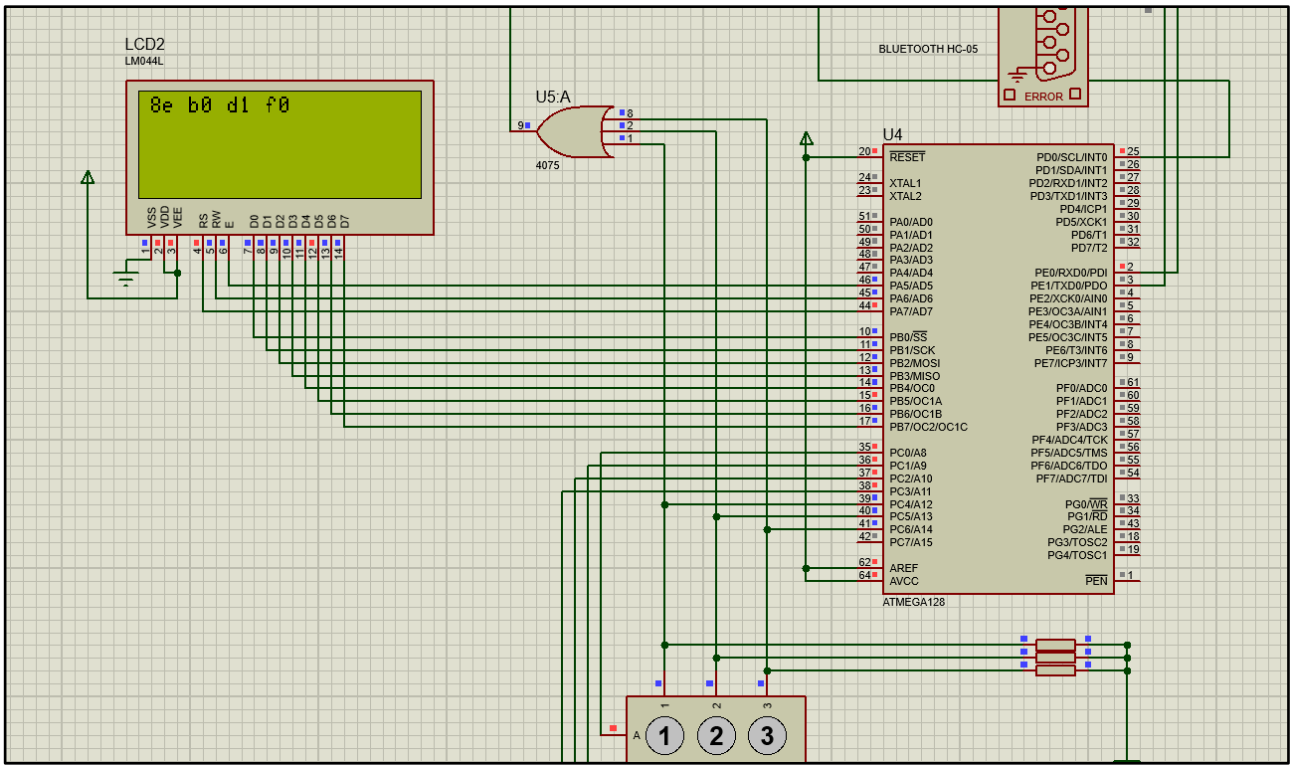
To force reset on the sensor we send a reset request from the data logger, and this will trigger the reset function in the sensor that will activate watchdog timer with 15ms followed by while(1); which will result in resetting the sensor and empty all its RAM and registers in 15ms and that's why we are seeing 0's in the display and the motor stopped suddenly because the sensor started from the beginning again.



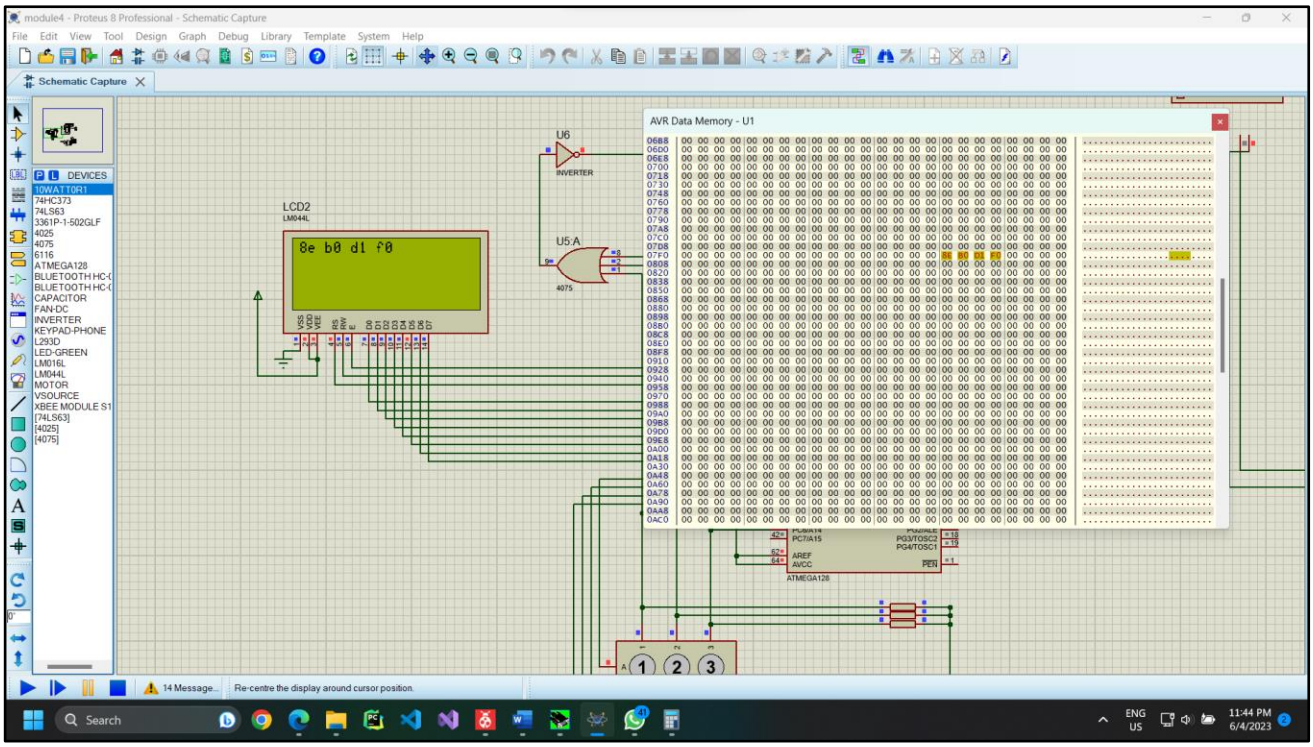

4) Testing Last Entry after 10 seconds have passed, and values have been logged.



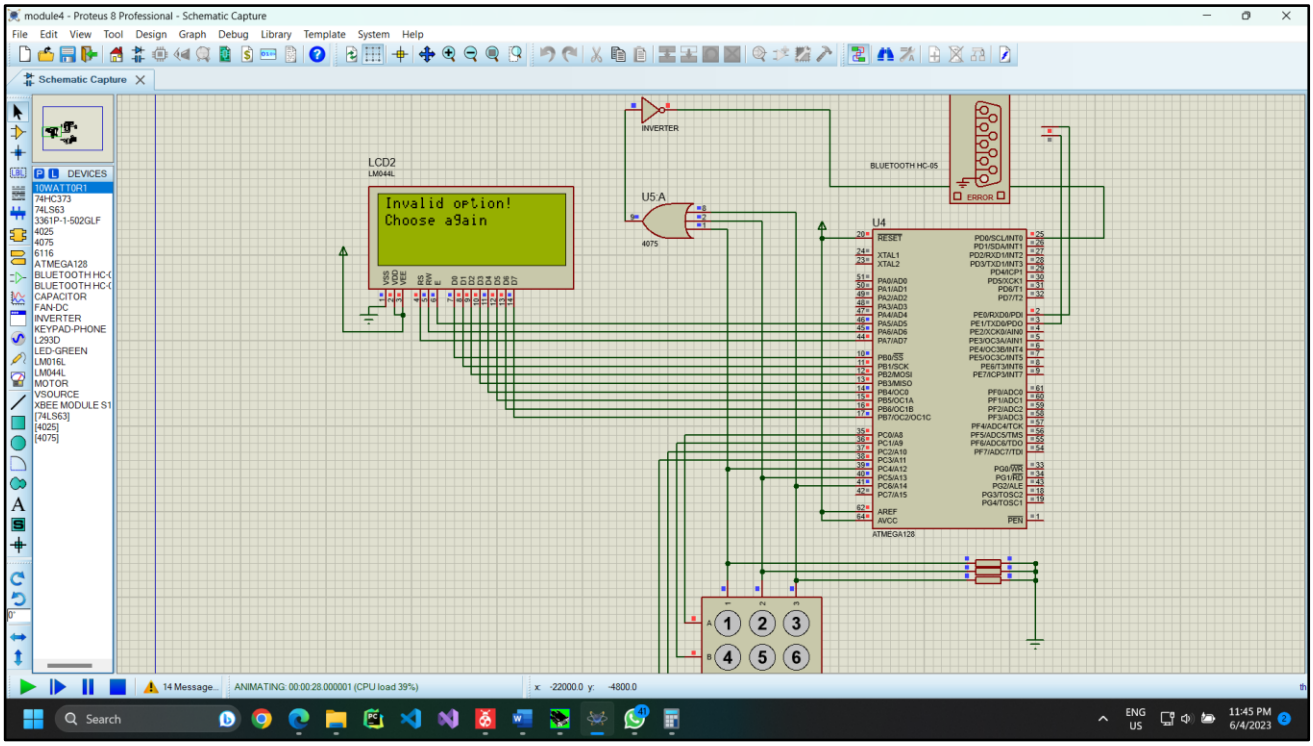

5) Testing Memory dump:




Making sure these are the correct values to be logged by showing the logged memory/sensor side:



6) Testing Invalid keypad inputs




# Codes

Sensor node:

```
/* Includes */
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <stdio.h> // for sprintf
#include <stdlib.h>
#include <string.h>
#include <util/delay.h>
#include <avr/wdt.h>

/* Define statements*/


#define F_CPU 4000000UL
// ((F_CPU/16/BaudRate)-1) = 0x19
#define BR_Calc 0x19

#define gen 0xD4 // CRC polynomial

#define COMMAND_TYPE(x)     (x&0x60)

// LCD
```

```
//Command_to_LCD(0x10);     //shift cursor to right
#define LCD_START_LINE_1 0x80
#define LCD_START_LINE_2 0xC0
#define LCD_INIT_8BIT 0x38 // 2 Lines and 5x7 matrix (D0-D7, 8-bit)
#define LCD_DISPLAY_ON_CURSOR_OFF 0x0C //Display on, cursor off
#define LCD_CLEAR_DISPLAY 0x01
#define LCD_LEFT_TO_RIGHT 0x06

#define LCD_DDR DDRE
#define LCD_PORT PORTE // for sending data to LCD
#define LCD_CTRL_DIR DDRD
#define LCD_CTRL PORTD // for controlling the LCD flags
// PORTD bits
#define RS 5
#define RW 6
#define E 7

// commands
#define Reset_command 0x00
#define Acknowledge 0x40
#define Log_command 0x20
#define Error_repeat 0x60

// ADC
#define ADC_DDR DDRF


// sensor indexes
#define TEMPERATURE 0
#define MOISTURE 1
#define WATER 2
#define BATTERY 3

// MOTOR
// since our PWM wave will be on OC1B which is PB6 and we can also use the other pins for
the L293D motor driver
#define MOTOR_DDR DDRB
#define MOTOR_PORT PORTB

/* Global variables*/
uint8_t TOS; // in case we received a repeat request
uint8_t is_tos_empty = 1; // 1 empty, 0 full

uint8_t central_tr_buffer; // for transmitting byte to the central MCU
uint8_t central_rc_buffer; // to receive byte from the central MCU

uint8_t sensor_input_buffer[4] = {0, 0, 0, 0}; // because we have 4 sensors

/* Prototypes*/
void initialize();
void message_central(uint8_t);
void init_timer3_10s();

// helper
void pushTOS(uint8_t);
void popTOS();

// LCD functions
void lcd_command(uint8_t);
void lcd_print(unsigned char*);
void lcd_display_sensor_screen();
void lcd_display_battery_low();
```

```
// sensor functions
void read_sensors();
uint16_t get_sensor(uint8_t);
void receive();
void reset_sensor();


// CRC functions
uint8_t CRC3(uint8_t);
uint8_t CRC11(uint8_t);
uint8_t CRC3_CHECK(uint8_t);


int main(void){
	initialize();
    while (1){
		sleep_enable();
		sleep_cpu();
    }
}

// to send data for the central MCU
ISR(USART1_UDRE_vect){
	sleep_disable();

	while(!(UCSR1A & (1<<UDRE1))); // double check
	UDR1 = central_tr_buffer;
	_delay_ms(100);

	// disabling transmitting and enabling receiving again
	UCSR1B &= ~((1 << TXEN1) | (1 << UDRIE1));
	UCSR1B |= (1 << RXEN1) | (1 << RXCIE1);
}
// to receive data from the central MCU
ISR(USART1_RX_vect){
	sleep_disable();

	while (!(UCSR1A & (1 << RXC1))); //double check as in slides ;)
	uint8_t input = UDR1; // capture input

	// data logger will only send repeat requests, reset command, or acknowledge
	sei();
	if(!CRC3_CHECK(input)){
		// send error repeat request
		message_central(CRC3(Error_repeat));
	}

	else if(Reset_command == COMMAND_TYPE(input)){
		reset_sensor();
	}
	else if(Error_repeat == COMMAND_TYPE(input)){
		message_central(CRC3(TOS));
	}

	else if(Acknowledge == COMMAND_TYPE(input)){
		// since acknowledge we can discard TOS contents
		popTOS();
	}

	else {
		// send repeat error
```

```
                message_central(CRC3(Error_repeat));
        }


}

// monitor conversion
ISR(ADC_vect){
        // this will prompt us back to get sensor
        sleep_disable();
}

// work every 10s
ISR(TIMER3_COMPA_vect){
        sleep_disable();
        // read sensors
        read_sensors();
        // check battery
        // 3.2V -> 0x04 digital from the Battery equation
        if(sensor_input_buffer[BATTERY] < 0x04){
                lcd_display_battery_low();
        }
        else{
                // display values
                lcd_display_sensor_screen();
        }


        // send values
        sei();
        for(uint8_t i =0; i<0x04; i++){

                // set data flag, data id, 5 bit data parameter as explained in module 2
                uint8_t data = (0x80 | (i<<5)) | sensor_input_buffer[i];
                // get crc11 for the data with the log request
                uint8_t log_request = CRC11(data);

                // send data first
                message_central(data);
                pushTOS(data);
                // then send log request
                message_central(log_request);
                pushTOS(log_request);
                receive();

        }
        _delay_ms(100);

        init_timer3_10s();

}

// start motor after 10s
ISR(TIMER1_COMPB_vect){
        sleep_disable();
        // start the motor
        // equation explained in the report

        // phase correct PWM, inverted, 1024 prescaler
        TCCR0= (1<<WGM00) | (1<<COM01) | (1<<COM00) | (1<<CS02) |(1<<CS01);//|(1<<CS00);
        // proportional to Moisture
        OCR0 = ((sensor_input_buffer[MOISTURE]*0x5FA - 0xBF4)/0x118) + 0x33;
```

```
}

// stop motor after the 5s
ISR(TIMER1_OVF_vect){
	sleep_disable();
	// stop the motor
	// note since OC0 connected to EN in L293D all we need is to stop the PWM
	TCCR0 = 0;

}

void pushTOS(uint8_t value){
	TOS = value;
	is_tos_empty = 0;
}

void popTOS(){
	is_tos_empty=1;
}

void reset_sensor(){
	// will force reset in 15ms
	wdt_enable(WDTO_15MS);
	while(1);
}

void init_timer3_10s(){
	// reset timer
	TCNT3 = 0;
	// set output compare
	OCR3A = 40000;
	// set 1024 prescaling and activate CTC mode
	TCCR3B = (1<<CS32) | (1<<CS30)| (1<<WGM32);
	// Enable interrupt on a compare match
	ETIMSK = (1<<OCIE3A);
}

void receive(){
	// disabling transmitting and enabling receiving again
	UCSR1B &= ~((1 << TXEN1) | (1 << UDRIE1));
	UCSR1B |= (1 << RXEN1) | (1 << RXCIE1);

}

void initialize(){
	// USART initialization
	UBRR1H = (uint8_t)(BR_Calc>>8);
	UBRR1L = (uint8_t)(BR_Calc);
	UCSR1C = (1<<UCSZ11)|(1<<UCSZ10); // setting USART to 8 bit width
	UCSR1B |= (1 << RXEN1) | (1 << RXCIE1); // enabling receive and receive interrupt

	// ADC initialization
	ADC_DDR = 0x00; // input chanels for ADC, but only the first 4 bits
	//ADMUX = (1<<REFS0); // choose AVCC as the voltage reference
	// enable ADC, and prescaling by 64 to make the 4.096MHZ -> 64KHZ to be valid
	ADCSRA =  (1<<ADEN) | (1<< ADPS2) | (1<< ADPS1);

	// initialize LCD
	LCD_CTRL_DIR |= 0xE0;
	LCD_DDR = 0xFF; // D[7:0] of LCD
	_delay_ms(20); // documentation says +15ms
```

```
	lcd_command(LCD_INIT_8BIT); // make it 8 bit mode and 2 lines
	lcd_command(LCD_LEFT_TO_RIGHT);
	lcd_command(LCD_DISPLAY_ON_CURSOR_OFF); // display on cursor off (or cursor on with
0x0E)
	lcd_command(LCD_CLEAR_DISPLAY); // clear display screen
	lcd_display_sensor_screen();

	// configure timer 3 to wait 10s
	init_timer3_10s();



	// configure timer 1 for Fast PWM mode 15
	// we will have 10s waiting and 5s serving the motor
	// make interrupt for the match that will occur after 10s to start timer0 PWM to
serve the motor
	// thus duty cycle for timer1 is 33.33%
	// since we are using mode 15 we can define our TOP to be 60000 cycle of Clk_mcu
	// 40000(10s) then sets it for 20000(5s) until counter reach the TOP and clear it
	// set output compare
	OCR1A = 60000; // TOP
	OCR1B = 40000; // 10s
	TCCR1A = (1<<WGM11) | (1<<WGM10); // WGM for setting to mode 15 also keeping COM
zeros to disconnect OC
	TCCR1B = (1<<CS12) | (1<<CS10) | (1<<WGM13) | (1<<WGM12);     // set 1024 prescaling
	TIMSK = (1<<OCIE1B) | (1<<TOIE1);// Enable interrupt on compare match and on overflow
	MOTOR_DDR = 0xFF; // make PORTB an output
	MOTOR_PORT = 0x01; // sets the direction
	// no need to activate COMM since we are not concerned of the output of OC1
	// enable interrupts
	sei();
}

void message_central(uint8_t packet){
	central_tr_buffer = packet;

	// disabling the receive and the receive interrupt
	UCSR1B &= ~((1 << RXEN1) | (1 << RXCIE1));
	// enabling the transmitting and the USART buffer empty interrupt
	UCSR1B |= (1 << TXEN1) | (1 << UDRIE1);


}

uint16_t get_sensor(uint8_t index){
	// choose the input voltage to read from
	ADMUX = (1<<REFS0)|index;
	// begin A to D and enable interrupts
	//ADCSRA |= (1<<ADSC) | (1<<ADIE) ;

	//sleep_enable();
	//sleep_cpu();


	// stop ADC conversion
	//ADCSRA &= ~((1<<ADSC) | (1<<ADIE));

	// polling
	ADCSRA |= (1<<ADSC);
	while((ADCSRA & (1<<ADIF)) == 0);

	//uint16_t value, valueL;
```

```
	//valueL = ((uint16_t)ADCL);
	//value = ((uint16_t)ADCH) * 256;
	//
	//value = value + valueL;
	//return (value);
	return ADC;
}

void read_sensors(){

	// we are using  Vref 5 volts and 10 bits resolution
	// therefore step size = 5/1024 = 4.88 mV (5/1024)
	// and digital output will be Vin/step size
	// after that we will scale each one based on its limits to become 5bits

	// calculations and equations are in the report

	uint32_t temp;

	// TEMPERATURE
	// m = 15/256
	// c = -21
	temp = get_sensor(TEMPERATURE);
	temp = (15*temp / 256) - 21;
	sensor_input_buffer[TEMPERATURE] = (uint8_t)temp;
	_delay_ms(1);

	// MOISTURE
	// m = 175/3072
	// c = -19
	temp = get_sensor(MOISTURE);
	//temp = 0.0568*temp - 19;
	temp = (175*temp / 3072) - 19 + 1;
	sensor_input_buffer[MOISTURE] = (uint8_t)temp;
	_delay_ms(1);

	// WATER
	// m = 275/2048
	// c = -51
	temp = get_sensor(WATER);
	temp = (275*temp / 2048) - 51;
	sensor_input_buffer[WATER] = (uint8_t)temp;;
	_delay_ms(1);

	// BATTERY
	// m = 75/1024
	// c = -44
	temp = get_sensor(BATTERY);
	//temp = 0.074*temp - 44;
	temp = (75*temp / 1024) - 44 + 1;
	sensor_input_buffer[BATTERY] = (uint8_t)temp;;
	_delay_ms(1);

}

void lcd_command(uint8_t command){
	LCD_CTRL &= ~((1<<RS) | (1<< RW)); // select command register and set it as writing
	LCD_PORT = command;

	// now we send H-To-L pulse on E to apply changes
	// delays might be excessive here but they are safe as shown in the lecture slides
	LCD_CTRL |= (1<<E);
```

```
	_delay_ms(1);
	LCD_CTRL &= ~(1<<E);
	_delay_ms(20);

}

void lcd_print(unsigned char* str){
	for(uint8_t i = 0; str[i]; i++){
		LCD_CTRL |= (1<<RS); //select data register
		LCD_CTRL &= ~(1<<RW); // set it for writing
		LCD_PORT = str[i];
		LCD_CTRL |= (1<<E);
		_delay_ms(1);
		LCD_CTRL &= ~(1<<E);
		_delay_ms(20);
	}

}

void lcd_display_sensor_screen(){
	unsigned char str[17];

	//clear screen
	lcd_command(LCD_CLEAR_DISPLAY);

	// put the cursor at the beginning of the first line
	lcd_command(LCD_START_LINE_1);
	sprintf(str, " T = %2x  M = %2x ", sensor_input_buffer[TEMPERATURE],
sensor_input_buffer[MOISTURE]);
	lcd_print(str);

	// put the cursor at the beginning of the second line
	lcd_command(LCD_START_LINE_2);
	sprintf(str," W = %2x  B = %2x ", sensor_input_buffer[WATER],
sensor_input_buffer[BATTERY]);
	lcd_print(str);
}

void lcd_display_battery_low(){
	//clear screen
	lcd_command(LCD_CLEAR_DISPLAY);
	lcd_command(LCD_START_LINE_1);
	lcd_print(" Change Battery \0");
	lcd_command(LCD_START_LINE_2);
	lcd_print(" Immediately :( \0");
}

uint8_t CRC3(uint8_t c){
	uint8_t new_c = c & 0xE0; //extract the first three bits of the command packet
	uint8_t temp = new_c;
	int msb;
	int i;
	for(i=0; i<3;i++)
	{
		msb = 1 << 7;

		if (temp & msb)   // if msb is set we do the xor operation
		temp =  temp ^ gen;

		temp = temp << 1;  //shift left

	}
```

```
	temp = temp >> 3;   //to shift the CRC bits so that they are in the correct position.


	new_c |= temp;   //appending the crc bits.

	return new_c;

}

uint8_t CRC3_CHECK(uint8_t command_in){
	uint8_t trueCRC= CRC3(command_in);

	return trueCRC == command_in ? 1 : 0 ; // if the data is corrupted return 0 else 1

}

uint8_t CRC11(uint8_t data){
	uint8_t temp = data;
	uint8_t c_new = Log_command & 0xE0;  //extract the first three bits of the command
packet

	uint8_t c_cpy =  c_new;

	int msb;
	int count=0;

	for(int i=0; i<11;i++){
		msb = 1 << 7;

		if (temp & msb)   // if msb is set we do the xor operation
		temp =  temp ^ gen;

		temp = temp << 1;  //shift left

		if(count!=3) {
			temp= temp + (   (c_new & msb)  >> 7);
			c_new = c_new << 1;
			count++;
		}

	}


	temp = temp >> 3;   //to shift the CRC bits so that they are in the correct position.



	c_cpy |= temp;   //appending the crc bits.

	return c_cpy;


}
```

User node:

```
/* Include statements*/
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <stdio.h> // for sprintf
#include <stdlib.h>
#include <string.h>
#include <util/delay.h>
#include <avr/wdt.h>
#include <avr/eeprom.h>


/* Define statements */
#define F_CPU  4000000UL
#define BR_Calc  0x19



// Variables and ports definitions
#define LCD_PORT PORTB   //writes to LCD
#define LCD_DDR  DDRB
#define LCD_CTRL PORTA   // for RS, RW and E
#define LCD_CTRL_DIR DDRA
#define KEY_PORT PORTC
#define KEY_DIR      DDRC
#define KEY_PIN      PINC   //for user input to KEYPAD


#define ROW1     0xFE
#define ROW2     0xFD
#define ROW3     0xFB
#define ROW4       0xF7
#define LINEONE    0x80
#define LINETWO    0xC0
#define LINETHREE  0x94
#define LINEFOUR   0xD4

#define LCD_INIT_8BIT 0x38 // 2 Lines and 5x7 matrix (D0-D7, 8-bit)
#define LCD_DISPLAY_ON_CURSOR_OFF 0x0C //Display on, cursor off
#define LCD_CLEAR_DISPLAY 0x01
#define LCD_LEFT_TO_RIGHT 0x06



#define RS 7
#define RW 6
#define E 5

#define central_rc_buffer_SIZE 100



/*Global variables*/

// to ensure not writing while reading and vice versa
uint8_t bluetooth_mutex = 0x01; // (0: mutex locked, 1: mutex unlocked)
unsigned char central_rc_buffer[central_rc_buffer_SIZE] = "";
```

```
uint8_t central_rc_ind = 0;
uint8_t line = 1;


/* functions */
void lock_bluetooth();
void unlock_bluetooth();
void central_transmit(unsigned char);
void lcd_print(unsigned char);
void sysINIT();
void lcd_command(uint8_t);
unsigned char getKeyPressed();
void lcd_choose_line();
void display_message();


int main(){
	sysINIT();

	while(1){
		sleep_enable();
		sleep_cpu();
	}

	return 0;

}


//keypad interrupt
ISR(INT0_vect){
	sleep_disable(); // disable sleep once an interrupt wakes CPU up
	// get row and column from keypad



	unsigned char userKey = getKeyPressed();
	KEY_PORT = 0x0F;

	// Send character to main logger

	central_transmit(userKey);
	central_transmit('.');

}

// the user interface module is getting data from the data logger module
ISR(USART0_RX_vect){
	sleep_disable(); // disable sleep once an interrupt wakes CPU up

	// bluetooth mutex is locked and user buffer didn't finish yet
	if(!bluetooth_mutex){
		central_rc_buffer[central_rc_ind] = UDR0;
		if( central_rc_buffer[central_rc_ind] == '\0'|| central_rc_ind ==
central_rc_buffer_SIZE){
				unlock_bluetooth();
				return;
		}
		central_rc_ind++;
	}

}
```

```
void lcd_print(unsigned char ch){
	LCD_CTRL |= (1<<RS); //select data register
	LCD_CTRL &= ~(1<<RW); // set it for writing

	LCD_PORT = ch;

	LCD_CTRL |= (1<<E);
	_delay_ms(1);
	LCD_CTRL &= ~(1<<E);
	_delay_ms(15);
}

void sysINIT(){


	// initialize LCD
	LCD_CTRL_DIR = 0xE0;
	LCD_DDR = 0xFF; // D[7:0] of LCD
	_delay_ms(20); // documentation says +15ms
	lcd_command(LCD_INIT_8BIT); // make it 8 bit mode and 2 lines
	lcd_command(LCD_DISPLAY_ON_CURSOR_OFF); // display on cursor off (or cursor on with
0x0E)
	lcd_command(LCD_LEFT_TO_RIGHT);
	lcd_command(LCD_CLEAR_DISPLAY);
	lcd_choose_line(line);


	UBRR0H = (BR_Calc >> 8);
	UBRR0L = BR_Calc;
	UCSR0C = (1 << UCSZ00) | (1 << UCSZ01);
	lock_bluetooth();

	//these depend on the connections
	//to enable input from keypad
	KEY_DIR = 0x0F;
	KEY_PORT = 0x0F;

	// keypad interrupt
	EIMSK = 0x01;
	EICRA |= (1<<1);

	sei();
}

void lock_bluetooth(){
	// mutex is locked
	// means last transmition didn't finish yet

	// lock mutex to send data
	bluetooth_mutex = 0; // locked and we can receive the entire buffer
	UCSR0B &= ~((1 << TXEN0) | (1 << UDRIE0)); // disabling
	UCSR0B |= (1 << RXEN0) | (1 << RXCIE0);
}

void unlock_bluetooth(){
	bluetooth_mutex = 1;// unlock the mutex
	central_rc_ind = 0;
	// disable receiving interrupt from the user and enabling only transmitting
	UCSR0B &= ~((1 << RXEN0) | (1 << RXCIE0));
	UCSR0B |= (1 << TXEN0) | (1 << UDRIE0);
```

```
	display_message();
}

void central_transmit(unsigned char data_packet){
	UCSR0B &= ~((1 << RXEN0) | (1 << RXCIE0));
	UCSR0B |= (1 << TXEN0) | (1 << UDRIE0);
	while(!(UCSR0A		& (1<<UDRE0)));
	UDR0= data_packet;

}

void lcd_command(uint8_t command){
	LCD_CTRL &= ~((1<<RS) | (1<< RW)); // select command register and set it as writing
	LCD_PORT = command;

	// now we send H-To-L pulse on E to apply changes
	// delays might be excessive here but they are safe as shown in the lecture slides
	LCD_CTRL |= (1<<E);
	_delay_ms(1);
	LCD_CTRL &= ~(1<<E);
	_delay_ms(15);

}

unsigned char getKeyPressed(){

	//toggle the rows one by one and check for which column is activated

	KEY_PORT = ROW1;  //make ROW1 = 0  ->grounding

	if(KEY_PIN == 0xEE ) return '1';
	else if(KEY_PIN == 0xDE) return '2';
	else if(KEY_PIN == 0xBE) return '3';

	KEY_PORT = ROW2;  //make ROW2 = 0  ->grounding
	if(KEY_PIN == 0xED ) return '4';

	else if(KEY_PIN == 0xDD) return '5';
	else if(KEY_PIN == 0xBD) return '6';

	KEY_PORT =  ROW3;   // make ROW3= 0  -> grounding

	if(KEY_PIN == 0xEB ) return '7';
	else if(KEY_PIN == 0xDB) return '8';
	else if(KEY_PIN == 0xBB) return '9';

	KEY_PORT = ROW4;  //make ROW4=0  -> grounding

	if(KEY_PIN == 0xE7) return '*';
	else if(KEY_PIN == 0xD7) return '0';
	else if(KEY_PIN == 0xB7) return '#';

	return 0;
}

void lcd_choose_line(){
	if (line == 1)lcd_command(LINEONE);
	else if (line == 2)lcd_command(LINETWO);
	else if (line == 3)lcd_command(LINETHREE);
	else if (line == 4) lcd_command(LINEFOUR);
	else{
```

```
                lcd_command(LCD_CLEAR_DISPLAY);
                line =0;
        } // display is full so clear
        line++;
}

void display_message(){

        cli();

        unsigned char ch;

        for(uint8_t i = 0; central_rc_buffer[i]; i++){
                ch = central_rc_buffer[i];
                if(ch == '\n'){
                 lcd_choose_line();
                 continue;
                }


                lcd_print(ch);
        }

        lock_bluetooth();
        sei();

}
```

Updated Central

Updates are minimal, for example changed the messages a little bit to fit in the display LCD. We also added delays for between transmitting big messages to give time for the LCD to display, and we changed the receive from the user since the it will receive the entire packet directly from the sensor not two ASCII characters for each packet from terminal like in module3.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <stdio.h> // for sprintf
#include <stdlib.h>
#include <string.h>
#include <util/delay.h>
#include <avr/wdt.h>
#include <avr/eeprom.h>



#define F_CPU 4000000UL
#define BR_Calc 0x19

#define gen 0xD4

#define IMEM_START 0x0800
#define IMEM_END (RAMEND- 0x100) // to reserve 128bytes for stack: 0x10FF - 0x800 = 0x08FF
//4351 - 2048 = 2303 bytes from internal SRAM for the logger
#define XMEM_START (RAMEND+1) // 0x1100
```

```
#define XMEM_END 0X18FF // because we are using 2KB external memory

// commands
#define Reset_command 0x00
#define Acknowledge 0x40
#define Log_command 0x20
#define Error_repeat 0x60

// Maximum buffer sizes for transmission:
#define USER_TR_BUFFER_SIZE 100

// messages indexes
#define USER_MENU 0
#define SENSOR_RST_MESSAGE 1
#define INVALID_OPTION 2
#define NOTHING_TO_SHOW 3
#define MEM_DUMP_START 4
#define MEM_DUMP_END 5
#define LAST_ENTRY_START 6
#define LAST_ENTRY_END 7
#define INIT_DONE 8
#define MASTER_WD_MENU 9
#define SLAVE_WD_MENU 10
#define SLAVE_TIMER_EXPIRED 11

// global variables

const unsigned char messages [][USER_TR_BUFFER_SIZE] = {
       "1-Memory Dump\n2-Last Entry\n3-Restart\0",
       "Sensor reset!\n\0",
       "Invalid option!\nChoose again\0",
       "Nothing to show\nlog file is empty\0",
       "Memory dump:\n",
       "Dump finished\0",
       "Last Entry: \n",
       "Finished\0",
       "Init Done\n",
       "Master Watchdog:\n1-30ms\n2-250ms\n3-500ms\0",
       "Slave Watchdog:\n1-11s\nB-13s\n3-15s\0",
       "Timer Expired!\0"
};

uint8_t TOS;
uint8_t is_tos_empty = 1; // 1 empty, 0 full
uint8_t* logFile = IMEM_START;
// to ensure not writing while reading and vice versa
uint8_t bluetooth_mutex = 0x01; // (0: mutex locked, 1: mutex unlocked)

// USART0 communication buffers and indexes
unsigned char user_tr_buffer[USER_TR_BUFFER_SIZE] = "";
uint8_t user_tr_ind = 0;
unsigned char user_rc_buffer;// because will only hold one character option

// USART1 communication buffers
uint8_t sensor_tr_buffer;



// functions prototypes
void initialize();
void usart_init();
void init_sensor();
```

```
void memory_dump();
void last_entry();
void user_transmit(unsigned char);
void sensor_transmit(uint8_t);
void repeat_request();
void log_request();

void message_user(uint8_t);

// WD functions
void ConfigMasterWD();
void ConfigSlaveWD();
void slave_wdt_reset();

// CRC functions
uint8_t CRC3(uint8_t);
uint8_t CRC11(uint8_t);
uint8_t CRC3_CHECK(uint8_t);
uint8_t CRC11_CHECK(uint8_t);

// helper functions
void pushTOS(uint8_t);
void popTOS();
void log_data(uint8_t);
uint8_t is_data_packet(uint8_t);
uint8_t log_file_empty();
void lock_bluetooth();
void unlock_bluetooth();

void display_data(unsigned char*);


uint8_t str_to_hex(unsigned char*); // for accepting inputs from sensor
void hex_to_str(uint8_t, unsigned char *); //  for user transmitting

int main(){

	initialize();

	while(1){
		//wdt_reset();
		sleep_enable();
		sleep_cpu();
	}

	return 0;
}

// check for user buffer when transmitting is completed
ISR(USART0_UDRE_vect){
	sleep_disable();
	// bluetooth mutex is locked and user buffer didn't finish yet
	if(!bluetooth_mutex){
		user_transmit(user_tr_buffer[user_tr_ind]);

		if( user_tr_buffer[user_tr_ind] == '\0'|| user_tr_ind == USER_TR_BUFFER_SIZE){
			unlock_bluetooth();
			return;
		}
		user_tr_ind++;
		_delay_ms(50);
	}
```

```
}

// receiving input from user
ISR(USART0_RX_vect){
	sleep_disable();

	// check for period to know that the previous input was the option
	while (!(UCSR0A & (1 << RXC0)));
	unsigned char input = UDR0;

	// check for period to know that the previous input was the option
	if(input != '.'){
		user_rc_buffer = input;
		return;
	}


	sei();
	// here the input was dot and we want to check the user input
	switch(user_rc_buffer){
		// memory dump
		case '1':
		memory_dump();
		break;

		case '2':
		last_entry();
		break;

		case '3':
		// reset sensor
		init_sensor();
		break;

		default:
		// invalid option
		message_user(INVALID_OPTION);
	}

}

ISR(USART1_UDRE_vect){
	sleep_disable();

	while(!(UCSR1A & (1<<UDRE1)));
	UDR1 = sensor_tr_buffer;
	_delay_ms(10);


	UCSR1B &= ~((1 << TXEN1) | (1 << UDRIE1));
	UCSR1B |= (1 << RXEN1) | (1 << RXCIE1);

}

// receiving input from sensor
ISR(USART1_RX_vect){
	sleep_disable();

	uint8_t packet_in = UDR1;
```

```
	// if packet in is data packet
	if(is_data_packet(packet_in)){
		pushTOS(packet_in);
		return;
	}
	// packet_in is command packet
	if(is_data_packet(TOS)){
		// if check fails empty the TOS, send repeat request and return
		if(!CRC11_CHECK(packet_in)){
			popTOS();
			repeat_request();
			return;
		}
		// crc11 passed
		log_request();
		return;
	}

	// no data in TOS do crc3 check
	// check if it fails crc3
	if(!CRC3_CHECK(packet_in)){
		popTOS();
		repeat_request();
		return;
	}

	// mask the 7th and the 6th bit
	if((packet_in & 0x60) == Acknowledge){
		popTOS();
		return;
	}
	else if((packet_in & 0x60) == Error_repeat){
		if(is_tos_empty)return;

		// if TOS is not empty
		sensor_transmit(TOS);
	}


}

// slave watchdog will use timer1
ISR(TIMER1_COMPA_vect){
	sleep_disable();
	sei();
	TIMSK &= ~(1<<OCIE1A);

	message_user(SLAVE_TIMER_EXPIRED);
	init_sensor();
	ConfigSlaveWD();
}


uint8_t is_data_packet(uint8_t packet){
	return packet & 0x80;
}

void pushTOS(uint8_t value){
	TOS = value;
	is_tos_empty = 0;
}
```

```
void popTOS(){
	is_tos_empty=1;
}

void usart_init(){

	// User USART0
	// setting the Baud rate
	UBRR0H = (uint8_t)(BR_Calc >> 8); // loading the most significant byte
	UBRR0L = (uint8_t)BR_Calc;
	//setting the width to 8 bits
	UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);


	// Sensor USART1
	UBRR1H = (uint8_t)(BR_Calc >> 8);
	UBRR1L = (uint8_t)BR_Calc;
	UCSR1C = (1 << UCSZ11) | (1 << UCSZ10);


}

void init_sensor(){
	pushTOS(CRC3(Reset_command));
	sensor_transmit(TOS);
	message_user(SENSOR_RST_MESSAGE);
}

void initialize(){

	MCUCR = (1 << SRE); // External memory enable
	XMCRB = (1<<XMM2)|(1<<XMM0); // release C pins that are not needed

	// enable USART communications
	usart_init();

	// set global interrupt enable
	sei();

	// enable watchdog timers
	//ConfigMasterWD(); // master will wdt_enable at the end is commented
	//ConfigSlaveWD(); // for sensor: diabled for testing


	UCSR1B &= ~((1 << TXEN1) | (1 << UDRIE1));
	UCSR1B |= (1 << RXEN1) | (1 << RXCIE1);
	// initialize sensor
	init_sensor();

	// show menu to the user
	message_user(USER_MENU);
}

void log_request(){
	*(logFile) = TOS;
	logFile++;
	if(logFile > XMEM_END) logFile = IMEM_START; // reset to the beginning
	else if(logFile> IMEM_END) logFile = XMEM_START; // skip stack

	pushTOS(CRC3(Acknowledge));

	sensor_transmit(TOS);
```

```
}

uint8_t log_file_empty(){
	// nothing to show
	if(logFile == IMEM_START){
		message_user(NOTHING_TO_SHOW);
		return 1;
	}

	return 0;
}

void lock_bluetooth(){
	// mutex is locked
	// means last transmition didn't finish yet
	while(!bluetooth_mutex){
		sleep_enable();
		sleep_cpu();
	}

	// lock mutex to send data
	bluetooth_mutex = 0; // locked and we can send the entire buffer
	// disable receiving interrupt from the user and enabling only transmitting
	UCSR0B &= ~((1 << RXEN0) | (1 << RXCIE0));
	UCSR0B |= (1 << TXEN0) | (1 << UDRIE0);
}

void unlock_bluetooth(){
	bluetooth_mutex = 1;// unlock the mutex
	user_tr_ind = 0;

	UCSR0B &= ~((1 << TXEN0) | (1 << UDRIE0)); // disabling
	UCSR0B |= (1 << RXEN0) | (1 << RXCIE0);
}

void message_user(uint8_t message_index){
	_delay_ms(300);
	strcpy(user_tr_buffer, messages[message_index]);
	user_tr_ind = 0;
	lock_bluetooth();

}

void display_data(unsigned char* data){
	_delay_ms(300);
	strcpy(user_tr_buffer, data);
	user_tr_buffer[2] = ' ';
	user_tr_buffer[3] = '\0';
	user_tr_ind = 0;


	lock_bluetooth();
}

void memory_dump(){

	if(log_file_empty()) return;

	//message_user(MEM_DUMP_START);
	uint8_t* ptr = IMEM_START;

	unsigned char temp[4] = "";
```

```
	while(ptr< logFile){
		hex_to_str(*ptr, temp);

		display_data(temp);
		ptr++;
		if(ptr > IMEM_END) ptr = XMEM_START; // skip the stack
	}

	//message_user(MEM_DUMP_END);



}

void last_entry(){
	if(log_file_empty()) return;

	//message_user(LAST_ENTRY_START);
	unsigned char temp[4] = "";
	hex_to_str(*(logFile-1), temp);

	display_data(temp);

	//message_user(LAST_ENTRY_END);

}

void repeat_request(){
	uint8_t out = CRC3(Error_repeat);
	sensor_transmit(out);
}

void user_transmit(unsigned char ch){
	while(!(UCSR0A & (1<<UDRE0)));
	UDR0 = ch;
}

void sensor_transmit(uint8_t packet_out){

	sensor_tr_buffer = packet_out;

	UCSR1B &= ~((1 << RXEN1) | (1 << RXCIE1));
	UCSR1B |= (1 << TXEN1) | (1 << UDRIE1);

}

uint8_t str_to_hex(unsigned char* buffer){
	return strtol(buffer,NULL, 16);
}

void hex_to_str(uint8_t hex, unsigned char * buffer){
	sprintf(buffer,"%x",hex);
	buffer[2]=' ';
	buffer[3]='\0';
}

uint8_t CRC3(uint8_t c){
	uint8_t new_c = c & 0xE0; //extract the first three bits of the command packet
	uint8_t temp = new_c;
	int msb;
```

```
	int i;
	for(i=0; i<3;i++)
	{
		msb = 1 << 7;

		if (temp & msb)   // if msb is set we do the xor operation
		temp =  temp ^ gen;

		temp = temp << 1;  //shift left

	}

	temp = temp >> 3;   //to shift the CRC bits so that they are in the correct position.


	new_c |= temp;   //appending the crc bits.

	return new_c;

}

uint8_t CRC3_CHECK(uint8_t command_in){
	uint8_t trueCRC= CRC3(command_in);

	return trueCRC == command_in ? 1 : 0 ; // if the data is corrupted return 0 else 1

}

uint8_t CRC11(uint8_t command_in){
	//get upper byte from TOS
	uint8_t data = TOS;
	uint8_t temp = data;
	uint8_t c_new = command_in & 0xE0;  //extract the first three bits of the command
packet

	uint8_t c_cpy =  c_new;

	int msb;
	int count=0;

	for(int i=0; i<11;i++){
		msb = 1 << 7;

		if (temp & msb)   // if msb is set we do the xor operation
		temp =  temp ^ gen;

		temp = temp << 1;  //shift left

		if(count!=3) {
			temp= temp + (   (c_new & msb)  >> 7);
			c_new = c_new << 1;
			count++;
		}

	}


	temp = temp >> 3;   //to shift the CRC bits so that they are in the correct position.



	c_cpy |= temp;   //appending the crc bits.
```

```
	return c_cpy;


}

uint8_t CRC11_CHECK(uint8_t command_in){
	uint8_t trueCRC = CRC11(command_in);

	return trueCRC == command_in ? 1 : 0  ;   // if the data is corrupted return 0 else 1

}

void ConfigMasterWD(){

	uint16_t temp = eeprom_read_word((const uint16_t*)0x50);

	uint16_t config = 'C';

	if(temp == 0xFFFF){
		message_user(MASTER_WD_MENU);
		_delay_ms(10);

		// take input from user using polling
		uint8_t input = 0;

		cli(); // disable interrupts to so we don't mix things with user menu
		UCSR0B |= (1 << RXEN0);
		while (1){
			while(input != '.'){
				config = input;
				while(!(UCSR0A & (1<<RXC0)));
				input = UDR0;
			}
			if(!(config >= '1' && config <= '3')){
				sei();
				message_user(INVALID_OPTION);
				cli();
				config='3';
			}
			else{
				break;
			}
		}

		eeprom_write_word((const uint16_t*)0x50, config);

		UCSR0B &= ~(1 << RXEN0);



	}

	else {
		config = temp;
	}

	uint8_t duration;
	switch(config){
		// configure master watch dog
		case '1':
		duration = WDTO_30MS;
```

```
                break;

                case '2':
                duration = WDTO_250MS;
                break;

                case '3':
                duration = WDTO_500MS;
                default:
                duration = WDTO_500MS;
        }

        wdt_reset();
        //wdt_enable(duration);

        sei(); // enable interrupts again

}

void ConfigSlaveWD(){

        uint16_t temp = eeprom_read_word((const uint16_t*)0x0006);

        uint16_t config = 'C';

        if(temp == 0xFFFF){
                message_user(SLAVE_WD_MENU);
                _delay_ms(10);

                // take input from user using polling
                uint8_t input = 0;

                cli(); // disable interrupts to so we don't mix things with user menu
                UCSR0B |= (1 << RXEN0);
                while (1){
                        while(input != '.'){
                                config = input;
                                while(!(UCSR0A & (1<<RXC0)));
                                input = UDR0;
                        }
                        if(!(config >= '1' && config <= '3')){
                                sei();
                                message_user(INVALID_OPTION);
                                cli();
                                config='3';
                        }
                        else{
                                break;
                        }
                }

                eeprom_write_word((const uint16_t*)0x0006, config);

                UCSR0B &= ~(1 << RXEN0);
                sei(); // enable interrupts again



        }

        else {
                config = temp;
```

```
        }
        // using pre-scaler 256 so we dont have fractions
        uint16_t duration;
        switch(config){
                // configure slave watch dog
                case '1':
                duration = 44000; // 11s
                break;

                case '2':
                duration = 52000; // 13s
                break;

                case '3':
                duration = 60000; // 15s
                break;
                default:
                duration = 60000;
        }


        // reset timer
        TCNT1 = 0;
        // set output compare
        OCR1A = duration;
        // set 256 pre-scaling and activate CTC mode
        TCCR1B = (1<<CS12) | (1<<WGM12);

        // Enable interrupt on A match
        TIMSK = (1<<OCIE1A);


}

void slave_wdt_reset(){
        TCNT1 = 0;
}
```