INTRODUCTION TO MICROPROCESSORS | EMBEDDED SYSTEMS DEVELOPMENT

CNG 336

MODULE 3 REPORT

Fatma Erem Aksoy – 2315075

Ece Erseven – 2385383

3.4 DESIGN AND REPORTING

3.4.1 Preliminary Questions – Answers

a) We wanted to implement two types of interrupts in our system. The different types of interrupts used in this system are Timer interrupt and USART Receive-Transmit. The Timer interrupt is used in the point that there is no input or system hang. On the other hand, the incoming data and command bits from the user interface are handled by Receive interrupts.

b) If the MCU program stops running or runs erratically for some reason, the Watchdog timer detects where the program runs out of control or fails. Moreover, it may trigger a processor reset if needed. By constantly monitoring MCU, the Watchdog timer prevents system damage. The Watchdog timers use an RC oscillator, and the pre-scalar can be set to a value that determines the number of clock pulses. WDP[3:0] bits are used for the Watchdog Timer pre-scaling. If WDE(Watchdog Enable) is equal to 1, the Watchdog timer enables; if equal to 2, it will be disabled. To disable the watchdog timer, write logic 1 to WDCE and WDE; within the next four clock cycles, write logic 0 to WDE. Before disabling the operation starts, logic one should be written in WDE.

c) Xbee transmission power is 1mW (0dBm). 63 mW (18 dBm) for the Xbee Pro and 10 mW (10 dBm) for the international version. The HC-05 Bluetooth module supports up to 4dBm RF transmit power and works in master/slave mode. We need to check the output power graph to determine if the module consumes the maximum power when receiving or transmitting data. When receiving data, the output graph will appear positive; however, a negative graph will appear in the data transmission. HC-05 Bluetooth dissipates max power during the transmitting mode, while Xbee consumes max power in the receiving mode.

d) ADC Noise Reduction Mode: This mode stops the CPU and all I/O modules except asynchronous timer, PTC, and ADC, to minimize switching noise during ADC conversions. It's used when a high-resolution ADC measurement is required. ADC measurements are then implemented when the core is put to sleep.

Idle Mode: Idle mode enables the MCU to wake up from external interrupts as well as internal ones like the Timer Overflow and USART Transmit Complete interrupts. If wake-up from the Analog Comparator interrupt is not required, the Analog Comparator can be powered down by setting the ACD bit in the Analog Comparator Control and Status Register – ACSR. This will reduce power consumption in Idle mode.

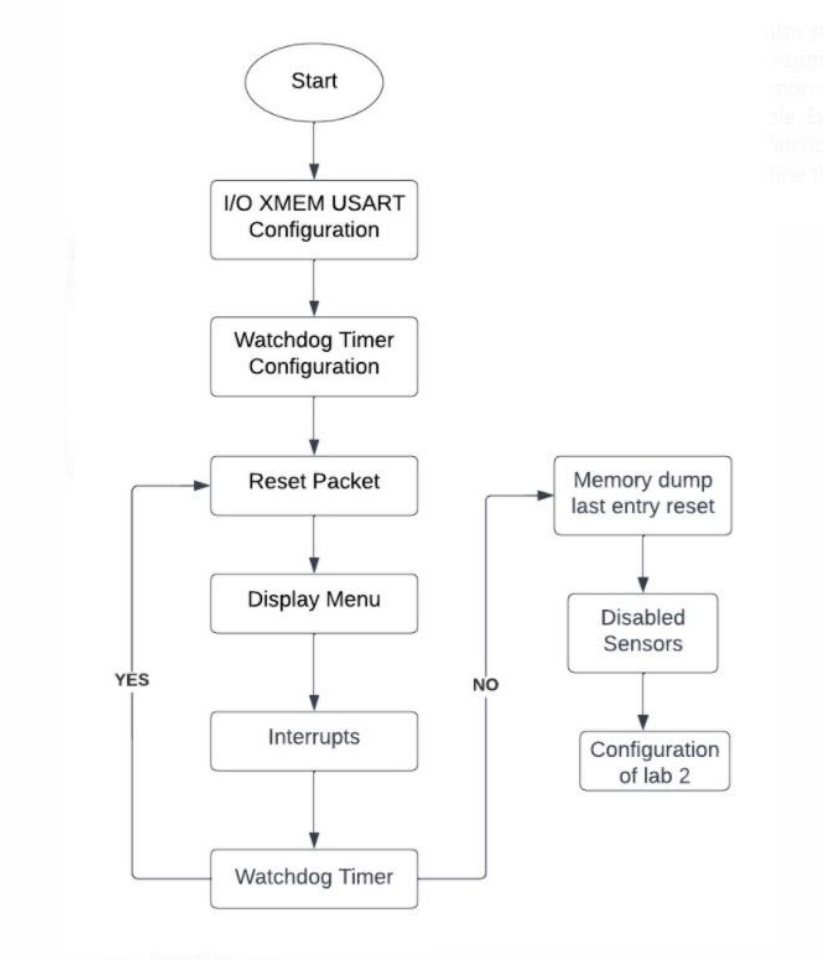
Power-Down Mode: When waking up from Power-Down mode, there is a delay from the wake-up condition occurs until the wake-up becomes effective. This allows the clock to restart and become stable after having been stopped. The wake-up period is defined by the same CKSEL Fuses that define the Reset Time-out period.

Power-save Mode: If Timer/Counter2 is not running, Power-down mode is recommended instead of Power-save mode. The Timer/Counter2 can be clocked both synchronously and asynchronously in Power-save mode. If Timer/Counter2 is not using the asynchronous clock, the Timer/Counter Oscillator is stopped during sleep. If Timer/Counter2 is not using the synchronous clock, the clock source is stopped during sleep. Even if the synchronous clock is running in Power-save, this clock is only available for Timer/Counter2.

Standby Mode: When the **SM[2:0]** bits are written to **'110'** and an external clock option is selected, the SLEEP instruction makes the MCU enter Standby mode. This mode is identical to Power-Down with the exception that the Oscillator is kept running. From Standby mode, the device wakes up in six clock cycles.

Extended Standby Mode: When the **SM[2:0]** bits are written to **'111'** and an external clock option is selected, the SLEEP instruction makes the MCU enter Extended Standby mode. This mode is identical to Power-Save mode with the exception that the Oscillator is kept running. From Extended Standby mode, the device wakes up in six clock cycles.

e)



3.4.2 Design

The Code:

main.c

interrupt.h — C:\Program Files (x86)\Atmel\Studio\7.0\toolchain\avr8\avr8-gnu-toolchain\avr\include\avr\interrupt.h

```
/*
 * main.c
 *
 * Created: 12/20/2022 10:20:03 PM
 *  Author: Erem
 */


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

#define USER_MENU "\r\rEnter choice (and period):\r1-Mem Dump  2-Last Entry  3-Restart\nChoice: \0"
#define MST_WD_MENU "\rEnter MCU WD Choice (& period):\r1-30ms\r2-250ms\r3-500ms\rChoice: \0"
#define MST_SS_MENU "\rEnter Sensors WD Choice (& period):\r1-0.5s\r2-1s\r3-2s\rChoice: \0"

// USART communication definitions at the program header
#define F_CPU 8000000
#define Baudrate 9600
#define BR_Calc ((F_CPU/16/Baudrate)-1)         //Baud rate calculator

// Maximum buffer sizes for transformation:
#define USER_TR_BUFFER_SIZE 128
#define SENSOR_TR_BUFFER_SIZE 5

// USART0 communication buffers and indexes
char user_tr_buffer [USER_TR_BUFFER_SIZE] = "";
char user_tr_index = 0;         // keeps track of character index in buffer
char new_user_read_char;            // reception one character at a time
```

```
// USART1 communication buffers and indexes
char sensor_tr_buffer [SENSOR_TR_BUFFER_SIZE] = "";
char sensor_tr_index = 0;
char new_sensor_read_char;
int TOS;
int delay_mcu_ms;
int delay_sensor_ms;
char PACKET_OUT;
char PACKET_IN;

// Memory definitions
#define MEM_START 0x500
#define STACK_LIMIT 0x10EB
#define EXTENDED_SRAM_LIMIT 0x18FF
#define EXTENDEND_SRAM_START 0x1100     // as internal SRAM finishes at 0x10FF

// Data memory pointers
int *x;         // points to the next data memory entry
int *z;         // used in memory dumps
int *y;         // can be used in memory dumps, if wished
void ReceiverUserResponse(char);
char CRC3(char);
void ConfigMasterWD();

void Enable_MCU_WD(){
    if(delay_mcu_ms==30){
        WDTCR|=(1<<WDCE)|(1<<WDE);      // setting change enable and WD enable bit to 1
        WDTCR|=(1<<WDE) |(1<<WDP0);     // setting the pre_scalar values
        WDTCR&=~(1<<WDCE);              // setting change enable to 0
    }
    else if(delay_mcu_ms==250){
        WDTCR|=(1<<WDCE)|(1<<WDE);      // setting change enable and WD enable bit to 1
        WDTCR|=(1<<WDE)|(1<<WDP2);
        WDTCR&=~(1<<WDCE);              // setting change enable to 0
    }
```

```
    else {
        WDTCR|=(1<<WDCE)|(1<<WDE);      // setting change enable and WD enable bit to 1
        WDTCR|=(1<<WDE)|(1<<WDP2)|(1<<WDP0);
        WDTCR&=~(1<<WDCE);              // setting change enable to 0
    }
}

// this func puts the MCU to sleep and waits for interrupts
void Sleep_and_Wait()
{
    sleep_enable();     // arm sleep mode
    sei();              // global interrupt enable
    sleep_cpu();        // put CPU to sleep
    sleep_disable();    // disable sleep once an interrupt wakes CPU up
}

// Energy saving while transmitting everything in a transmit buffer sleeping, instead of polling
void UserBufferOut(void)
{
    char i = 0;
    UCSR0B |= (1 << TXEN0) | (1 << TXCIE0);         // enabling the TX to the sensor (USART1 TXen and TX complete interrupt)
    while (user_tr_index > 0)
    {
        user_tr_index--;
        while(!(UCSR0A & (1 << UDRE0)));            // waiting
        UDR0 = user_tr_buffer[i++];
        Sleep_and_Wait();
    }
}

void last_entry()
{
    x = z;      // initializing a pointer to where we last did write
```

```
    if(z==MEM_START)        // checking if the address is the start of memory,
    {
        z = EXTENDED_SRAM_LIMIT;        // then going back to the extended SRAM limit (as we did z++ after writing)
    }
    else if(z==EXTENDEND_SRAM_START)        // if already at the extended start address,
    {
        z = STACK_LIMIT-1;              // then  going to the address of stack start - 1 (0x10EB -1)
    }
    else {z -=1;}       // decrementing the z value
    strcpy(user_tr_buffer,"\rLast Entry: ");

    char buffer[10];
    sprintf(&buffer,"%X",*z);
    strcat(user_tr_buffer,buffer);
    user_tr_index = 23;
    z = x;      // setting the value z to the value where we last did write
    //Disable_MCU_WD();
    UserBufferOut();
}

void mem_dump()
{
    new_user_read_char = "";
    x = MEM_START;      // starting from the address of memory start
    strcpy(user_tr_buffer,"\rMEM Dump:\r");
    user_tr_index = 11;
    UserBufferOut();
    _delay_ms(1000);

    UCSR0B= ((1<<RXEN0) | (1<<RXCIE0));         // enabling the receive and its interrupts to see if the user is pressed '.'

    while(1)
    {
        Enable_MCU_WD();        // enabling the master watchdog
```

```
        if(x==STACK_LIMIT)      // if x is at the stack limit, then making it the extended limit start
        {
            x = EXTENDED_SRAM_LIMIT;
        }
        else if(x==EXTENDED_SRAM_LIMIT+1)       // round robin
        {
            x = MEM_START;
        }
        if(new_user_read_char=='.')         // stop when user enters .
        {
            //Disable_MCU_WD();             // disabling the master watchdog
            break;
        }

        char buffer [10];       // displaying memory contents
        sprintf(&buffer,"%x: %X\r" ,x,*x);
        strcat(user_tr_buffer,buffer);

        user_tr_index = 9;
        //_delay_ms(9999);

        x++;
        //Disable_MCU_WD();
        UserBufferOut();
        _delay_ms(1000);
    }
}

// this func sends and prints a text message to User screen to ask for remote sensor watchdog timer delay
void ConfigSlaveWD(){
    strcpy(user_tr_buffer, MST_SS_MENU);
    user_tr_index=63;
    UserBufferOut();            // choosing sensor delay by sending the menu to the user
    ReceiverUserResponse(2);    // waiting for response from the user

    while(EECR&(1<<EEWE));      // waiting for EEPROM to finish the last write operation

    // setting address pointers
    EEARL=0x00;
    EEARL=0x04;
    EEDR=delay_sensor_ms;       // writing the EEPROM delay low byte
    EECR|=(1<<EEWE);        // enabling the master write
    EECR|=(1<<EEWE);        // enabling the write

    while(EECR&(1<<EEWE));
    EEARL=0x03;
    EEDR=(delay_sensor_ms>>8);      // writing the EEPROM delay high byte
    EECR|=(1<<EEWE);        // enabling the master write
    EECR|=(1<<EEWE);        // enabling the write
}

void WD()
{
    while (EECR & (1<< EEWE));      // waiting for EEPROM to finish the last write operation

    // setting address pointers
    EEARH=0x00;
    EEARL=0x01;
    EECR=(1<<EERE);     // setting read enable

    if(EEDR==0xFF){
        EEARL=0x02;     // setting address pointer to the next byte
        if(EEDR==0xFF){
            ConfigMasterWD();       // calling the config master WD func
        }
    }
    else{
        delay_mcu_ms=0x100* EEDR;   // putting high byte into delay
        EEARL=0x02;             // setting address pointer to next byte
        delay_mcu_ms+=EEDR;         // putting lower byte into delay
    }

    while(EECR & (1<<EEWE));    // waiting for EEPROM to finish the last write operation
```

```
    // setting address pointers
    EEARH=0x00;
    EEARL=0x03;
    EECR |=(1<<EERE);           // setting read enable

    if(EEDR==0xFF){
        EEARL=0x04;             // set address pointer to next byte
        if(EEDR=0xFF){
            ConfigSlaveWD();        // calling the config slave WD func
        }
    }
    else{
        delay_sensor_ms=0x100* EEDR;    // putting high byte into delay
        EEARL=0x04;                 // setting address pointer to next byte
        delay_sensor_ms+=EEDR;          // putting lower byte into delay
    }
}

void SensorBufferOut(void)
{
    unsigned char i = 0;
    UCSR1B |= (1 << TXEN1) | (1 << TXCIE1);         // enabling the TX to the sensor (USART1 TXen and TX complete interrupt)
    while (sensor_tr_index > 0)
    {
        sensor_tr_index--;
        while(!(UCSR1A & (1 << UDRE1)));
        UDR1 = sensor_tr_buffer[i++];
        Sleep_and_Wait();
    }
    //Enable_Sensors_WD();
}

// this func transmits the packet_out
void transmit(unsigned char packet)
{
    sensor_tr_buffer[sensor_tr_index++] = packet;       // sending the packet to the sensor
    SensorBufferOut();
    _delay_ms(100);
}

void initialize_io(){
    z=MEM_START;        // initializing the memory start address (both x and z) to 0x500
    x=MEM_START;
    MCUCR=0x80;         // Setting XMEM SRE to 1
    XMCRB=0x05;         // pinC 0-2 are reserved for external memory addressing
    UCSR1C=0x06;        // asynchronous mode with 8-bit data frame, no parity and 1 stop bit
    UBRR1H=0;           // setting the baud rate
    UBRR1L= BR_Calc;
    UCSR0C=0x06;        // asynchronous mode with 8-bit data frame, no parity and 1 stop bit
    UBRR0H=0;           // setting the baud rate
    UBRR0L=BR_Calc;
    DDRD=0x01;          // chip enable
    UCSR1B |= (1<<RXEN1) |(1<<RXCIE1);      // enable RX sensor (USART1 RXen and RX complete interrupt)
}

void init()     // Repeat Request func
{
    PACKET_OUT = 0x00;      // Sending reset request to the sensor
    PACKET_OUT = CRC3(PACKET_OUT);
    transmit(PACKET_OUT);
    strcpy(user_tr_buffer, "\r\rReset Request sent to the sensors.\r");

    // notifying the user that the reset request is sent
    user_tr_index=36;
    UserBufferOut();
    TOS = PACKET_OUT;
}
```

```
void restart()
{
    // going back to the start point
    initialize_io();
    WD();
    init();
}


void ReceiverUserResponse(char fun_num)
{
    _delay_ms(1000);
    UCSR0B = ((1<<RXEN0) | (1<<RXCIE0));    // enabling RX and RXC interrupts
    char i = 0;
    char Choice;
    new_user_read_char = "";

    while (new_user_read_char !='.')
    {
        Choice = new_user_read_char;
        Sleep_and_Wait();           // making the MCU sleep and wait for the interrupts
    }

    UCSR0B &= ~((1 << RXEN0) | (1 << RXCIE0));      // disabling the interrupt

    if(fun_num==1)      // checking the choice from the user menu
    {
        switch(Choice)
        {
            case '1': mem_dump();
            break;      // go to memory dump and enable master watchdog
            case '2': Enable_MCU_WD(); last_entry();
            break;      // enable master watchdog and then go to last
            case '3': restart();
            break;      //restart
            default: strcpy(user_tr_buffer,"\rError. Enter again: ");
            user_tr_index=21;
            UserBufferOut();
            ReceiverUserResponse(1);
        }
    }
    else        // the case where the choice is from the sensor delay or user delay
    {
        switch(Choice)
        {
            // if user delay equals to 0, then store it in the user delay: else store in sensor delay
            case '1': (fun_num == 0) ? (delay_mcu_ms = 30) : (delay_sensor_ms = 500);
            break;
            case '2': (fun_num == 0) ? (delay_mcu_ms = 250) : (delay_sensor_ms = 1000);
            break;
            case '3': (fun_num == 0) ? (delay_mcu_ms = 500) : (delay_sensor_ms = 2000);
            break;
            default: strcpy(user_tr_buffer,"\rError. Enter again: ");
            user_tr_index=21; UserBufferOut();
            ReceiverUserResponse(fun_num);
        }
    }
}

 // if the first 2-byte entry in EEPROM is 0xFFFF, EEPROM content is set high, this func sends a text message to User screen to ask for MCU watchdog timer delay
void ConfigMasterWD(){
    strcpy(user_tr_buffer, MST_WD_MENU);
    user_tr_index = 65;
    UserBufferOut();            // sending user delay menu to the user
    ReceiverUserResponse(0);    // waiting for the user to enter a delay value

    while (EECR&(1<<EEWE));     // waiting for EEPROM to finish the last write operation

    // setting address pointers
    EEARH=0x00;
```

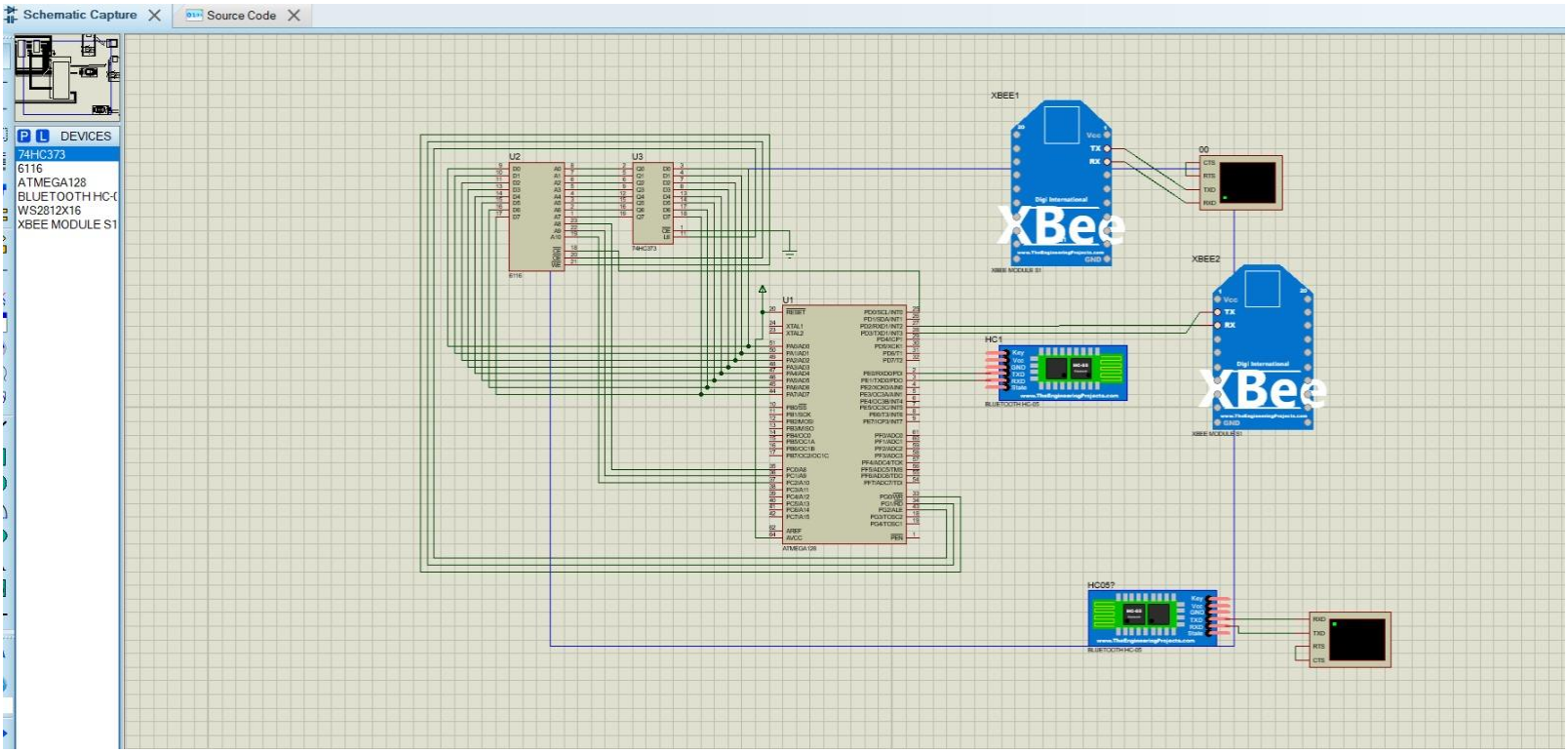
```
    EEARL=0x02;
    EEDR=delay_mcu_ms;      // writing the EEPROM delay low byte
    EECR|=(1<<EEWE);        // enabling the master write
    EECR|=(1<<EEWE);        // enabling the write

    while(EECR&(1<<EEWE));
    EEARL=0x01;
    EEDR=(delay_mcu_ms>>8);     // writing the EEPROM delay high byte
    EECR|=(1<<EEWE);            // enabling the master write
    EECR|=(1<<EEWE);
}

void Disable_MCU_WD(){
    WDTCR|=(1<<WDCE);
    WDTCR=0x00;         // turning the WD timer off
}

void Enable_Sensors_WD(){
    unsigned int timer_clock = F_CPU/256;       // calculating the timer 1 counter clock value
    unsigned int timer_period = 1/timer_clock;
    int value;
    if(delay_sensor_ms==500)
    value=65536- (0.5)/timer_period;
    else if(delay_sensor_ms==1000)
    value=65536- (1)/timer_period;
    else
    value=65536-(2)/timer_period;
    TCNT1=value;            // setting the TCNT initial value
    TCCR1B=(1<<CS12);       // assigning pre_scaler to 256
    TIMSK=(1<<TOIE1);       // enabling the interrupt
    Sleep_and_Wait();
}
void Disable_Sensors_WD(){
    TIMSK&=~(1<<TOIE1);     // disabling the interrupt
}

char CRC3(char packet)
{
    char temp = packet;
    char G = 0xD4;          // shifted version of G(53) by 2 which becomes 0xD4
    int i;

    for(i=0;i<3;i++)
    {
        if((temp & 0x80))       // checking if MSB of data has a 0
        {
            temp ^=G;           // XORing temp with G (generator)
        }
        temp = temp<<1;         // shifting temp by 1 to left
    }

    // shifting temp value right 3 times to make CRC result fit into 5 bits
    temp = temp >>1;
    temp = temp >>1;
    temp = temp >>1;
    packet |= temp;         // putting CRC result at the end of the current packet
    return packet;
}

ISR(TIMER1_OVF_vect){
    Disable_Sensors_WD();
    init();         // reinitializing the sensor
}

char CRC_CHECK11(){
    int G = 53<<10; // shifted version of G(53) by 10 which becomes 0xD400
    int temp;
    temp = 0x100* TOS;
    temp += PACKET_IN;          // storing new coming packet_in (the second one) in the temp too

    for(char i=0;i<11;i++){
        if((temp & 0x8000)){        // checking if the MSB of data has a 0
            temp ^= G;      // XORing temp with G
        }
        temp = temp <<1;    // shifting temp to left
    }

    if(temp == 0){          // if the check is passed, tenmp==0, return 1
        return 1;
    }
    return 0;
}
```

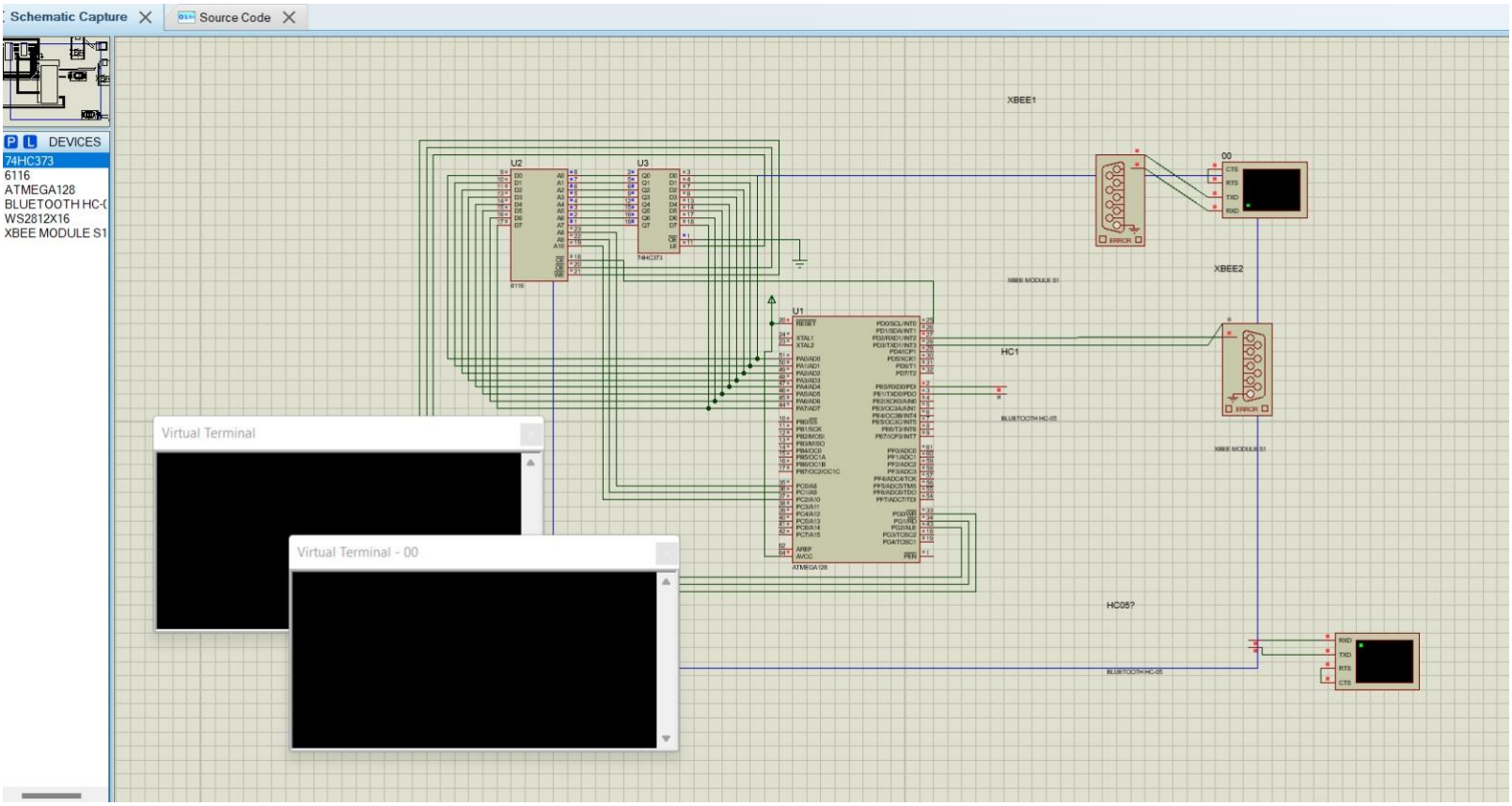
```
char CRC_CHECK3()
{
    char G = 53 << 2;       // shifted version of G(53) by 2 which becomes 0xD4
    char temp;
    temp = PACKET_IN;
    for (char i=0;i<3;i++)
    {
        if((temp & 0x80))       // checking if the MSB of data has a 0
        {
            temp ^= G;          // XORing temp with G
        }
        temp = temp << 1;       // shifting temp by 1 to left
    }
    if(temp==0)         // if the check is passed, tenmp==0, return 1
    {
        return 1;
    }
    return 0;
}

void SensorTRBufferInit()
{
    // making the whole buffer 0
    char i=0;
    while(i<5)          // executing 5 times - according to the limit given at the beginning of the program
    {
        sensor_tr_buffer[i++] = 0x00;
    }
}

void UserTrBufferInit()
{
    // making the whole buffer 0
    char i = 0;
    while(i<128)        // executing 128 times - according to the limit given at the beginning of the program
    {
        user_tr_buffer[i++] = 0x00;
    }
}

void logger()
{
    if(z==STACK_LIMIT)      // skipping the part of the memory that is reserved for the stack
    {
        z = EXTENDEND_SRAM_START;
    }
    else if(&z == EXTENDED_SRAM_LIMIT+1)            // round robin
    {
        z = MEM_START;
    }

    if(z <= EXTENDED_SRAM_LIMIT && z >= EXTENDEND_SRAM_START)       // checking the address in the extended memory space
    {
        PORTD = 0x00;            // turning the chip select on
    }
    else        // otherwise; turning the chip select off
    {
        PORTD = 0x01;
    }

    *z = TOS;       // writing whatever in the TOS to the memory
    z++;            // incrementing the pointer value
    TOS = 0x40;     // sending the acknowledge signal to the sensor
    TOS = CRC3(TOS);
}

void repeat_request()
{
    PACKET_OUT = 0x60;      // sending a repeat request to the sensor, assigning the packet_out value to repeat
    PACKET_OUT = CRC3(PACKET_OUT);      // calling the CRC3 func with the packet_out
    transmit(PACKET_OUT);               // transmitting packet_out
}

void service_readout()
{
    strcpy(user_tr_buffer,USER_MENU);       // sending service readout menu to the user
    user_tr_index = 72;
    UserBufferOut();
    ReceiverUserResponse(1);
}
```

```
// processing the data packet (packet_in)
void process_packet()
{
    PACKET_IN = new_sensor_read_char;       // reading the sensor for the new value and assigning it to packet_in

    if(PACKET_IN & (1<<7))          // checking if the packet_in is data type
    {
        TOS = PACKET_IN;
        return;
    }
    else        // checking if the packet_in is command type
    {
        if(TOS&(1<<7))      // checking if the TOS has a data packet
        {
            if(CRC_CHECK11())       // then applying CRC_CHECK11 and checking if it passes or failles
            {
                if((PACKET_IN&(0x60))==0x20)        // as it passes the CRC_CHECK11, now checking if the packet_in has log request
                {
                    logger();       // if it does have a log request, then writing to the memory
                    PACKET_OUT = TOS;           // assigning packet_out value as the TOS
                    transmit(PACKET_OUT);       // and transmitting the packet_out
                }
                else        // the case where the packet_in doesn't have a log request
                {
                    // going back to the main to initialize the stack pointer
                    initialize_io();
                    WD();
                    init();
                }
            }
            else        // the case where the CRC_CHECK11 is failed
            {
                TOS = 0x00;     // popping TOS
                repeat_request();       // making the packet_out value repeat, calling the CRC3 func with the packet_out and then transmitting the packet_out
            }
        }
        else        // the case where TOS does not have a data packet
        {
            if(CRC_CHECK3())        // applying CRC_CHECK3 and checking if it's passed or failed, if pass then go inside the if statement
            {
                if((PACKET_IN&(0x60))==0x40)        // checking if packet_in has an acknowledge request
                {
                    if(TOS!=0x00)           // checking if the stack is (TOS)empty
                    {
                        TOS = 0x00;         // if not, then pop TOS
                    }
                    else                // if the stack is already empty, then go back to service readout
                    {
                        service_readout();
                    }
                }
                else
                {
                    if((PACKET_IN&(0x60))==0x40)    // checking if the packet_in has a repeat request
                    {
                        if(TOS!=0x00)           // checking if the stack is (TOS)empty, if not then execute the next lines
                        {
                            PACKET_OUT = TOS;       // initializing the packet_out value to TOS
                            transmit(PACKET_OUT);   // transmitting the packet_out
                        }
                    }
                }
            }
            else        // the case where the CRC_CHECK3 is failed
            {
                repeat_request();
            }
        }
    }
}
```

```
// Interrupt handler for USART1 when a TX is done
ISR(USART1_TX_vect)
{
    if(sensor_tr_index==0)
    {
        SensorTRBufferInit();       // re-initializing the buffer for the next one
        UCSR1B &= ~((1 << TXEN1) | (1 << TXCIE1));      // disabling the interrupt
    }
}

// Interrupt handler for USART0 when a TX is done
ISR(USART0_TX_vect)
{
    if(user_tr_index==0)
    {
        UserTrBufferInit();         // re-initializing the buffer for the next one
        UCSR0B &= ~((1 << TXEN0) | (1 << TXCIE0));      // disabling the interrupt
    }
}

// Interrupt handler for USART0 when a RX is done
ISR(USART0_RX_vect)
{
    new_user_read_char = UDR0;      // getting a value from the user
}

// Interrupt handler for USART1 when a RX is done
ISR(USART1_RX_vect)
{
    //Disable_Sensors_WD();         // disabling the slave watchdog timer
    new_sensor_read_char = UDR1;    // getting a value from the sensor
int main(void)
{
    initialize_io();        // initializing input & output ports
    WD();                   // watchdog timer master and slave
    init();                 // initializing sensors
    service_readout();  // displaying readout menu
    Sleep_and_Wait();
    while (1)
    {

    }
}
```

Proteus Design:





We have implemented the Proteus design but due to some license problems on the program itself, it throws an error and doesn't show the User Menu. It compiles fine on both Microchip Studio and Proteus, and it shows the screens (as shown on the figures above) but then gives an error and doesn't display the menu options. This problem will be fixed till the demo date.