Introduction to Microprocessors | Embedded Systems Development

EEE 347 | CNG 336

LAB MODULE #3:

HIGH-LEVEL PROGRAMMING of an EMBEDDED SYSTEM with TIMERs, INTERRUPTS and SERIAL I/O INTERFACEs

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."

Objective

In this module, we have had to incorporate timers, interrupts and serial I/O interface to our old design and update it such that the smart data logger system is more energy efficient and sustainable. We have used USARTs to implement serial interfaces for communication between the logger and remote sensor. Major objective is to use interrupts instead of polling and make use of watch dog timer for better synchronization and reliability.

Preliminary Work

a) *What are the different types of interrupts in this system and which interrupts should be allowed to occur at the same time?*

There will 5 interrupts in the system (ordered below from highest to lowest priority)

Two for each USART and one for the manual slave watchdog:

Slave watchdog:

- TIMER1_COMPA_vect: this interrupt will be services when Timer1 is equal to the values in OCR1A which will cause the system to reset.

User interface side interrupts

- USART0_UDRE_vect: serviced when USART0 data register becomes empty and its interrupt UDRIE0 is enabled via message_user() function.
- USART0_RX_vect: serviced when USART0 receives a message from the user and its interrupt RXCIE0 is enabled.

Sensor side interrupts

- USART1_UDRE_vect: serviced when USART1 data register becomes empty and its interrupt UDRIE1 is enabled via sensor_transmit () function.
- USART1_RX_vect: serviced when USART1 receives a packet from the sensor and its interrupt RXCIE1 is enabled.

Other interrupts would be the Master watchdog, and even though its configured is enable function is commented as asked in the manual.

The interrupt that we allowed other interrupts to occur with is USART0_RX_vect so can print on the virtual terminal while treating the user request with USART0_UDRE_vect, also we alloed interrupts to happen for TIMER1_COMPA_vect because again we are going to print on the screen when slave watchdog expires to inform the user.

b) *Explain how the watchdog timer feature is enabled and how it is programmed to support different watchdog delays. Provide a sequence of instructions for Watchdog timer configuration for different delays and enabling.*

All we need to do to enable it is to include the watchdog timer library <avr/wdt.h> and choose the duration (WDTO_XMS) we want from it and enable it with, and of course reset it in the main while(1) loop.

Below is an example of how to do it for 500ms

```
#include <avr/wdt.h>

wdt_reset();
wdt_enable(WDTO_500MS);
```

and here is how we did it inside our code (note enable is commented as instructed in the manual)

```
void ConfigMasterWD(){

	uint16_t temp = eeprom_read_word((const uint16_t*)0x0);

	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 >= 'A' && config <= 'C')){
				sei();
				message_user(INVALID_OPTION);
				cli();
				config='C';
```

```
				}
				else{
					break;
				}
			}

			eeprom_write_word((const uint16_t*)0x0, config);

			UCSR0B &= ~(1 << RXEN0);


		}

		else {
			config = temp;
		}

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

			case 'B':
				duration = WDTO_250MS;
			break;

			case 'C':
				duration = WDTO_500MS;
			default:
				duration = WDTO_500MS;
		}
		wdt_reset();
		//wdt_enable(duration);
		sei(); // enable interrupts again

	}
```

c) Study the datasheets for HC-05 Bluetooth and Xbee Radio.

i. When do each of these components dissipate maximum power? Is it when the module is receiving data, transmitting data, or neither receiving nor transmitting?

ii. What do you think is the reason behind the specified behavior in (i)

HC-05 Bluetooth dissipates maximum power while transmitting data. As per the documentation, while transmitting, it needs 20mA. Xbee radio also dissipates maximum power while transmitting data.

It makes sense, because transmission in general requires more power as the device has to amplify the signal enough so that it reaches the target. While during receive mode, it just needs to demodulate and process the signal instead of amplifying it. Also, during receiving, it might be conserving power and sleeping until the data has been fully received.

d) Study ATmega128 datasheet and carefully explain the differences in six different power management / sleep modes Which one do you think represents the lowest power mode? Clarify your assumptions.

There are 6 power management/sleep modes that can be chosen using the MCU controller register MCUCR by adjusting the sleep mode select bits SM[2:0] and setting the sleep enable bit SE.

After adjusting the MCU, SLEEP instruction must be called to enter the mode.

- **Idle Mode**: SM[2:0] = 000
  Stop the CPU but allowing SPI, USART, Analog Comparator, ADC, Two-wire Serial Interface, Timer/Counters, Watchdog, and the interrupt system to continue operating. This mode only stops the clkCPU and clkFLASH.

- **ADC Noise Reduction Mode**: SM[2:0] = 001
  Stop the CPU but allowing the ADC, the External Interrupts, the Two-wire Serial Interface address watch, Timer/Counter0 and the Watchdog to continue operating.This mode only stops the clkI/O, clkCPU, and clkFLASH.

- **Power-down Mode** : SM[2:0] = 010
  In this mode, the External Oscillator is stopped, while the External Interrupts, the Two-wire Serial Interface address watch, and the Watchdog continue operating . Only an External Reset, a Watchdog Reset, a Brown-out Reset, a Two-wire Serial Interface address match interrupt, an External Level Interrupt on INT7:4, or an External Interrupt on INT3:0 can wake up the MCU. This sleep mode basically stops all generated clocks, allowing operation of asynchronous modules only.
  Note when waking up from this mode the MCU should take its time to reset all the clocks and give them time to become stable.

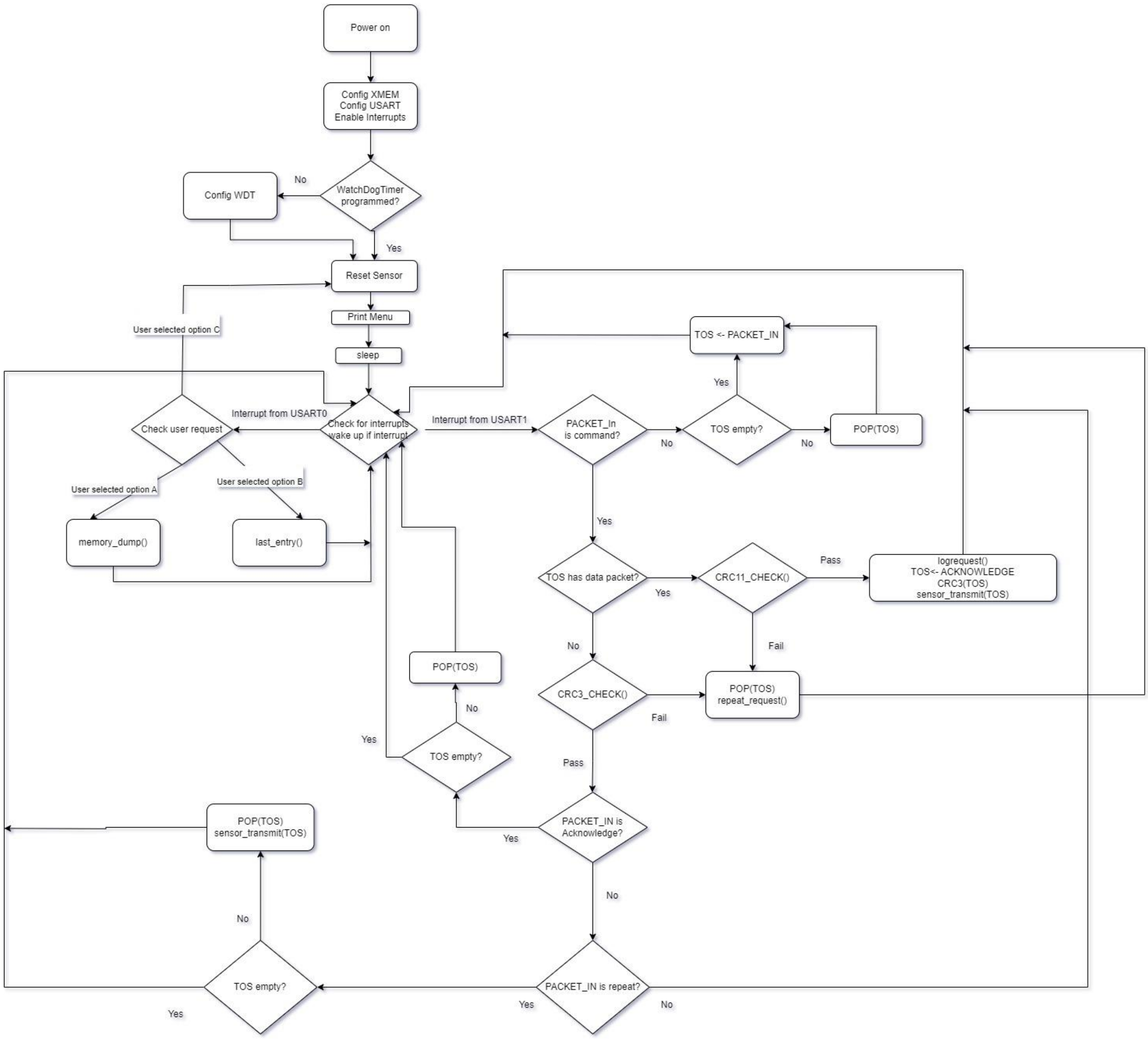
- **Power-save Mode**: SM[2:0] = 011
  This mode is identical to Power-down, with one exception: If Timer/Counter0 is clocked asynchronously, Timer/Counter0 will run during sleep. The device can wake up from either Timer Overflow or Output Compare event from Timer/Counter0, so this mode stops all the clocks except clkASY.

- **Standby Mode**: SM[2:0] = 110
  If External Crystal/Resonator clock option is selected, 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 6 clock cycles.

- **Extended Standby Mode**: SM[2:0] = 111
  Also, if External Crystal/Resonator clock option is selected, the MCU enter Extended Standby mode. This mode is identical to the one before it with the exception that the Oscillator is kept running from Extended Standby mode, the device also wakes up in 6 clock cycles.

Based on the above we can clearly see that the lowest power mode is **Power-down mode** because all clocks are stopped in this mode to save energy meanwhile other modes there are some clocks that still working.

e) After studying Figure 3.4 above and Section 3.4.2 below, draw an algorithmic flowchart, like the one you were provided in Module 2, to outline how your overall code will work. Consider the design guidelines and subroutines described in Section 3.4.2 in sketching your flowchart.



Code

Notes on the code:

Uint8_t and unsigned char are the same thing but one used when dealing with integers and the other one when dealing with ASCII characters for readability.

For the user messages that we will print we didn't define them as macros (#define) so we don't generate new string inside the data memory each time service an interrupt, and we make it as fixed global array which we can see starts from 0x100, and that why we made or loging starts from 800

For the stack we reserved 0x100 (256) bytes and made sure to jump over him to the external memory when reading or writing from it

```
#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 8000000
#define BaudRate 9600
#define BR_Calc ((F_CPU/16/BaudRate)-1) //Baud rate calculator as in slides equals
0x33 (51)

#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
```

```c
#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
#define TMP_message 12

// global variables

const unsigned char messages [][USER_TR_BUFFER_SIZE] = {
       "\rEnter your option:\rA-Memory Dump\rB-Last Entry\rC-Restart \0",
       "\rSensor reset completed \0",
       "\r**Invalid option**\rPlease choose again \0",
       "\rNothing to show, log file is empty \0",
       "\rMemory dump process: \0",
       "\rMemory dump process finished \0",
        "\rLast Entry: \0",
        "\rLast entry finished \0",
        "\rInitialization succeeded \0",
        "\rEnter your option for Master Watchdog (& period):\rA-30ms\rB-250ms\rC-500ms
\0",
        "\rEnter your option for SLAVE Watchdog (& period):\rA-0.5s\rB-1s\rC-2s \0",
        "\rTimer Expired!, Sensor took so long \0"
};

uint8_t TOS;
uint8_t is_tos_empty = 1; // 1 empty, 0 full
uint16_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 and indexes
unsigned char sensor_tr_buffer;
//uint8_t sensor_tr_ind = 0;
unsigned char sensor_rc_buffer[3] = "\0\0\0";
uint8_t sensor_rc_ind = 0;

// Watchdogs variables
uint8_t WD_master_configured = 0; // 0: not configured, 1:configured


// 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();

		user_tr_ind++;
		_delay_ms(10);
	}

}

// 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 'A':
			memory_dump();
			break;

			case 'B':
			last_entry();
			break;

			case 'C':
			// 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();
		slave_wdt_reset();

		sensor_rc_buffer[sensor_rc_ind++] = UDR1;
		if(sensor_rc_ind < 2)return;

		sensor_rc_ind = 0;
		uint8_t packet_in = str_to_hex(sensor_rc_buffer);
		// if packet in is data packet
		if(packet_in & (1<<7)){
			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 & (1<<7);
}

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

	// initialize sensor
	init_sensor();

	// show menu to the user
	message_user(INIT_DONE);
	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;
	}
```

```c
	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){

	strcpy(user_tr_buffer, messages[message_index]);
	user_tr_ind = 0;
	lock_bluetooth();

}

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

void memory_dump(){

	if(log_file_empty()) return;

	message_user(MEM_DUMP_START);
	uint16_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

}
```

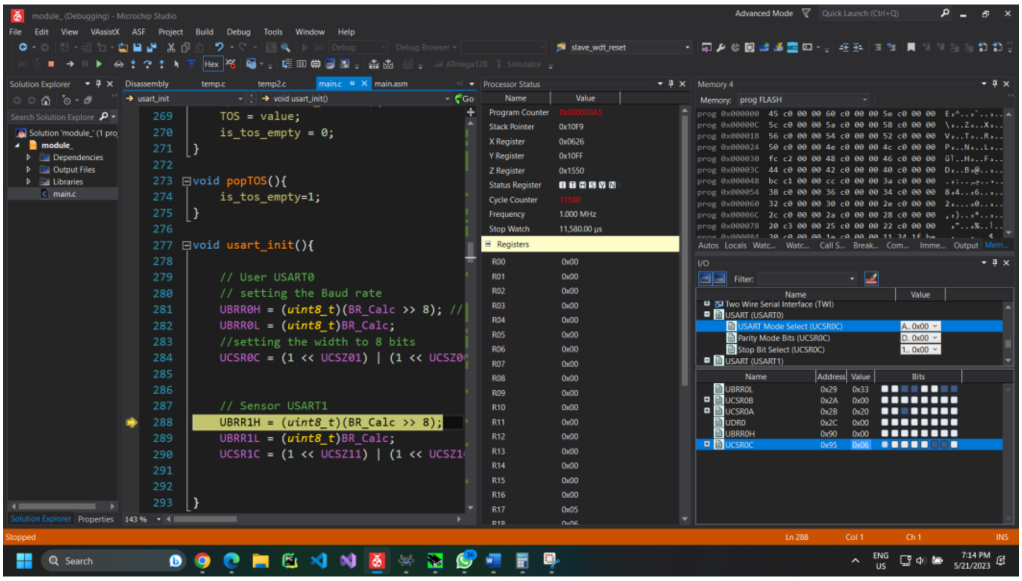
```c
void ConfigMasterWD(){

	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 >= 'A' && config <= 'C')){
				sei();
				message_user(INVALID_OPTION);
				cli();
				config='C';
			}
			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 'A':
			duration = WDTO_30MS;
		break;

		case 'B':
			duration = WDTO_250MS;
		break;

		case 'C':
			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*)0x0002);

		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 >= 'A' && config <= 'C')){
					sei();
					message_user(INVALID_OPTION);
					cli();
					config='C';
				}
				else{
					break;
				}
			}

			eeprom_write_word((const uint16_t*)0x0002, 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 master watch dog
			case 'A':
				duration = 15625; // 0.5s
			break;

			case 'B':
				duration = 31250; // 1s
			break;
```

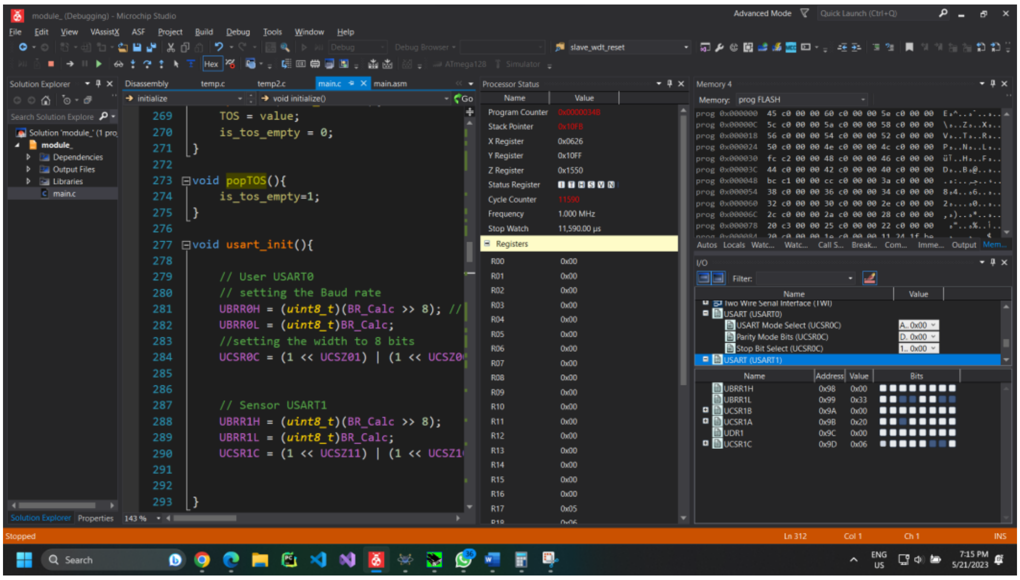
```
		case 'C':
			duration = 62500; // 2s
		break;
		default:
			duration = 62500;
	}


	// 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;
}
```

Verifying Initialization with Atmel Studio

Verifying USART0



Verifying USART1



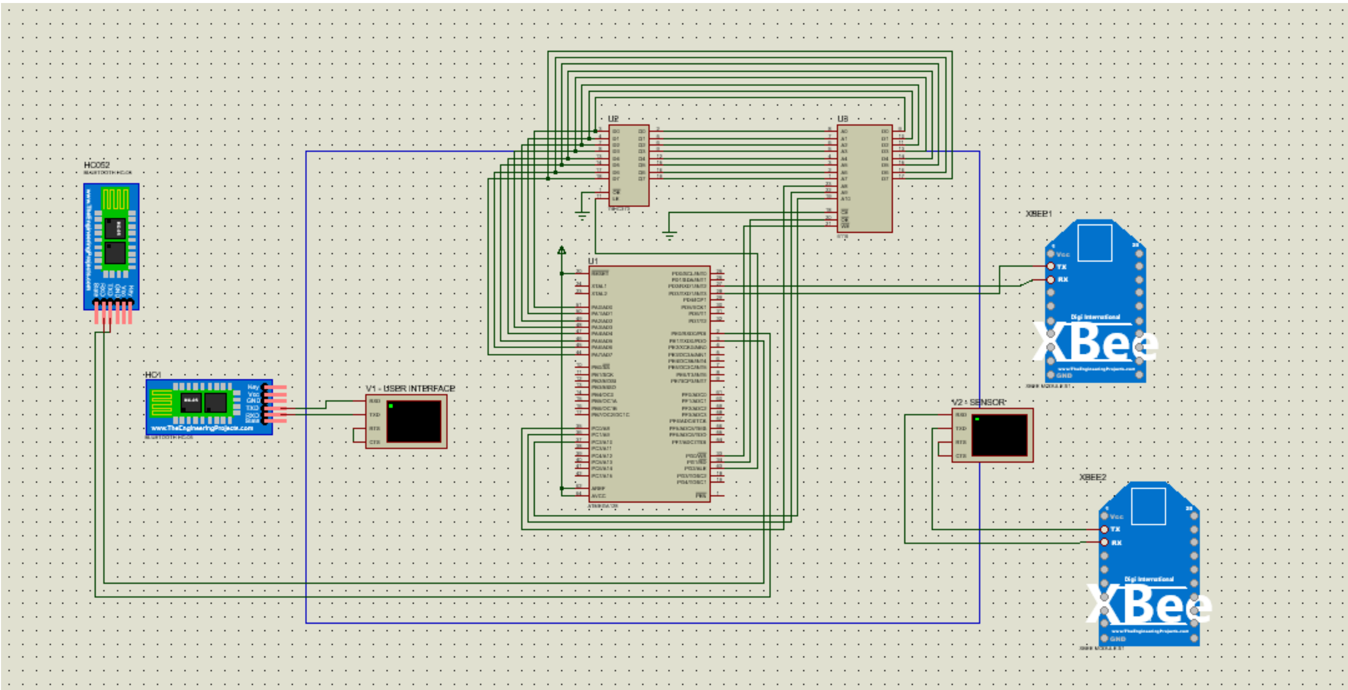
Initializing external memory

Not all C pins are needed since we are going to use only 2KB as we did in module2.

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

```

Proteus design

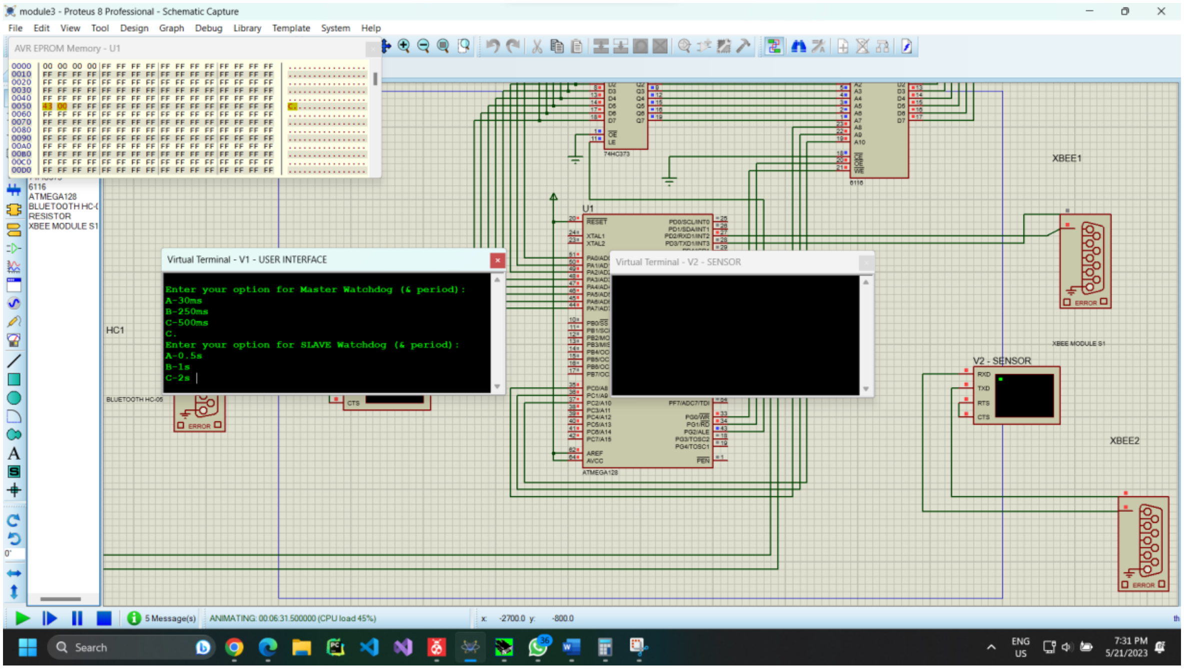
This is how our proteus design looks like, with connections made according to the instructions given in the lab manual. The external memory connections are the same as module2, but we have added new components such as Bluetooth, XBee and Virtual Terminals, as part of the lab specifications for module3.



Proteus Simulation

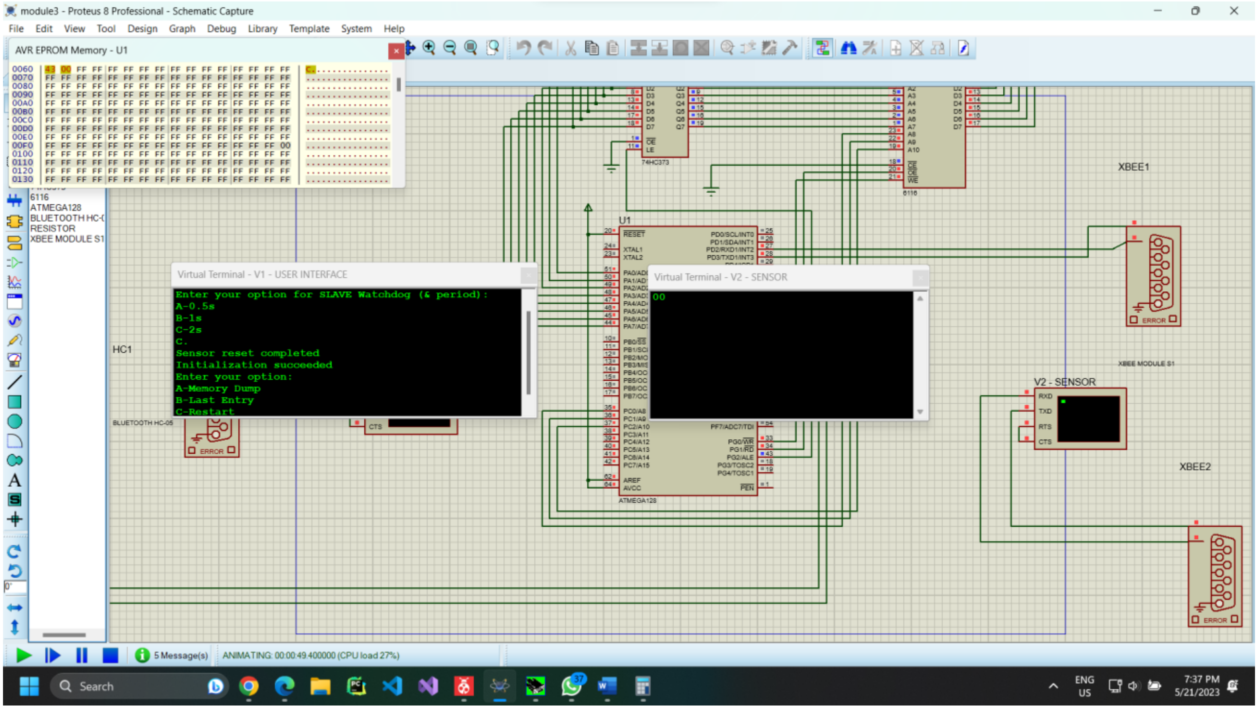
1) Configuring master watchdog

since the first 4 bytes by default are 0x00 in EEPROM but in the manual it says to only configure when the first 2 bytes are 0xFFFF we slightly change the code to check 0x50 and 0x51 to see the menu and choose an option
as you can see below, we chose C and we wrote our decision in EEPROM to read it next time.
C is associated to enable watchdog with WDTO_500MS which is 500ms

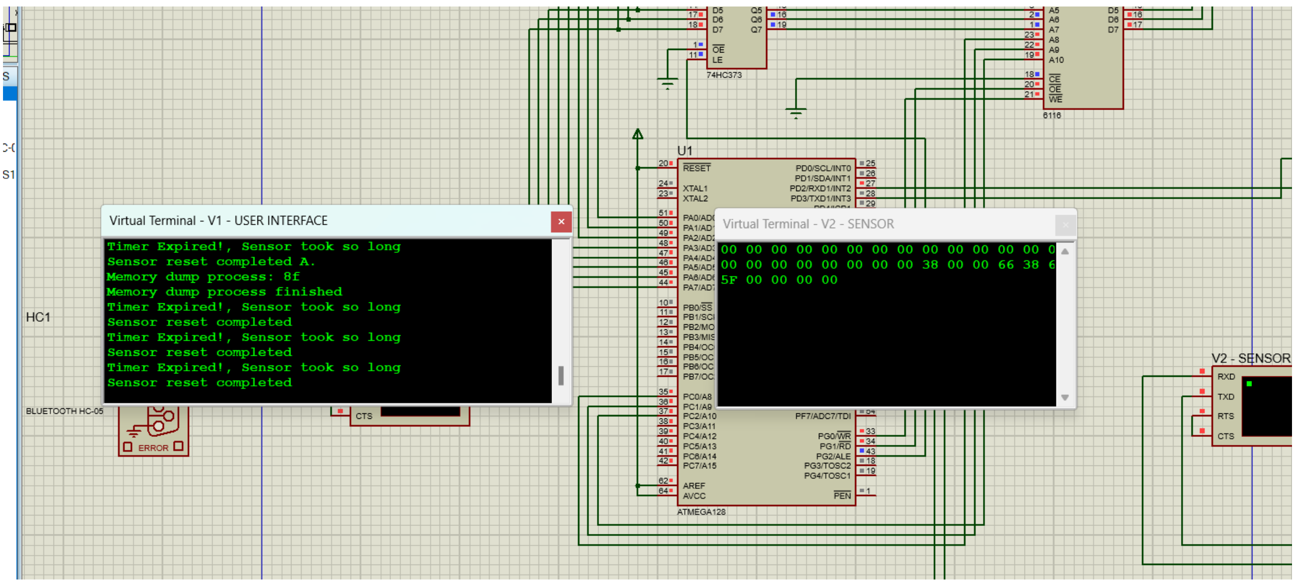


2) Configuring Slave Watchdog

Again, we modified the code slightly because the second 2 bytes are also 0x00
Here we are reading and writing our choice to 0x60 and 0x61
Also for the duration we are loading the choice to the output compare to compare it with the value of the timer
And we are using presale of 256 to not have fractions and can fit them in 16 bit
0.5s will load 0.5s * 8MHZ / 256 = 15625 cycles
1s will load 1s * 8MHZ / 256 = 31250 cycles
2s will load 2s * 8MHZ / 256 = 62500 cycles
Finally we complete the rest of the initialization process.

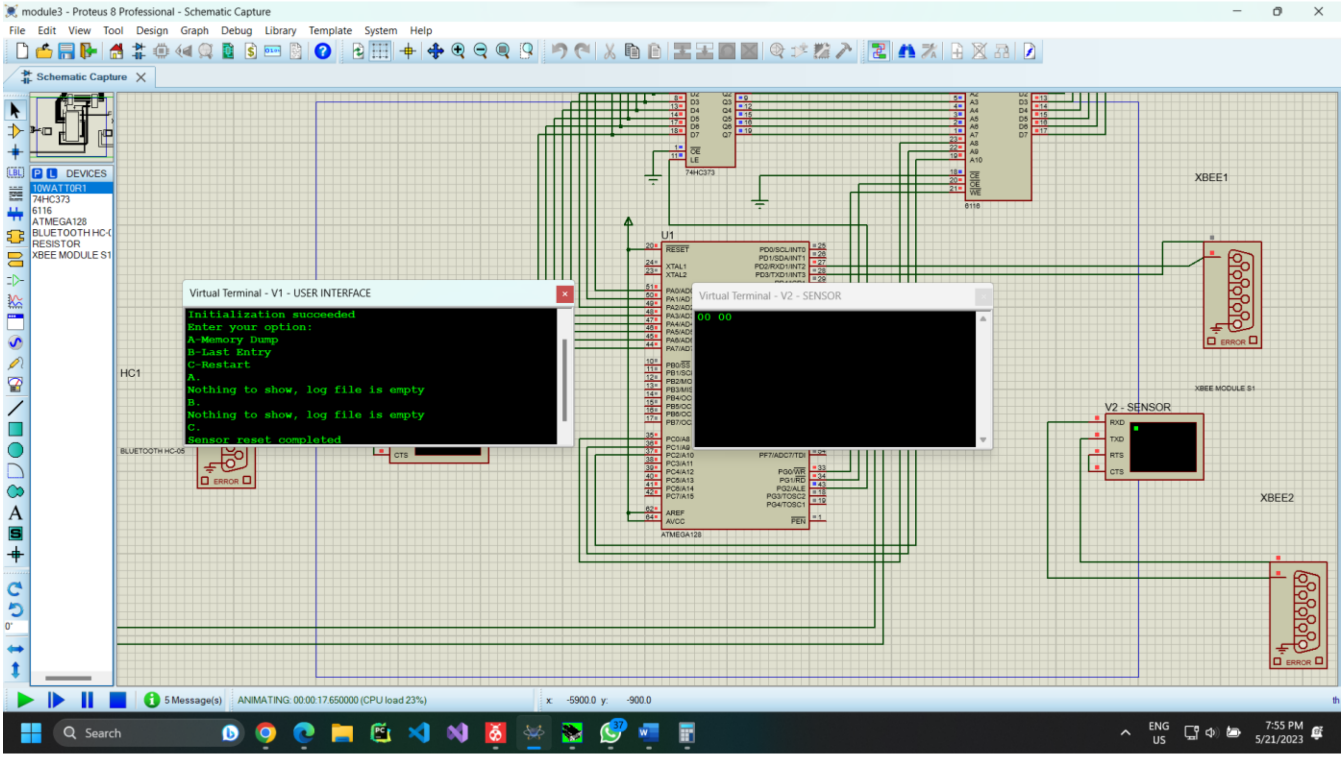


Also, because slave watchdog will keep resetting the sensor each time, we don't receive a packet from it and inform the user, our terminals will be crowded very fast, so we decided to disable it while testing but even if it was enabled everything still works as expected and, in the figure, below we are dumping memory but it resets every 2s because the slave watchdog expires.



3) *Basic user inputs*

It prints nothing to show when we choose A or B because we didn't log anything yet.
Also, sensor prints 00 twice because first time it was system initialization, and the second time we did it manually with selecting the option C.



Now let us test more use cases.

Note that we will be testing all our use cases from module2 and additionally checking if the communication is taking place properly.

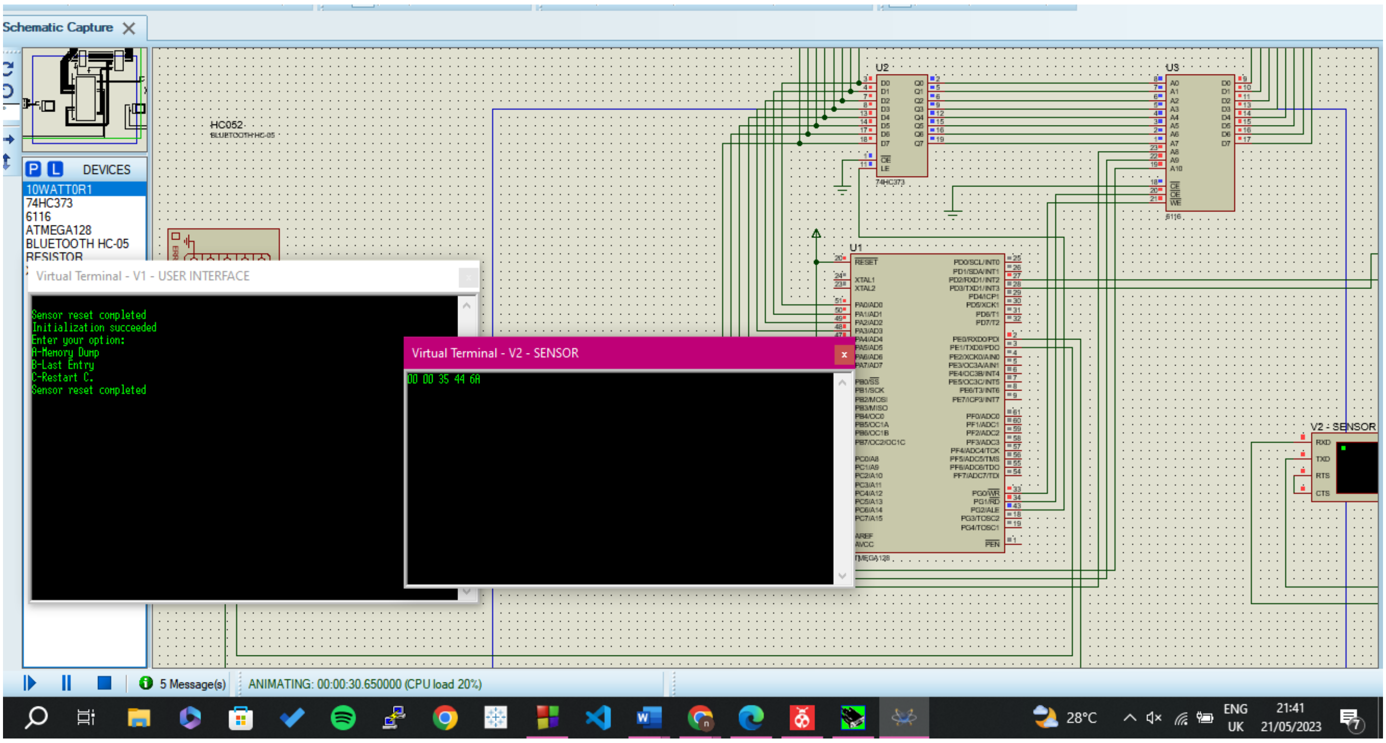
CASE 1 (a):

Enter a corrupted acknowledge packet at sensor VT.

We input an Acknowledge packet with the wrong CRC at Sensor terminal 010 11101 (5D).

We expect to see a REPEAT REQUEST as PACKET_OUT -> 011 01010 (6A).

And it will be shown on the sensor terminal after of course it has been transmitted to the sensor successfully.



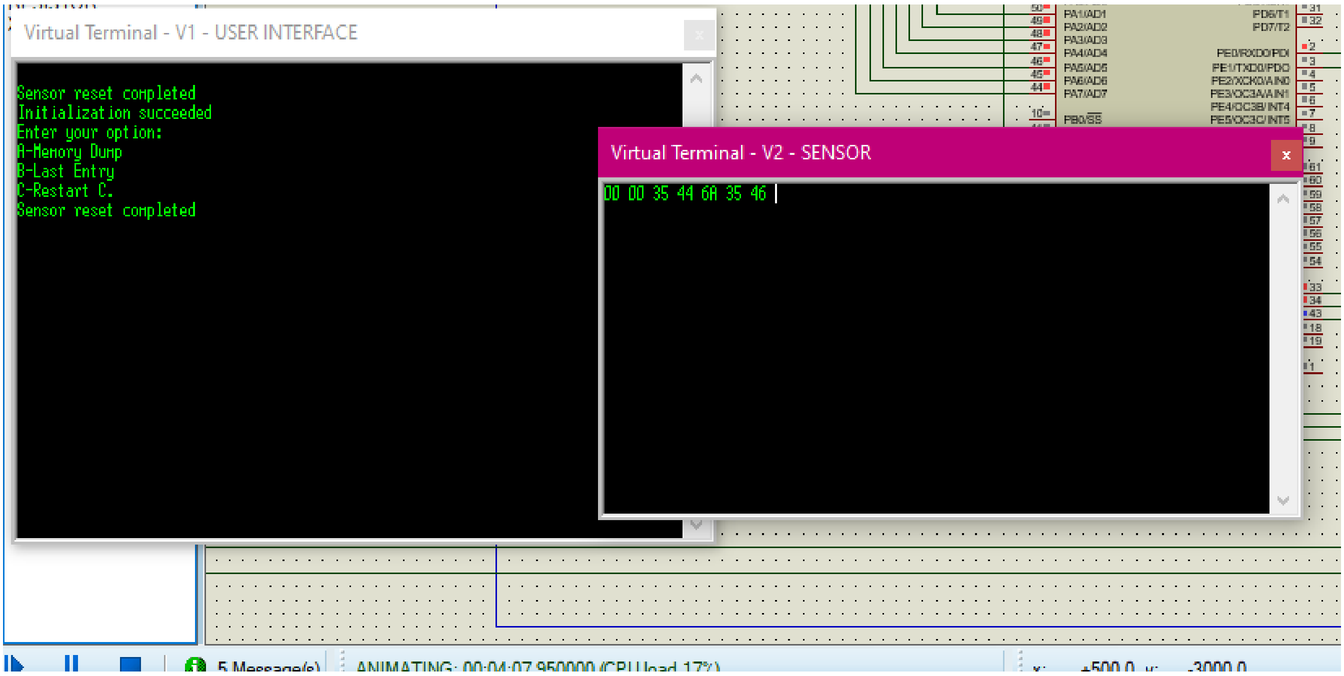
As seen, 6A is seen on the terminal after I input 5D (which is shown as the ASCII 34 and 44 on the terminal).

CASE 1 (b):

Enter a correct Acknowledge packet at Sensor VT.

We input an Acknowledge packet with the right CRC at Sensor terminal 010 11111 (5F).

We don't expect to see any output on the Sensor VT.



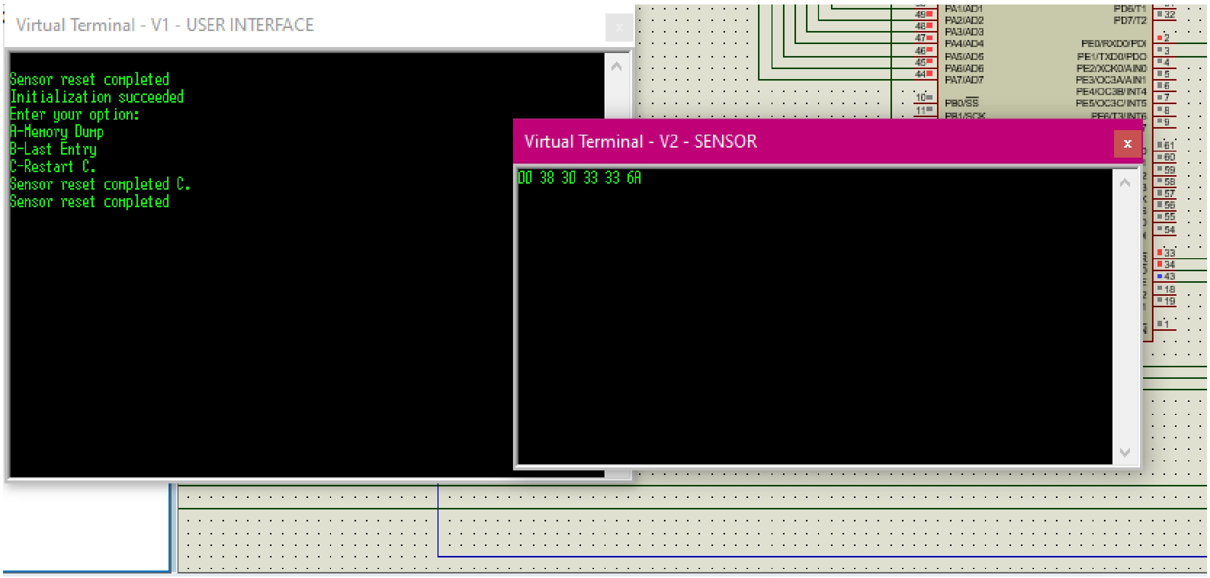
As you can see, I entered 5F which is shown as the ASCII 35 and 46, and as expected, there is no repeat signal transmitted to SENSOR or shown at the SENSOR terminal.

CASE 2 (a):

Data packet followed by incorrect log request.

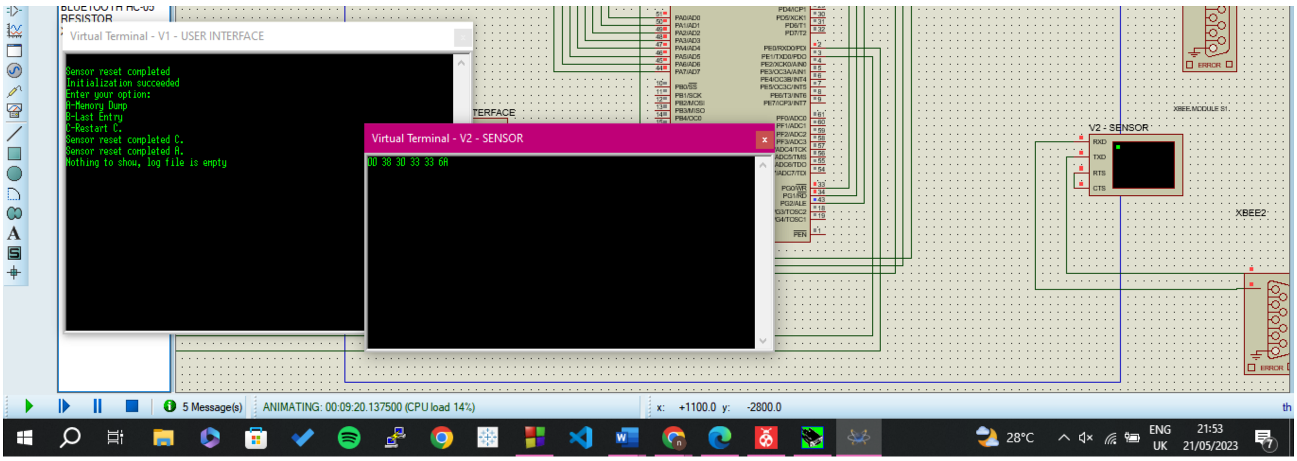
I entered data packet: 0x80 followed by incorrect log request which is 0x33.
As you can see on the SENSOR VT, these values are shown in ASCII format.

As expected, the REPEAT_REQUEST packet (0x6A) is received by the SENSOR and is shown on the SENSOR VT.



Another verification of this can be making sure nothing was logged in memory because the log request was corrupted.

This can be done by using memory dump instruction at the USER VT exactly after this.

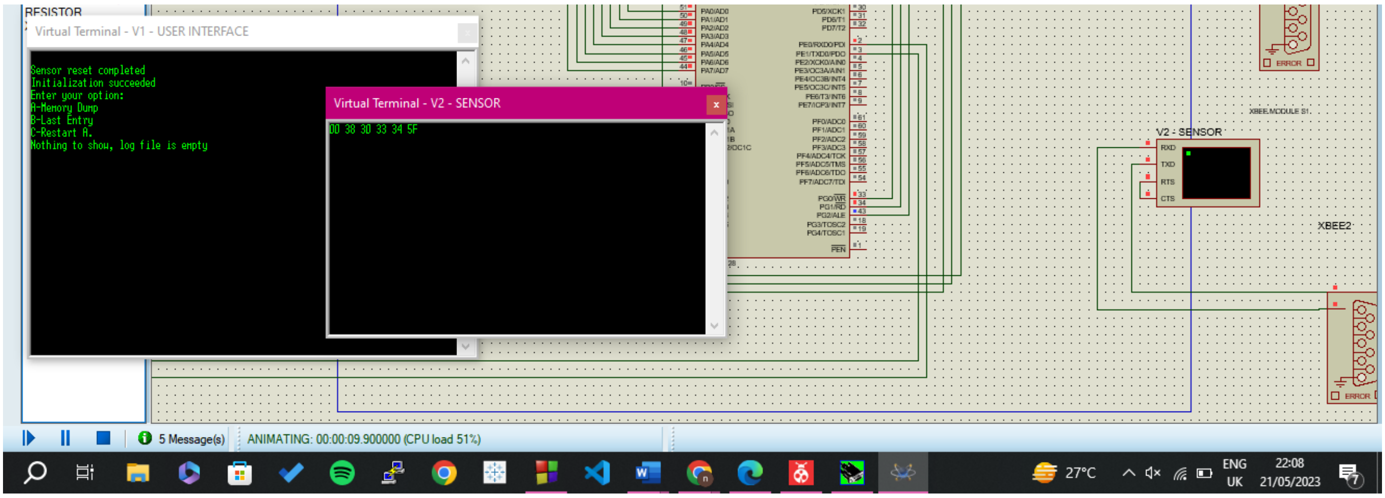


As seen, the message "NOTHING TO SHOW IN MEMORY" is printed at User VT.

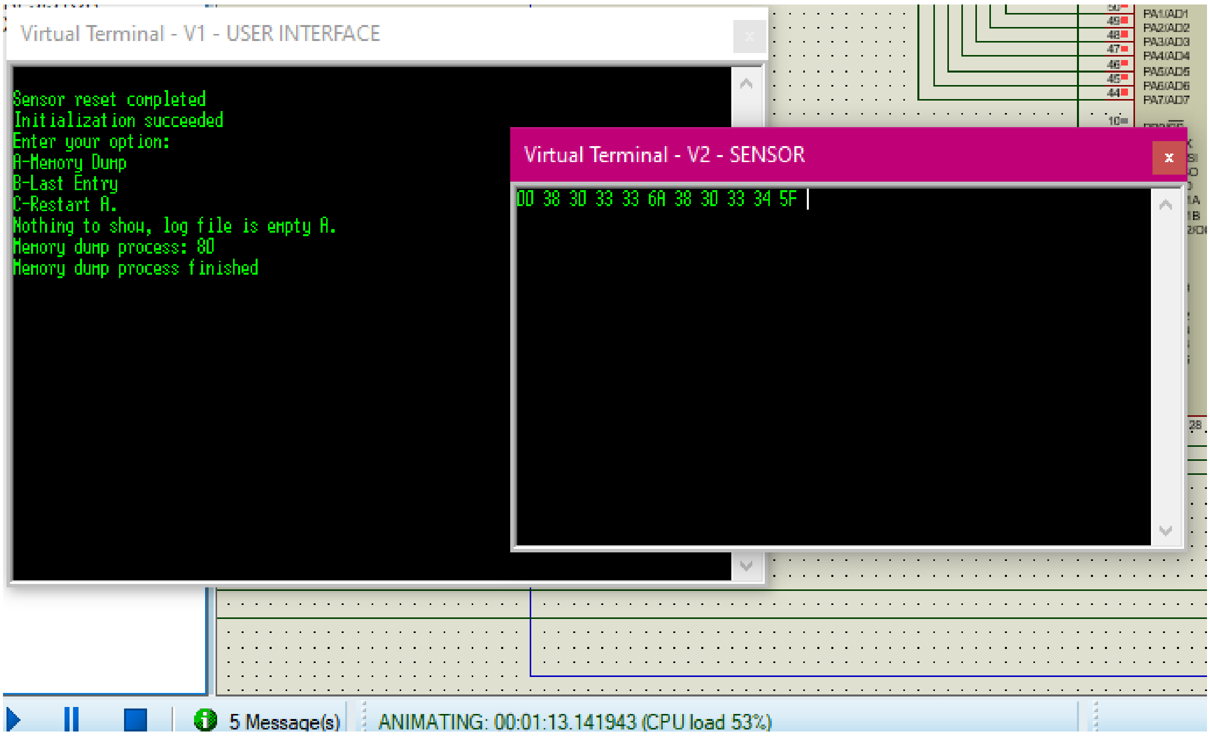
CASE 2 (b):

Data packet followed by correct log request.

Now I input the data packet 0x80 again on the SENSOR VT followed by the correct log request which is 0x34. As expected, the Acknowledge packet should be transmitted and received at the SENSOR VT and after it has been received, we can see that it is echoed on the SENSOR VT.



Again, to make sure that the logging was successful, we could additionally use the memory dump functionality right after this and check if 0x80 has really been logged.



As expected, memory dump shows that 80 was logged successfully.

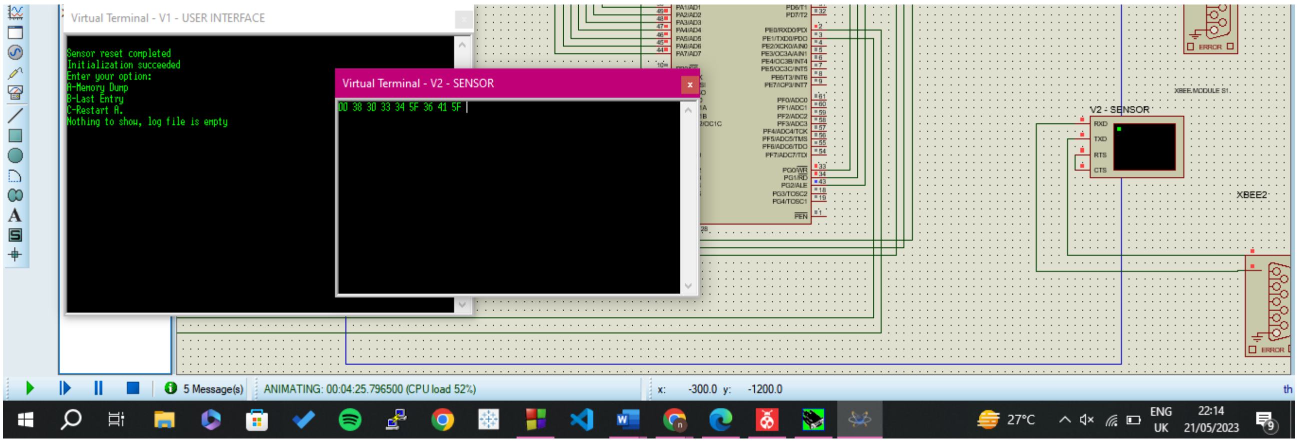
CASE 3(a):

Enter a REPEAT request at the sensor TERMINAL.

And expect TOS to be transmitted and received by the SENSOR.

Since I am doing this right after CASE2(b), TOS contains the Acknowledge packet (5F) and I expect to see that on the SENSOR VT.

I will enter the repeat request: 011 01010 (6A).



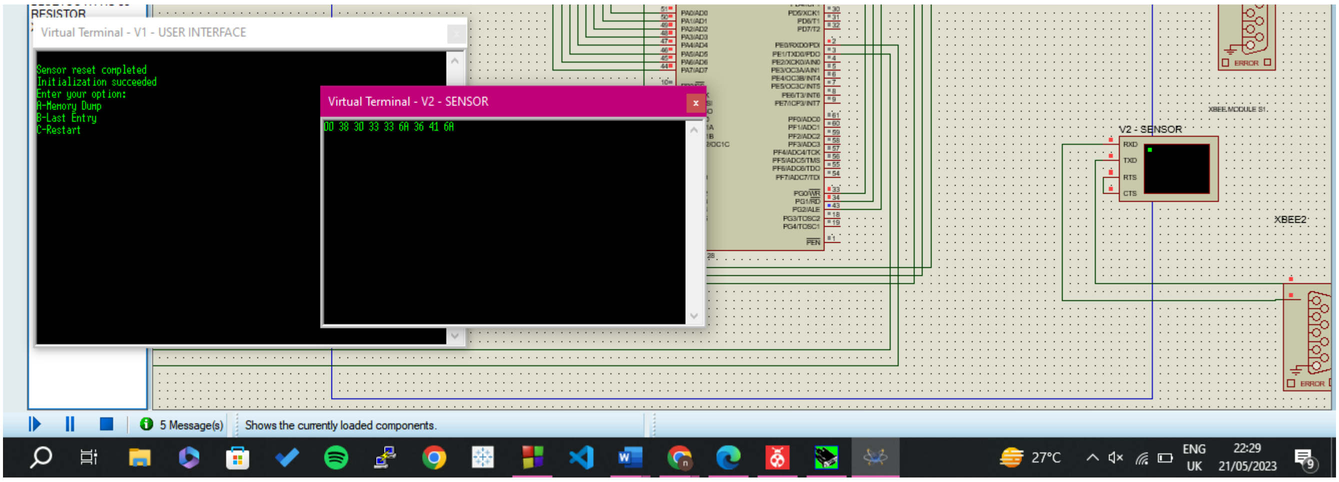
As you can see, I entered the repeat request 6A which translated to ASCII 36 and 41, as seen on the VT (this screenshot is followed by the last screenshot),

As expected, the Acknowledge packet (5F) is seen on the Sensor VT.

CASE3 (b):

Now assume that TOS contains the repeat request, for example right after CASE2(a).

If I enter the repeat request at the SENSOR VT, I expect to see the TOS (0x6A) as the packet received by the SENSOR.

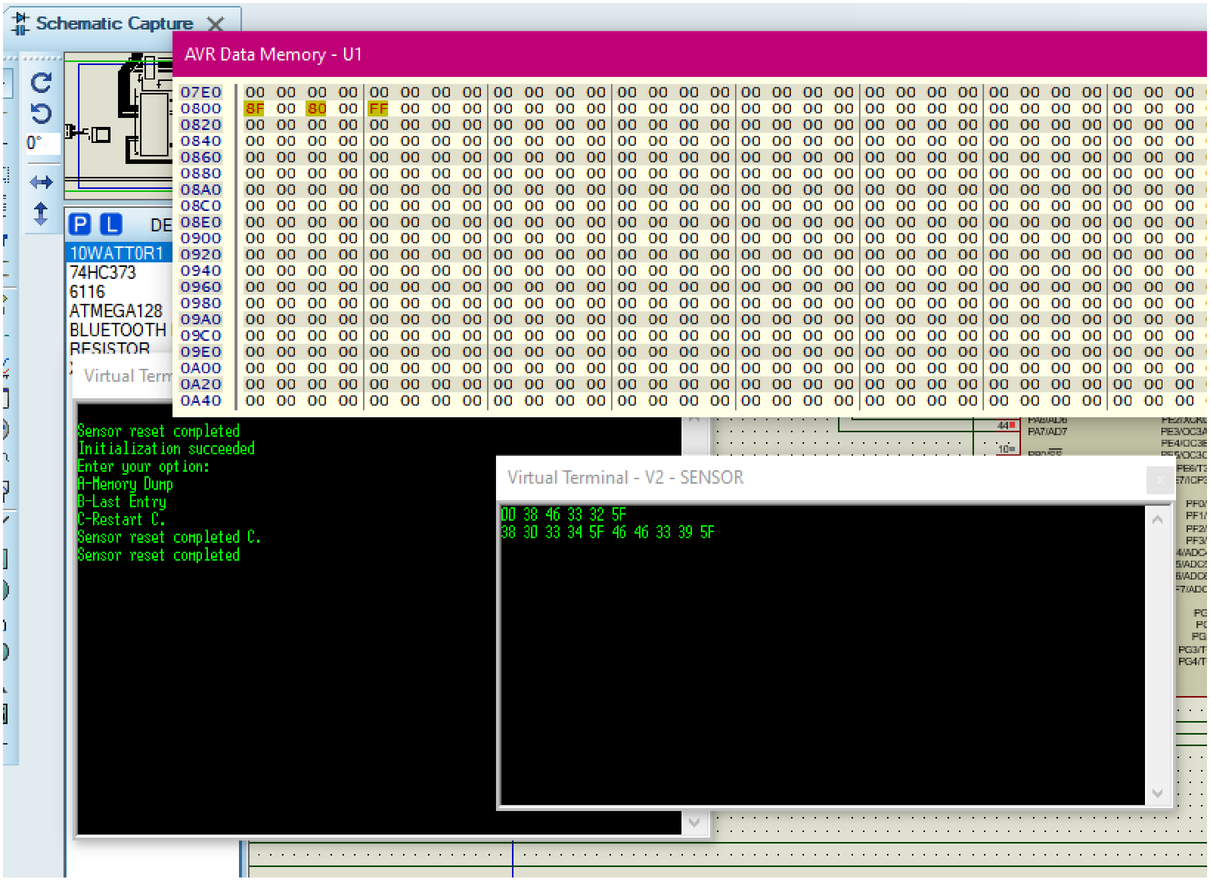


As you can see, after I enter 6A which corresponds to ASCII 36 and 41, I see the expected output. which is 6A (because that is what was present in TOS).

CASE 4:

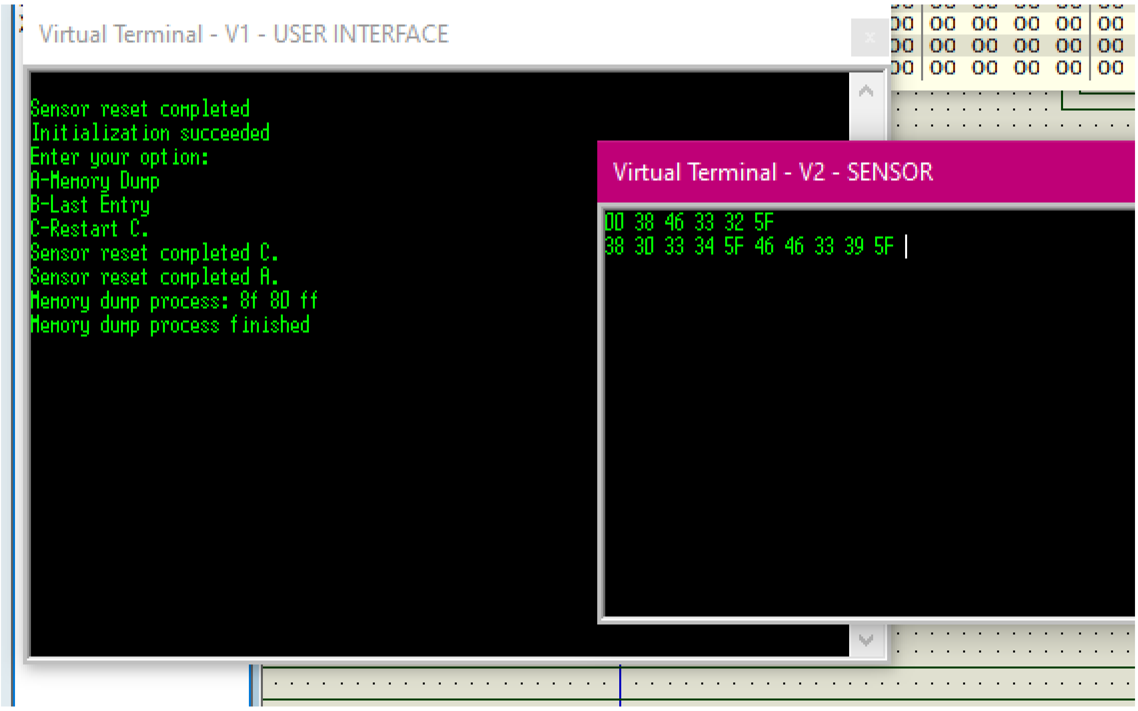
Memory dump when we have logged multiple values.

Let’s check the AVR Data memory for the logged values.



As seen, I have logged 8F, 80 and FF.

Now let’s check if memory dump, when selected on User VT, will echo all of them.



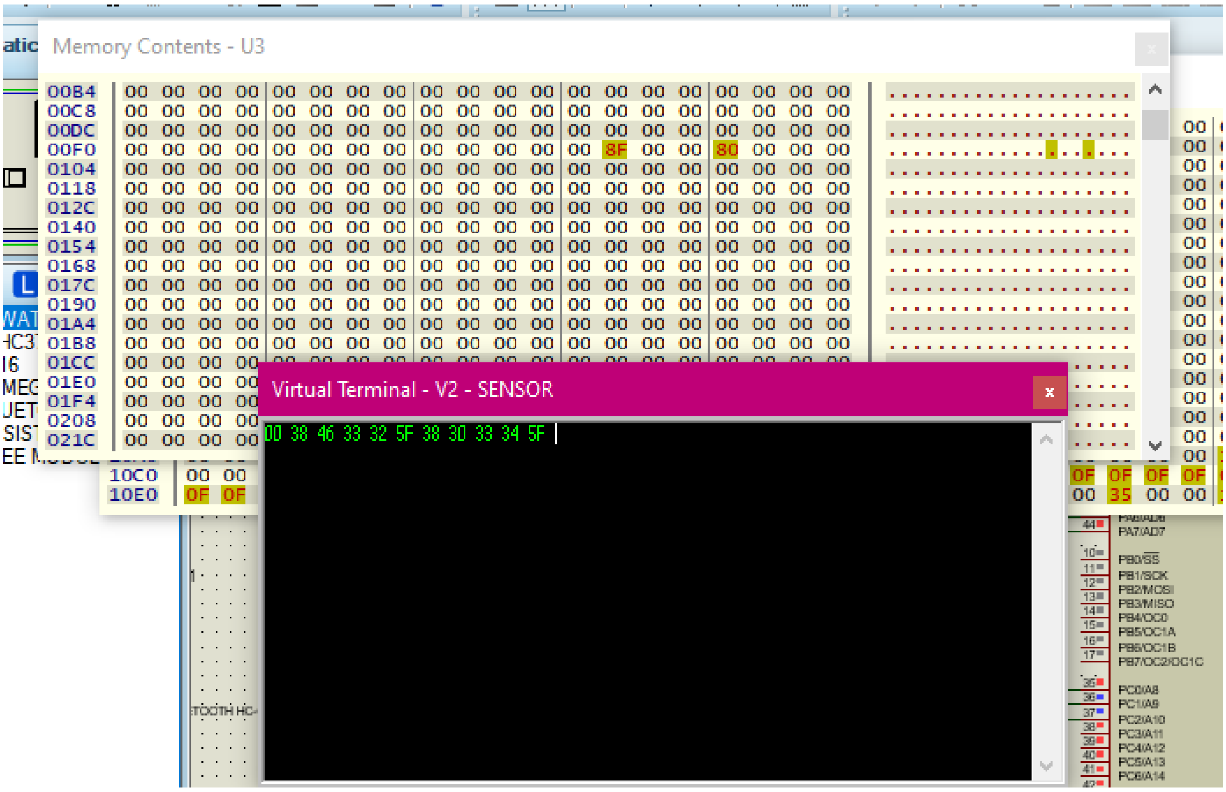
And it does!!

Memory dump functionality is also correct.

CASE 6:

Writing to External Memory:

Note that, like module2, for this purpose we will update the IMEM_START = 0x18FD because we quickly want to jump to external memory and do not want to log a lot of data to be able to check external memory functionality.



As seen, I input valid data packets with log requests (their validity is confirmed by the 5F acknowledge packet seen on the terminal) and it does log data to external memory, as expected.

Conclusion

We have successfully demonstrated how we achieved all the objectives of this module, how we modularized our code, how we did the system testing and verification on proteus and we have also demonstrated a successful overall result. We have made sure that we build on top of the previous module and take the smart data logger system as close to its final version and kept making it even better. In conclusion, we have gained vital embedded system programming and design skills, learnt about the importance of hardware enabled interrupts and the close relationship between hardware and software development.