KUZEY KIBRIS KAMPÜSÜ
ODTÜ METU
NORTHERN CYPRUS CAMPUS

CNG 336 / EEE 347

Module 1

Report

Name & ID:

Muhammad Somaan (2528404)

Muhammad Maarij (2486785)

Declaration: The content of the report represents the work completed by the submitting team only, and no material has been copied from the other source.

Objectives: The objective of this lab is to get familiar with

- assembly instructions of ATMega 128
- debugging on Atmel Microchip Studio
- running simulation on Proteus
- AVRFlash
- ATMega 128 architecture

Design Details

System Layout:

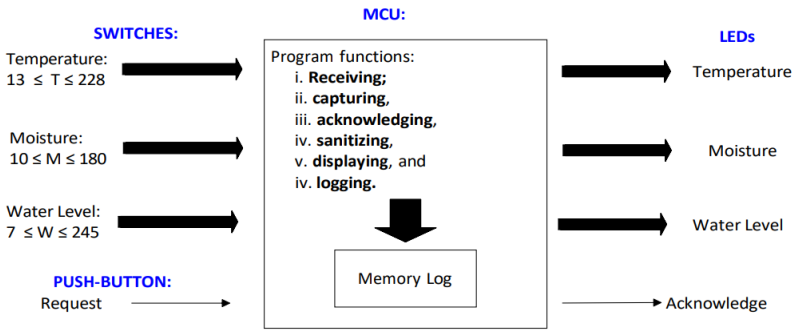


Figure 1. Main functions and interfaces of the data logging system [1

Flow Chart:

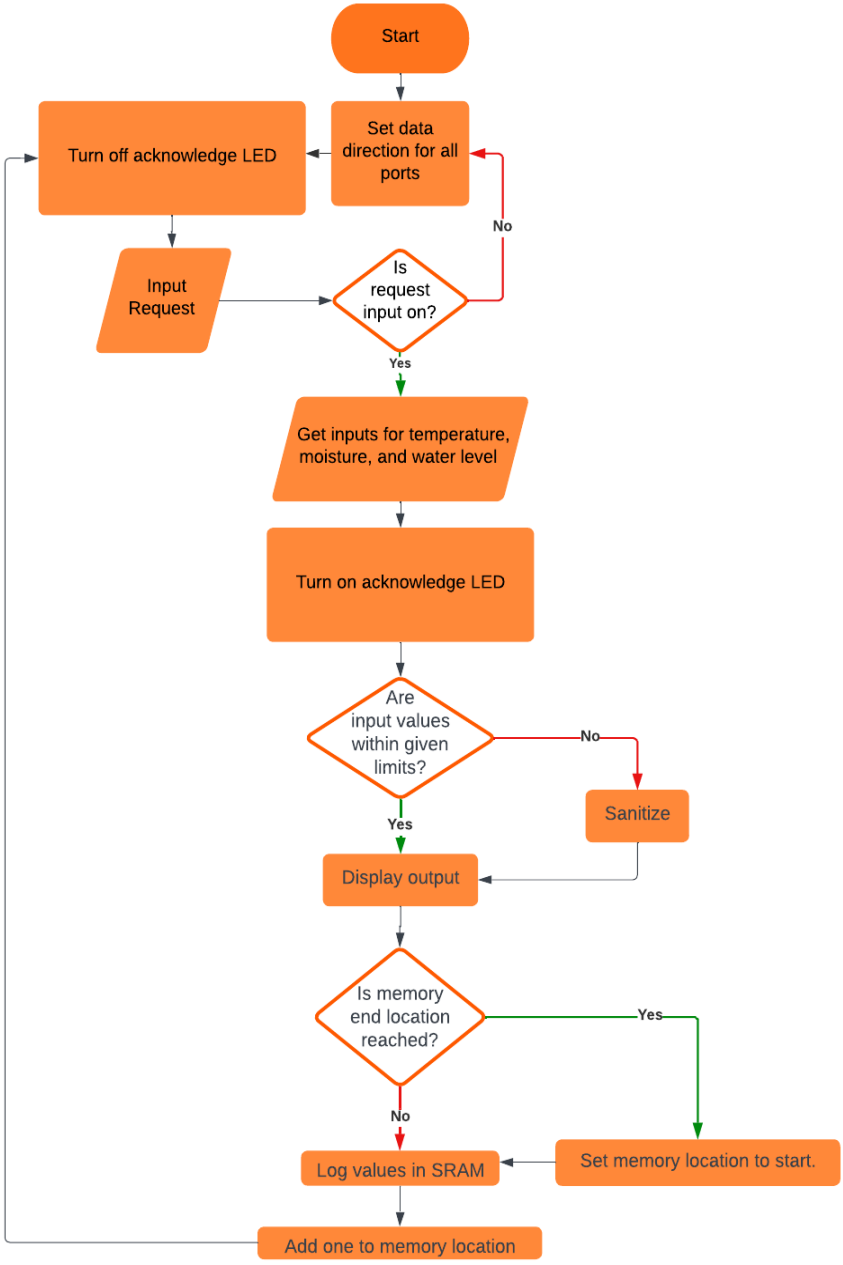


Figure 2. Flow chart of problem statement

System Hardware Connectivity:

Table 1. Table for input/output mapping to ATMega128 pins

Pin/Port	Component	Description
A	Switch	Input for temperature
B	Switch	Input for moisture
C	Switch	Input for water level
D	8-LEDs	Output for temperature
E	8-LEDs	Output for moisture
F	8-LEDs	Output for water level
G	Bit 0: Push Button Bit 4: 1-LED	Bit 0: Request Bit 4: Acknowledge

Code:

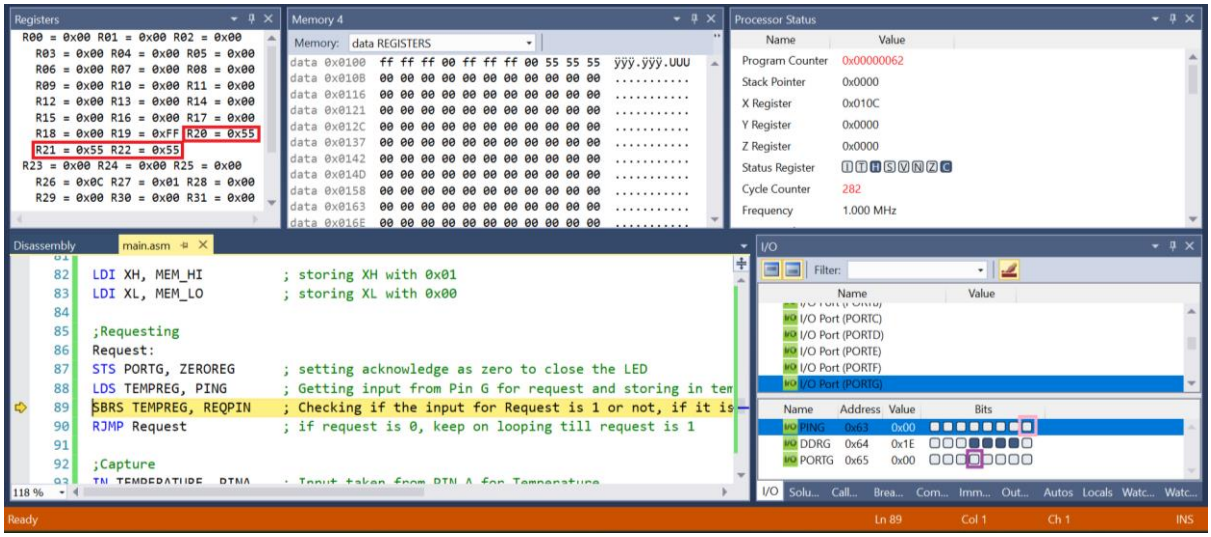
```
; Lab Module 1.asm
; Created: 07-Apr-23 2:10:03 PM
; Code
.INCLUDE "m128def.inc"
.EQU ZEROS = 0x00          ; defining constant ZEROS = 0000 0000
.EQU ONES = 0xFF           ; defining constant ONES = 1111 1111
.EQU T_LO_LMT = 0x0D       ; setting temperature lower limit
.EQU T_HI_LMT =0xE5        ; setting temperature higher limit plus 1 so we can use BRLO and make our code easier
.EQU M_LO_LMT = 0x0A       ; setting moisture lower limit
.EQU M_HI_LMT =0xB5        ; setting moisture higher limit plus 1 so we can use BRLO and make our code easier
.EQU W_LO_LMT = 0x07       ; setting water level lower limit
.EQU W_HI_LMT =0xF6        ; setting water level higher limit plus 1 so we can use BRLO and make our code easier

; Registers
; defining registers for easy readability of code
.DEF TEMPREG = R17                ; defining R17 as TEMPREG (temporary register)
.DEF ZEROREG = R18                ; defining R18 as ZEROREG
.DEF ONESREG = R19                ; defining R19 as ONESREG
.DEF TEMPERATURE = R20            ; defining R20 as TEMPERATURE
.DEF MOISTURE = R21               ; defining R21 as MOISTURE
.DEF WATER = R22                  ; defining R22 as WATER

.EQU REQPIN = 0                   ; setting Request input to be in bit 0
.EQU ACKPORT = 4                  ; setting Acknowledge output to be in bit 4

.EQU MEM_START = 0x100            ; memory start from 0x100
.EQU MEM_HI = 0x01                ; memory high 0x01
.EQU MEM_LO = 0x00                ; memory low 0x00
.EQU MEM_END = 0x10FF             ; memory end at 0x10FF
.CSEG
; Start using the Program Memory at the following address
.ORG 0x0050
Start:
; first input is going to be the input register
.MACRO SANITIZE                   ; defining macro for Sanitizing the input
LDI @0, 0xFF                      ; setting the first input to 1111 1111
.ENDMACRO
```

```
; first input is going to be the input register
; second input is going to be the lower limit of the input
; third input is going to be the upper limit of the input

.MACRO CHECK_LIMITS      ; defining macro to check limits of inputs
CPI @0, @1               ; comparing input value with lower limit
BRLO Sanitize_Jmp        ; branch if the input value is lower than the lower limit
CPI @0, @2               ; comparing input value with upper limit
BRLO Continue            ; branch if the input value is lower than the upper limit
Sanitize_Jmp:            ; label for Sanitize so the upper branch can jump here
SANITIZE @0              ; calling SANITIZE macro with the input value register
Continue:                ; continue label so the branch can skip sanitization if the input value is lower than the
upper limit
.ENDMACRO                ; end macro

.MACRO CHECK_AND_LOOP_RAM_END ; setting macro to check if RAM is full
LDI TEMPREG, 0x11             ; loading 0x11 to TEMPREG
CPSE XH, TEMPREG              ; when memory is full XH= 0x11 so compare to check if memory is full
RJMP End_Check                ; if memory is full skip this line, else, jump to End_Check
LDI XH, MEM_HI                ; loading MEM_HI i.e. 0x01 to XH so RAM memory can be overwritten
End_Check:                    ; End_Check label if memory is not full branch here to skip overwritting
.ENDMACRO                     ; end macro

.MACRO LOG                    ; setting macro to log values into the memory
ST X+, @0                     ; store indirect the input in X and increment automatically
CHECK_AND_LOOP_RAM_END        ; check if ram end reached after every input
.ENDMACRO                     ; end macro


;Initialization
LDI ZEROREG, ZEROS
LDI ONESREG, ONES

OUT DDRA, ZEROREG             ; Input for temp
OUT DDRB, ZEROREG             ; Input for moisture
OUT DDRC, ZEROREG             ; Input for water

OUT DDRD, ONESREG             ; Output for temp LED
OUT DDRE, ONESREG             ; Output for moisture LED
STS DDRF, ONESREG             ; Output for water LED

LDI TEMPREG, 0b11110          ; Bit 0 for request input and bit 4 for led output for acknowledge
STS DDRG, TEMPREG             ; Output for acknowledge from LED 4 and Input for request from Pin 0

LDI XH, MEM_HI                ; storing XH with 0x01
LDI XL, MEM_LO                ; storing XL with 0x00

;Requesting
Request:
STS PORTG, ZEROREG            ; setting acknowledge as zero to close the LED
LDS TEMPREG, PING             ; Getting input from Pin G for request and storing in temprorary register
SBRS TEMPREG, REQPIN          ; Checking if the input for Request is 1 or not, if it is move to Capture phase
RJMP Request                  ; if request is 0, keep on looping till request is 1

;Capture
IN TEMPERATURE, PINA          ; Input taken from PIN A for Temperature
NOP                           ; setting no operation to prevent problems in input
IN MOISTURE, PINB             ; Input taken from PIN B for Moisture
NOP                           ; setting no operation to prevent problems in input
IN WATER, PINC                ; Input taken from PIN C for Water
```

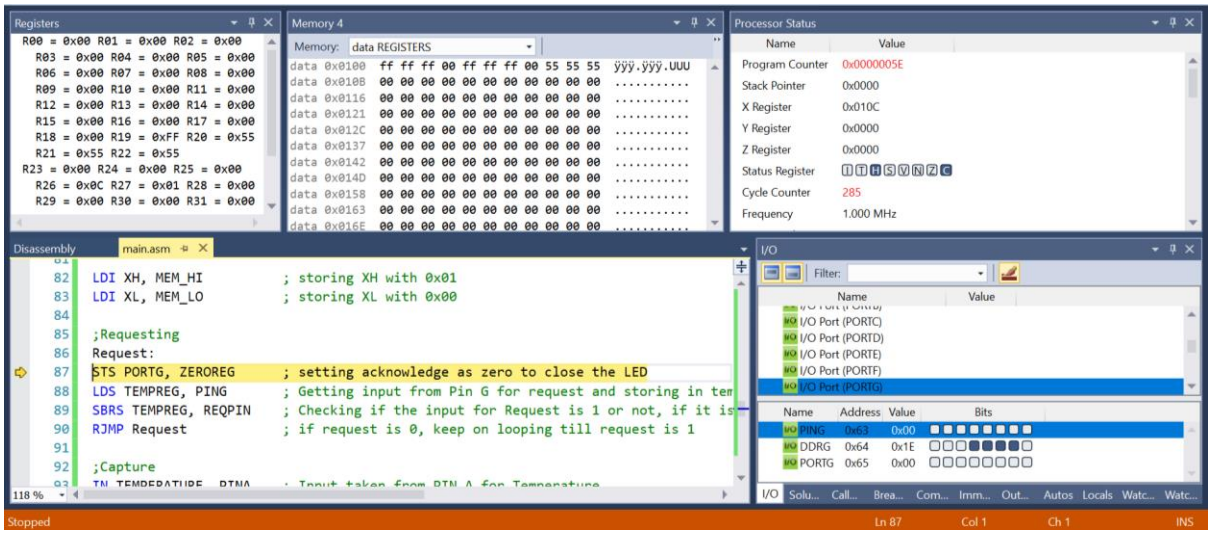
```
;Acknowledge
LDI TEMPREG, 0x10
STS PORTG, TEMPREG              ; storing 0x10 in port G to turn on acknowledge LED

;Santize
CHECK_LIMITS TEMPERATURE, T_LO_LMT, T_HI_LMT ; calling macro to check if temperature is within limits, else sanitize
CHECK_LIMITS MOISTURE, M_LO_LMT, M_HI_LMT    ; calling macro to check if moisture is within limits, else sanitize
CHECK_LIMITS WATER, W_LO_LMT, W_HI_LMT       ; calling macro to check if water level is within limits, else sanitize

;Displaying
OUT PORTD, TEMPERATURE          ; output to port D to turn on LEDs for temperature value
OUT PORTE, MOISTURE             ; output to port E to turn on LEDs for moisture value
STS PORTF, WATER                ; output to port F to turn on LEDs for water level value

;Logging
LOG TEMPERATURE                 ; store value of temperature in memory and round robin if memory is full
LOG MOISTURE                    ; store value of moisture in memory and round robin if memory is full
LOG WATER                       ; store value of water level in memory and round robin if memory is full
LDI TEMPREG, 0                  ; Loading ASCII 0 i.e. null character, in temporary register
LOG TEMPREG                     ; store null character in memory

RJMP Request                    ; loop from Request to input and log next set of values
```

Debugging and Verification:

I. Program starts running and data within expected range is received, but is not captured without a Request input. Acknowledge is also not asserted without the Request input.

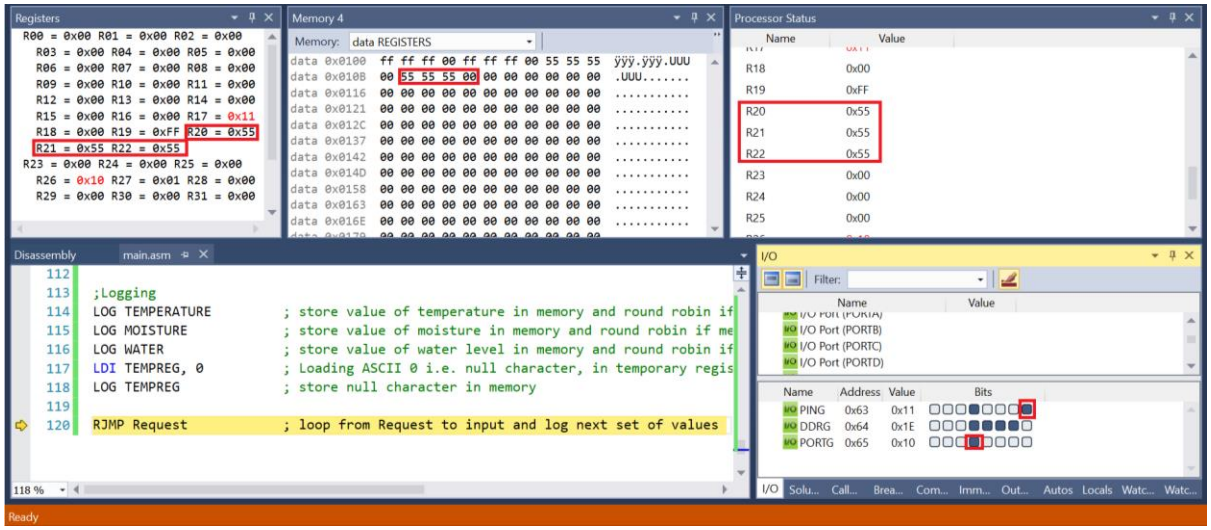


As seen from the red boxes in the “Registers” tab, the data is provided within the expected range i.e. 0x55 for temperature (R20), moisture (R21), and water level (R22). Before capture, the code is stuck in a loop due to request input (bit 0 of pin G) being “0” as shown by the pink box in “I/O” tab. Acknowledge is also not asserted as shown by (port G bit 4) the purple box in the “I/O” tab since the request input is “0”.

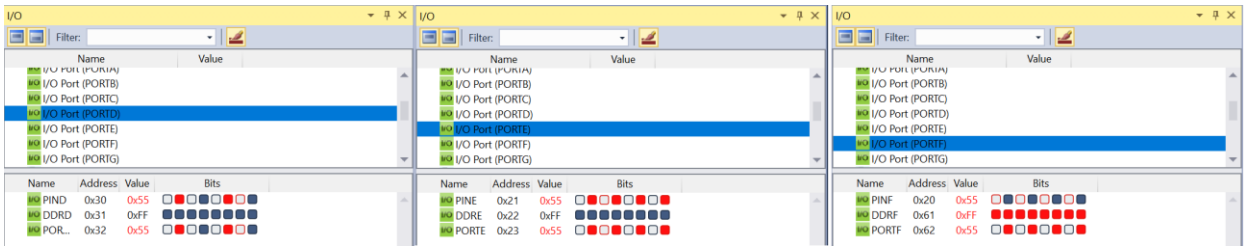


As seen above, the code moves to line 90 and jumps to line 87 since bit 0 for pin G is not set to 1. This keeps repeating until bit 0 for pin G is set to 1 which makes SBRS (skip if bit in register is set) skip the line which jumps to line 87.

II. Registers showing captured data same as inputs, asserted Acknowledge in response to Request, and output data from the first Request, and corresponding logged data to memory within expected range for the first data set.

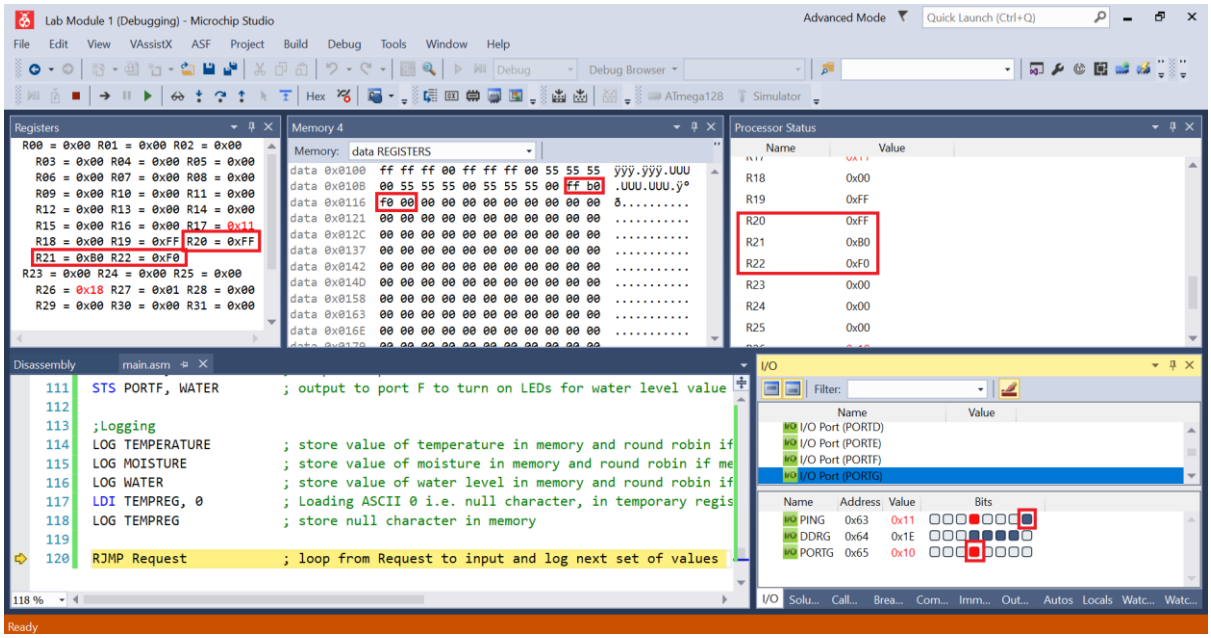


As seen from the red boxes in “Registers” tab and “Processor Status” tab, registers have been given the input 0x55 for temperature (R20), moisture (R21), and water level (R22). The “I/O” tab shows the Request input from pin G bit 0 and Acknowledge is asserted as shown by port G bit 4 (both outlined by red boxes). The “Memory” tab shows the Logged values into the memory (outlined by red box) where the 00 is due to the null pointer (ASCII ‘0’).

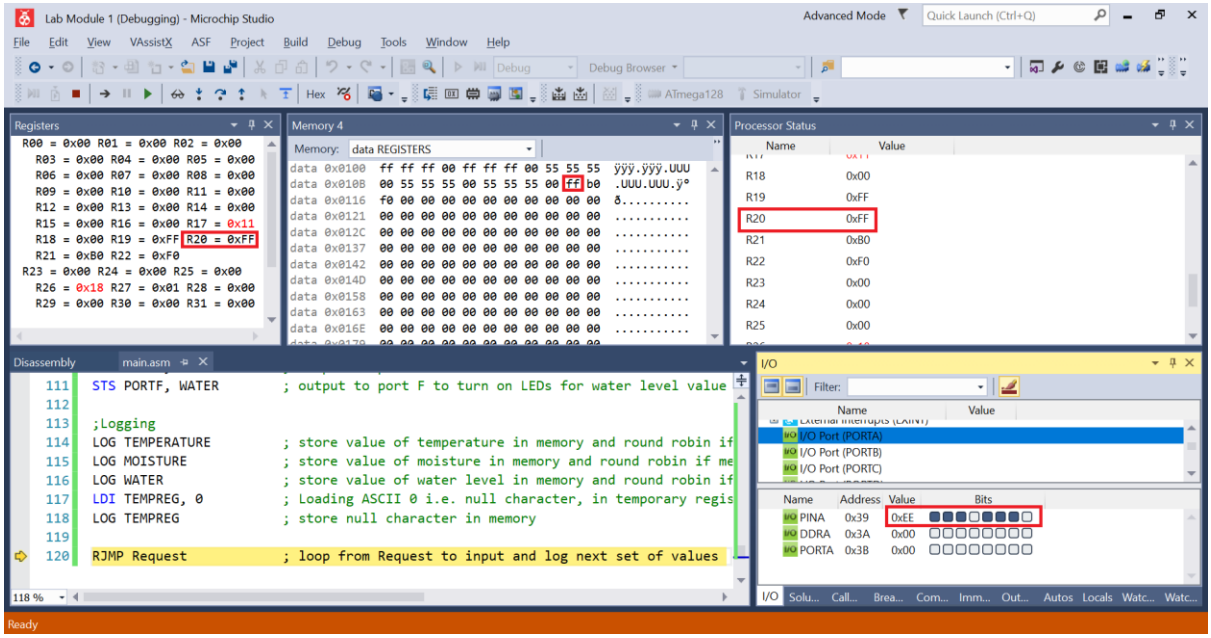


Outputs for temperature, moisture, and water level are shown by port D, port E, and port F respectively as shown above.

III. Same as the previous scenario, except one of the parameter values is outside the expected range.

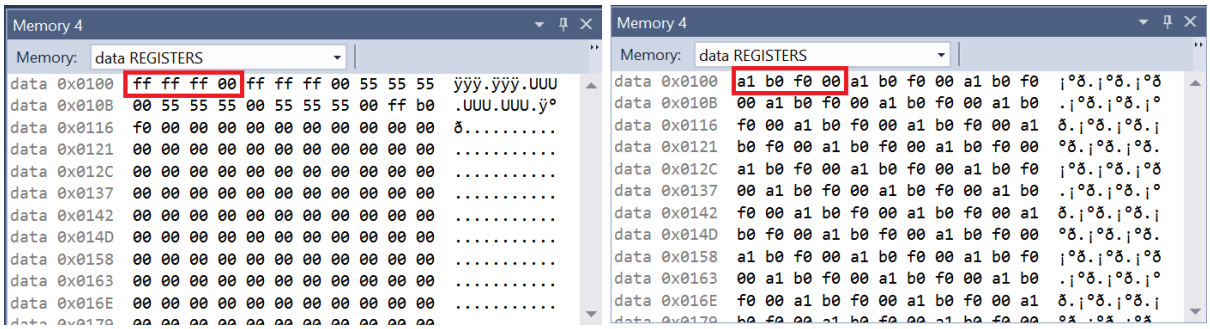


As seen from the red boxes in “Registers” tab and “Processor Status” tab, registers have been given the input 0xB0 and 0xF0 for moisture (R21), and water level (R22). The “I/O” tab shows the Request input from pin G bit 0 and Acknowledge is asserted as shown by port G bit 4 (both outlined by red boxes). The “Memory” tab shows the Logged values into the memory (outlined by red box) where the 00 is due to the null pointer (ASCII ‘0’). In this case the only difference is that temperature value assigned (0xEE) was out of range so the value is Sanitized. The output and Log is therefore displayed as 0xFF as shown in the picture and below.

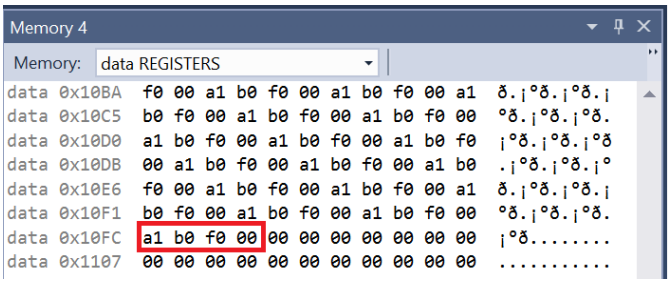


As seen above, the input in port A is 0xEE which is out of the range so the value is Sanitized and the output is 0xFF and so is the logged value in the memory (seen in “Registers”, “Memory 4”, and “Processor Status” tabs).

IV. The case when SRAM data memory is full (address $10FF has been written) and the last received set of data is logged by overwriting the first data set at the beginning of the memory (address $100).

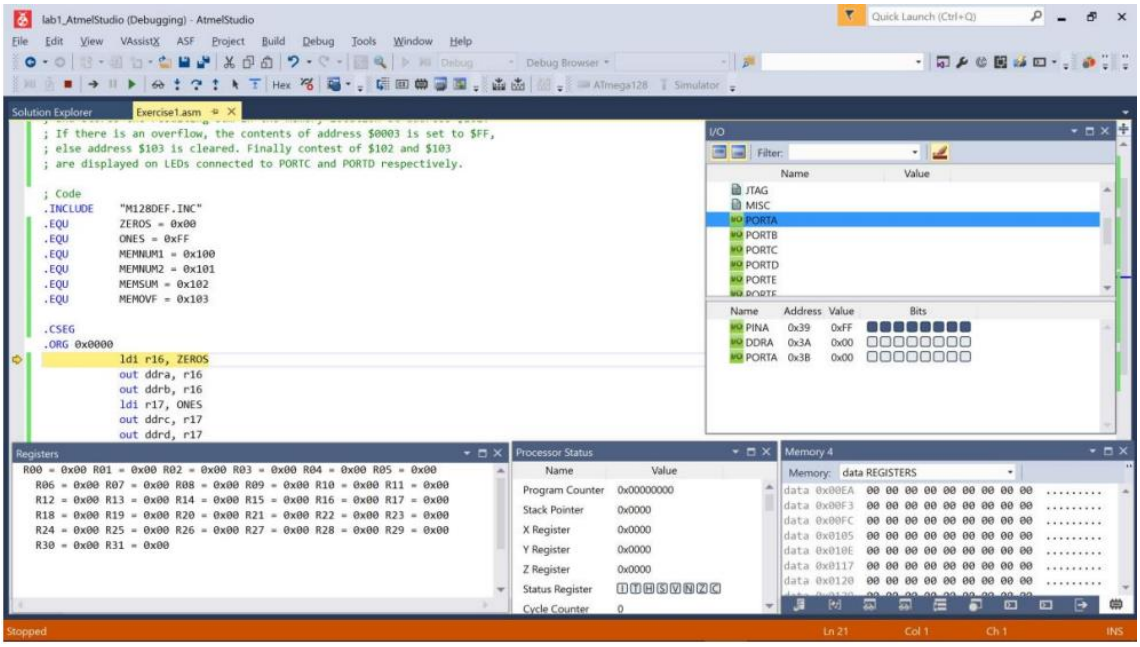


In the image above, the left side shows the memory from case III above where the red box indicates that the first four values are ff, ff, ff, and 00. When we fill the memory using inputs 0xA1, 0xB1, and 0xF0, the last received set of data is logged by overwriting the first data set at the beginning of the memory (i.e. $100).



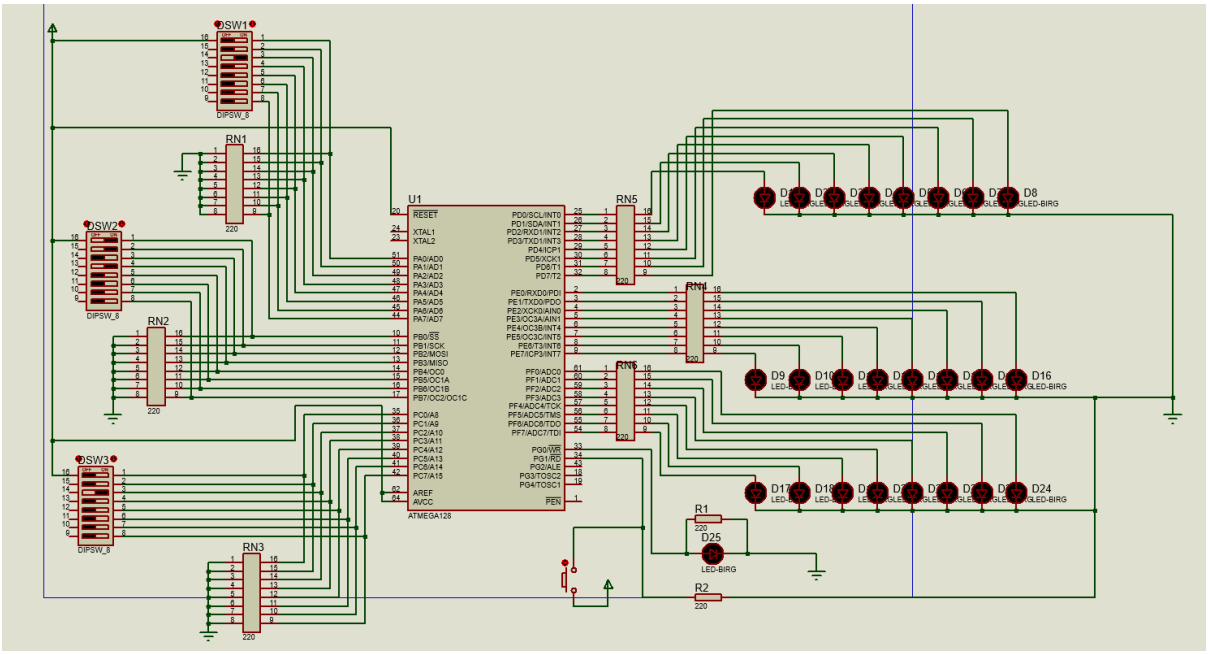
The image above proves that the memory was filled up to 0x10FF and that the last value was the null pointer so the next value i.e. 0xA1 is stored by overwriting memory address 0x100.

V. Which lines in the code do not affect the machine state at all?



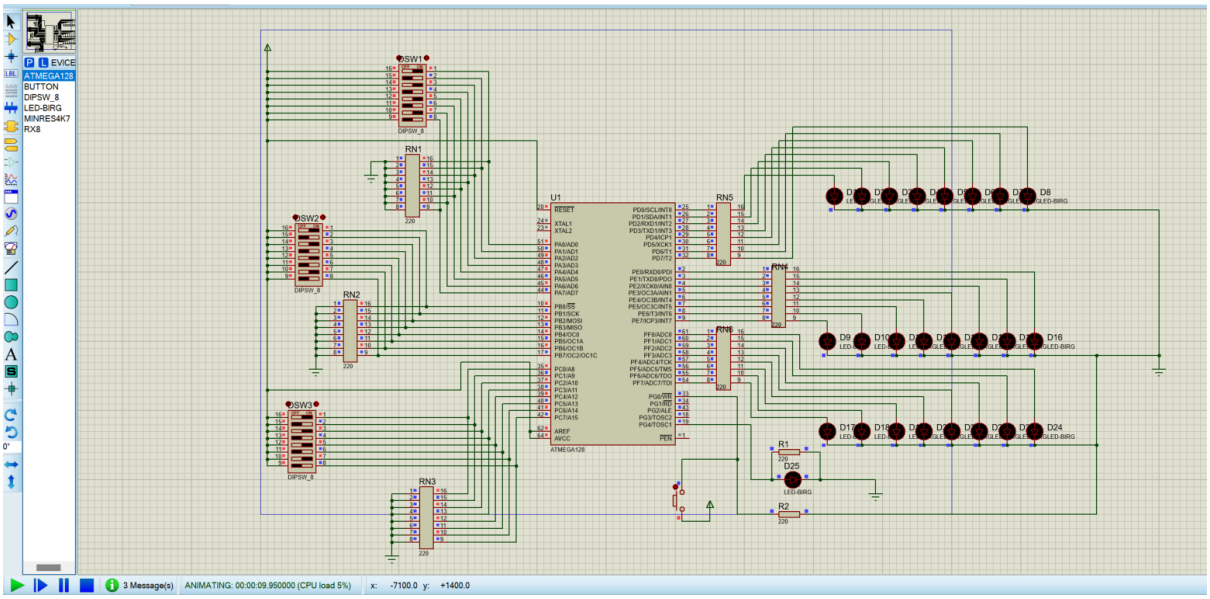
From .INCLUDE to the .CSEG, the doesn't affect machine state since they are just there to include external library and to define constants to make the code easier to read. ORG sets value of program counter, LDI changes the register value, and OUT also changes the register value in the I/O.

Schematic:



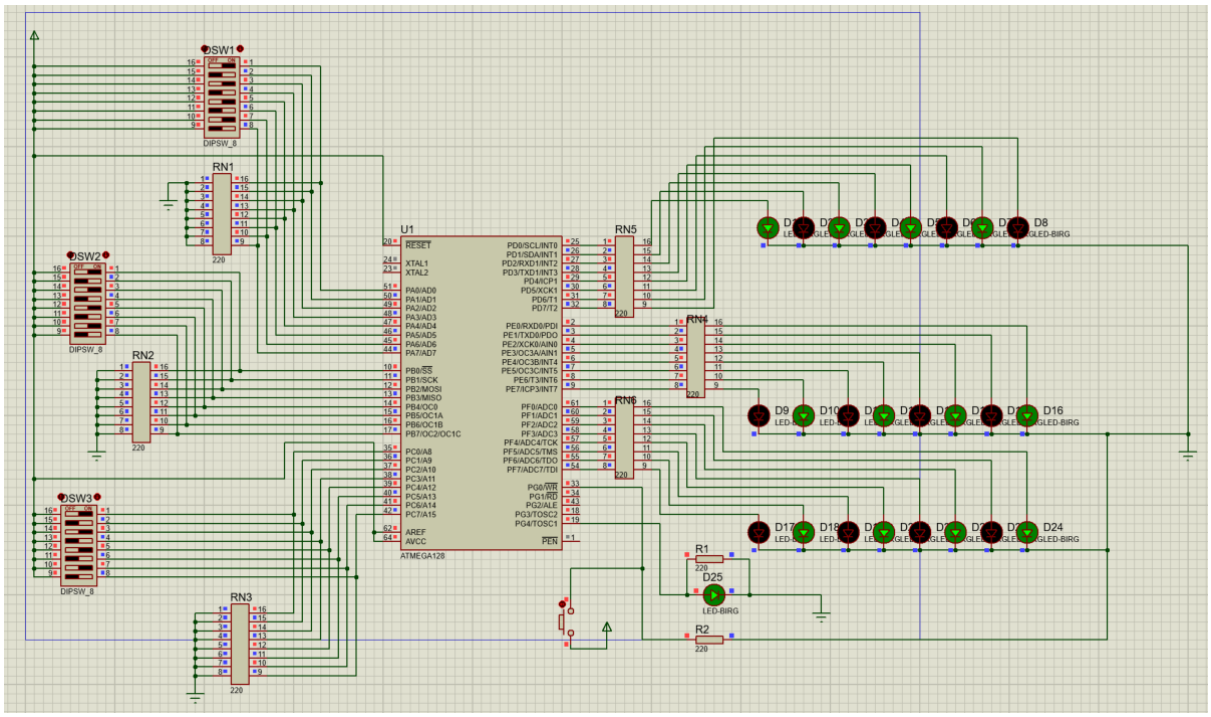
Simulation:

I. All inputs are within acceptable range but the Request button not pushed.



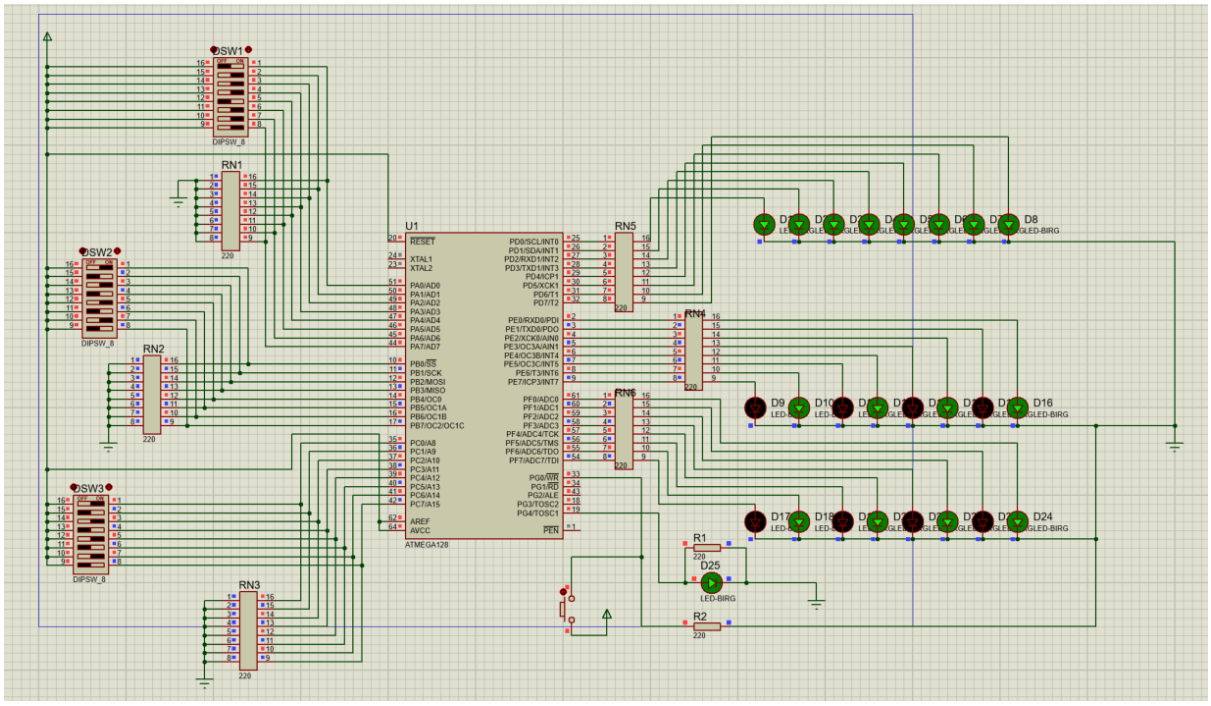
Over here, the request button is not pushed so the values are not taken into the MCU therefore the LEDs are all turned off.

II. All inputs are within acceptable range and the Request button is pushed.



As is seen above, the Acknowledge LED is on along with all the output LEDs that display the same value as the input switches as per expectations.

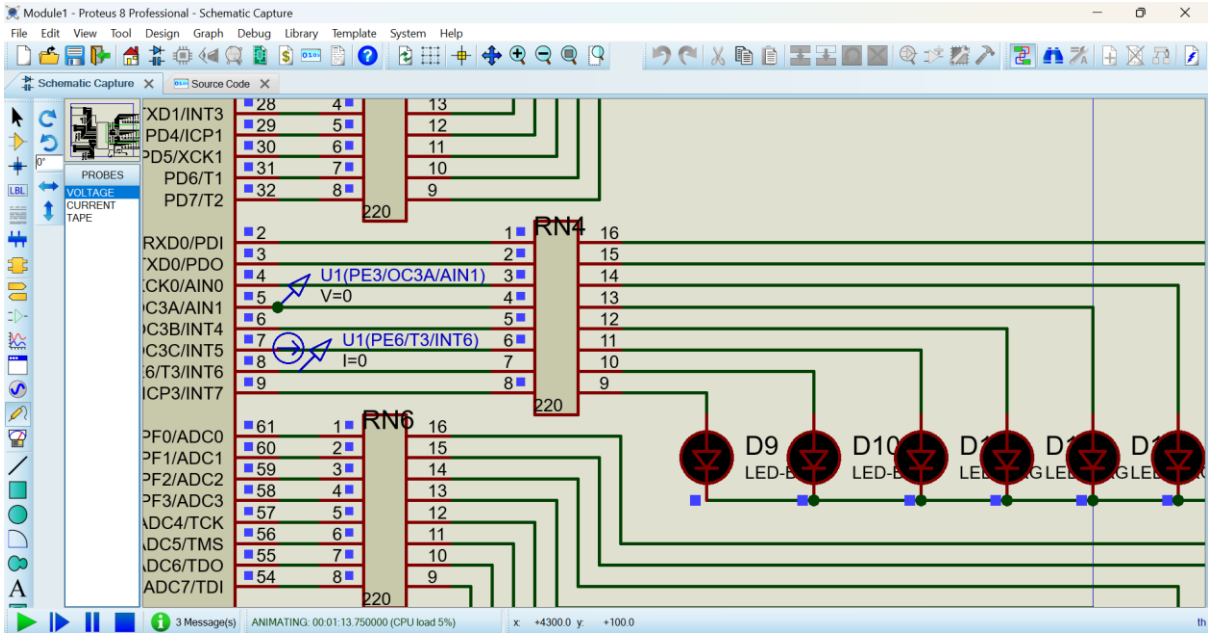
III. Same as previous case but one input is outside acceptable range.



Temperature is given a value beyond its range and as a result it gets Sanitized and all the LEDs in the temperature output light up representing 0xFF. The remaining output LEDs are on according to the given input since they are within the defined range. The Acknowledge LED is also on.

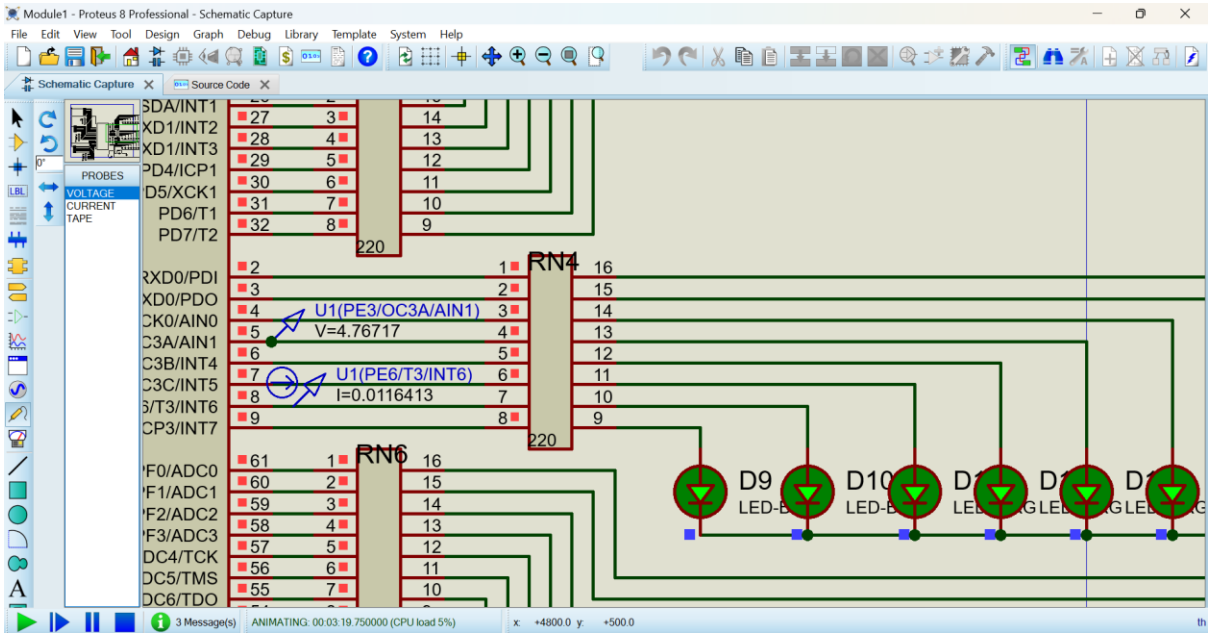
Probe Measurements:

I. When the output port is off (signal is low).



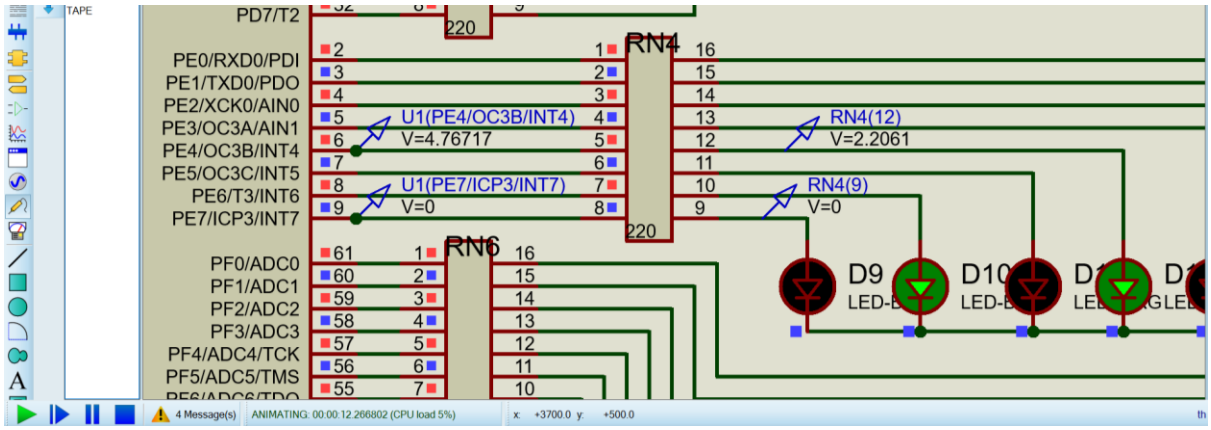
As expected, when the Request button is not pushed, there is no output and LEDs do not light up which mean no voltage (V = 0V) and no current (I = 0A).

II. When the output port is on (signal is high).



As seen, when the LEDs light up (by pushing the Request button and giving any input) the voltage is V = 4.76717 V and the current is I = 0.0116513 A (using a 220 ohm resister).

III. Voltage drops across LEDs



When the LED is off, the voltage remains zero so the voltage drop V_{drop} = 0 V.

When the LED is on, the voltage drops and the value V_{drop} = 4.76717-2.2061 = 2.56107 V where the voltage going into the LED is V = 2.2061 V.

Questions:

I. What is the difference between RAM and ROM? Why is there a need for both a Flash Memory and an EEPROM?

Computer memory comes in the forms of RAM and ROM. Data that the microcontroller needs to access rapidly is stored in RAM, or random access memory. ROM, which stands for Read-Only Memory, is used to store permanent information that cannot be altered while a device is in use. The primary distinction between RAM and ROM is that RAM is volatile, meaning that when power is taken, its contents are lost, but ROM is non-volatile, meaning that even when power is removed, its data are kept.

Flash memory and EEPROM are both used because they have unique qualities that are appropriate for various uses. Program code is kept in flash memory while EEPROM is used to store data that must be kept even after the power is turned off. Flash memory can only be written to in huge chunks but is comparatively quick and may be programmed numerous times. EEPROM, on the other hand, is slower and has a limited number of write cycles, but it can be written to in smaller increments. This makes it better suited for storing small amounts of data that need to be updated frequently.

II. What is the function of the control unit? How does the control unit get the instruction that it must execute? How does it know the number of bytes in an instruction?

Control unit is an important part of an MCU as it executes functions such as fetching instructions from memory, decoding them, and executing them in proper sequence. The control unit gets the instructions from memory (stored at specific locations) and uses the program counter to keep track of the memory address of the next instruction to be fetched. The control unit is able to determine the number of bytes in an instruction via its opcode and since AVR is RISC architecture, the instruction size is either two or four bytes.