EMT 3104: Mechatronic System Programming I

Location: Mechatronic Laboratory

Number of Sessions: 9:00 am to 12:00 pm and 2:00pm to 5:00 pm

GENERAL INTRODUCTION

DEFINITION OF MECHATRONIC SYSTEMS

Mechatronics combines various disciplines in engineering, including mechanics, computing/programming, control, and electronics. Mechatronic systems are thus mechanical and electrical systems that are controlled through programming implemented using sensors, actuators and other embedded electronic devices.

Programming is the design and implementation of computer understandable instructions in order to achieve efficient and low cost operation of systems. These computer instructions are written using programming languages which are specific to different embedded systems and controllers. Examples of embedded system controllers include Microprocessors (μP), Microcontrollers (μC), Programmable Logic Controllers (PLC), and Field Programmable Gate Arrays (FPGA).

WHAT IS AN EMBEDDED SYSTEM?

An embedded system is one that uses one or more microcomputers running custom dedicated programs and connected to specialized hardware, to perform a dedicated set of functions. This can be contrasted with a general-purpose computer such as the familiar desktop or notebook, which are not designed to run only one dedicated program with one specialized set of hardware. Due to the continued changes in current technology, this definition continues to be refined. Some examples of embedded systems are:

- Alarm / security system
- Automobile cruise control
- Heating / air conditioning thermostat
- Microwave oven
- Traffic light controller
- Vending machine
- Gas pump
- Game console (Play Station)
- Irrigation system controller
- Oscilloscope
- Mars Rover

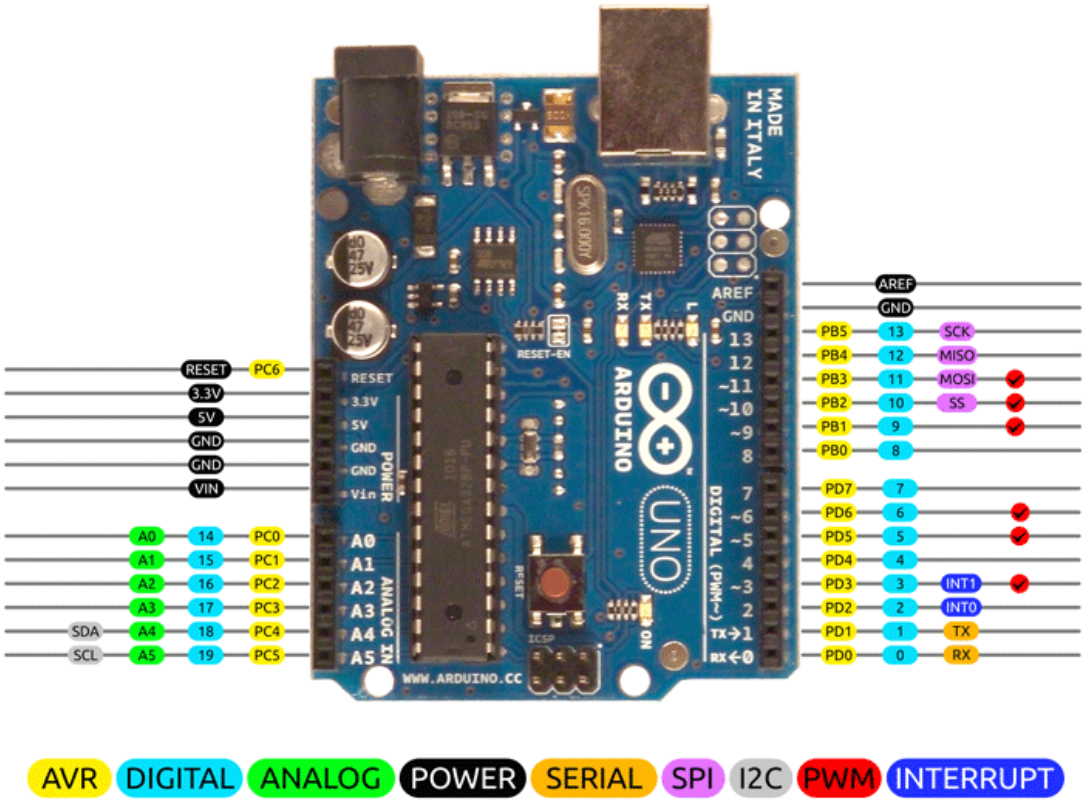
WHAT MICROCONTROLLER FAMILIES WILL WE CONSIDER?

To give a bit of an overview of the different flavours of microcontrollers available, this unit will be written around one 8-bit family (the Atmel AVR) i.e. Atmega328P and is generally coded in C language. Other popular microcontroller families do exist such as PIC and ARM microcontrollers but these are left for the student to consider. The main programming concepts remain the same it is just the syntax that changes.

WHAT ELSE IS REQUIRED FOR THIS SESSION?

Proteus electronic development platform and a coding environment
Atmel Studio 7 and above.
Since it is best to practically verify the concepts learnt in class, it is strongly recommend that the students strives to obtain, either a microcontroller training/development board, or even just a bare μC chip, assorted components and a powered breadboard. (Making your own microcontroller board).
Arduino IDE

AVR to Arduino UNO



Port Registers

Port registers allow for lower-level and faster manipulation of the i/o pins of the microcontroller on an Arduino board. The chips used on the Arduino board (ATmega168/ATmega328p) have three ports:

- B (digital pin 8 to 13)
- C (analog input pins)
- D (digital pins 0 to 7)

Each port is controlled by three registers, which are also defined variables in the arduino language. The DDR register, determines whether the pin is an INPUT or OUTPUT. The PORT register controls whether the pin is HIGH or LOW, and the PIN register reads the state of INPUT pins set to input with pinMode(). DDR and PORT registers may be both written to, and read. PIN registers correspond to the state of inputs and may only be read.

PORTD maps to Arduino digital pins 0 to 7

DDRD - The Port D Data Direction Register - read/write
PORTD - The Port D Data Register - read/write
PIND - The Port D Input Pins Register - read only

PORTB maps to Arduino digital pins 8 to 13. The two high bits (6 & 7) map to the crystal pins and are not usable

DDRB - The Port B Data Direction Register - read/write
PORTB - The Port B Data Register - read/write
PINB - The Port B Input Pins Register - read only

PORTC maps to Arduino analog pins 0 to 5. Pins 6 & 7 are only accessible on the Arduino Mini

DDRC - The Port C Data Direction Register - read/write
PORTC - The Port C Data Register - read/write
PINC - The Port C Input Pins Register - read only

Each bit of these registers corresponds to a single pin; e.g. the low bit of DDRB, PORTB, and PINB refers to pin PB0 (digital pin 8).

Examples

Referring to the pin map above, the PortD registers control Arduino digital pins 0 to 7.
You should note, however, that pins 0 & 1 are used for serial communications for programming and debugging the Arduino, so changing these pins should usually be avoided unless needed for serial input or output functions. Be aware that this can interfere with program download or debugging.
DDRD is the direction register for Port D (Arduino digital pins 0-7). The bits in this register control whether the pins in PORTD are configured as inputs or outputs so, for example:

```
DDRD |= 0b11111100;          // this is safer as it sets pins 2 to 7 as outputs
                             // without changing the value of pins 0 & 1, which are RX & TX
```

PORTD is the register for the state of the outputs. For example;

```
PORTD |= 0b10101000; // sets digital pins 7,5,3 HIGH
```

You will only see 5 volts on these pins however if the pins have been set as outputs using the DDRD register
PIND is the input register variable It will read all of the digital input pins at the same time.

NB: Programming microcontrollers using C is an important skill to master and below are some of the benefits when compared to programming using simplified IDEs such as the arduino IDE.

Pros:

Here are some of the positive aspects of direct port access:

- You may need to be able to turn pins on and off very quickly, meaning within fractions of a microsecond. If you look at the source code in lib/targets/arduino/wiring.c, you will see that digitalRead() and digitalWrite() are each about a dozen or so lines of code, which get compiled into quite a few machine instructions. Each machine instruction requires one clock cycle at 16MHz, which can add up in time-sensitive applications. Direct port access can do the same job in a lot fewer clock cycles.
- Sometimes you might need to set multiple output pins at exactly the same time. Calling digitalWrite(10,HIGH); followed by digitalWrite(11,HIGH); will cause pin 10 to go HIGH several microseconds before pin 11, which may confuse certain time-sensitive external digital circuits you have hooked up. Alternatively, you could set both pins high at exactly the same moment in time using PORTB |= B1100;
- If you are running low on program memory, you can use these tricks to make your code smaller. It requires a lot fewer bytes of compiled code to simultaneously write a bunch of hardware pins simultaneously via the port registers than it would using a for loop to set each pin separately. In some cases, this might make the difference between your program fitting in flash memory or not!

Cons

Despite the above positives, here are some other reasons why you would want to use the arduino IDE to program your microcontroller.

- The code is much more difficult for you to debug and maintain when written in C and is harder for other people to understand. It only takes a few microseconds for the processor to execute code, but it might take hours for you to figure out why it isn't working right and fix it. But the computer's time is very cheap, measured in the cost of the electricity you feed it. Usually it is much better to write code the most obvious way.
- The code is less portable. If you use digitalRead() and digitalWrite(), it is much easier to write code that will run on all of the Atmel microcontrollers, whereas the control and port registers can be different on each kind of microcontroller.
- It is a lot easier to cause unintentional malfunctions with direct port access. Notice how the line DDRD = 0b11111110; above mentions that it must leave pin 0 as an input pin. Pin 0 is the receive line (RX) on the serial port. It would be very easy to accidentally cause your serial port to stop working by changing pin 0 into an output pin! Now that would be very confusing when you suddenly are unable to receive serial data.

Programming Arduino UNO from Atmel Studio 7

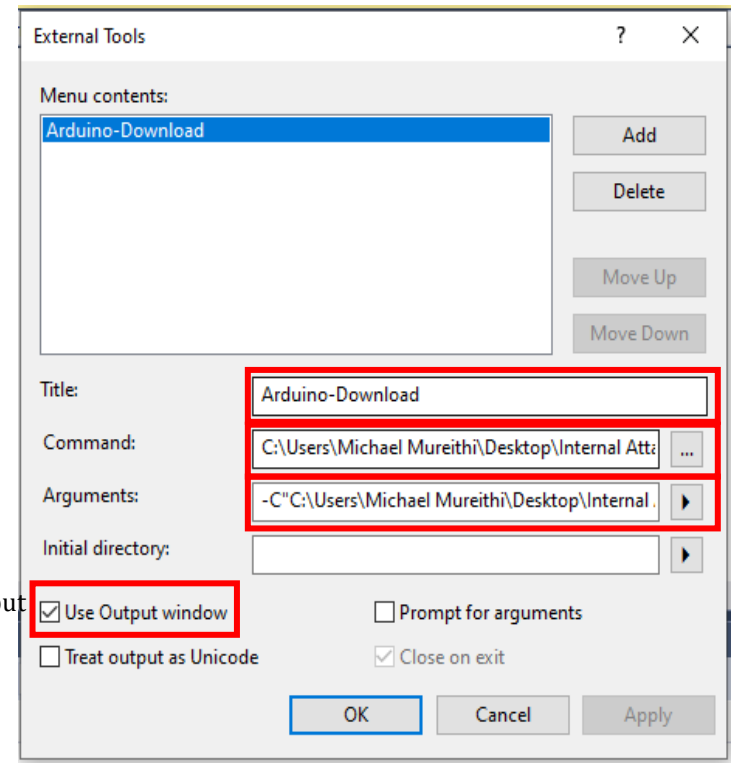
1. Download the latest version of the Arduino IDE.
2. Connect an Arduino UNO to the computer and open the IDE.
3. In the File/preferences section, select the "Show verbose output during: upload".
4. In the Tools/Board and Tools/Port sections, ensure you have selected the right options.
5. Compile and upload the blink example to ensure the Arduino board is working.
6. Once uploaded, proceed to the IDE terminal and access and copy paste the command line from the 3rd line of the terminal output to a notepad.
   Example:
   C:\Users\Michael Mureithi\Desktop\Internal Attachment\Softwares\arduino-1.8.10\hardware\tools\avr/bin/avrdude -CC:\Users\Michael Mureithi\Desktop\Internal Attachment\Softwares\arduino-1.8.10\hardware\tools\avr/etc/avrdude.conf -v -patmega328p -carduino -PCOM4 -b115200 -D -Uflash:w:C:\Users\MICHAE~1\AppData\Local\Temp\arduino_build_418246/Blink.ino.hex:i
7. C:\Users\Michael Mureithi\Desktop\Internal Attachment\Softwares\arduino-1.8.10\hardware\tools\avr/bin/avrdude
   →This is the command segment of this line, add an .exe to the end i.e.
   C:\Users\Michael Mureithi\Desktop\Internal Attachment\Softwares\arduino-1.8.10\hardware\tools\avr/bin/avrdude.exe
8. -CC:\Users\Michael Mureithi\Desktop\Internal Attachment\Softwares\arduino-1.8.10\hardware\tools\avr/etc/avrdude.conf -v -patmega328p -carduino -PCOM4 -b115200 -D -Uflash:w:C:\Users\MICHAE~1\AppData\Local\Temp\arduino_build_418246/Blink.ino.hex:i
   This is the argument section. Add quotation marks ("") to the file name areas and change the last file name to "$(ProjectDir)Debug\$(TargetName).hex"
   The final argument section should look similar to the following example:
   -C"C:\Users\Michael Mureithi\Desktop\Internal Attachment\Softwares\arduino-1.8.10\hardware\tools\avr/etc/avrdude.conf" -v -patmega328p -carduino -PCOM4 -b115200 -D -Uflash:w:"$(ProjectDir)Debug\$(TargetName).hex":i

9. Open Atmel studio and start an new project "blink_test". Select the Atmega328P as the microcontroller. Write a short blink LED code as shown below:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>


int main(void){
   DDRB |= (1<<5);
   PORTB |= (1<<5);
    while (1) {
           PORTB |= (1<<5);
           _delay_ms(500);
           PORTB &= ~(1<<5);
           _delay_ms(500);
    }
}
```

10. Proceed to the top menu bar/Tools/External tools…



Select a suitable tool

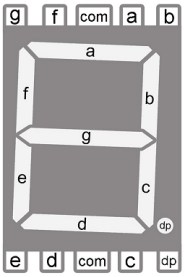
Paste the command line you developed earlier

Paste the argument line you developed earlier

Select the use output window checkbox

11. Build the program and upload to the arduino using Arduino-Download option in tools.

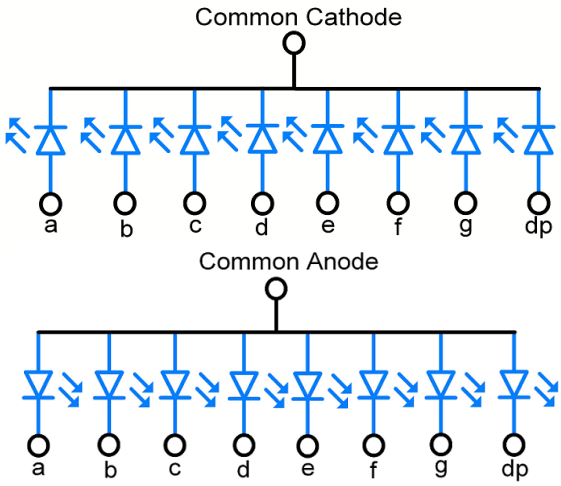
7-segment LED Display



So far we have handled simple single pin output devices such as an LED but now we move on to more complex methods of display. To begin with, we will consider a typical 7-segment LED displays are formed by LED segments. It is basically used to display numerical values from 0 to 9, and other alphanumeric characters (A, B, C, D, E, and F). One more segment is also present there which is used as decimal point. Several of these segment LED can be combined to form a digital display.

Connection Types

In 7-segment displays there are two types, common anode and common cathode.

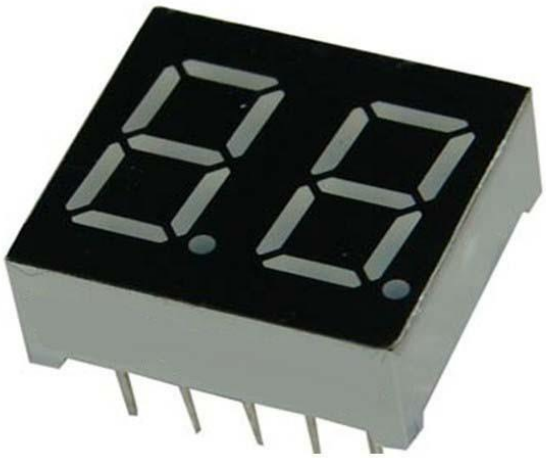


1. **Common Anode (CA)**

In common anode display, all anode pins are connected together to VCC and LEDs are controlled via cathode terminals. It means to turn ON LED (segment), we have to make that cathode pin logic LOW or Ground.

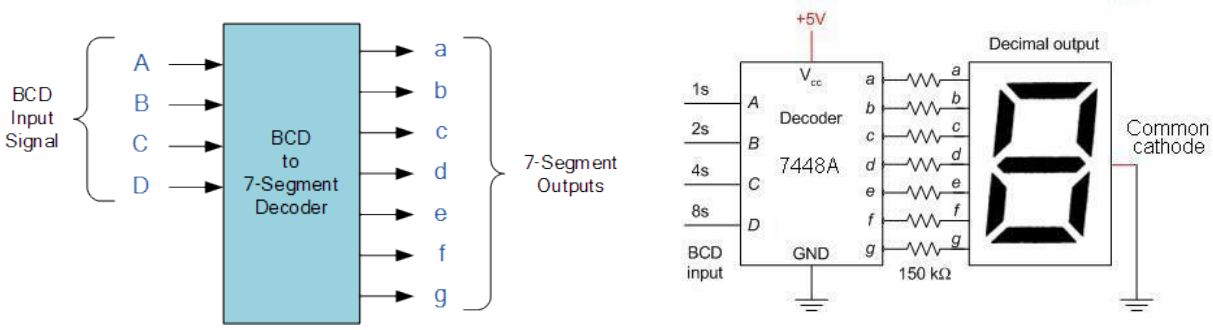
2. **Common Cathode (CC)**

In common cathode display, all cathode pins are connected together and led are controlled via anode terminal. It means to turn ON LED (segment), we have to apply proper voltage to the anode pin.



	Binary	HEX
0	00111111	0x3F
1	00000110	0x06
2	01011011	0x5b
3	01001111	0x4f
4	01100110	0x66
5	01101101	0x6d
6	01111101	0x7d
7	01110000	0x70
8	01111111	0x7f
9	01101111	0x6f

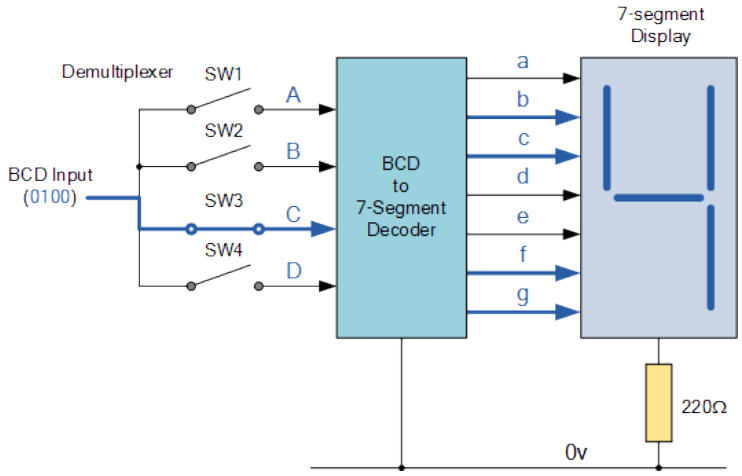
BCD to 7-Segment Decoder



The use of **packed** BCD allows two BCD digits to be stored within a single byte (8-bits) of data, allowing a single data byte to hold a BCD number in the range of 00 to 99.

An example of the 4-bit BCD input (0100) representing the number "4" is given below.

Display Decoder Example No1



In practice current limiting resistors of about 150Ω to 220Ω would be connected in series between the decoder/driver chip and each LED display segment to limit the maximum current flow. There are different display decoders and drivers available for the different types of available displays, either LED or LCD. For example, the 74LS48 for common-cathode LED types, the 74LS47 for common-anode LED types, or the CMOS CD4543 for liquid crystal display (LCD) types.

Exercise 1.1: 1 Digit 7-Segment Circuit

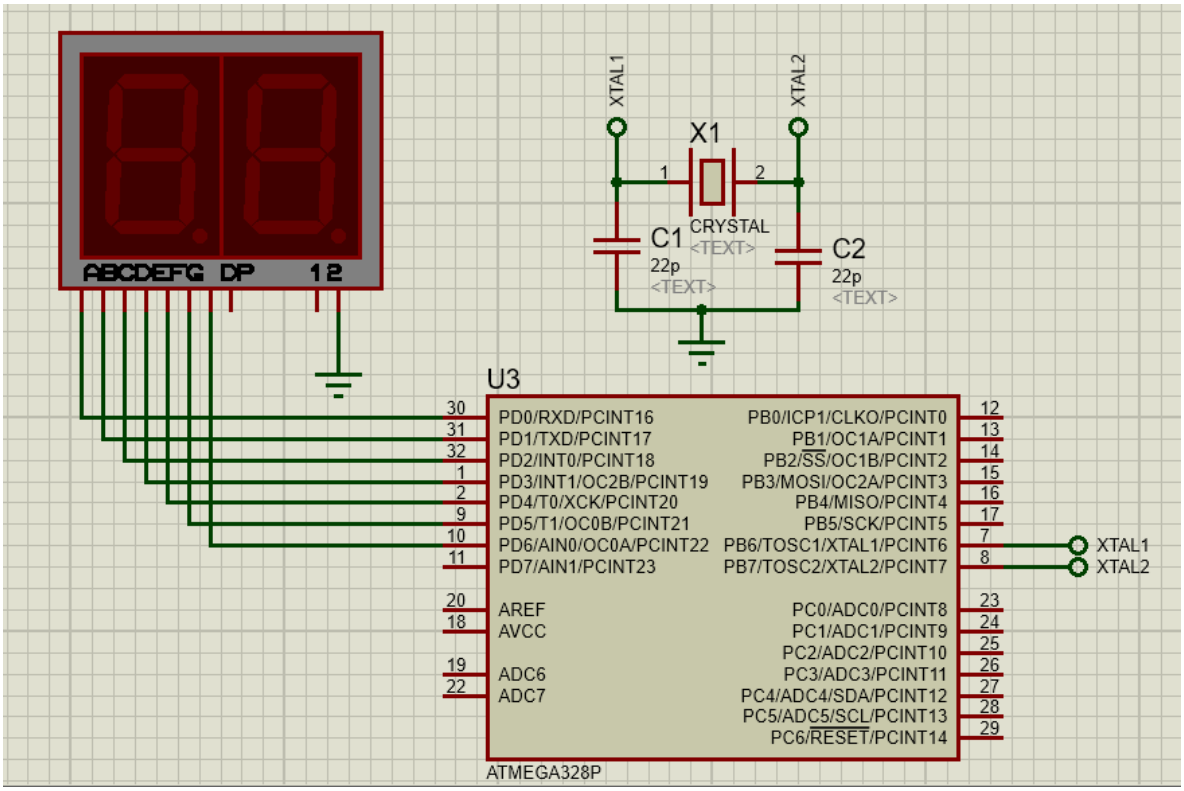
The aim of this task is to be able to control a 1 digit 7 segment LED display, using the Atmega328p. First is to

The aim of this task is to be able to control a 1 digit 7 segment LED display, using the Atmega328p. First is to simulate the circuit in Proteus simulation software and then build the actual circuit using the Arduino UNO. It should be possible to control the 1 digits, separately and cycle through the digits 0-9 with an interval of around 1000ms.

Circuit Components:

1. 2 digit seven segment display
2. Arduino UNO
3. Connecting wire
4. Breadboard
5. Computer for uploading the software.

Proteus Circuit:



Task:

1. Create the circuit above in Proteus Simulation software.
2. Using the code template bellows, write a control program to light the 7-SEG LEDs as stated above.
3. Using the components mentioned above, create the above circuit on a bread board and upload the code and observe the operation of the system.
4. ***NB: Call the supervisor to confirm your circuit before finalizing your connection and powering it.***
5. Record your findings by recording the circuit diagram that you develop and its operations using pictures.
6. Write a group report on the exercise to be submitted for assessment.

Code Example

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#define LED_Dir DDRD        /*define LED Directory*/
#define LED_Port PORTD      /*define LED PORT*/

void display(int digit);
int main(void)
{
	/*initialization*/
	LED_Dir |= 0xff;
	LED_Port = 0xff;

	while (1){
		for(int i=0;i<10;i++){
			display(i);          /* Function to display a digit */
			_delay_ms(1000);     /* wait for 1 second */
		}
	}
}

void display(int digit){
	/*We light each LED one at a time*/
	/* write hex value for Common Cathode display from 0 to 9 */
	char numbers[] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
	if(digit < 10 ){
		LED_Port = numbers[digit];
	}
}
```

Exercise 1.2 : 2 Digit 7-Segment Circuit

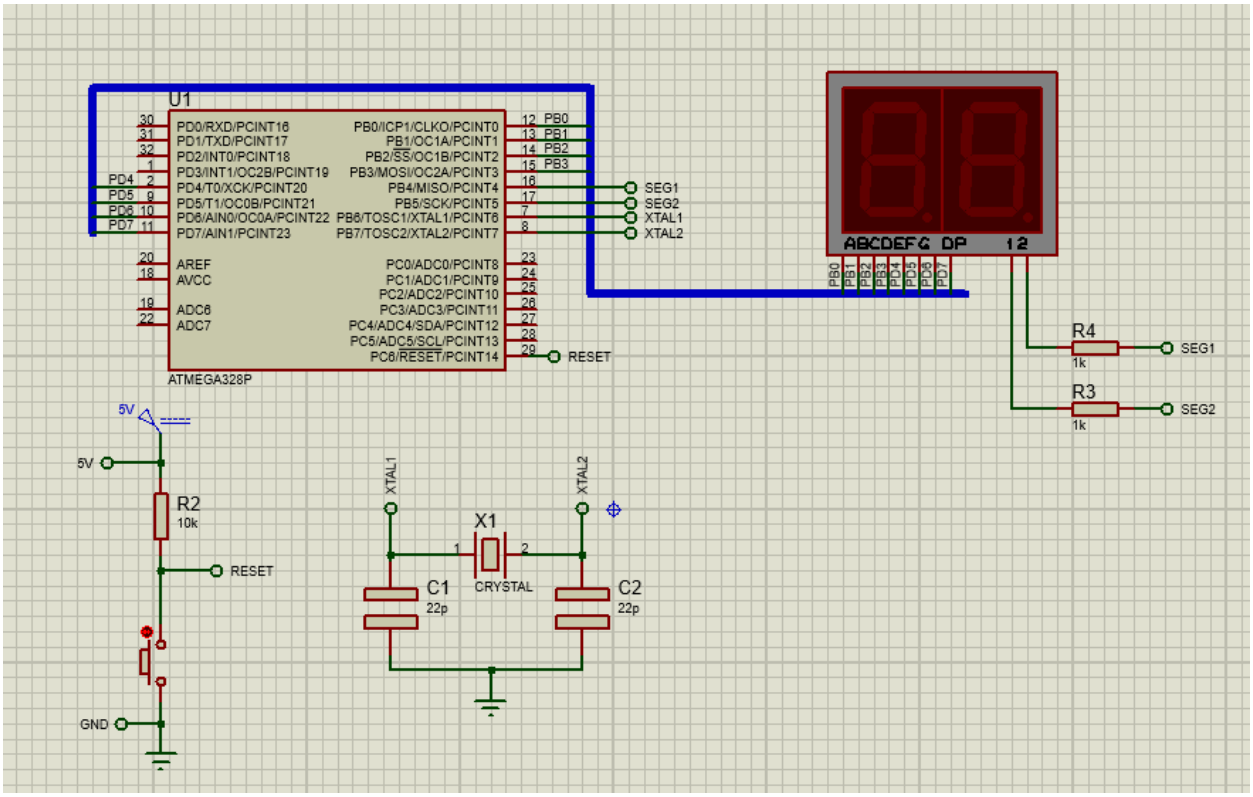
The aim of this task is to be able to control a 2 digit 7 segment LED display, using the Atmega328p. First is to

The aim of this task is to be able to control a 2 digit 7 segment LED display, using the Atmega328p. First is to simulate the circuit in Proteus simulation software and then build the actual circuit using the Arduino UNO. It should be possible to control the 2 digits, separately and cycle through the digits 0-9 with an interval of around 500ms.

Circuit Components:

1. 2 digit seven segment display
2. Arduino UNO
3. Connecting wire
4. Breadboard
5. Computer for uploading the software.

Proteus Circuit:



Task:

7. Create the circuit above in Proteus Simulation software.
8. Using the code template bellows, write a control program to light the 7-SEG LEDs as stated above.
9. Using the components mentioned above, create the above circuit on a bread board and upload the code and observe the operation of the system.
10. ***NB: Call the supervisor to confirm your circuit before finalizing your connection and powering it.***
11. Record your findings by recording the circuit diagram that you develop and its operations using pictures.
12. Write a group report on the exercise to be submitted for assessment.

Code Example

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

char display[] = {0x3F ,0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x67};
int SEG1 = 4;
int SEG2 = 5;

void displayLED(int input){
        char BH = (~PORTB & 0xF0);
        char BL = (display[input] & 0x0F);
        PORTB = ~(BH|BL);

        char DL = (~PORTD & 0x0F);
        char DH = (display[input] & 0xF0);
        PORTD = ~(DH|DL);
}

int main(void){
   DDRB |= 0xFF;
   DDRD |= 0xFF;
   int count = 0;

    while (1) {
                PORTB |= (1<<SEG1);
                PORTB |= (1<<SEG2);

                if(count < 10){
                        displayLED(count);
                        count++;
                        _delay_ms(100);
                }
                if(count == 10){
                        count = 0;
                }
        }
}
```

Timers in AVR ATmega328p:

Timers is a feature in micro-controllers which enables the precise counting of time. In a lot of application, we require to process data, inputs (sensor data) or outputs (variable voltage – PWM), generate waveforms and delays, count events etc. with precise timing. Thus we will look at how to interface with onboard timers that are available on the Atmega328p.
In AVR ATmega328p, there are two timers:

- **Timer0**: 8-bit timer
- **Timer1**: 16-bit timer

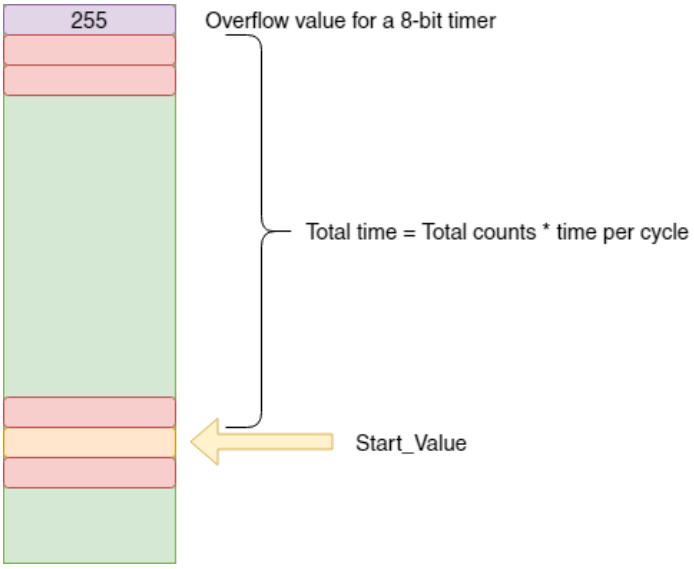
TCNTn: Timer / Counter Register
Every timer has timer/counter register. It is zero upon reset. We can access value or write a value to this register. It counts up with each clock pulse.
TOVn: Timer Overflow Flag
Each timer has Timer Overflow flag. When timer overflows, this flag will get set.
TCCRn : Timer Counter Control Register
This register is used for setting the modes of timer/counter.
OCRn : Output Compare Register
The value in this register is compared with the content of the TCNTn register. When they are equal, OCFn flag will get set. It is often used to set a limit/set point value for the timer.
Let us look at Timer0 as an example to understand its operation.

Prescaler:

From the above calculation we can observe that with a 16 bit timer, the max count value is 65535. With an operation frequency of 16MHz, the maximum time period that we can count is $65535 * 6.25 * 10^{-8}s = 4.1ms$. We can notice that this is a somewhat limited time range. In order to expand our time range, we use Prescaler.

In essence with the prescaler, the time per cycle calculation becomes:

$$Time\ cycle = \frac{1}{F_{osc}} * Prescaler$$



15.11.2 TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – ICNC1: Input Capture Noise Canceler**

Setting this bit (to one) activates the input capture noise canceler. When the noise canceler is activated, the input from the input capture pin (ICP1) is filtered. The filter function requires four successive equal valued samples of the ICP1 pin for changing its output. The input capture is therefore delayed by four oscillator cycles when the noise canceler is enabled.

- **Bit 6 – ICES1: Input Capture Edge Select**

This bit selects which edge on the input capture pin (ICP1) that is used to trigger a capture event. When the ICES1 bit is written to zero, a falling (negative) edge is used as trigger, and when the ICES1 bit is written to one, a rising (positive) edge will trigger the capture.

When a capture is triggered according to the ICES1 setting, the counter value is copied into the input capture register (ICR1). The event will also set the input capture flag (ICF1), and this can be used to cause an input capture interrupt, if this interrupt is enabled.

When the ICR1 is used as TOP value (see description of the WGM13:0 bits located in the TCCR1A and the TCCR1B register), the ICP1 is disconnected and consequently the input capture function is disabled.

- **Bit 5 – Reserved Bit**

This bit is reserved for future use. For ensuring compatibility with future devices, this bit must be written to zero when TCCR1B is written.

- **Bit 4:3 – WGM13:2: Waveform Generation Mode**

See TCCR1A register description.

- **Bit 2:0 – CS12:0: Clock Select**

The three clock select bits select the clock source to be used by the Timer/Counter, see Figure 15-10 on page 106 and Figure 15-11 on page 106.

Table 15-6. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (no prescaling)
0	1	0	$clk_{I/O}$/8 (from prescaler)
0	1	1	$clk_{I/O}$/64 (from prescaler)
1	0	0	$clk_{I/O}$/256 (from prescaler)
1	0	1	$clk_{I/O}$/1024 (from prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

If external pin modes are used for the Timer/Counter1, transitions on the T1 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

15.11.4 TCNT1H and TCNT1L – Timer/Counter1

Bit	7	6	5	4	3	2	1	0	
(0x85)	TCNT1[15:8]								TCNT1H
(0x84)	TCNT1[7:0]								TCNT1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The two Timer/Counter I/O locations (TCNT1H and TCNT1L, combined TCNT1) give direct access, both for read and for write operations, to the Timer/Counter unit 16-bit counter. To ensure that both the high and low bytes are read and written simultaneously when the CPU accesses these registers, the access is performed using an 8-bit temporary high byte register (TEMP). This temporary register is shared by all the other 16-bit registers. See Section 15.3 “Accessing 16-bit Registers” on page 91.

Modifying the counter (TCNT1) while the counter is running introduces a risk of missing a compare match between TCNT1 and one of the OCR1x registers.

Writing to the TCNT1 register blocks (removes) the compare match on the following timer clock for all compare units.

Internal Timer Interrupts:

Internal interrupts as mentioned previously are interrupts triggered by internal devices such as Timers, ADC, UART, SPI etc. We will see how timers can be configured to produce interrupts when the timer overflows. The TIMSK register is used to do this setup

TIMSK: Timer / Counter Interrupt Mask Register

We have to set TOIE1 (Timer1 Overflow Interrupt Enable) bit in TIMSK register to set the timer1 interrupt, so that as soon as the Timer1 overflows, the controller jumps to the Timer1 interrupt routine.

15.11.8 TIMSK1 – Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7, 6 – Res: Reserved Bits**

These bits are unused bits in the Atmel® ATmega328P, and will always read as zero.

- **Bit 5 – ICIE1: Timer/Counter1, Input Capture Interrupt Enable**

When this bit is written to one, and the I-flag in the status register is set (interrupts globally enabled), the Timer/Counter1 input capture interrupt is enabled. The corresponding interrupt vector (see Section 11. "Interrupts" on page 49) is executed when the ICF1 flag, located in TIFR1, is set.

- **Bit 4, 3 – Res: Reserved Bits**

These bits are unused bits in the Atmel ATmega328P, and will always read as zero.

- **Bit 2 – OCIE1B: Timer/Counter1, Output Compare B Match Interrupt Enable**

When this bit is written to one, and the I-flag in the status register is set (interrupts globally enabled), the Timer/Counter1 output compare B match interrupt is enabled. The corresponding interrupt vector (see Section 11. "Interrupts" on page 49) is executed when the OCF1B flag, located in TIFR1, is set.

- **Bit 1 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable**

When this bit is written to one, and the I-flag in the status register is set (interrupts globally enabled), the Timer/Counter1 output compare A match interrupt is enabled. The corresponding interrupt vector (see Section 11. "Interrupts" on page 49) is executed when the OCF1A flag, located in TIFR1, is set.

- **Bit 0 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the status register is set (interrupts globally enabled), the Timer/Counter1 overflow interrupt is enabled. The corresponding interrupt vector (see Section 11. "Interrupts" on page 49) is executed when the TOV1 flag, located in TIFR1, is set.

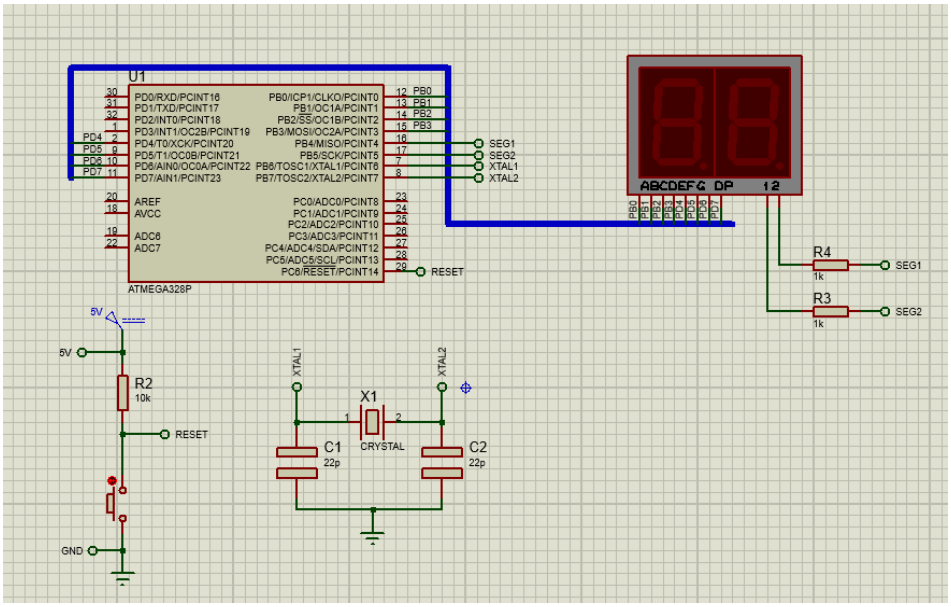
Exercise 2: Control of the 2 digit 7 segment Display.

The aim of this task is to be able to control a 2 digit 7 segment LED display, using the Atmega328p so as to display 2 separate values on the 2 digits of the LED display. The key idea is to use the concept of "Persistence of Vision". In this we switch between the 2 digits with a high enough frequency so that the eye does not perceive that the LED has been switched off. To accomplish this, we need to create a timer interrupt of 30ms which we use to refresh the display. When we refresh the timer, we will display each digit for a period of 5ms.

Circuit Components:

1. 2 digit seven segment display
2. Arduino UNO
3. Connecting wire
4. Breadboard
5. Computer for uploading the software.

Proteus Circuit:



Task:

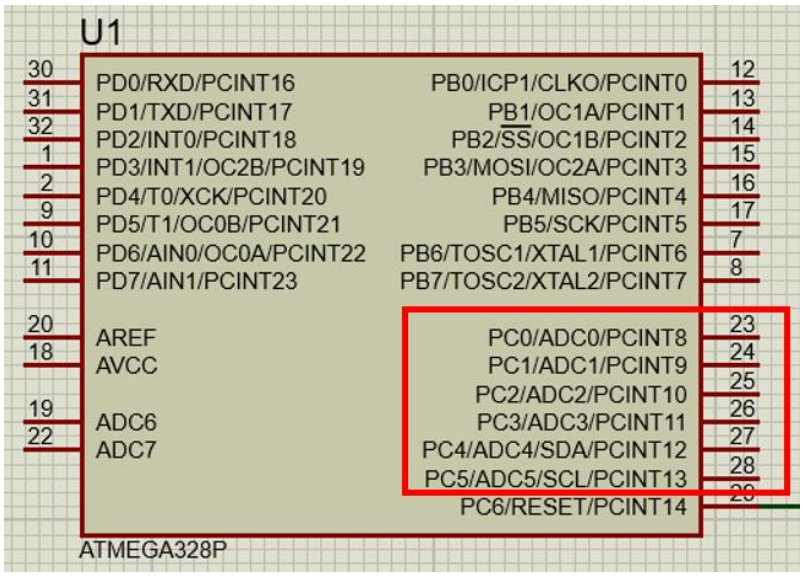
1. Determine the correct timer to use; Timer0 (8 bits) or Timer1 (16 bits) in order to achieve the desired 30ms time period. Having decided the correct timer, calculate the appropriate value that should be loaded into the timer TCNT. Express your answer in hexadecimal values.
2. Implement a persistence of vision program using the template code provided to display the number 1 and 2 simultaneously in Atmel studio 7 and simulate it in Proteus.
3. Once the simulation is complete, load your program into the circuit built in exercise 1.
4. ***NB: Once successful, implement a looping 10 second countdown. This is to be demonstrated to the supervisor for successful completion of the exercise.***
5. Write a group report on the exercise to be submitted for assessment.

```c
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
char display[] = {0x3F ,0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x67};
int SEG1 = 4;
int SEG2 = 5;
int delayvalue = 5000;
int i = 0;
int j = 0;
/*A function to act as a persistence of vision delay*/
void vision_persist(){
	int delay = delayvalue;
	while (0 < delay){
		_delay_us(1);
		--delay;
	}
}
/* A function to reload the timer initial value after the timer overflows */
void sevseg_refreshDisplay(){
	TCNT1L = 0x15;		/* Load TCNT1, count for 30ms*/
	TCNT1H = 0x9F;
	TCCR1B |= (1<<CS11);	/* Start timer1 with 8 pre-scaler*/
	TIMSK1 |= (1<<TOIE1);	/* Enable Timer1 overflow interrupts */
}
void displayLED(int input){ /*Function to display a number in the LED*/
	char BH = (~PORTB & 0xF0);
	char BL = (display[input] & 0x0F);
	PORTB = ~(BH|BL);

	char DL = (~PORTD & 0x0F);
	char DH = (display[input] & 0xF0);
	PORTD = ~(DH|DL);
}
ISR(TIMER1_OVF_vect){ /*Interrupt Service Routine for Timer1*/
	PORTB ^= (1<<SEG1);	/* Toggle SEG1 */
	PORTB ^= (1<<SEG2);	/* Toggle SEG2 */
	displayLED(i);
	vision_persist();

	PORTB ^= (1<<SEG1);	/* Toggle SEG1 */
	PORTB ^= (1<<SEG2);	/* Toggle SEG2 */
	displayLED(j);
	vision_persist();
	sevseg_refreshDisplay();
}
int main(void){
	int count = 0;
	DDRB |= 0xFF;
	DDRD |= 0xFF;
	PORTB |= (1<<SEG1);
	/*********************************************************************/
	/*Interrupt Setup*/
	sevseg_refreshDisplay();
	sei();			/* Enable Global Interrupt */
	/*********************************************************************/
	while (1) {	/*Change this code segment to achieve a 10s countdown*/
		if(count < 10){
			i = count;
			j = 9-count;
			count++;
		}
		if(count == 10){
			count = 0;
		}
		_delay_ms(500);
	}
}
```

Analogue to Digital Conversion (ADC) in AVR ATmega328p:



The controller has 10 bit ADC, means we will get digital output 0 to 1023. i.e. When input is 0V, digital output will be 0V & when input is 5V (and Vref=5V), we will get highest digital output corresponding to 1023 steps, which is 5V. So controller ADC has 1023 steps and

- Step size with Vref=5V : 5/1023 = 4.88 mV.
- Step size with Vref=2.56 : 2.56/1023 = 2.5 mV.

So Digital data output will be Dout = Vin / step size.

ADC Registers: Consulting the Data Sheet

In AVR ADC, we need to understand four main register -

1. ADCH: Holds digital converted data higher byte
2. ADCL: Holds digital converted data lower byte
3. ADMUX: ADC Multiplexer selection register
4. ADCSRA: ADC Control and status register
5. ADCH : ADCL register

We can select any divisor and set frequency Fosc/2, Fosc/4 etc. for ADC, But in AVR, ADC requires an input clock frequency less than 200KHz for max. accuracy. So we have to always take care about not exceeding ADC frequency more than 200KHz. Suppose your clock frequency of AVR is 8MHz, then we must have to use devisor 64 or 128. Because it gives 8MHz/64 = 125KHz, which is lesser than 200KHz.

23.9 Register Description

23.9.1 ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6 – REFS1:0: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in Table 23-3. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

Table 23-3. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, internal V_{REF} turned off
0	1	AV_{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 1.1V voltage reference with external capacitor at AREF pin

- **Bit 5 – ADLAR: ADC Left Adjust Result**

The ADLAR bit affects the presentation of the ADC conversion result in the ADC data register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC data register immediately, regardless of any ongoing conversions. For a complete description of this bit, see Section 23.9.3 "ADCL and ADCH – The ADC Data Register" on page 219.

- **Bit 4 – Res: Reserved Bit**

This bit is an unused bit in the Atmel® ATmega328P, and will always read as zero.

- **Bits 3:0 – MUX3:0: Analog Channel Selection Bits**

The value of these bits selects which analog inputs are connected to the ADC. See Table 23-4 on page 218 for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

From the description in the controller's data sheet on how to use the ADC, we can generate the following functions in order to interact with the ADC.

Code example:

```
void InitADC(){
	// Select Vref=AVcc
	ADMUX |= (1<<REFS0);
	//set prescaller to 128 and enable ADC
	ADCSRA |= (1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)|(1<<ADEN);
}

uint16_t ReadADC(uint8_t ADCchannel){
	//select ADC channel with safety mask
	ADMUX = (ADMUX & 0xF0) | (ADCchannel & 0x0F);
	//single conversion mode
	ADCSRA |= (1<<ADSC);
	// wait until ADC conversion is complete
	while( ADCSRA & (1<<ADSC) );
	return ADC;
}
```

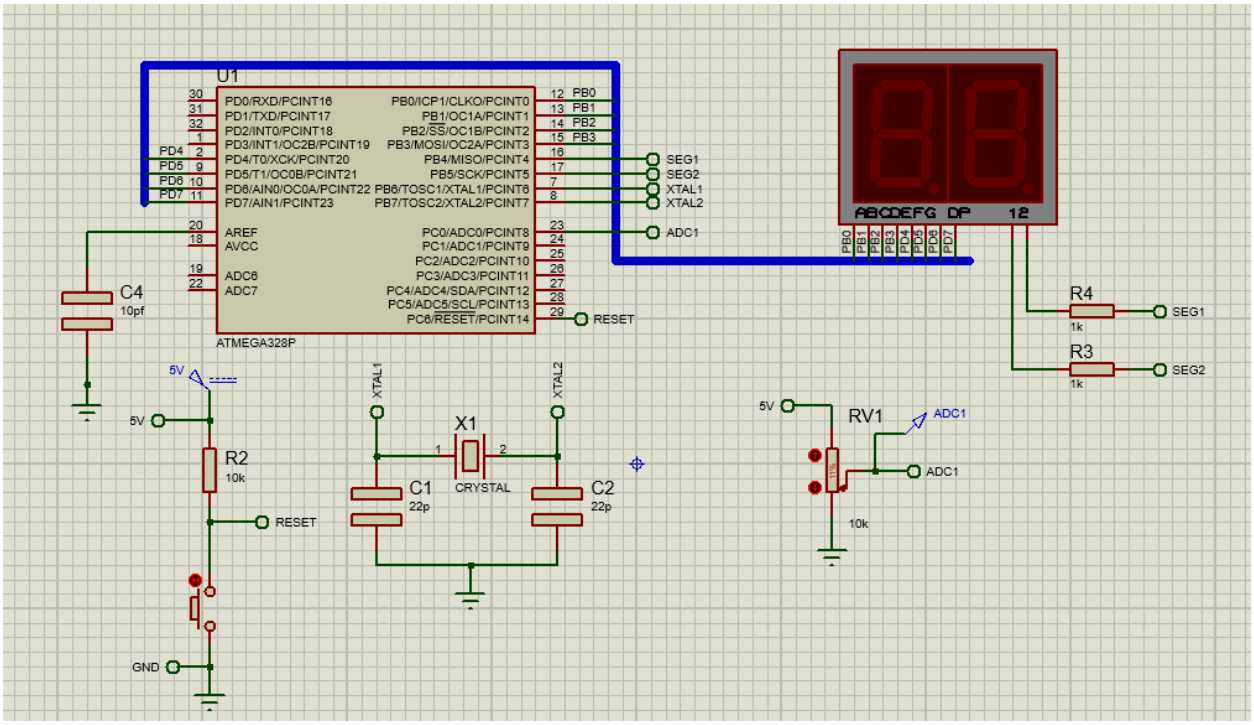
Exercise 3: Using the ADC and display the result on the LED display

The aim of this task is to be able to read a varying voltage using the controller ADC, scale it between a value from 0 to 99 and display it on the 2 digit 7 segment LED display. To accomplish this we need to use the recommended method discussed in the microcontroller data sheet thus there is need to fully understand what the data sheet highlights to successfully perform this task.

Circuit Components:

1. 2 digit seven segment display
2. 10k variable resistor
3. Arduino UNO
4. Connecting wire
5. Breadboard
6. Computer for uploading the software.

Proteus Circuit:



Task:

1. Develop an appropriate flowchart and C program that can read the value of the ADC channel 0 as well as a suitable equation to display the value of the ADC within the range of 0 to 99.
2. Build and compile this program into your Proteus model and simulate.
3. Once the simulation is complete, load your program into the circuit built in exercise 1.
4. ***NB: Once successful, by varying the variable resistor, the value displayed on the 7 segment should range from 0 to 99 which should be demonstrated to the supervisor for successful completion of the exercise.***
5. Write a group report on the exercise to be submitted for assessment.