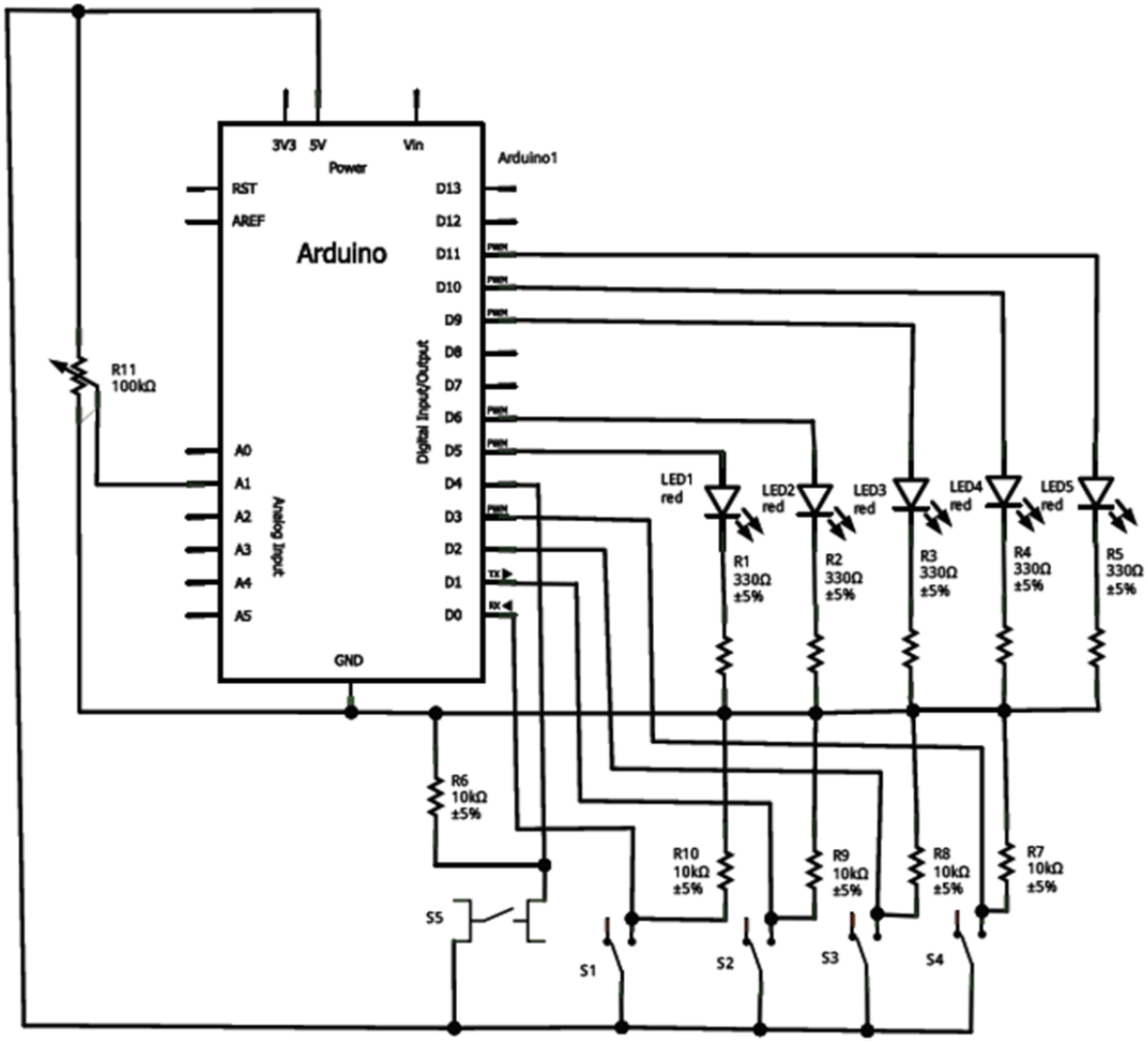
CG2271 Real Time Operating Systems

Tutorial 4

Question 1

This question carries on from last tutorial, where you built a binary to BCD converter. A possible solution to Questions 1a and 2a is shown below:



In summary, in last week's tutorial you were asked to:

i. Implement a binary to BCD converter.
ii. Add in code to read the potentiometer and write the value to a global variable called adc_in.
iii. Add in a timer so that the switches are read and LEDs updated every 5 ms instead of whenever the push-button S5 is pushed.

For this week, you will modify the code so that you control the LED brightness using a 200 Hz PWM frequency (or something as close as possible).

a. Complete the following table showing the mappings between PWM the Uno's digital/PWM pins, the associated timer output pin, and the timer connected to that pin. The second entry has been filled for you.

Arduino Pin Number	Timer Output Pin	Timer
5	OC0B	TIMER0
6	OC0A	TIMER0
9	OC1A	TIMER1
10	OC1B	TIMER1
11	OC2A	TIMER2

b. What is a suitable prescaler value for this program? Can it be achieved? If not what is the closest PWM frequency that you can get?

We have 200=(16000000/510*P), solve for P and we get 16000000/(200*510)
P=156.8

Using P=256 gives us 16000000/(510*256) = 122 Hz. Using P=64 gives us 16000000/(510*64) = 490 Hz. We pick P=256 which gives value that's closer to 200 Hz.

This question demonstrates the unfortunate thing about phase-correct PWM on the AVR: You can't generate many frequencies accurately. This is usually not critical though.

c. Modify your program from Tutorial 3 so that the LEDs are fully off when adc_in is 0, fully on when adc_in is at its maximum value, and intermediate levels of brightness in between.
Note: You can use any timer to implement part c, regardless of whether you used them previously in Tutorial 3. This doesn't work in the real world but our objective here is to see whether you understand PWM programming.

```
// Initialize the timers
void initPWM()
{
	TCNT0=TCNT1H=TCNT1L=TCNT2=0;

	// Enable the OCIExA and OCIExB interrupts for timers 0 and 1
	TIMSK0|=0b110;
	TIMSK1|=0b110;

	// Enable OCIE2A interrupt
	TIMSK2|=0b010;

	// COMxA1 and COMxA0=0b10 (x=0, 1 or 2) in TCCRxA.
	// WGMx2, WGMx1, and WGMx0 = 001. On timer1, WGM13, WGM12, WGM11
	// and WGM10 = 0001 to choose 8-bit phase-correct PWM. WGM13 and WGM12
	// are in TCCR1B. For TCCRxB we set everything to 0. This means that CSx2, CSx1
	// and CSx0 which hold the prescaler value is set to 0, stopping the timer.

	TCCR0A=TCCR1A=TCCR2A=0b10000001;
	TCCR0B=TCCR1B=TCCR2B=0;
}

// Start the timers
void startPWM()
{
	// Set prescaler of 256. This means that CSx2, CSx1 and CSx0 = 100
	TCCR0B|=0b100;
	TCCR1B|=0b100;
	TCCR2B|=0b100;
	sei();
}

// Contains the duty cycle value to be written
unsigned int duty_value;

// ISR for pin 5 (OC0B)
ISR(TIMER0_COMPB_vect)
{
	OCR0B=duty_value;
}
```

```
// ISR for pin 6 (OC0A)
ISR(TIMER0_COMPA_vect)
{
        OCR0A=duty_value;
}

// ISR for pin 9 (OC1A)
ISR(TIMER1_COMPA_vect)
{
        OCR1A=duty_value;
}


// ISR for pin 10 (OC1B)
ISR(TIMER1_COMPA_vect)
{
        OCR1B=duty_value;
}


// ISR for pin 11 (OC2A)
ISR(TIMER2_COMPA_vect)
{
        OCR2A=duty_value;
}

// Code from last week
…
int main()
{
        …
        while(1)
        {
                // Convert adc_in to 0-255 range
                duty_value=(int) (adc_in/1024.0*255.0);
                output(B4, B3toB0);
        }
}
```

Question 2

What are the relative advantages and disadvantages of round robin, round robin with interrupts, function queue scheduling and RTOS-based software architectures? What sorts of applications are best suited to each of these architectures?

Architecture	Advantages	Disadvantages
Round Robin	**Very simple.**	**Not robust - Loop must complete before any sensor is due to be read again or readings may be missed. Addition of slow computations may make response times unacceptable.** **Poor design architecture, hard to extend.**
RR with interrupts.	**Interrupts ensure that sensors are read even if the processing loop is long.**	**More complicated than RR, need to write interrupt handlers and set up interrupt vectors, control registers, etc.** **Cannot prioritize handling.**
Function Q scheduling	**Gives priorities to more important processing functions.**	**Even more complicated than RR with interrupts. Need queue management which adds overheads.**

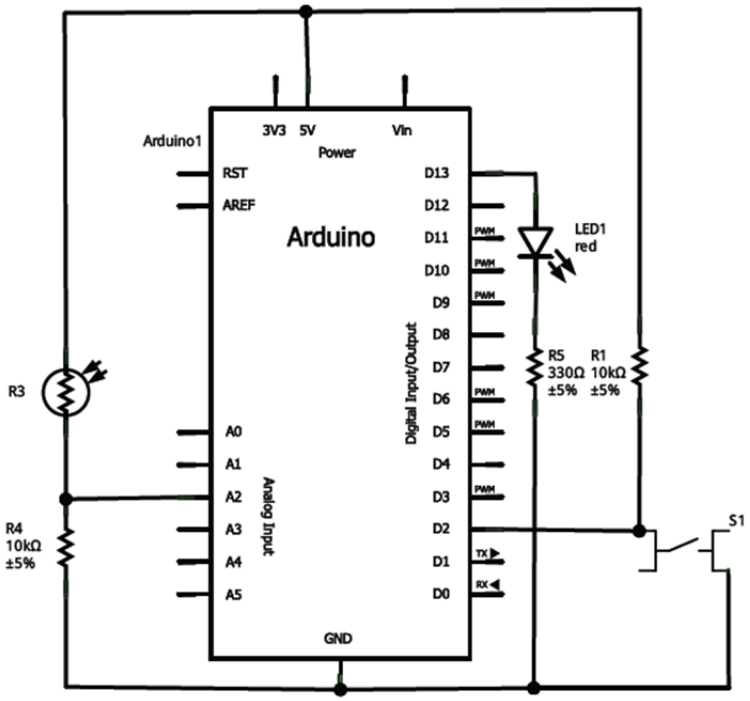
Question 3

Through Google or any other means, research on the INT0 and INT1 external interrupts available on the Atmega microcontrollers.

a. Which Arduino pins do INT0 and INT1 connect to? (Hint: There is an Atmega data sheet in the Lectures folder).

INT0 is connected to PD2 (digital pin 2) and INT1 is connected to PD3 (digital pin 3).

b. Design a circuit so that the pin on the Arduino corresponding to INT0 is pulled LOW when a push button is pressed, and is held high when the button is not pressed. See the "electronics.pdf" file in the Lab folder in IVLE for details of how to use a pull-up resistor. In addition up a photoresistor to analog channel 2 and an LED to digital pin 13. Your circuit must be properly designed with the proper protection resistors, voltage dividers, etc.



c. Write a program that toggles the LED on and off with each button press. I.e. turn on the LED the first time the button is pressed, turn it off the second time, on the third time, etc. You must use INTERRUPTS to do this, not simple GPIO programming. Research on how to set up and handle INT0 properly.

Note: Many online resources show a different way of setting up INT0 that appears to be for versions of AVR that are not compatible with the Atmega series. For example ISC00 and ISC01 are set in MCUCR instead of EICRA. MCUCR in the Atmega does not have ISC00 and ISC01 bits and these methods won't work.

To handle INT0 we need to:

1. **Set up the ISR for INT0_vect**

```
ISR(INT0_vect)
{
    …
}
```

2. **Select the way we intend to trigger INT0 by setting the ISC00 and ISC01 bits in EICRA (External Interrupts Control Register A).**
3.

Table 12-2. Interrupt 0 Sense Control

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

We can use mode 00 or 10 since we are pulling PD2 low. We will use mode 0. This gives us:

EICRA

7	6	5	4	3	2	1	0
-	-	-	-	ISC11	ISC10	ISC01	ISC00
0	0	0	0	?	?	0	0

C statement is: EICRA &= ~(0b11);

4. **Enable the INT0 interrupt in EIMSK (Extern Interrupt Mask) register:**

EIMSK

7	6	5	4	3	2	1	0
-	-	-	-	-	-	INT1	INT0
0	0	0	0	0	0	?	1

C statement is: EIMSK|=0b1;

5. **Finally enable global interrupts: sei()**

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define BIT5  0b00100000

void ledOn()
{
	PORTB |= BIT5;
}

void ledOff()
{
	PORTB &=~BIT5;
}

int toggle;

ISR(INT0_vect)
{
	toggle=!toggle;
}

int main(void)
{
	toggle=1;
	DDRB|=0b00100000;


	// Disable interrupts until we set up everything.
	cli();

	// Set up EICRA
	EICRA &=~(0b11);

	// Set up EIMSK
	EIMSK|=0b1;

	// Enable interrupts
	sei();

    while(1)
    {
		if(toggle)
			ledOn();
		else
			ledOff();
    }

}
```

d. In Tutorial 2 you looked briefly at how to do debouncing. Set up Timer 0 and use it to debounce the push button.

Additional code is shown in *bold italics.*

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define BIT5  0b00100000

void ledOn()
{
	PORTB |= BIT5;
}

void ledOff()
{
	PORTB &=~BIT5;
}

int toggle;
unsigned long saved_timer, timer;

ISR(INT0_vect)
{
	// Ensure at least 10 ms between toggles.
	if(timer-saved_timer>10)
	{
		toggle=!toggle;
		saved_timer=timer;
	}
}

ISR(TIMER0_COMPA_vect)
{
	// Counts number of milliseconds passed
	timer++;
}

int main(void)
{
	toggle=1;
	timer=saved_timer=0;

	// Set pin 5 for output
	DDRB|=0b00100000;


	// Disable interrupts until we set up everything.
	cli();

	// Set up EICRA
	EICRA &=~(0b11);

	// Set up EIMSK
	EIMSK|=0b1;


	// Set up TIMER0 to trigger every ms
	TCNT0=0; // Zero the counter
```

```
        OCR0A=249; // Roll over every 250 counts.
                   // Together with a prescaler of 64 this gives 1 ms interrupts.

        // Set up TCCR0A. Unlike the lecture example, here we disconnect OC0A
        // so that it is not affected by the counter. So
        // COM0A1, COM0A0=00. We also choose CTC mode so WGM02 WGM01 WGM00 = 010
        TCCR0A=0b00000010;

        // Enable OCIE0A interrupt
        TIMSK0|=0b10;

        // Start Timer 0 with a prescaler of 64
        TCCR0B=0b00000011;

        // Enable interrupts
        sei();

    while(1)
    {
              if(toggle)
                     ledOn();
              else
                     ledOff();
    }

}
```

We will extend on this question in Tutorial 5.