CG1112 Engineering Principles and Practices II for CEG

Week 5 Studio 1 – Timers

INTRODUCTION

In this studio you will learn about using timers on the AVR (the microcontroller that is on your Arduino) to manage code execution on the Arduino. We will do two activities today:

Activity 1:

Switches make use of springed contacts that can bounced when pressed, causing spurious on-and-off signals on your input pins. To prevent this, we ignore on or off signals that are spaced less than a few milliseconds apart, assuming that these are caused by the springed contacts bouncing.

In this activity we learn how to "debounce" a switch using a timer. This activity is useful in teaching you how to write routines that time events – in this case, switch presses.

Activity 2:

In Week 2 Studio 2 we had flashing green and red LEDS, and we switched between them by pressing a switch. In this activity we will use the timer to automatically switch between the LEDs. This activity is useful in showing you how to use timers to trigger events at fixed intervals.

This studio is UNGRADED.

TEAM FORMATION

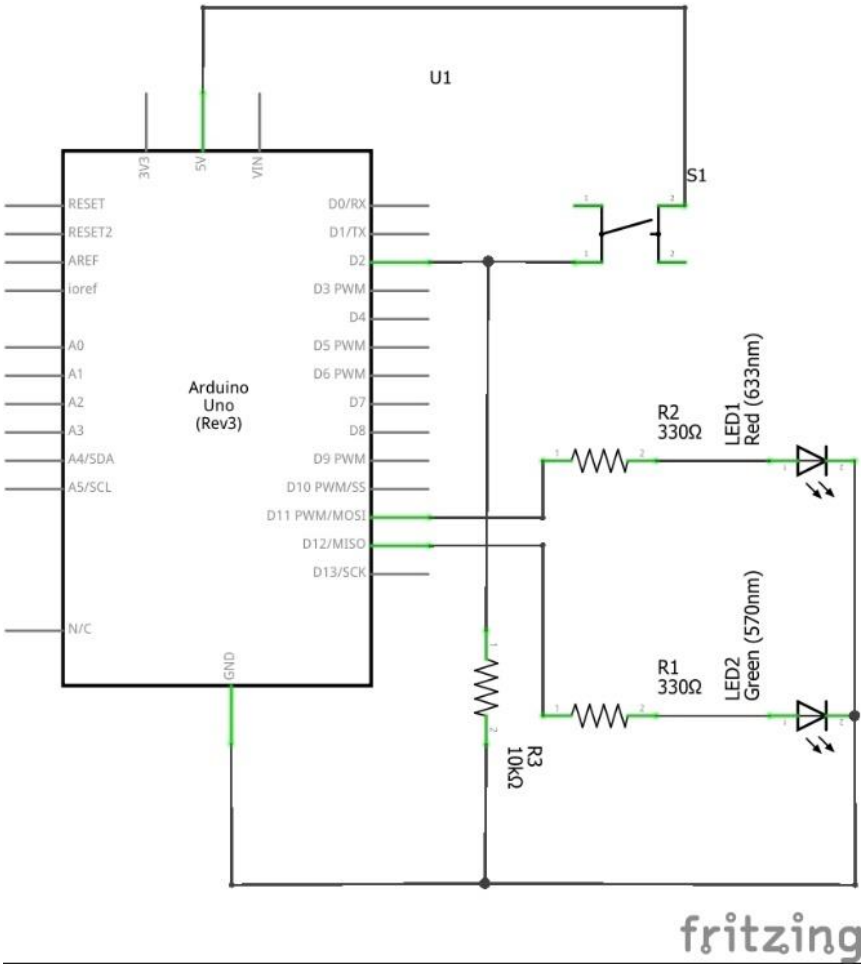
Please work within your assigned sub-teams of 2 to 3 students.

COMPONENT LIST AND CIRCUIT

You will be provided with the following:

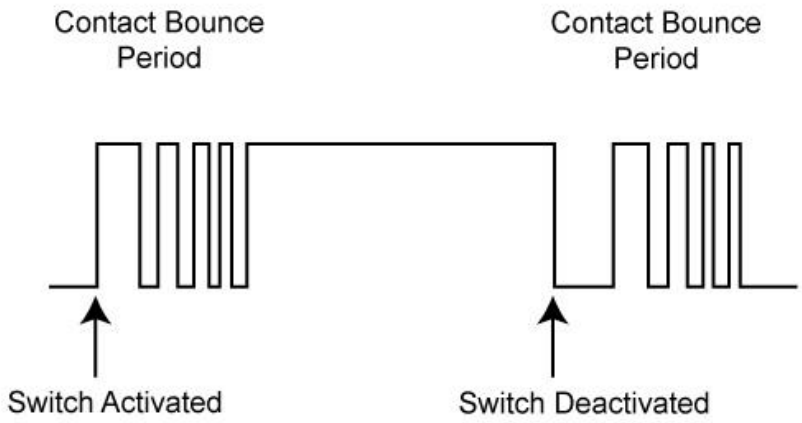
Arduino UNO	x1
LED, Red	x1
LED, Green	x1
Resistor, 330 ohm, 0.25W	x2
Resistor, 10K, 0.25W	x1
Pushbutton Switch	x1
Breadboard	x1
Jumper wires	

Assemble the exact same circuit as in Week 2 Studio 2:



ACTIVITY 1 – BARE-METAL SWITCH DEBOUNCING

You may have observed in Week 2 Studio 2 that switches do not operate reliably. To understand why, we have to remember that the push button switch (and just about all switches) are mechanical devices with metal springs. When you push the switch, the metal contacts within the switch vibrates, causing the switch contacts to open and close repeatedly during the "contact bounce period". This is shown in the diagram below:



Effectively this is like as though you are pressing the switch repeatedly, causing the LED to switch on and off too quickly to be seen, making it appear as though the switch press wasn't registered at all.

To fix this problem, we note from the diagram above that the "keypresses" made by the vibrations are very rapid. If we ignore key presses with very short intervals, we are effectively ignoring any contacts made due to switch bouncing.

This is called "debouncing".

How do we do this on the Arduino? The Arduino library has a millis() function that returns the number of milliseconds since the Arduino was powered up.

We can then use the following algorithm to debounce the switch:

```
currTime = millis();
if(currTime - lastTime > THRESHOLD)
{
   lastTime = currTime;
   /* DO WHATEVER WE NEED TO DO WHEN WE PRESS THE SWITCH */
}
```

We place this algorithm in the ISR that is triggered when the button is pressed. Each time the ISR is triggered, this algorithm checks the current "time" as returned by millis, and subtracts away the last time we recognized a switch press (tracked by "lastTime"). If the difference is more than a certain number of milliseconds given by THRESHOLD, we recognize this as a switch press, and save the time in lastTime.

Open the Arduino sketch called w5s1p0. Start Arduino and open this sketch. Here you will find a solution to the switch bouncing problem using the Arduino wiring language.

We see function called pinISR that we set up as an ISR for Arduino pin 2:

```
static volatile int turn=0;

static unsigned long lastTime=0, currTime;
#define THRESHOLD    10

void pinISR()
{
  currTime=millis();

  if(currTime - lastTime > THRESHOLD)
  {
    // Toggles turn
    turn = 1 - turn;
    lastTime = currTime;
  }
}
void setup() {
  // put your setup code here, to run once:

  pinMode(REDPIN, OUTPUT);
  pinMode(GREENPIN, OUTPUT);
  pinMode(SWITCHPIN, INPUT);
  attachInterrupt(digitalPinToInterrupt(SWITCHPIN), pinISR, RISING);
}
```

You can see here that the function calls “millis” to get the amount of time in millseconds since the program started running on the Arduino and stores it in “currTime”. It compares currTime against lastTime, and if more than THRESHOLD milliseconds have passed, it will recognize the button press and toggle “turn”.

We will now learn how to do this using the timer instead of millis().

Open the w5s1p1 Arduino sketch to complete this section.

Step 1. Setting Up The Timer

We will now write the code for setting up the timer. We want to trigger an interrupt every 100 microseconds (100 us). To help you along we will answer some questions:

Question 1.

Complete the following table (refer to slides 6 to 11 of the lecture if you’re not sure what’s going on here):

T_{cycle} = 100 us (us = microsecond)

Prescalar Value	Resolution (us) (16MHz)	V
1	0.0625	
8	0.5	
64	4	
256	16	
1024	64	

Question 2.

Based on Question 1, what is the best pre-scalar and V value? Does the V value you chose give exactly 100 us? If not, what is the actual interval produced?

Question 3a.

We are going to use Timer 2 in CTC mode. Unlike in the lecture notes, we do **NOT** want to generate any waveforms on any of the AVR's pins. Write down suitable binary values for the TCCR2A register.

Note: The layout for TCCR2A and TCCR2B are identical to TCCR0A and TCCR0B, so you can use the lecture notes to guide you on how to fill in the bit values.

TCCR2A

COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20

Write down the C statement(s) for setting TCCR2A. You may directly assign the bits to TCCR2A without using bitwise operations.

We also need to enable the OCIE2A interrupt. Write down a suitable bit pattern for TIMSK2:

TIMSK2

-	-	-	-	-	OCIE2B	OCIE2A	TOV2

Write down the C statement(s) for setting these values. Again you may assign directly to TIMSK2.

Question 3b.

Write down suitable values for TCNT2 and OCR2A

TCNT2 =

OCR2A =

Based on your answers above, write the code for setupTimer(). Cut and paste the code into your report.

Step 2. Starting the Timer

We will now write the code to start the timer. We start the timer by setting the prescalar in TCCR2B to any value other than 0. (Of course we will set it to the value we found in Question 2).

Question 5.

Based on your answer for Question 2, write down a suitable bit pattern for TCCR2B. Again write X for bits you don't intend to set/clear:

TCCR2B

FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20

Write down the C statement(s) to directly set TCCR2B.

Now use your answers to Question 5 to implement startTimer().

Step 3. Building the Timer ISR

Locate the following segment of code in the w5s1p1 sketch:

```
/*
        ISR(...)
        {
        _timerTicks++;
        }
*/
```

Replace the "…" between the brackets with the proper interrupt vector name (remember that we are running Timer 2 in CTC mode and using OCR2A), then delete the /* and */ to uncomment the ISR.

Cut and paste the ISR to your report.

Step 4.

We are keeping track of the number of 100 us ticks passing by using the _timerTicks variable. Modify the ISR for INT0 to use _timerTicks to implement debouncing.

Cut and paste your new INT0 ISR into your report.

ACTIVITY 2 – USING TIMERS TO TRIGGER EVENTS

In this activity we will look at how to use timers to trigger events. In our case, our event is to switch between flashing the green LED to flashing the red LED, or vice-versa.

Create a new Arduino sketch called w5s1p2.

For this activity we will use:

Timer: Timer 2

Mode: CTC

Compare register: OCR2A

Timer tick interval: 1 second

Question 6.

Complete the following table to choose your prescalar and suitable V value:

T_{cycle} = 1000000 us (us = microsecond)

Prescalar Value	Resolution (us) (16MHz)	V
1	0.0625	
8	0.5	
64	4	
256	16	
1024	64	

Explain based on your answer why Timer 2 cannot be used for this application.

Since we cannot use Timer 2, we will now switch to Timer 1, which is a 16-bit timer and can take counts of up to 65535. Write down suitable V and Prescalar values for Timer 1 (Note: you should always choose the largest possible V value to get the smallest possible prescalar resolution):

V =

Prescalar =

Note our new specification is now:

Timer: Timer 1

Mode: CTC

Compare register: OCR1A

Timer tick interval: 1 second

The timer 1 registers TCNT1 and OCR1A are now 16 bit registers instead of 8 bit registers. However the bad news is that the TCCR1A and TCCR1B registers are laid out somewhat differently from TCCR0A/TCCR2A and TCCR0B/TCCR2B. The specifications for these registers are shown below:

20.14.1. TC1 Control Register A

Name: TCCR1A
Offset: 0x80
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	COM1	COM1	COM1	COM1			WGM11	WGM10
Access	R/W	R/W	R/W	R/W			R/W	R/W
Reset	0	0	0	0			0	0

20.14.2. TC1 Control Register B

Name: TCCR1B
Offset: 0x81
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	ICNC1	ICES1		WGM13	WGM12	CS12	CS11	CS10
Access	R/W	R/W		R/W	R/W	R/W	R/W	R/W
Reset	0	0		0	0	0	0	0

Note that the waveform generation mode (WGM) configuration bits are now 4 bits long and spread across TCCR1A and TCCR1B. For our purposes we will set all COM1 bits in TCCR1A and the ICNC1 and ICES1 bits in TCCR1B to 0. These bits control waveform generation on the external pins, and input noise removal, which are beyond the scope of our course.

The table below shows how to set the WGM10, WGM11, WGM12 and WGM13 bits for various timer modes.

Table 20-6. Waveform Generation Mode Bit Description

Mode	WGM13	WGM12 (CTC1)[1]	WGM11 (PWM11)[1]	WGM10 (PWM10)[1]	Timer/ Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM

Mode	WGM13	WGM12 (CTC1)[1]	WGM11 (PWM11)[1]	WGM10 (PWM10)[1]	Timer/ Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	-	-	-
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

We are interested in mode 0b0100, which is CTC mode.

Question 7.

Write down the bit pattern for TCCR1A. Note we will program the WGM13 and WGM12 bits in TCCR1B later on.

TCCR1A

COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10

Write down the C statements to program TCCR1A. Remember that we want to set all the COM1 (COM1A1, COM1A0 etc.) to 0. You can assign directly to TCCR1A.

Also, write down the statement to assign the V value from Question 6 into the OCR1A register and set TCNT1 to 0.

(Note: The OCR1A register is actually a 16-bit register. However the AVR is an 8-bit MCU. For that reason OCR1A is internally represented by OCR1AH and OCR1AL, representing the high and low bytes of OCR1A respectively. You should actually program the value for OCR1AH first then OCR1AL in two separate assignments, but the C compiler handles this for you. Yay!)

Lastly we need to set the appropriate bit in the TIMSK1 register. Since we are going to use the OCR1A compare register in CTC mode, we will be enabling the OCIEA bit in TIMSK1:

20.14.12. Timer/Counter 1 Interrupt Mask Register

Name: TIMSK1
Offset: 0x6F
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
			ICIE			OCIEB	OCIEA	TOIE
Access			R/W			R/W	R/W	R/W
Reset			0			0	0	0

Write down the C code to do this, using bitwise operations:

Question 8.

Using what you have done in Question 7, fill in the code for setupTimer(). Cut and paste the code into your report.

We will now figure out the prescalar bit values for TCCR1B. Note that the bit values for Timer 1 are different from Timers 0 and 2 for each respective prescalar value.

Bits 0, 1, 2 – CS10, CS11, CS12: Clock Select 1 [n = 0..2]
The three Clock Select bits select the clock source to be used by the Timer/Counter. Refer to Figure 20-10 and Figure 20-11.

Table 20-7. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
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	clkI/O/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)

Based on a 16MHz clock, the prescalars above will have the following resolutions:

CS12	CS11	CS10	Description	Resolution
0	0	0	Timer stopped	-
0	0	1	CLKIO/1	1/16000 = 0.0625us
0	1	0	CLKIO/8	8/16000000 = 0.5us
0	1	1	CLKIO/64	64/16000000=4 us
1	0	0	CLKIO/256	256/16000000 = 16 us
1	0	1	CLKIO/1024	1024/16000000 = 64 us

Question 9.

Based on your selection for WGM13 and WGM12 bits in Question 6 / 7 and the prescalar you have chosen, fill in the bit pattern for TCCR1B.

TCCR1B

ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10

Question 10.

Using your answers from Question 9, Write down the C statement to set TCCR1B, using direct assignment, not bitwise operations.

Write the code for startTimer() and cut and paste the code into your report.

Question 11.

Now modify your w5s1p2 program so that the program flashes the green LED first for 5 seconds, then the red for 5 seconds, then the green for 5 seconds, etc. **NOTE: YOU MUST REMOVE ALL SETUP AND ISR CODE FOR INT0 OR YOUR SOLUTION WOULD BE INCORRECT.**

(Hint: We have included a _timerTick variable for you to record how many timer interrupts have occurred, and you've set Timer 1 to trigger once per second)

Cut, paste and explain your code in your report.