Students: Kevin (Kaiwen) Xue - 20814292, Anni Wang - 20849760

# BME 393L: Lab 6

## 4. In Lab Procedure

### 4.1 Timer block - Stopwatch Timer

```
const int switchPin = 2; // the number of the input pin
long startTime; // store starting time here
long duration; // variable to store how long the timer has been running
float secduration; // variable to store the duration in seconds
int switchPinValue = HIGH;
void setup()
{
  pinMode(switchPin, INPUT_PULLUP);
  Serial.begin(9600);
}

void loop()
{
    switchPinValue = digitalRead(switchPin);
    if (switchPinValue == LOW){
    Serial.println("Button pushed");
    startTime = millis();
    while (1) {
        if (digitalRead(switchPin) == HIGH) {
        break;
        }
    }
    duration = millis() - startTime;
    secduration=(float)duration / (1000) ;
    Serial.print("Button released after "); // print out your results
    Serial.print(secduration);
    Serial.println(" seconds");
    }
}
```

**Figure 1:** Arduino Code for Stopwatch Timer

**Questions**

1. *How accurate is your timer?*
   The timer is accurate to the nearest millisecond.
2. *What is the limiting factor in the effectiveness of this overall design?*
   The execution of the program is stuck on polling the input to see when the button gets released, before allowing the rest of the program to execute. Another limiting factor would be the accuracy of the timer, which is limited by the clock speed of the Arduino. This affects the **millis()** function, which was used to measure the duration/period of the signal.

## 4.2 Frequency counter - Limits of a timer

```
const int switchPin = 2;
const int buttonPin = 8;
long startcount;
long endcount;
long duration;
float period;
float scaled_period;
float frequency;

void setup()
{
  pinMode(switchPin, INPUT);
  pinMode(buttonPin, INPUT_PULLUP);
  Serial.begin(9600); //sets up communication with the serial monitor
//  TCCR0B = ( _BV(CS02) | _BV(CS00));
//  TCCR0B = ( _BV(CS02));

}

void loop()
{
  if (digitalRead(buttonPin) == LOW){
    if (digitalRead(switchPin) == HIGH){
        while (digitalRead(switchPin) == HIGH){}
        startcount = millis();
        while (digitalRead(switchPin) == LOW){}
        while (digitalRead(switchPin) == HIGH){}
        endcount = millis();
        duration = endcount - startcount;
        period = (float)(((float)duration / 1000));
```

```
        scaled_period = period;
        frequency = 1 / scaled_period;
        Serial.print(" Frequency is = ");
        Serial.println(frequency);
    }
  }
}
```

**Figure 2:** Arduino code for Frequency Counter

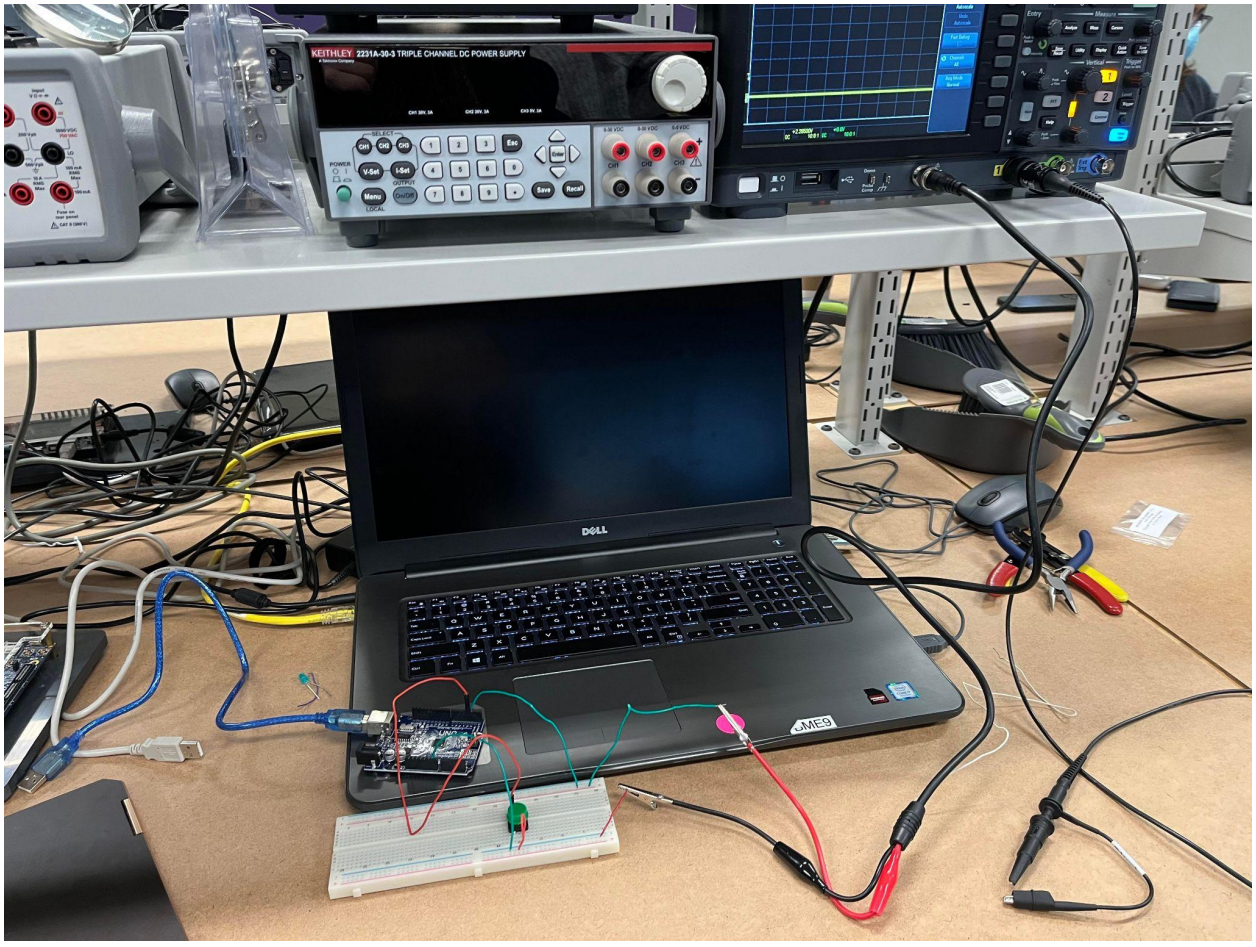


**Figure 3:** Setup for Frequency Counter

A frequency was considered valid if its rounded value from the serial monitor matched the value on the function generator.

To calculate the frequency of the signal, a full period was measured.

**Questions**

1. *What frequency range is your design valid for?*
   The design is valid up until ~50 Hz.

2. *What is the limiting factor in this design?*
   The limiting factor in this design is the Arduino Timer, which was equivalent to the clock speed of 16 MHz prescaled by 64, and the **millis()** function, which was used to measure the full period.
3. *Calculate the theoretical frequency limit of your design and compare it to the implemented design.*
   The theoretical frequency limit of our design would be based on the frequency of the prescaled Timer, and the frequency of our duration measurement function, which in this case, is the **millis()** function. In this case, the limit should be 1 kHz, as the smallest duration that can be measured is 1 ms. However, looking at the implemented design, which had a frequency limit of ~50 Hz, it is as expected. This is because at some point (highest valid frequency), the process of recording the current time, polling for changes, finding the duration, etc., take considerable amounts of time relative to the period of the signal being measured, resulting in deviations of the measured period from the true period, and thus, large deviations of the calculated frequency from the true frequency.

## 4.3 Frequency counter - Improved Version

```
const int switchPin = 2;
const int buttonPin = 8;
long startcount;
long endcount;
long duration;
float period;
float scaled_period;
float frequency;

void setup()
{
  pinMode(switchPin, INPUT);
  pinMode(buttonPin, INPUT_PULLUP);
  Serial.begin(9600); //sets up communication with the serial monitor
  TCCR0B = ( _BV(CS02) | _BV(CS00));

}

void loop()
{
  if (digitalRead(buttonPin) == LOW){
    if (digitalRead(switchPin) == HIGH){
        while (digitalRead(switchPin) == HIGH){}
        startcount = millis();
        while (digitalRead(switchPin) == LOW){}
```

```
        while (digitalRead(switchPin) == HIGH){}
        endcount = millis();
        duration = endcount - startcount;
        period = (float)(((float)duration / 1000));
        scaled_period = 16*period;
        frequency = 1 / scaled_period;
        Serial.print(" Frequency is = ");
        Serial.println(frequency);
    }
  }
}
```

**Figure 4:** Arduino code for Frequency Counter improved, with prescaler value of 1024

The same code from **Section 4.2** was used, except the period scale and changing the TCCR0B register in the setup to the corresponding prescaler value. The method for determining whether a frequency is valid is also the same as the one from **Section 4.2**.

**Table 1:** Tabulating the results from changing prescaler values and measuring the highest valid frequency

| CS02:CS00 | Prescaler value | Highest Valid Frequency (Hz) |
|---|---|---|
| 5 | 1024 | 6 |
| 4 | 256 | 16 |
| 3 | 64 | 50 |
| 2 | 8 | 400 |
| 1 | 1 | 4000 |

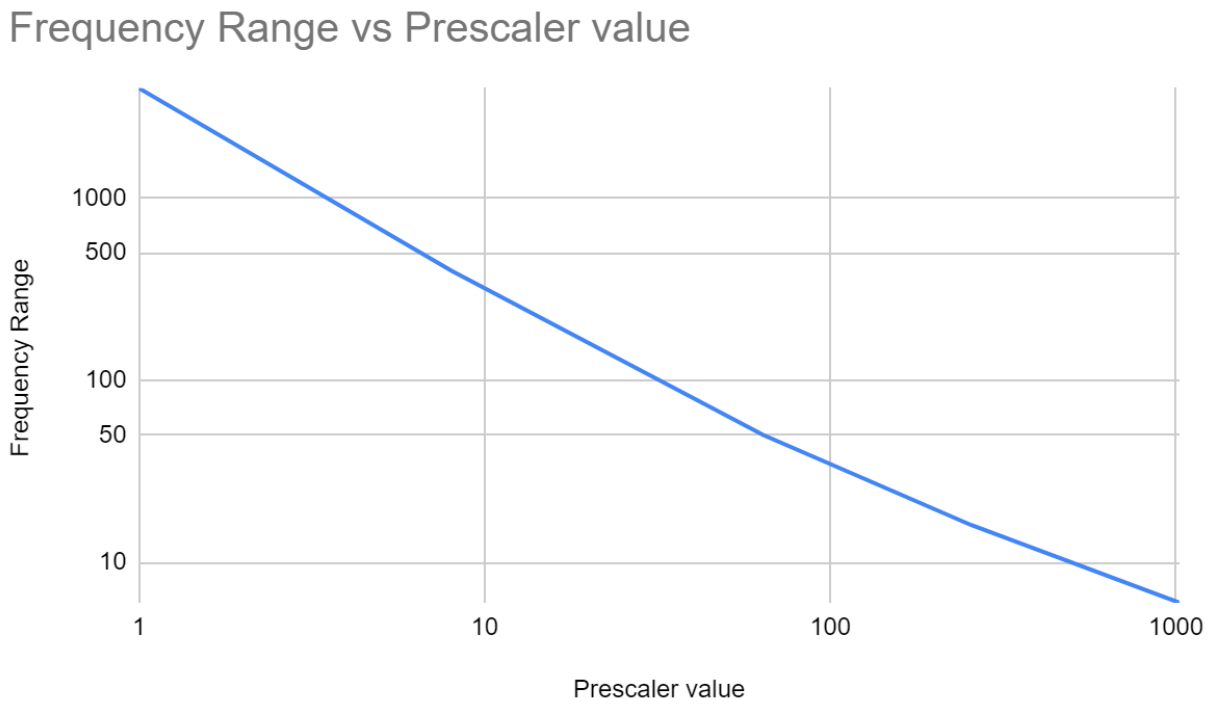


**Figure 5:** Plotting the highest valid frequency (aka Frequency Range) vs the prescaler value. Note that in the plot above, the axes are of a logarithmic scale. This was done to improve presentability.

**Questions**

1. *What frequency range is your design valid for?*
   The design is valid up until ~4 kHz.
2. *What is the limiting factor in this design?*
   The limiting factor in this design is the base clock speed of the Arduino, which is 16 MHz (in reality, the base clock speed of the actual Arduino used is 12 MHz). This affects the **millis()** function, which was used to measure the duration/period of the signal.
3. *Calculate the theoretical frequency limit of your design and compare it to the implemented design.*
   Similar to **Section 4.2**, **Question 3**, the theoretical frequency limit of our design would be based on the frequency of the prescaled Timer, and the frequency of our duration measurement function, which in this case, is the **millis()** function. In this case, the limit should be 64 kHz, as the smallest duration that can be measured is 15.625 µs. However, looking at the implemented design, which had a frequency limit of ~4000 Hz, it is as expected. This is because at some point (highest valid frequency), the process of recording the current time, polling for changes, finding the duration, etc., take considerable amounts of time relative to the period of the signal being measured, resulting in deviations of the measured period from the true period, and thus, large deviations of the calculated frequency from the true frequency.

Looking at **Figure 5**, it is evident that as the prescaler value decreases logarithmically, the highest valid frequency (aka Frequency Range) increases logarithmically, and vice versa. This is expected, as decreasing the prescaler value increases the effective clock speed. This increased effective clock speed results in the process of recording the current time, polling for changes, finding the duration, etc., taking considerably less amounts of time relative to the period of the signal being measured, resulting in smaller deviations of the measured period from the true period at lower true frequencies, and allowing for the measurement of higher frequencies before large deviations are present from the calculated frequency to the true frequency.

## 4.4 Interrupt Block - Background Task

The code provided, *Tone_generator.ino*, was run on the Arduino to confirm that it works, which it did.

Demo:
https://drive.google.com/file/d/1LKUvZn6XLGuPnlXwJSmz5xY2GE1Xx7zG/view?usp=sharing

## 4.5 LED control by push-button

```
const int led_pin = 8; // the input pin
const int interrupt_pin_number = 2;
volatile byte led_state = LOW;


void setup()
{
  pinMode(led_pin, OUTPUT);
  pinMode(interrupt_pin_number, INPUT_PULLUP);
  attachInterrupt (digitalPinToInterrupt(interrupt_pin_number), ISR_blink,
CHANGE);
}

void loop()
{
    digitalWrite(led_pin, led_state);
}

void ISR_blink(){
  led_state = ~led_state;
}
```

**Figure 6:** Arduino code for LED control by push-button

Demo:
https://drive.google.com/file/d/1LbM77FN5I_BdeJ2_U8x76OiA-2L8zyZn/view?usp=sharing

## 4.6 Effect of ISR complexity on the main task

```
#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
```

```
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
```

```
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978

int melody[] = {
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4
};

int noteDurations[] = {
  4, 8, 8, 4, 4, 4, 4, 4
};

const int interrupt_pin_number = 2;

void setup() {
  pinMode(interrupt_pin_number, INPUT_PULLUP);
  attachInterrupt (digitalPinToInterrupt(interrupt_pin_number), ISR_delay,
FALLING);
}

void loop() {
  for (int thisNote = 0; thisNote < 8; thisNote++) {
    int noteDuration = 1000 / noteDurations[thisNote];
    tone(8, melody[thisNote], noteDuration);

    int pauseBetweenNotes = noteDuration * 1.30;
    delay(pauseBetweenNotes);
    noTone(8);
  }
}

void ISR_delay(){
  delay(5000);
}
```

**Figure 7:** Arduino code for “Effect of ISR complexity on the main task”

The amount of delay in the ISR, *ISR_delay*, was varied to change the amount of audible distortion. The minimum amount of delay that caused an audible amount of distortion was 1000 milliseconds, or 1 second.

# 5. Post Lab

**Question 1** – We didn't use the 16 bit counter for section 4.2, but review this timers capabilities (see pages 136 to 137 of the ATMega328 datasheet TCCR1B register) and determine if (and why/why not) using this register would be beneficial for a frequency counter. In other words, will it allow you to measure higher frequencies?

Using the 16 bit counter would allow for the measurement of higher frequencies. This is because the ability to measure higher frequencies depends on the speed of the clock, which determines how fast the CPU can fetch, decode, and execute instructions to ensure that everything occurs in time to measure from the first rising edge to the next.

**Question 2** – Suppose the following interrupt service routine (ISR) is written to respond to an interrupt service request for a specific problem. The code is running on an Arduino Uno board with 16 MHz clock-pulse. If the maximum time allowed for the execution of the ISR is limited to 10 µSec, how many lines of code from the set $\{in, out, add, sub, and, or, cp, cpi, tst\}$ can be embedded in the ISR (as `some_code` shown below) including PUSH/POP/ret instructions?

It can be found from the ATMega328 datasheet that each instruction in this set, {in, out, add, sub, and, or, cp, cpi, tst}, takes 1 clock cycle to execute. It can also be found from the ATMega328 datasheet that each instruction in this set, {push, pop}, takes 2 clock cycles to execute.

$$(10 \times 2\,cycles \times \frac{1}{16\,M}\,s/cycle) + (4\,cycles \times \frac{1}{16\,M}\,s/cycle) = 1.5\,\mu s$$

$$10\,\mu s - 1.5\,\mu s = 8.5\,\mu s$$

$$\frac{8.5\,\mu s}{(1/16\,M)\,s/cycle \times 1\,cycle} = 136$$

Therefore, **136 lines of code fit**

What if the allowed set would be $\{sbi, cbi, ld, st\}$?

$$(10 \times 2\,cycles \times \frac{1}{16\,M}\,s/cycle) + (4\,cycles \times \frac{1}{16\,M}\,s/cycle) = 1.5\,\mu s$$

$$10\,\mu s - 1.5\,\mu s = 8.5\,\mu s$$

$$\frac{8.5\,\mu s}{(1/16\,M)\,s/cycle \times 2\,cycle} = 68$$

Therefore, **68 lines of code fit**

**Question 3** – Suppose a fictitious microcontroller board with single core CPU 16 MHz is used for a real-time application that is running a multi-tasking operating system. A timer block inside the microcontroller is generating a 1000 Hz pulse that is used as tick interrupt for the operating system. It means, the CPU core is getting an interrupt every millisecond in order to switch from one task to another. Saving variables for the current task (before loading variables for the new task) into memory is done during the interrupt service routine (ISR). Maximum time allocated for each ISR is only 5% of the time allocated per task. Suppose an instruction that needs access to memory (like LOAD and STORE) takes 8 clock-cycles to complete, and all other instructions take 2 clock-cycle for execution (per instruction).
What would be the maximum number of lines of code (include all push/pop commands) for an ISR? Assume the CPU has 32 registers, and that all of them in general are needed for each task execution. Show your calculations.

The maximum time for each ISR would be 5% of the time allocated per task, which can be found by taking the period of the 1000 Hz pulse. This would be 5% of 1 ms, which is 50 μs. The duration of 1 clock cycle for a CPU with a clock speed of 16 MHz would be 1/(16 MHz), which is 62.5 ns. This means that the maximum number of clock cycles for the ISR would be 50 μs / 62.5 ns, which is 800 clock cycles. As 32 registers are needed for each task execution, LOAD and STORE would each be called 32 times within the ISR to store all the variables of the current task from the registers into memory, and load all the variables of the next task from memory into the registers. In total, loading and storing would take (32 * 8 * 2) clock cycles, which is 512 clock cycles. In total loading and storing would take 32 * 2 lines of code, which is 64 lines of code. The remaining clock cycles not taken up by loading and storing can be used for other instructions. The number of remaining clock cycles is 800 - 512, which is 288. As the other instructions take 2 clock cycles to execute per instruction, the number of lines of code for the other instructions can be found by 288 / 2, which is 144 lines of code. In total, the maximum number of lines of code for the ISR is 64 + 144, which is **208 lines of code**.