EEET2490 – Embedded Systems: Operating Systems and Interfacing

LABORATORY 1

**ATmega32 Microcontroller Essentials & Functions**

**Lecturer:** Dr Andrew Smith

**Student Name:** Tran Thi Hong Phuong

**Student ID:** s3623386

**Date:** 10th August, 2018

**Table of Contents**

[1. INTRODUCTION 3](#_Toc521248934)

[2. TASK 1 – Toggling LEDs 3](#_Toc521248935)

[2.1. Sourcing 3](#_Toc521248936)

[2.2. Sinking 4](#_Toc521248937)

[3. TASK 2 – Controlling the Brightness of LEDs 5](#_Toc521248938)

[3.1. Toggle LED at the rate of 100Hz 5](#_Toc521248939)

[3.2. Dim the LED using different duty cycles 6](#_Toc521248940)

[4. TASK 3 – Controlling LEDs with a Push Button 10](#_Toc521248941)

[4.1. LED on when button pushed, off when button released 11](#_Toc521248942)

[4.2. LED toggle on button pushed 12](#_Toc521248943)

[5. TASK 4 – Controlling the Brightness of an LED with a Timer 13](#_Toc521248944)

[6. TASK 5 – Toggling an LED Using a Timer 15](#_Toc521248945)

[7. TASK 6 – Toggling the Flashing Rate of an LED Using Interrupts 16](#_Toc521248946)

[7.1. Toggle LED on and off on push 17](#_Toc521248947)

[7.2. Toggle LED flashing rate on push 18](#_Toc521248948)

[8. CONCLUSION 20](#_Toc521248949)

[9. REFERENCES 21](#_Toc521248950)

[10. APPENDIX 22](#_Toc521248952)

# INTRODUCTION

This lab activity was carried out under the purpose of helping students to learn how to build basic electronic circuits which interface with ATmega32 microcontroller inside the Open-USB-IO (OUSB) board. The tasks are about writing embedded programs to perform simple functioning to control an LED and receive digital input from a push button. Other than that, several basic peripheral features of the microcontroller are also explored to resolve the same problems. Upon completing these laboratory exercises, students are expected to be familiar with Atmel Studio development environment and C programming language to perform basic microcontroller functions.

# TASK 1 – Toggling LEDs

## Sourcing

In this task, the OUSB is used to toggle an LED via a General Purpose Input/Ouput (GPIO) pin. For each GPIO pin of the ATmega32 microprocessor, the output DC voltage and current is 5V and 40.0mA respectively [1]. However, common LEDs operates ideally at forward voltage varying between 1.7 to 3.5V and forward current at around 20mA [2]. The LED used in this task is a red one, thus the typical operating voltage is 2.2V [2]. Therefore, some resistance should be added to the circuit in serial with the LED, so as to limit the current and ensure the life of the LED. The resistance value is calculated as followed:

Since a 140 was not available, it was replaced with a 150 resistor instead. Below is the schematic of the circuit, where the microcontroller is sourcing the LED.

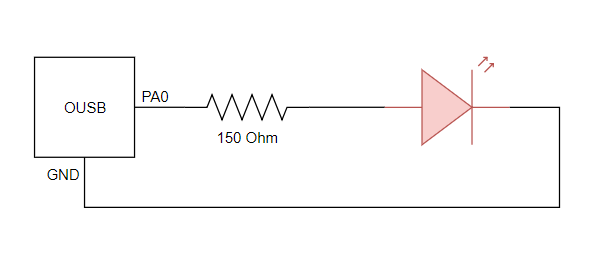


Figure 1. OUSB sourcing LED

In the first scenarior, the program is designed to toggle the LED every second, which means that the Pin 0 of Port A (PA0) is set to output HIGH signal, followed by a delay of 1000 millisecond, and then set to output LOW signal with another delay of the same time. The same method is applied to the second scenarior where the LED is required to toggle five times every second, or five times every 1000 millisecond, so the delay between each command is reduced to 200 millisecond.

The waveform result of this program, which is measured using NI VirtualBench, is presented below. D0 signal line is connected to PA0. The time delay between each toggle, the interval between the two red cursor lines, is displayed at the bottom right corner of the figure, and it is 201.5ms, approximately equal to the specified delay in the code.

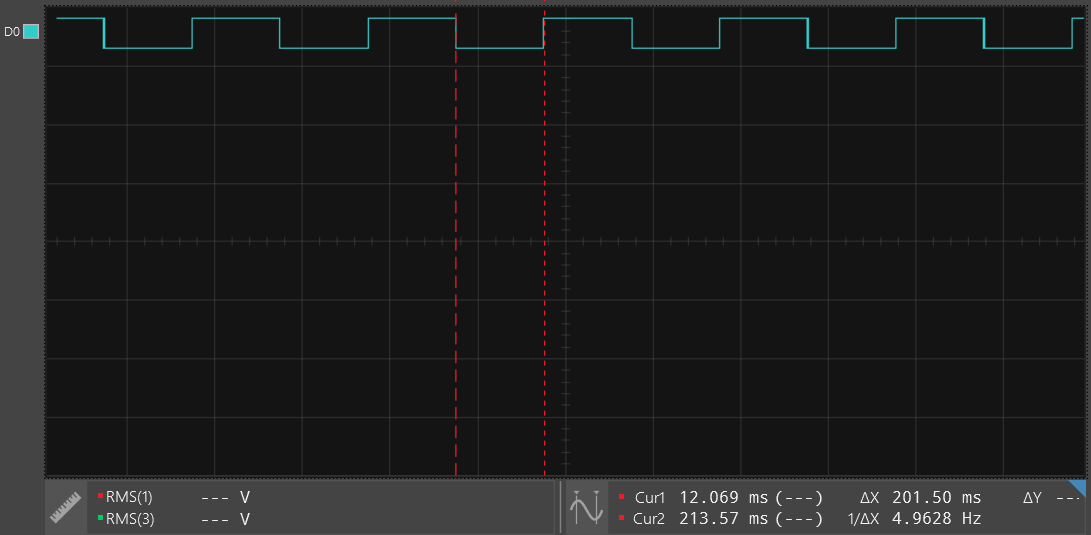


Figure 2. Task 1.1 waveform

## Sinking

*Refer to Figure 6 in Appendix 10.1 for the code.*

Next, a green LED is used to create a connection where the OUSB board is sinking it, while the red LED circuit is kept as above. The HIGH voltage source is now set to VCC instead of a GPIO pin, and GND is substituted with GPIO Pin 1 of Port A (PA1). The schematic is described as followed:

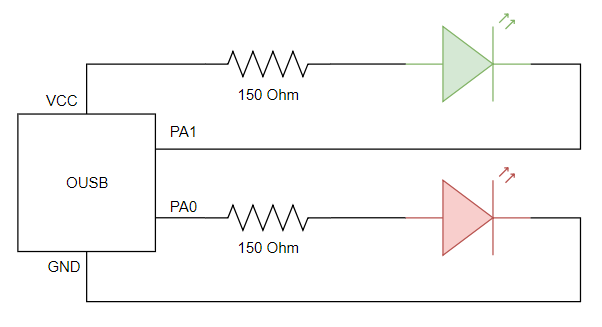


Figure 3. OUSB sourcing red LED and sinking green LED

In sourcing connection, the LED takes HIGH voltage source from the GPIO pin, for this case is PA0. When PA0 is at logic 1, it outputs 5V signal thus creating current flow through the LED, turning it on. When PA0 is at logic 0, there is no current flow, and LED is off. On the other hand in sinking connection, when PA1 is at logic 1, there is no voltage drop thus no current flow through the LED, hence it is off. When PA1 is at logic 0, there is a voltage difference between two ends of the circuit creating current flow through the LED, thus it is on. In another word, the behaviour of a sunken LED contrasts with a sourced one.

In the code, within the infinite main loop, PA0 and PA1 are set to logic 1 for 200ms at the same time, then to logic 0 for another 200ms. When the code is executed, the two LEDs behave differently, they blink in sequence. When red LED is on, green LED is off, and vice versa. The waveform result of this program is presented below.

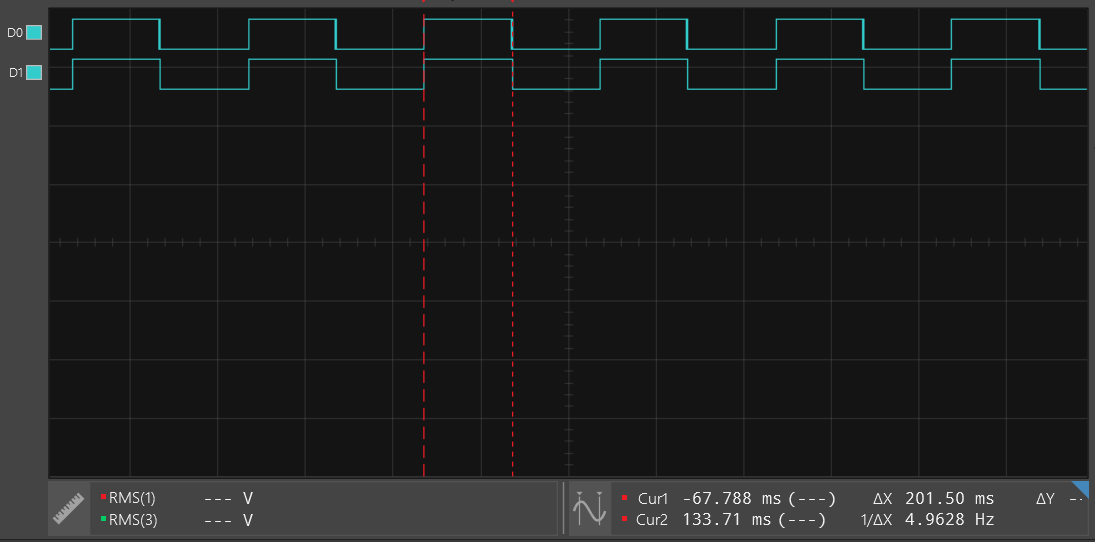


Figure 4. Task 1.2 waveform

# TASK 2 – Controlling the Brightness of LEDs

The activity of this task is learning how to control the brightness of an LED using fundamental principles, without the help of special functions provided by the microcontroller. The electronic circuit for this task is the same as that described in *Figure 1* in *Task 1* section. Adjusting brightness of an LED can be implemented using Pulse Width Modulation, or PWM. PWM is a technique to retrieve analog results from digital on-off signals. This on-off pattern can simulate voltage level varying between full on (for the OUSB board, its output voltage is 5V) and off (0V) by changing the duty cycle, which is the amount of time the signal is at HIGH state over the total time of one cycle [3].

## Toggle LED at the rate of 100Hz

*Refer to Figure 7 in Appendix 10.2 for the code.*

The LED is required to toggle at the rate of 100Hz, so the time period of one on/off cycle is:

So the LED is toggled every .

The waveform result of this program is presented below. The program successfully generates a 100Hz signal.

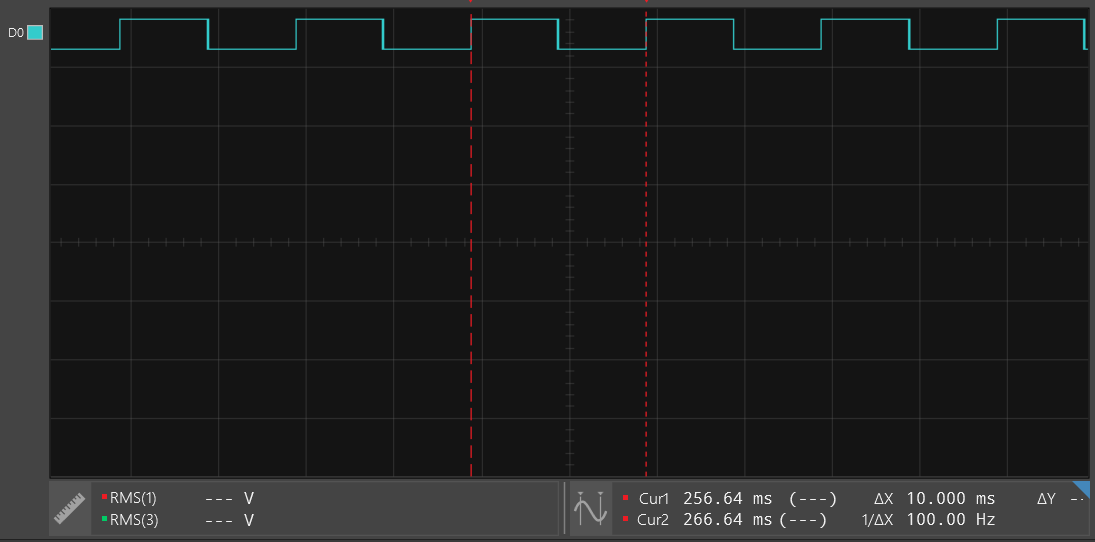


Figure 5. Task 2.1 waveform

## Dim the LED using different duty cycles

*Refer to Figure 8 in Appendix 10.2 for the code.*

The second activity of this task is modifying the above to adjust the brightness of the LED. In order to create a visual illusion that the LED is steadily dimmer than its actual brightness, the flashing, toggling on and off of the LED should not be visible to human eye. This refers to a concept called flicker fusion threshold, a frequency high enough to create persistence of vision, which is when a light stimulus appears steady to the average human observer [5]. The flicker artifacts are reported to be relatively invisible to most viewers at the rate above 30Hz [6]. Therefore, 100Hz toggle rate setting in *Section 3.1* above is more than enough to achieve an apparently stable light output from the LED.

The next step is to apply PWM technique to adjust the brightness of the LED. The brightness of an LED is proportional to the amount of time it stays at HIGH state within each cycle. Without PWM configuration, for one cycle, the amount of time the LED stays at HIGH state is equal to that at LOW state, which is 50% duty cycle. This duty cycle can be modulated by changing the delay time between each LED toggle. If this duty cycle is increased, which means the toggle on delay is longer than the toggle off delay, the LED will appears brighter, and vice versa. To justify this statement, the code provided in *Appendix 10.2* is designed to drive the LED at different duty cycle values (10%, 20%, 30%, 40%, 50%, 75% and 100%) using the function *set\_duty\_cycle()* and observe the brightness difference in each case. The delay time for each case is calculated as:

In the above equations, is the time period of one cycle, which is calculated as 10ms in the previous section. In the code, this value is assigned to a constant named *PERIOD* for later adjustment of frequency, if desired. Therefore in theory, the corresponding delay time for each case listed above is as followed:

|  |  |  |
| --- | --- | --- |
| **Duty cycle** | **Delay HIGH (ms)** | **Delay LOW (ms)** |
| 10% | 1 | 9 |
| 20% | 2 | 8 |
| 30% | 3 | 7 |
| 40% | 4 | 6 |
| 50% | 5 | 5 |
| 75% | 7 | 3 |
| 100% | 10 | 0 |

Table 1. Duty cycles and delay times

In the case of 75% duty cycle, the theoretically calculated is 7.5ms. However in the C program, data type *int* (integer) is used, so all decimal point values are omitted, thus the actual delay is 7ms. The waveform results of this program is presented below.

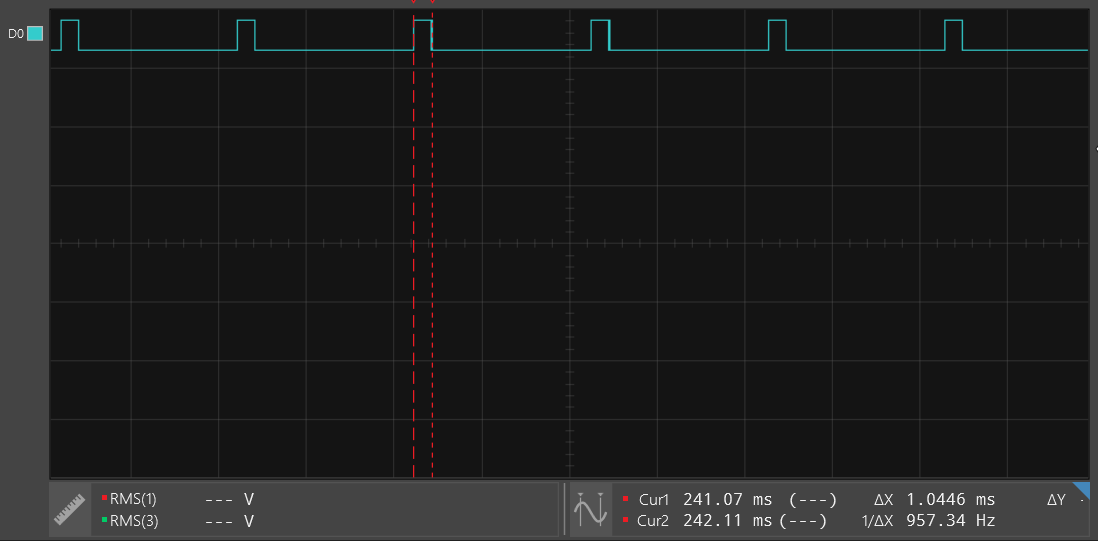


Figure 6. Task 2.2 waveform 10% duty cycle

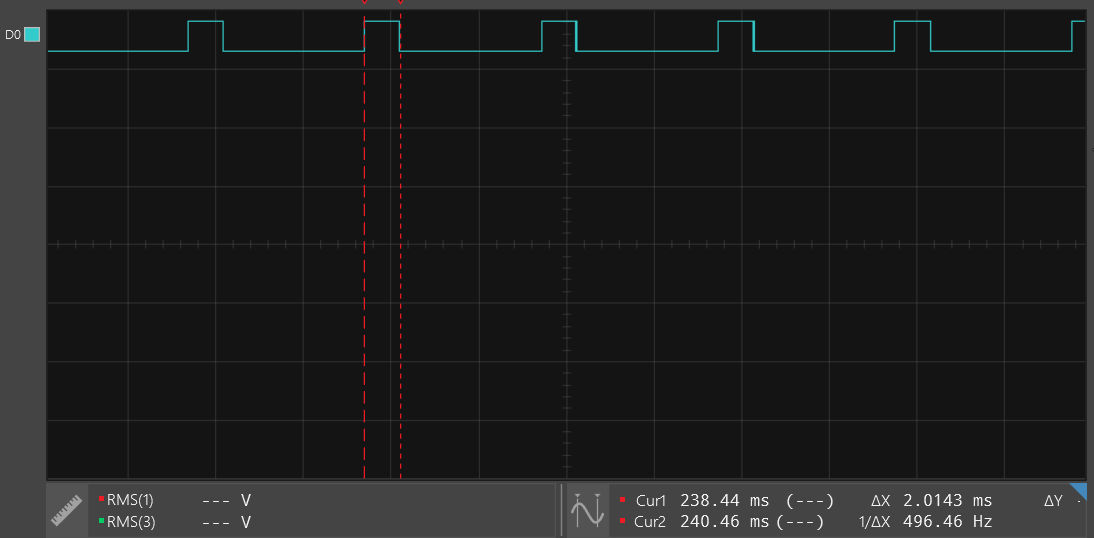


Figure 7. Task 2.2 waveform 20% duty cycle



Figure 8. Task 2.2 waveform 30% duty cycle



Figure 9. Task 2.2 waveform 40% duty cycle

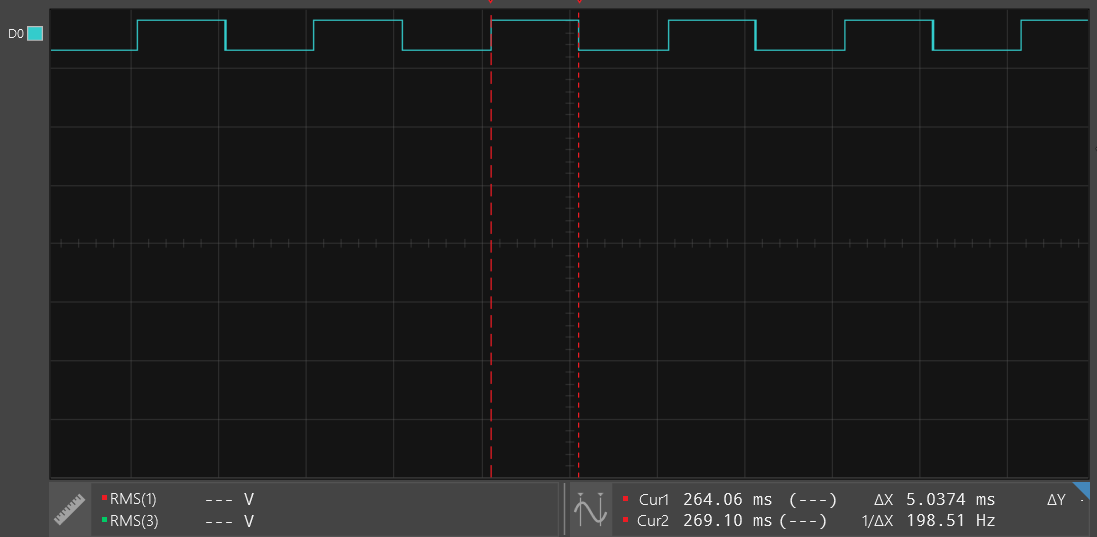


Figure 10. Task 2.2 waveform 50% duty cycle

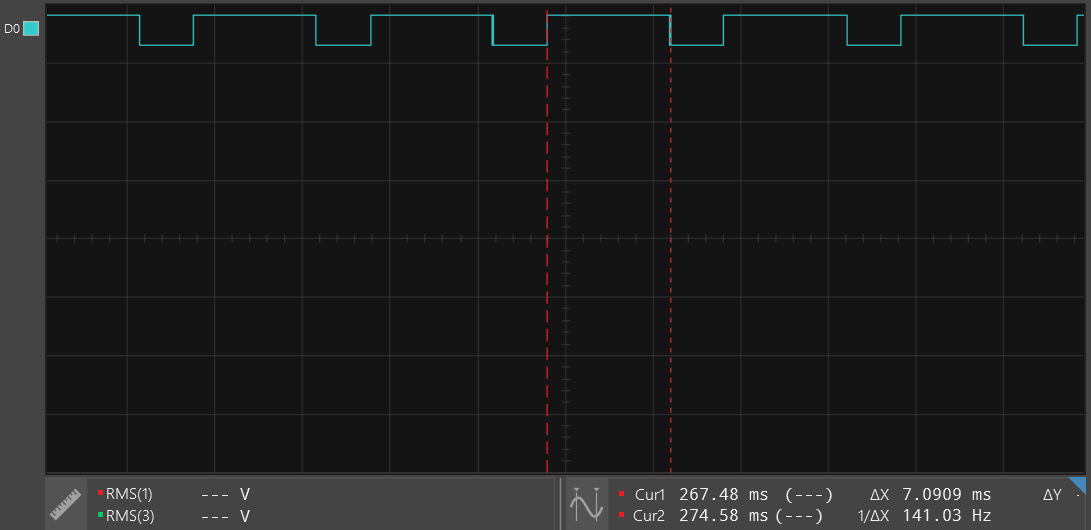


Figure 11. Task 2.2 waveform 75% duty cycle

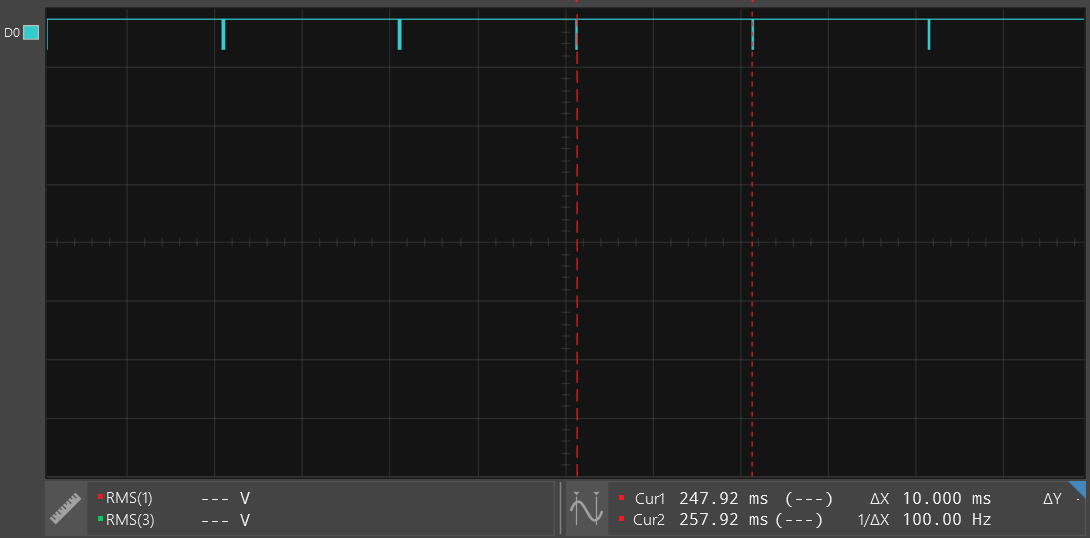


Figure 12. Task 2.2 waveform 100% duty cycle

# TASK 3 – Controlling LEDs with a Push Button

The first two tasks focus on using basic output signals from the OUSB board to turn an LED on and off sequentially. The next activity is to use an external input signal from a push button to control the behaviour of the LED. It is required that there is no more than 5mA of current flowing through the push button circuit, therefore the value of resistance is calculated as:

Since a single 1000-valued resistor is not available, it is replaced by a combination of the following smaller value resistors to achieve the almost equivalent resistance:

The push button is added, in high-acting mode, to the LED circuit in *Figure 1* as presented below.

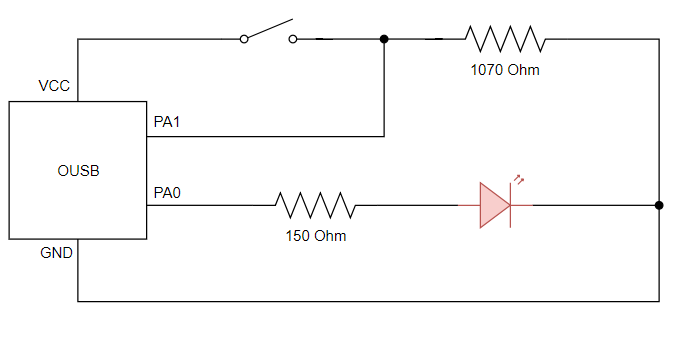


Figure 13. High-acting push button to control LED

When the connection was implemented, floating problem on the push button occurred many times, which made the LED blink in an unreasonable pattern. There were several possible causes that were observed during the process:

* The push button was not fully plugged into the breadboard
* The input signal wire to PA1 was connected to the wrong end of the push button

In the end, these issues were resolved successfully and the program worked as intended.

## LED on when button pushed, off when button released

*Refer to Figure 9 Appendix 10.3 for the code.*

PA0 is configured to be the output signal that drives the LED while PA1 receives input signal from the push button. In the main loop, the program continually checks the input value in PA1. When the button is pushed, since it is high-acting, PA1 is at logic 1, then PA0 is set to output HIGH signal to the LED. On the other hand, when the button is released, PA1 is at logic 0, PA0 is set to output LOW signal to the LED.

The waveform result of this program is presented below. D0 is connected to PA0 (LED), D1 is connected to PA1 (push button). Since debouncing is not applied in this exercise, it can be seen from the waveform that there is some bouncing here and there recorded by D1 signal line. When the push button is released completely at the end, the LED is not off immediately but after a small delay.

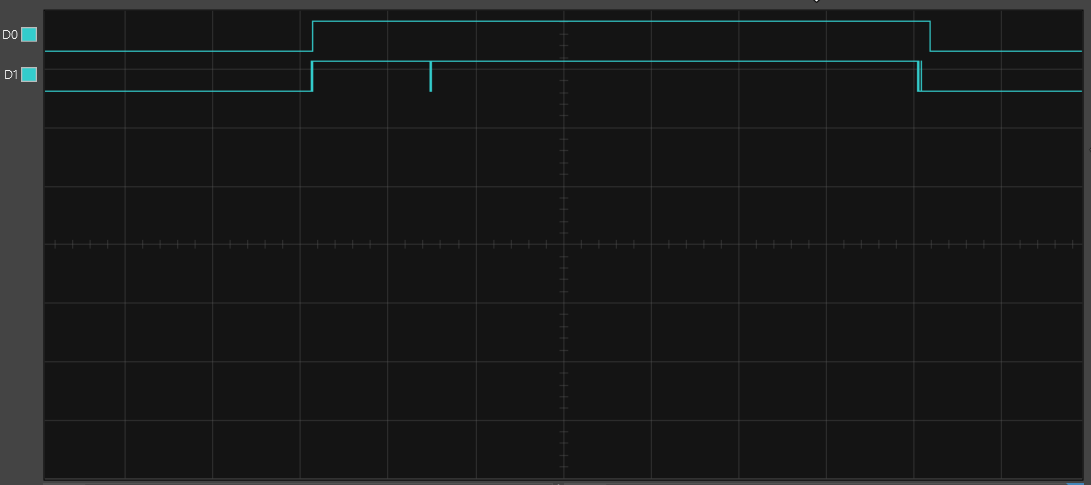


Figure 14. Task 3.1 waveform

## LED toggle on button pushed

*Refer to Figure 10 Appendix 10.3 for the code.*

Pin configuration is kept similar to that in the previous section. Everytime PA1 receives HIGH input signal, the state of toggled using XOR ^ operator with logic 1.

|  |  |  |
| --- | --- | --- |
| **A (Current State)** | **B (Logic 1)** | **A XOR B (Output)** |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Table 2. Truth table [8]

Without further work, the program will face bouncing issue. Bouncing issue usually occurs in push button, within the time the button is pressed, the signal does not stay constant but fluctuates rapidly from 0 to 5V. To solve this problem, a simple programmatical method is applied: a small delay time is generated after the very first high signal received from the button and lasts until after the pressing event is expected to finish, in order to ignore all the signal fluctuation happening inbetween.

The waveform result of this program is presented below. D0 is connected to PA0 (LED), D1 is connected to PA1 (push button). The LED is not toggled immediately after every rising edge of the input signal from the push button, but is toggled only after approximately 300ms after the first input rising edge. This time delay serves as a debouncing method, it makes the system wait until the push event is completely finished before proceeding with the LED toggle.

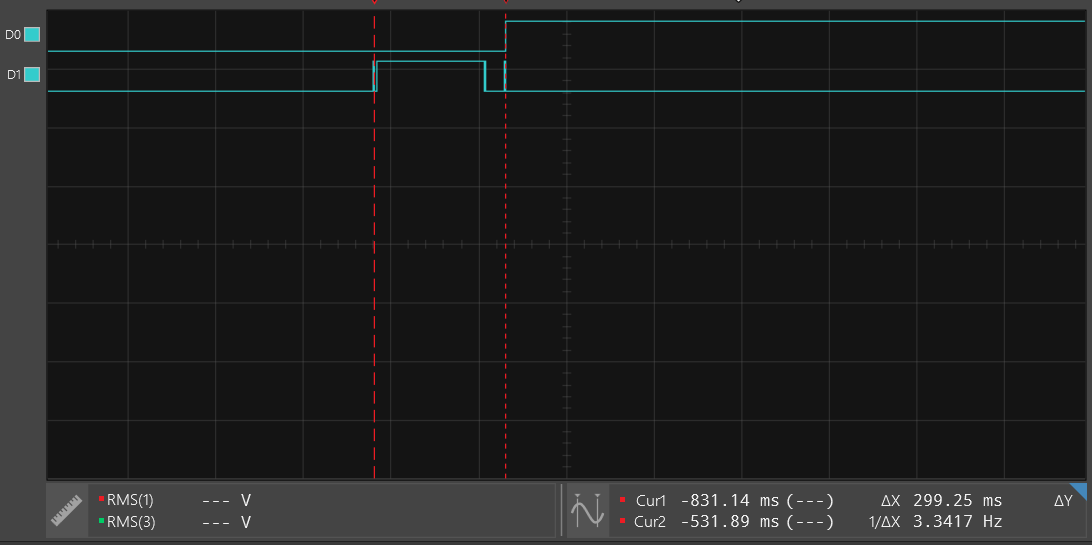


Figure 15. Task 3.2 waveform toggle on

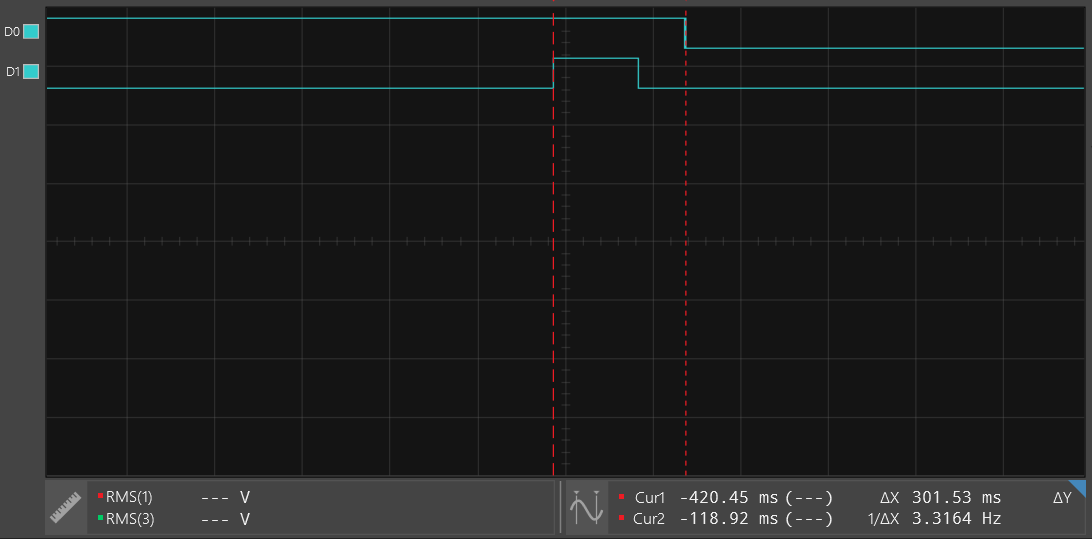


Figure 16. Task 3.2 waveform toggle off

# TASK 4 – Controlling the Brightness of an LED with a Timer

This task is also about applying PWM technique to adjust the brightness of an LED. Instead of the existing built-in function *\_delay\_ms\_()*, Timer/Counter module is used to control the delay time. On ATmega32 microcontroller, there are several Timer/Counter modules with different settings and capabilities. For this task, Timer/Counter0 (TCNT0) is chosen to operate in Fast PWM Mode. The counter continually counts from BOTTOM value (0) up to MAX value (255, because TCNT0 is an 8-bit counter) then restarts at BOTTOM [1]. There are two ways the output of TCNT0 is set, given a value within the range above to compare with its counting value. In non-inverting mode, output is cleared on compare match and set at BOTTOM. In inverting mode, output is set on compare match and cleared at BOTTOM [1]. With that intuition, the program is designed so that each full counting range is one cycle, from BOTTOM to the compare value is the time that the LED is on, and from the compare value to MAX is the time that the LED is off. The schematic is similar to that of *Task 1*, except the output pin from OUSB board is PB3 instead of PA0.

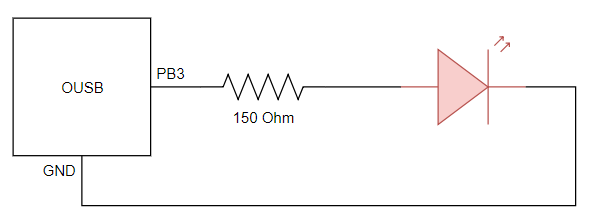


Figure 17. Control LED brightness using TCNT0

Refer to *Figure 11* in *Appendix 10.4* for the code of the program. Pin 3 of Port B (PB3) is configured as output signal for the LED. In addition to being a general digital I/O, PB3 also has an alternate function which is *Timer/Counter0 Output Compare Match Output* [1]. In order for TCNT0 to function as intended, several pin configuration is committed in *Timer/Counter Control Register* (TCCR0).

|  |  |  |
| --- | --- | --- |
| **Pin** | **Value** | **Result** |
| WGM01 | 1 | **Waveform Generation Mode**  Fast PWM |
| WGM00 | 1 |
| COM01 | 1 | **Compare Output Mode, Fast PWM Mode**  Clear output on Compare Match, set output at BOTTOM (non-inverting mode) |
| COM00 | 0 |
| CS02 | 1 | **Clock Select**  clkI/O / 1024 (from prescaler) |
| CS01 | 0 |
| CS00 | 1 |

Table 3. TCCR0 Pin configuration [1]

With the prescaler of 1024, the clock frequency used in TCNT0 is now:

Thus the time it takes for the TCNT0 to count from 0 to 255 is:

So the resulting toggle frequency of the LED is:

In *Section 3.2*, it is mentioned that the flicker artifacts are reported to be relatively invisible to most viewers at the rate above 30Hz, so the above frequency is acceptable for the LED to produce a stable light.

The output compare register (OCR0), which generates output signal in PB3, is required to produce a range of duty cycles similar to those in *Task 3*. The value assigned to this register is the number to be compared with the counting value in TCNT0. Given a specific duty cycle, this value is calculated by the function *calculate\_ocr()* using the formula (take example of 10% duty cycle):

The waveform result of this program is presented below. D0 is connected to PB3. The program successfully generates an approximately 46Hz signal. The figure represents the 10% duty cycle.

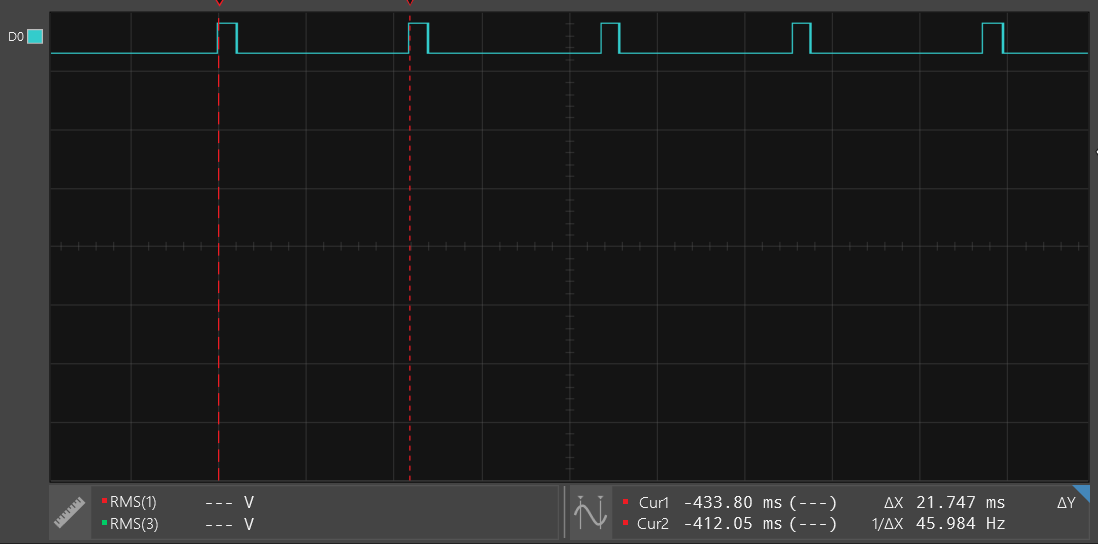


Figure 18. Task 4 waveform

# TASK 5 – Toggling an LED Using a Timer

In this task, Timer/Counter module is required to toggle an LED after an interval of five seconds. TCNT0 continues to be used and is set to operate in normal mode. In this mode, the counting direction is always up (incrementing) and there will be no counter clear [1]. The counter overflows at its MAX value, which is 255 because TCNT0 is an 8-bit counter [1], and restarts from its BOTTOM value, which is 0. Everytime TNCT0 restarts, the corresponding *Timer/Counter Overflow Flag* (TOV0) inside *Timer/Counter Interrupt Flag Register* (TIFR) is set to logic 1 [1].

The circuit design for this task is the same as that in *Task 1*. Refer to *Figure 12* in *Appendix 10.5* for the code of this program. In order for TCNT0 to function as intended, several pin configuration is committed in *Timer/Counter0 Control Register* (TCCR0).

|  |  |  |
| --- | --- | --- |
| **Pin** | **Value** | **Result** |
| WGM01 | 0 | **Waveform Generation Mode**  Normal mode (default) |
| WGM00 | 0 |
| COM01 | 0 | **Compare Output Mode, Non-PWM Mode**  Normal port operation, OC0 disconnected (default) |
| COM00 | 0 |
| CS02 | 1 | **Clock Select**  clkI/O / 1024 (from prescaler) |
| CS01 | 0 |
| CS00 | 1 |

Table 4. TCCR0 Pin configuration [1]

With the calculation in *Task 4*, it takes 21.675ms for the TCNT0 to count from 0 to 255, so to achieve 1 second, or 1000ms, TCNT0 needs to finish the following number of counting cycles:

A software variable called *counter* is declared to keep track of the number of cycles that TCNT0 has accomplished, and another variable called *second* to keep track of the elapsed seconds. Whenever TOV0 is at logic 1, meaning TCNT0 has restarted, the software writes 1 to TOV0 to clear it [1] and *counter* is incremented. When *counter* reaches 46, *second* is incremented and *counter* is reset. When *second* reaches 5, indicating 5 seconds has passed, the LED connected to PA0 is toggled using XOR operator, then *second* is reset.

The waveform result for this task could not be retrieved from NI VirtualBench since it can only display signal up to 1 second per frame, thus 5 seconds amount of time is too long to fit inside one frame. However in reality, the program successfully toggle the LED after approximately 5 seconds.

# TASK 6 – Toggling the Flashing Rate of an LED Using Interrupts

In *Task 3*, toggling LED using input signal from a push button is achieved with polling method. Polling is when the microcontroller constantly monitors the status of an event source and wait until a condition is met to proceed [9]. There is another method to achieve the same result and is more efficient in terms of resource usage called interrupt. Interrupt is a signal sent to the microcontroller to request the processor to stop the current program and execute another special piece of code called *Interrupt Service Routine* (ISR) [9]. The interrrupt signal may come from seval sources: external hardware device or internal interrupt sent by the microcontroller itself as a result of executing the program [9]. For this task, external interrupt signal from a push button is being used.

In ATmega32, there are three external *External Interrupt Input* pins [1]. *External Interrupt 1 Input* (INT1), which is the alternate function at Pin 3 of Port D (PD3), is chosen to be executed. The circuit design is presented as below.

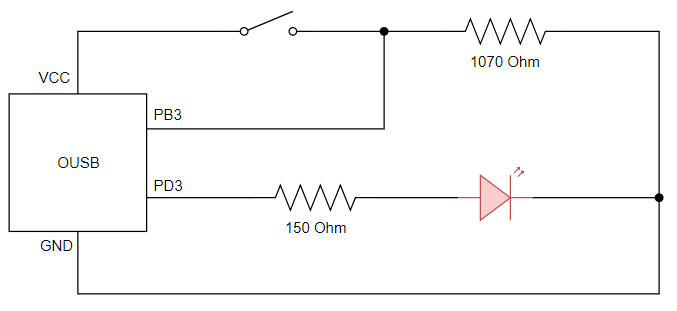


Figure 19. Toggle LED using interrupts

## Toggle LED on and off on push

*Refer to Figure 13 in* *Appendix 10.6 for the code.*

There are two registers to be configured to implement interrupt handler, *General Interrupt Control Register* (GICR) and *MCU Control Register* (MCUCR). GICR is used to specify which external interrupt request is being used in the program [1]. MCUCR determines that interrupt will be triggered by a falling or rising edge or a low level [1]. The below table shows the specific values assigned to each register.

|  |  |  |
| --- | --- | --- |
| **Pin** | **Value** | **Result** |
| GICR -> INT1 | 1 | **External Interrupt Request 1 Enable**  External pin interrupt 1 is enabled |
| MCUCR -> ISC11 | 1 | **Interrupt 1 Sense Control**  The rising edge of INT1 generates an interrupt request |
| MCUCR -> ISC10 | 1 |

Table 5. GICR and MCUCR Pin configuration [1]

After the pin configuration, *sei()* function is called to enable global interrupt flag in *The AVR Status Register* (SREG) [1]. A software variable *ledMode* is declared to store the operating mode of the LED. If ledMode is 1, the LED is on and if *ledMode* is 0, the LED is off. The code to be executed when interrupt is triggered is defined in the *ISR(INT1\_vect)* function. When an interrupt is sent, a small delay of 300ms is called to resolve the bouncing issue from the push button, then the value of *ledMode* is toggled using XOR operator. Inside the main loop, the state of the LED is set according to the value of *ledMode*. The waveform result of this program is presented below.

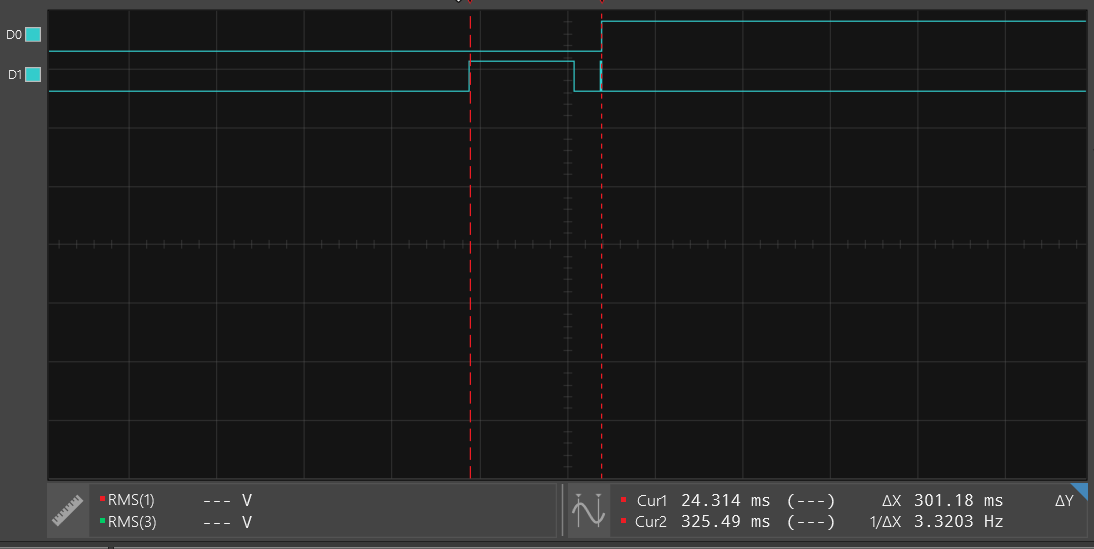


Figure 20. Task 6.1 waveform toggle on

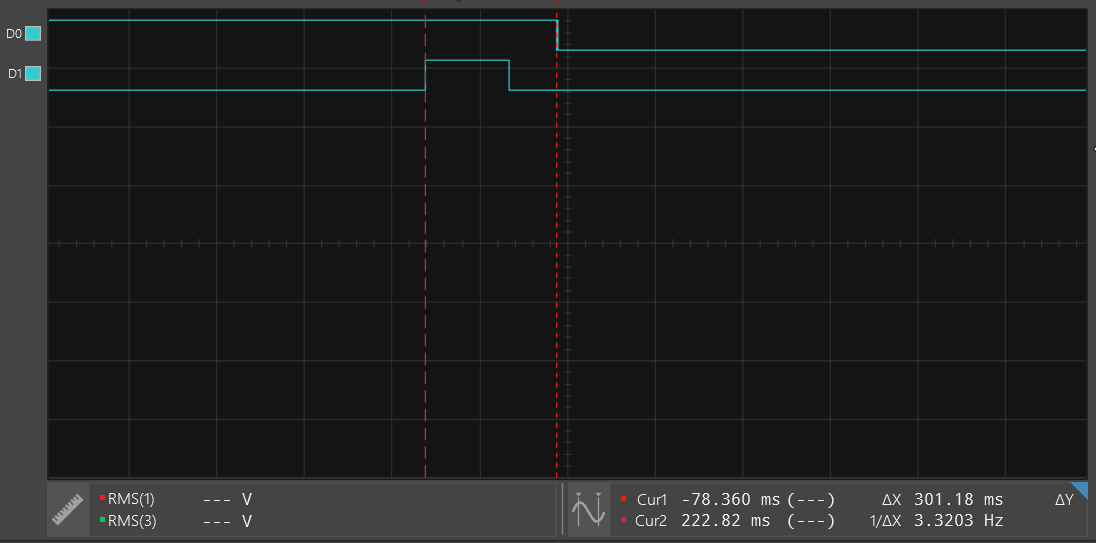


Figure 21. Task 6.1 waveform toggle off

## Toggle LED flashing rate on push

*Refer to Figure 14 in* *Appendix 10.6 for the code.*

This exercise is about toggling the duty cycle of an LED between 20% and 50% using external interrupt signal from a push button, a combination of *Task 4* and the exercise in *Section 7.1* above. TCNT0 and INT1 is used and configured with the values in the following table.

|  |  |  |
| --- | --- | --- |
| **Pin** | **Value** | **Result** |
| **TCNT0** | | |
| TCCR0 -> WGM01 | 1 | **Waveform Generation Mode**  Fast PWM |
| TCCR0 -> WGM00 | 1 |
| TCCR0 -> COM01 | 1 | **Compare Output Mode, Fast PWM Mode**  Clear output on Compare Match, set output at BOTTOM (non-inverting mode) |
| TCCR0 -> COM00 | 0 |
| TCCR0 -> CS02 | 1 | **Clock Select**  clkI/O / 1024 (from prescaler) |
| TCCR0 -> CS01 | 0 |
| TCCR0 -> CS00 | 1 |
| **INT1** | | |
| GICR -> INT1 | 1 | **External Interrupt Request 1 Enable**  External pin interrupt 1 is enabled |
| MCUCR -> ISC11 | 1 | **Interrupt 1 Sense Control**  The rising edge of INT1 generates an interrupt request |
| MCUCR -> ISC10 | 1 |

Table 6. TCNT0 and INT0 Pin configuration [1]

Inside the main loop, the state of the LED is set according to the value of *ledMode*. If *ledMode* is 1, the LED flashes at 50% duty cycle, and if *ledMode* is 0, the LED flashes at 20% duty cycle. As a result, initially, the LED shines at certain rate. After the first time the button is pushed, the LED becomes dim. Upon the next push, the LED returns to its previous brightness, and so on.

The waveform result of this program is presented below. One thing that can be observed when executing this program is that the bouncing issue is not resolved as smoothly as in *Task 3* although both tasks is applied with the same debounce method.



Figure 22. Task 6.2 waveform 20-50%

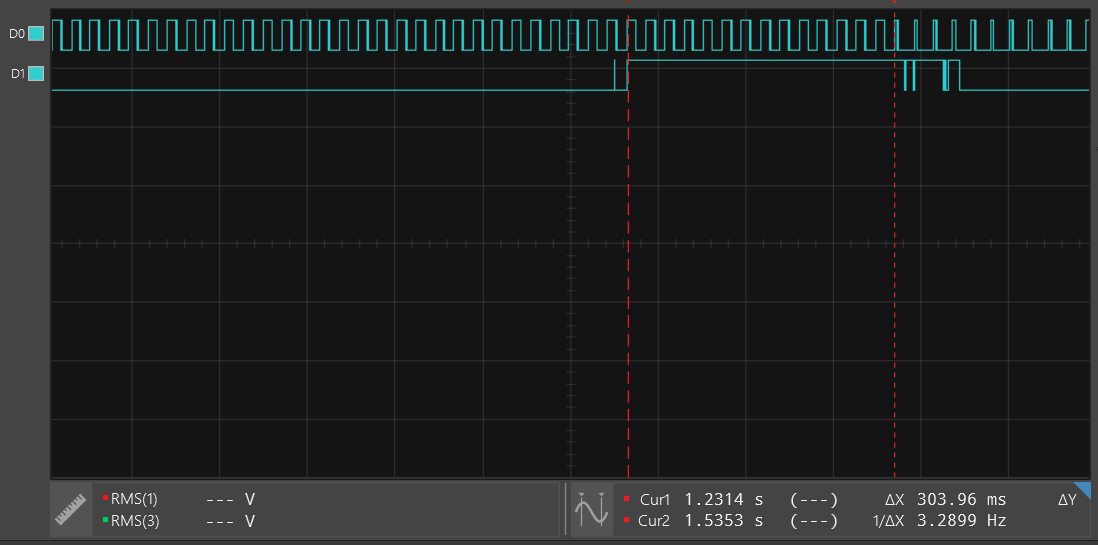


Figure 23. Task 6.2 waveform 50-20%

# CONCLUSION

On completing this lab activity, I have learned how to build basic electronic circuit and interface it with OUSB board. I have successfully written embedded programs using C programming language Atmel Studio development environment to perform simple functioning to control an LED and receive digital input from a push button. Other than that, I have gained basic knowledge of several basic peripheral features of the microcontroller, in particular Timer/Counter module and External Interrupt Request.

# REFERENCES

|  |  |
| --- | --- |
| [1] | Atmel Corporation, "ATmega32A Datasheet Complete," Aug 2016. |
| [2] | "Basic Light Emitting Diode guide", Dangerous Prototypes, Oct 2012. [Online]. Available: http://dangerousprototypes.com/docs/Basic\_Light\_Emitting\_Diode\_guide. [Accessed 20 Jul 2018]. |
| [3] | "What is a Pulse Width Modulation (PWM) Signal and What is it Used For?," National Instruments, May 2018. [Online]. Available: https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z0000019OkFSAU. [Accessed 23 Jul 2018]. |
| [4] | "Period And Frequency," Siyavula, [Online]. Available: https://www.siyavula.com/read/science/grade-10/transverse-waves/08-transverse-waves-06. [Accessed 30 Jul 2018]. |
| [5] | S. Davis, "Auditory and visual flicker-fusion as measures of fatigue," *The American Journal of Psychology,* vol. 68, no. 4, Dec 1955. |
| [6] | I. Hsu, "LED brightness adjustment: high-frequency PWM dimming", Texas Instruments, Aug 2016. [Online]. [Online]. Available: https://e2e.ti.com/blogs\_/b/powerhouse/archive/2016/08/26/led-brightness-adjustment-high-frequency-pwm-dimming. [Accessed Aug 2018]. |
| [7] | T. Floyd, *Principles of electric circuits : conventional current version*, 9th ed., Pearson Education Limited, 2014. |
| [8] | "Truth Table for Logical Operations," MathWorks, [Online]. Available: https://www.mathworks.com/help/matlab/matlab\_prog/truth-table-for-logical-operations.html. [Accessed 5 Aug 2018]. |
| [9] | V. Milosavljevic, "Polling vs. Interrupt," Electronics-Base, Dec 2011. [Online]. Available: http://www.electronics-base.com/useful-info/software-related/90-polling-vs-interrupt. [Accessed 5 Aug 2018]. |

# APPENDIX

## Task 1 – Toggling LEDs

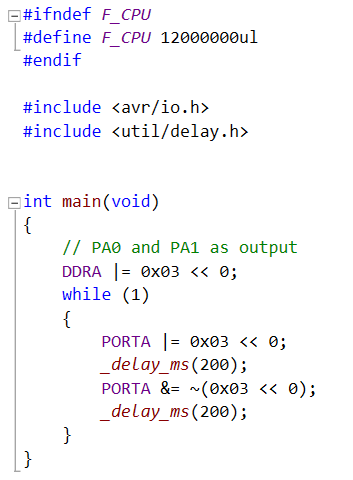


Figure 24. Task 1 - Toggling LEDs

## Task 2 – Controlling the Brightness of LED

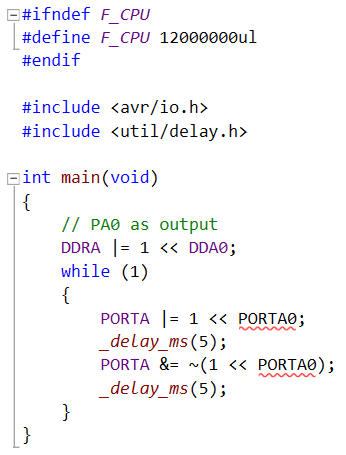


Figure 25. Task 2.1 - Flashing rate 100Hz

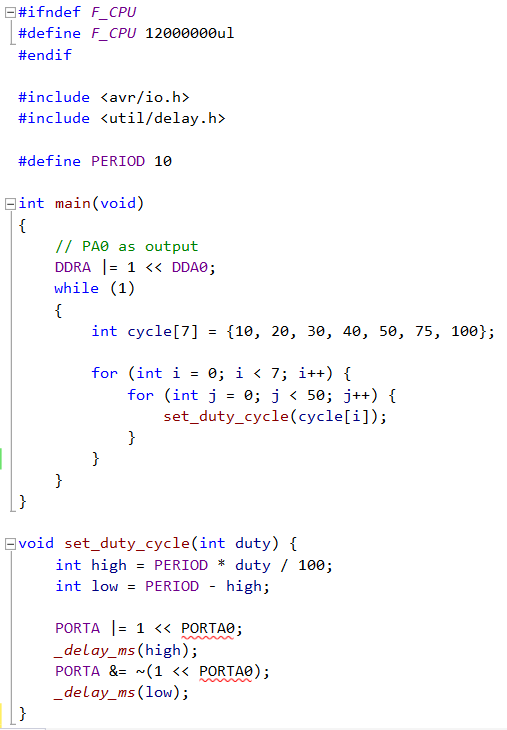


Figure 26. Task 2.2 - Various duty cycles

## Task 3 – Controlling LED with a Push Button

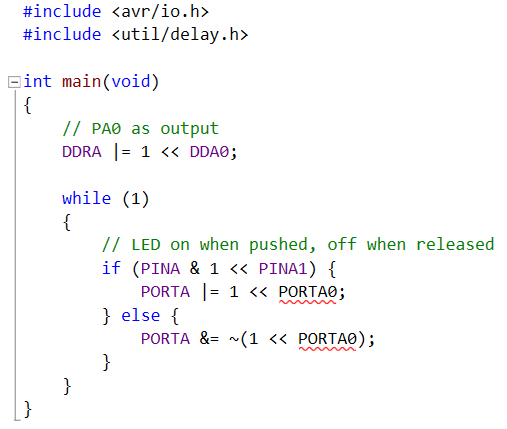


Figure 27. Task 3.1 - LED on when push, off when release

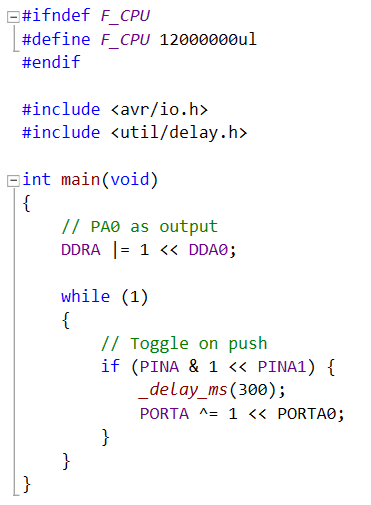


Figure 28. Task 3.2 - LED toggle on push

## Task 3 – Controlling the Brightness of LED using Timer

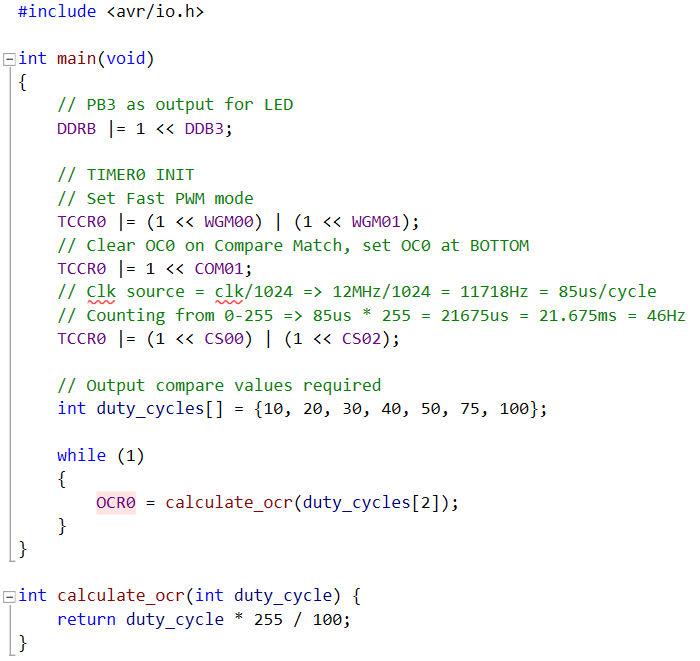


Figure 29. Task 4 - Control the Brightness of LED using Timer

## Task 5 – Toggle LED Using Timer

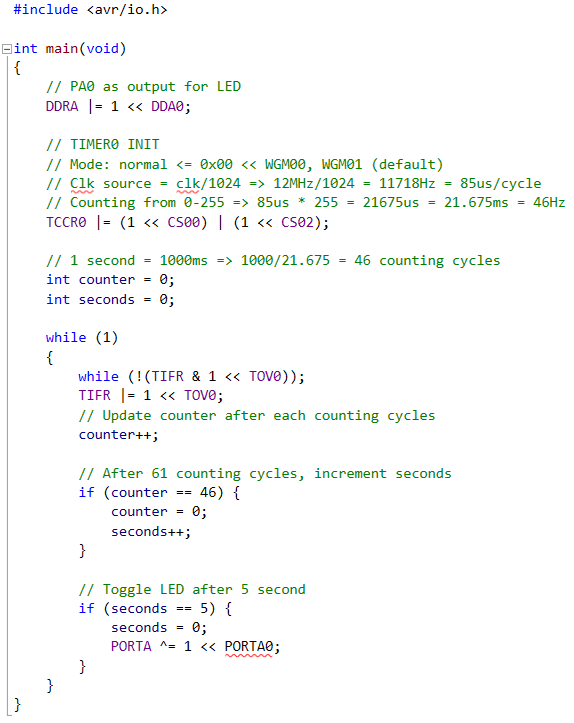


Figure 30. Task 5 - Toggle LED using Timer

## Task 6 – Toggle the Flashing Rate of LED using Interrupt

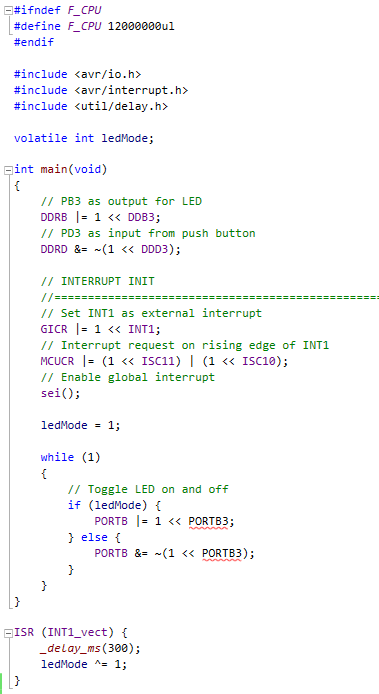


Figure 31. Task 6 - Toggle LED on push

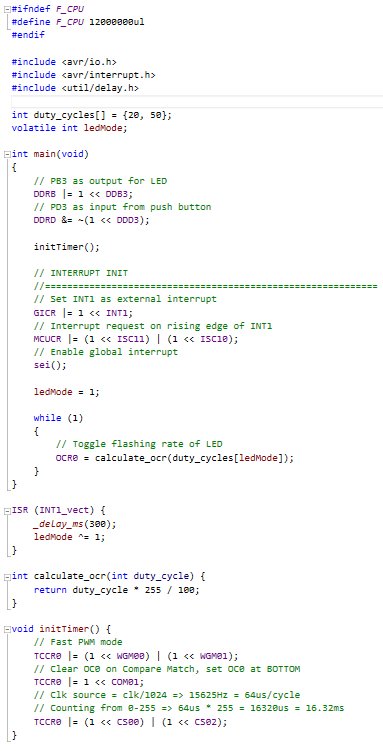


Figure 32. Toggle Flashing rate on Push