CAB202 Topic 10 – Analog to Digital Conversion and Pulse-Width Modulation

Authors:

- Luis Mejias QUT (2020)

Contents

Roadmap

Previously:

7. AVR ATMega328P Introduction to Microcontrollers; Digital Input/Output
8. Serial Communication – communicating with another computer/microcontroller
9. Debouncing, Timers and Interrupts. Asynchronous programming.

This week:

10. **Analogue to Digital Conversion; Pulse Width Modulation (PWM); Assignment 2 Q&A.**

Still to come:

11. LCD Display, sending digital signals to a device.

References

Recommended reading:

- Blackboard→Learning Resources→Microcontrollers→atmega328P_datasheet.pdf.

Analog to Digital Conversion

Introduction

Most of the physical quantities around us are continuous. By continuous we mean that the quantity can take any value between two extremes. For example the atmospheric temperature can take any value (within certain range). If an electrical quantity is made to vary directly in proportion to this value (temperature, etc.) then what we have is an analog signal which in most cases is a voltage. We have to convert this into digital form if we want to manipulate it with a digital microcontroller. For this an ADC or analog to digital converter is needed.

Analog signals have a frequency. A frequency is the number of occurrences of a repeating event per unit of time. For analog signals (in particular cyclical) is defined as a number of cycles per unit time. Frequency is measured in units of hertz which is equal to one occurrence of a repeating event per second. For signals that vary with time, samplig is defined as the measure of the value of the continuous signal every T seconds, which is called the sampling interval or the sampling period.

The Nyquist rate is the minimum sampling period required to avoid aliasing, equal to twice the highest frequency contained within the signal. Nyquist Rate = 2 x fmax. Therefore, we must be aware of the maximum frequency components of the analof signal so we can use the right samplig period. In our micronctrollers, this is defined by a pre-scaler. Out microcontroller has a fixed clock signal, so dividing this clock signal we can achive arbitrary frequencies that can be used to sample input signals connected to a particular micrcontroller pin.

ADC Register.

Our microntroller has 9 pins that can be used to read analog signal. These 9 pins are associated with a channel in the internal ADC circuitry. There is only one ADC circuitry, therefore these channels are multiplexed in time sharing the same core ADC converter.

The register associated with ADC are:

`ADMUX` – ADC Multiplexer Selection Register:

Bits	7	6	5	4	3	2	1	0
Name	`REFS1`	`REFS0`	`ADLAR`	`-`	`MUX3`	`MUX2`	`MUX1`	`MUX0`
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
initially	0	0	0	0	0	0	0	0

- `REFSx` = Reference Selection Bits: leave this at REFS0=1, REFS1=0. Ref voltage equal to Vcc (max input)
- `ADLAR` = ADC Left Adjust Result: leave this at 0
- `MUX[3:0]` = Analog Channel Selection Bits. See below

Values are:

`MUX3:0`	Input Channel Selection
`0b0000`	ADC0

0b0001	ADC1
0b0010	ADC2
0b0011	ADC3
0b0100	ADC4
0b0101	ADC5
0b0110	ADC6
0b0111	ADC7
0b1000	ADC8 (used for temperature sensors)

ADCSRA – ADC Control and Status Register A:

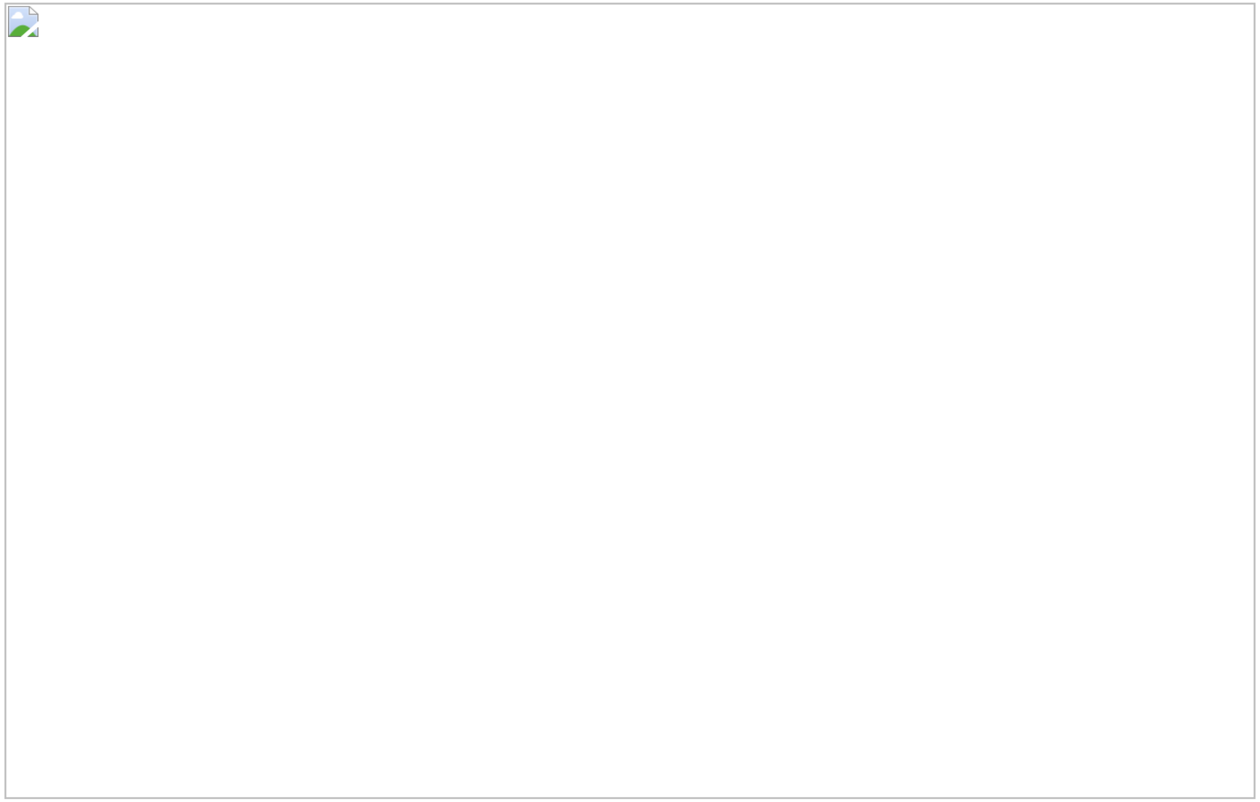
Bits	7	6	5	4	3	2	1	0
Name	**ADEN**	**ADSC**	**ADATE**	**ADIF**	**ADIE**	**ADPS2**	**ADPS1**	**ADPS0**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
initially	0	0	0	0	0	0	0	0

- **ADEN** = ADC Enable.Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off
- **ADSC** = ADC Start Conversion: In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion
- **ADATE** = When this bit is written to one, Auto Triggering of the ADC is enabled
- **ADIF** = ADC Interrupt Flag: This bit is set when an ADC conversion completes and the Data Registers are updated

- `ADIE` = ADC Interrupt Enable:When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.
- `ADPS[2:0]` = ADC Prescaler Select Bits

`ADPS2:0`	Pre-scaler
`0b000`	2
`0b001`	2
`0b010`	4
`0b011`	8
`0b100`	16
`0b101`	32
`0b110`	64
`0b111`	128

`ADC` – ADC Conversion result. 10 bits ADC9:0



Standard Configuration: Channel 0 in use, pre-scaler of 128

ADMUX

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0
0	1	0	0	0	0	0	0

ADCSRA

7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
1	1	0	0	0	1	1	1

Note: ADSC is set to one just before the conversion start, in single conversion mode.

The program below uses the ADC0 (channel 0) to read a potentiometer. During the main process a single conversion is performed, then we wait for the conversion to finish, then we read the result from the register **`ADC`**. Values are sent via UART for debugging.

The example1.c, ADCRead program illustrates ADC conversion.

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


/*
 *  Setting data directions in a data direction register (DDR)
 *
 *
 *  Setting, clearing, and reading bits in registers.
 *      reg is the name of a register; pin is the index (0..7)
 *  of the bit to set, clear or read.
 *  (WRITE_BIT is a combination of CLEAR_BIT & SET_BIT)
 */

#define SET_BIT(reg, pin)                   (reg) |= (1 << (pin))
#define CLEAR_BIT(reg, pin)               (reg) &= ~(1 << (pin))
#define WRITE_BIT(reg, pin, value)   (reg) = (((reg) & ~(1 << (pin))) | ((value) << (pin)))
#define BIT_VALUE(reg, pin)               (((reg) >> (pin)) & 1)
#define BIT_IS_SET(reg, pin)         (BIT_VALUE((reg),(pin))==1)

//uart definitions
#define BAUD (9600)
#define MYUBRR (F_CPU/16/BAUD-1)

// These buffers may be any size from 2 to 256 bytes.
#define  RX_BUFFER_SIZE  64
#define  TX_BUFFER_SIZE  64


//uart definitions
unsigned char rx_buf;

static volatile uint8_t tx_buffer[TX_BUFFER_SIZE];
static volatile uint8_t tx_buffer_head;
static volatile uint8_t tx_buffer_tail;
static volatile uint8_t rx_buffer[RX_BUFFER_SIZE];
static volatile uint8_t rx_buffer_head;
static volatile uint8_t rx_buffer_tail;


//Functions declaration
void setup(void);
void process(void);
```

```
void uart_init(unsigned int ubrr);
//uart functions
void uart_putchar(uint8_t c);
uint8_t uart_getchar(void);
uint8_t uart_available(void);
void uart_putstring(unsigned char* s);
void uart_getLine(unsigned char* buf, uint8_t n);
//ADC functions
uint16_t adc_read(uint8_t channel);
void adc_init();
//string convertion functions
void ftoa(float n, char* res, int afterpoint);
int intToStr(int x, char str[], int d);
void reverse(char* str, int len);

// END function declarations


//main loop
int main() {
        setup();

        for ( ;; ) {
                process();
                _delay_ms(50);
        }
}

//initialises ADC and UART port
void setup(void) {

    //init uart
        uart_init(MYUBRR);

        // Enable orange LED
        SET_BIT(DDRB, 5);

    // initialise adc
        // ADC Enable and pre-scaler of 128: ref table 24-5 in datasheet
    // ADEN  = 1
    // ADPS2 = 1, ADPS1 = 1, ADPS0 = 1
        ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);

   // select channel and ref input voltage
   // channel 0, PC0 (A0 on the uno)
   // MUX0=0, MUX1=0, MUX2=0, MUX3=0
   // REFS0=1
   // REFS1=0
    ADMUX = (1 << REFS0);

}


void process(void) {

   char temp_buf[64];
```

```
    // Start single conversion by setting ADSC bit in ADCSRA
         ADCSRA |= (1 << ADSC);

         // Wait for ADSC bit to clear, signalling conversion complete.
          while ( ADCSRA & (1 << ADSC) ) {}

         // Result now available in ADC
          uint16_t pot = ADC;

     //convert float to a string
     // ftoa(pot, temp_buf, 4);
     // convert uint16_t to string
      snprintf(temp_buf, sizeof(temp_buf), "%d", pot);

          //when converted value is above a threshold, perform an action
      if (pot > 512)
        SET_BIT(PORTB,PB5);
          else
        CLEAR_BIT(PORTB,PB5);


    //send serial data
    uart_putstring((unsigned char *) temp_buf);
    uart_putchar('\n');


}


/********** auxiliary functions *************/


// Reverses a string 'str' of length 'len'
void reverse(char* str, int len)
{
    int i = 0, j = len - 1, temp;
    while (i < j) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
        i++;
        j--;
    }
}

// Converts a given integer x to string str[].
// d is the number of digits required in the output.
// If d is more than the number of digits in x,
// then 0s are added at the beginning.
int intToStr(int x, char str[], int d)
{
    int i = 0;
    while (x) {
        str[i++] = (x % 10) + '0';
        x = x / 10;
    }

    // If number of digits required is more, then
```

```
    // add 0s at the beginning
    while (i < d)
        str[i++] = '0';

    reverse(str, i);
    str[i] = '\0';
    return i;
}

// Converts a floating-point/double number to a string.
void ftoa(float n, char* res, int afterpoint)
{
    // Extract integer part
    int ipart = (int)n;

    // Extract floating part
    float fpart = n - (float)ipart;

    // convert integer part to string
    int i = intToStr(ipart, res, 0);

    // check for display option after point
    if (afterpoint != 0) {
        res[i] = '.'; // add dot

        // Get the value of fraction part upto given no.
        // of points after dot. The third parameter
        // is needed to handle cases like 233.007
        fpart = fpart * pow(10, afterpoint);

        intToStr((int)fpart, res + i + 1, afterpoint);
    }
}



//PLEASE NOTE THIS VERSION OF UART USES INTERRUPTS

/*  ****** serial uart definitions ************ */
/******************  interrupt based  ********/

// Initialize the UART
void uart_init(unsigned int ubrr) {

        cli();

        UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)(ubrr);
        UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0);
        UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);
        tx_buffer_head = tx_buffer_tail = 0;
        rx_buffer_head = rx_buffer_tail = 0;

        sei();

}
```

```
// Transmit a byte
void uart_putchar(uint8_t c) {
        uint8_t i;

        i = tx_buffer_head + 1;
        if ( i >= TX_BUFFER_SIZE ) i = 0;
        while ( tx_buffer_tail == i ); // wait until space in buffer
        //cli();
        tx_buffer[i] = c;
        tx_buffer_head = i;
        UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0) | (1 << UDRIE0);
        //sei();
}

// Receive a byte
uint8_t uart_getchar(void) {
        uint8_t c, i;

        while ( rx_buffer_head == rx_buffer_tail ); // wait for character
        i = rx_buffer_tail + 1;
        if ( i >= RX_BUFFER_SIZE ) i = 0;
        c = rx_buffer[i];
        rx_buffer_tail = i;
        return c;
}


// Transmit a string
void uart_putstring(unsigned char* s)
{
    // transmit character until NULL is reached
    while(*s > 0) uart_putchar(*s++);
}


// Receive a string
void uart_getLine(unsigned char* buf, uint8_t n)
{
    uint8_t bufIdx = 0;
    unsigned char c;

    // while received character is not carriage return
    // and end of buffer has not been reached
    do
    {
        // receive character
        c = uart_getchar();

        // store character in buffer
        buf[bufIdx++] = c;
    }
    while((bufIdx < n) && (c != '\n'));

    // ensure buffer is null terminated
    buf[bufIdx] = 0;
}
```

```
uint8_t uart_available(void) {
        uint8_t head, tail;

        head = rx_buffer_head;
        tail = rx_buffer_tail;
        if ( head >= tail ) return head - tail;
        return RX_BUFFER_SIZE + head - tail;
}


// Transmit Interrupt
ISR(USART_UDRE_vect) {
        uint8_t i;

        if ( tx_buffer_head == tx_buffer_tail ) {
                // buffer is empty, disable transmit interrupt
                UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0);
        }
        else {
                i = tx_buffer_tail + 1;
                if ( i >= TX_BUFFER_SIZE ) i = 0;
                UDR0 = tx_buffer[i];
                tx_buffer_tail = i;
        }
}

// Receive Interrupt
ISR(USART_RX_vect) {
        uint8_t c, i;

        c = UDR0;
        i = rx_buffer_head + 1;
        if ( i >= RX_BUFFER_SIZE ) i = 0;
        if ( i != rx_buffer_tail ) {
                rx_buffer[i] = c;
                rx_buffer_head = i;
        }
}
```

TinkerCad implementation of the program:

https://www.tinkercad.com/things/4cldZq8ZPRB

Pulse-Width Modulation

Introduction

PWM is a modulation technique used to encode information into a signal, although its main use is for regulating power supplied to a load. It also can be used to generate analog signals using a digital source.

Consist of two main components that define its behavior: a duty cycle and a frequency.

- The duty cycle describes the amount of time the signal is in a high (on) stated as a percentage of the total time it takes to complete one cycle.
- The frequency determines how fast the PWM completes a cycle (i.e. 1000 Hz would be 1000 cycles per second), and therefore how fast it switches between high and low states. By cycling a digital signal off and on at a fast enough rate, and with a certain duty cycle, the output will appear to behave like a constant voltage analog signal when providing power to devices.

PWM signals are generated using timers. ATMega328P has 3 timers. Each timer can be connected to 2 or more output pins. Each timer has a counter, which cycles: In one direction, from 0 to TOP, at which point the timer wraps back to 0, or from TOP downward to 0. A timer can be set to repeatedly compare its counter to a threshold value set in a compare register. When the counter reaches the threshold, or when it hits 0, it can toggle the value of a digital output pin. We can use this to implement PWM in hardware or software.

In summary, what's involved in generating PWM signals.

- A counter, usually from a timer.
- A comparison value. This value is compared against the counter value.
- An output pin that toggles state, everytime the counter is equal to the comparison value.

Hardware-Based PWM

Timer0 registers in PWM mode (Datasheet, Section 15.9)

Each timer has a set of dedicated control and counter registers. Details are shown for Timer 0; you can look up the datasheet to find the corresponding registers for Timers 1 and 2.

- `TCCR0A` – Timer/Counter Control Register 0 A:

Bits	7	6	5	4	3	2	1	0
Name	`COM0A1`	`COM0A0`	`COM0B1`	`COM0B0`	-	-	`WGM01`	`WGM00`
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W
initially	0	0	0	0	0	0	0	0

  - `COM0Ax` = Compare Match Output A Mode.These bits control the Output Compare pin (OC0A) behavior
  -

COM0A1	COM0A0	Description
0	0	Normal port operation, OC0A disconnected.
0	1	Toggle OC0A on Compare Match
1	0	Clear OC0A on Compare Match
1	1	Set OC0A on Compare Match

  - `COM0Bx` = Compare Match Output B Mode.These bits control the Output Compare pin (OC0B) behavior

COM0B1	COM0B0	Description
0	0	Normal port operation, OC0B disconnected.
0	1	Reserved
1	0	Clear OC0B on Compare Match, set OC0B at BOTTOM, (non-inverting mode)
1	1	Set OC0B on Compare Match, clear OC0B at BOTTOM, (inverting mode)

- `WGM0x` = Waveform Generation Mode.Combined with the WGM02 bit found in the TCCR0B Register, these bits control the counting sequence of the counter

Mode	WGM02	WGM01	WGM00	Operation	Top	Update of OCRx at	TOV Flag Set on
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, phase correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	-	-	-
5	1	0	1	PWM,Phase correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	-	-	-
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOPM

- **MAX= 0xFF, BOTTOM= 0x00**

- `TCCR0B` – Timer/Counter Control Register 0 B:

Bits	7	6	5	4	3	2	1	0
Name	`FOC0A`	`FOC0B`	-	-	`WGM02`	`CS02`	`CS01`	`CS00`
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W

initially	0	0	0	0	0	0	0	0

- `FOC0x` = Force output compare: leave these at 0.
- `WGM02` = Waveform Generation Mode: should match the value used in `TCCR0A`.
- `CS02,CS01,CS00` = pre-scaler.
  Figures in this table assume that the CPU speed is set to 16MHz in the `setup` *phase.*
  Values are:

`CS02:0`	Counter updates
`0b000`	Never (Timer/Counter stopped)
`0b001`	Every clock cycle (No pre-scaling) == 16,000,000 ticks/sec
`0b010`	Every 8 clock cycles == 2,000,000 ticks/sec
`0b011`	Every 64 clock cycles == 250,000 ticks/sec
`0b100`	Every 256 clock cycles == 62,500 ticks/sec
`0b101`	Every 1024 clock cycles == 15,625 ticks/sec
`0b110`	External clock source on T0 pin. Clock on falling edge.
`0b111`	External clock source on T0 pin. Clock on rising edge.

  - We will not be using CS02:0 == 6 or CS02:0 == 7.

- `TCNT0` – Timer/Counter Register 0: an 8-bit numeric value. **Where the count is stored.**
- `OCR0A` – Output Compare Register 0 A: an 8-bit numeric value. Use this to adjut duty cycle
- `OCR0B` – Output Compare Register 0 B: an 8-bit numeric value. Use this to adjust duty cycle
- `TIMSK0` – Timer/Counter Interrupt Mask Register 0:

Bits	7	6	5	4	3	2	1	0
Name	-	-	-	-	-	`OCIE0B`	`OCIE0A`	`TOIE0`
Read/Write	R	R	R	R	R	R/W	R/W	R/W
initially	0	0	0	0	0	0	0	0

  - `OCIE0B` = Force output compare: leave these at 0
  - `OCIE0A` = Force output compare: leave these at 0

- `TOIE0` = Enable Timer Overflow Interrupt.

Case study: Generate a PWM signal using Timer0

In the present section we set up Timer 0, and see how to read the value of the clock.

- First, decide how fast we want the Timer/Counter register to update. That is the pre-scaler.
- Timer 0 is an 8 bit timer, so the Timer/Counter register will overflow every 256 ticks.
- We know the number of ticks per second from the datasheet, so we can calculate how long it will take for the timer to count from 0 to 255 (the overflow period) and how many times the counter will overflow per second (the overflow frequency).
  Definition: Frequency = 1 / Period.
  Figures in this table assume that the CPU speed is set to 16MHz in the `setup` *phase.*

`CS02:0`	Pre-scaler	Counter frequency	Overflow period = 256/freq	Overflow frequency
`0b000`	0	0	n/a	n/a
`0b001`	1	16MHz	0.000016s	62.5kHz
`0b010`	8	2MHz	0.000128s	7.8125kHz
`0b011`	64	250kHz	0.001024s	976.56Hz
`0b100`	256	62.500kHz	0.004096s	244.14Hz
`0b101`	1024	15.625kHz	0.016384s	61.035Hz

- Using the table, and balancing the update speed against our needs, we choose a pre-scaler.
- To set up Timer 0 to overflow about 7,8125 times per second, we choose `CS02:0 == 0b010 == 2`, which corresponds to a pre-scaler of 8.
- Starting the timer then consists of:
  - Set `TCCR0A = 0;`
  - Set `TCCR0B = 2;`
- Settng the PWM related register consist of:
  - Set `COM0A1` in `TCCR0A`.
  - Set `WGM01, WGM00` in `TCCR0A`.

- Set **OCR0A** to a value between 0 - 255, this sets the duty cycle.

This procedure is demonstrated in **example2.c, PwmTimer0**

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

int main(void)
{
    DDRD |= (1 << PD6);
    // PD6 is now an output

    OCR0A = 128;
    // set PWM for 50% duty cycle

    TCCR0A |= (1 << COM0A1);
    // set none-inverting mode

    // TinkerCAD Errata: timer clocking must be enabled before WGM
    // set prescaler to 8 and starts PWM
    TCCR0B = (1 << CS01);

    TCCR0A |= (1 << WGM01) | (1 << WGM00);
    // set fast PWM Mode


    while (1)
    {
        // write some code that changes the duty cycle
    }
}
```

It is recommended you use an oscillospoce and mutimeter to see the effect of the duty cycle.

Please note: the order of instructions, in tinkercad the Timer should be started before pwm is started.

[TinkerCad](https://www.tinkercad.com) implementation of the program:

https://www.tinkercad.com/things/6uUrw5Jl56j

Software-Based PWM

PWM signal can also be generate by software. The idea is simple. Define a timer with interrupt overflow, in the ISR routine increment a variable, then toggle the state of an output pin comparing this variable with a value used a comparator. This comparator value can be part of you main program. See pseudocode below

```
#define DELAY_MS 3

volatile uint8_t ISRcounter = 0; /* Count the number of times the ISR has run */

int main(void)
{
        //define comparison variable

        //set output pin

    //configure timer with interrupt overflow


   // enable global interrupts

        while(1){
                //set or increment comparator variable
                _delay_ms(DELAY_MS);
        }
        return 0;
}

ISR(TIMERx_OVF_vect)
{
        if(ISRcounter < comparator) {
                //set pin state high
        }else{
                //set pin state low
        }

        ISRcounter++;
}
```

WORKING WITH SERVOS

Frequency/period are specific to controlling a specific servo. A typical servo motor expects to be updated every 20 ms (at 50Hz) with a pulse between 1 ms and 2 ms, or in other words, between a 5% and 10% duty cycle on a 50 Hz waveform. With a 1.5 ms pulse, the servo motor will be at the natural 90 degree position.

With a 1 ms pulse, the servo will be at the 0 degree position, and with a 2 ms pulse, the servo will be at 180 degrees. You can obtain the full range of motion by updating the servo with a value in between.

Please note, that servos should be always calibrated, that is, to obtain either 0, 90, 180 degress the values won't be exactly 1ms, 1.5ms and 2 ms, but instead a value that close to it.

The idea is, define a signal that has a period of 20ms (50Hz) using the a timer and prescaler. If the combination of prescaler cannot achieve 50 Hz, pick the next value down, let's say 30Hz (33.3 ms). Then find the TOP count value that gives you 20ms. For example, if 65536 is the top count in 33.3 ms, then 39361 will give you approximately 20 ms. Then define you compare value so the time in the signal spend in high varies between 1ms and 2 ms.

Additional exercices:

1. Use the ADC to read a potenciometer and use this value to move a servo to a position that proportional to the potentiometer value.

Appendices

Appendix 1: Arduino UNO PIN D0 (PD0) UART RX

- The arduino PD0 (D0) RX pin is always high because the output from the USB / serial chip is high when it is not receiving anything. Therefore, connecting switches or any other input to D0 will always read High.

The End
