Day 1

Following is a concise report of the topics covered on Day 1 of the winter workshop.

1. Rectifiers

- Half wave rectifier, full wave rectifier, bridge rectifier (which is also a full wave rectifier).
- Addition of capacitor filter to reduce the ripple voltage to get an almost dc value – high value of capacitance should be taken.
- Further, use of linear voltage regulator IC 7805 to further reduce the ripples.

Ripples are fluctuations that can be potentially dangerous to the circuit elements if let be.

2. Filters

- Low pass, high pass, band pass, band reject filters.
- Concept of cutoff frequency- frequency at which gain decreases to -3dB of

it's original value.

- RC - low pass, CR - high pass.

3. Digital electronics basics

- AND, OR, NOT, NAND, NOR, XOR, XNOR gates.
- Universal gates - NOR, NAND - can implement all the other gates.
- Important theorems:
  - (A')'=A
  - AA'=0
  - A+A'=0
  - (A+B)'=A'B'
  - (AB)'=A'+B'

- XOR – High when odd number of inputs is high.
- XNOR – High when even number of inputs is high.

4. Multiplexer/Demultiplexer

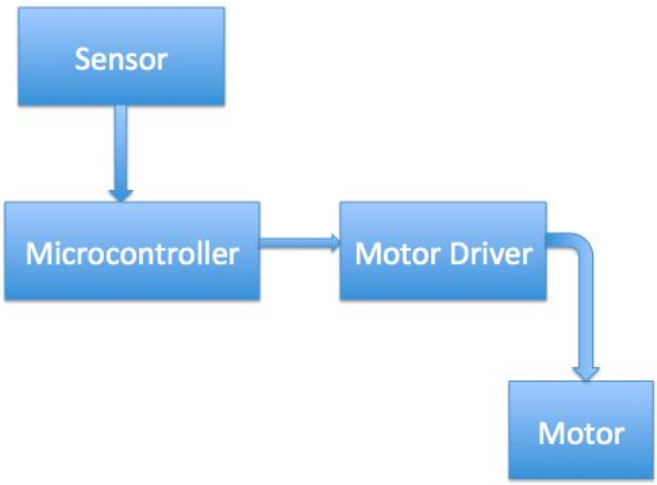
- Maps 2^n lines to one particular line so that the output of one particular line is mapped to output. This is decided by n extra lines called select lines. Eg. 4X1 MUX uses 2 select lines. DeMUX does the exact opposite job.

5. Signals

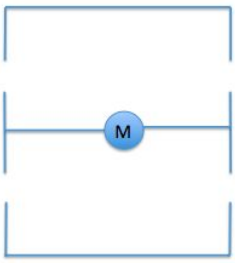
- Analog – harder to manipulate, but we have continuous data.
- Digital – simpler manipulation, but less data and noise may alter the value of bits (need parity checking)

6. Integrated Circuits
   - Datasheets contain all specifications of the device.
   - Monolithic packaged entity made up of transistors.
   - Never solder ICs directly as they cannot withstand the high temperature. Solder the IC bed first instead.
   - Surface Mount Devices (SMD) – good for when you are short on space. But, for prototyping, always use the bigger ICs.

   In our project, we will be using SONAR as a sensor to map the environment. Alongside this, we will be using a servomotor as an actuator to move the turret.



   H-Bridge structure to operate motor driver.



   We need a motor driver as an intermediate as the motor requires a current of about 20 amperes, while the microcontroller can supply a maximum of about a few milli-amperes.

   Switches are implemented using Si transistors.

   The motor is basically an inductive load, as it has coils. Now, when the current changes suddenly in an inductor, we will have high voltage spikes. (V=L.dI/dt)

But the operating voltage of Si switching devices is very less and they may get damaged due to the high voltage spikes. So we use flyback diodes to provide a safe path for the current.
TTL – Transistor Transistor Logic – 0,1 – 0V, 5V
CMOS – Complementary MOS – 0,1 – 0V, 3.3V

Motor:
0 0 Free Run – keeps running till frictional losses dissipate energy
1 0 Clockwise
0 1 Anticlockwise
1 1 Braking – stops immediately

Motor driver used : L293D – The 'D' stands for flyback diodes.
Metal strip used as a heat sink – at the ground pins.
This implements the H bridge. Here Vcc=5V for the logic and Vss=12 V is given for power. Generally, we want to isolate the power and logic components to reduce noise.
I1, I2 control M1 and I3, I4 control M2 (2 H-bridges)
0, 1 | 1, 0 | 0, 0 | 1, 1. Basically, these I inputs correspond to different states of the switches in the H-bridge (MOSFET switching)

Opto-isolator: Isolation of circuits using LED-Photodiode pair.

Operational Amplifier – Amplifies the voltage. We are going to use this as a differential amplifier. This uses bipolar dc power as the transistors inside need to have a proper quiescent point.

$V0= A(V_+ - V_-)$

Comparator – Modified OP-AMP. V+>V- ->V0 = 5 V – Pulled up (actually floating)
V+<V- ->V0= 0 V

Pulse Width Modulation (PWM) – Used to generate analog from digital without using DAC. This is done by varying the duty cycle of a square wave of value {0,Vcc}
Note: This will only work if the sampling rate is less than the frequency, ie. If it has a bad resolution.

Now, we give PWM to enable input. So, when the square wave has high value, the motor runs and when it is low, the motor is in free run. So, by using PWM, we can control speed of the motor.

Day 2

1. Microcontroller
   - Flash NVM – Non-Volatile Memory
   - EEPROM – Electrically Erasable Programmable Read-Only Memory
   - SRAM – Static Random Access Memory
   - We would be using AtMega16A
   - Data Path Bit Width – 8 bits – No. of bits in the registers
   - Flash Memory – 16KB
   - Synchronized device- common clock – generated by quartz oscillator.
   - Frequency of microcontroller – frequency of clock pulse -16 MHz for AtMega16A

2. AVR Architecture
   - RISC Processor – Reduced Instruction Set Computing
   - Simple set of instructions. All complex tasks can be accomplished as some combination of these instructions.
   - .c program compiled into a .hex file, which is written into the flash memory of the microcontroller.

We will be manipulating the bits in the registers to get the job done.
The bits are combined with logic gates in such a manner that changing a bit gets the job done.

3. Peripherals
   - GPIO – General Purpose Input and Output
   - Timer – to keep track of world time. All the microcontroller knows is the clock pulse. So, a timer is implemented by counting the number of clock cycles – PWM can be generated.
   - Analog to Digital Converter (ADC) – To convert PWM into digital components.
   - Protocol – common set of instructions that – USART (Universal Synchronous/Asynchronous Receiver/Transmitter)
   - Interrupts – Eg. Obstacle avoiding – keep running. If you encounter an obstacle, stop, turn appropriately and keep running.

SONAR – send a pulse, receive it, measure the time in between. Peripherals – GPIO; GPIO; Timer
Motor actuation – PWM

In this project, we would not be using ADC and USART for the most part – auxillary.

4. Interrupts
- Timer Interrupt – Interrupts that run when TCNT reaches TOP.
- External Interrupt – Interrupt depending on when the external signal is high or low.

GPIO pins are grouped into ports, each having 8 pins.
Port A 0-7
Port B 0-7
C 0-7
D

Now, each pin can be used for input or output. But how do we let the microcontroller know?
DDRX – 8 bit register for each port (DDRA…) – set value 0 for Input, 1 for output.

7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	1
I	O	O	I	O	I	O	O

Alongside this, to set the values, we also have a PORTX register

7	6	5	4	3	2	1	0
X – Don't care.	Can be set	Can be set	X	Can be set	X	Can be set	Can be set

Since the input bits depend on the input provided. So, it is useless to set those bits as they won't be taken into consideration.

How to set a bit? Take or with 1(which should be in the required position)
But DDRA|1 won't work as we will have DDRA|00000001 and only the last bit can be set this way.
So, instead: DDRA|(1<<n) or otherwise DDRA|(0b00010000) (n=4 ie. Wanting to set pin4 to 1)

DDRA=0b00001000 – Sets pin 4 as output.
PORTA=0b00001000 – Sets pin 4(output) as high

3 types of registers:
DDR: I/O
PORT: tells if output pins are set as high or low.
PIN: cannot be written into. Can only be accessed to obtain pin states.

Each port has a dual function apart from GPIO. For port A, it is ADC conversion.
0-5V is mapped to 0-255(8 bits)

Set – DDRA | (1<<n)
Clear - DDRA & ~(1<<n)
Toggle – PORTX^(1<<n)

Blink LED-
Loop 2 to 5

1. Switch ON
2. Wait
3. Switch OFF
4. Wait

We use PC6 for blink example

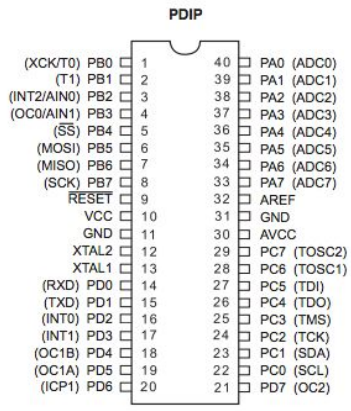
```
DDRC=(1<<6)

{
        PORTC=(1<<6);
        delay(t);
        PORTC& = ~(1<<6);
        delay(t);
}
```

A better way to do this:

```
DDRC=(1<<6);
{
        PORTC^=(1<<6);
        delay(t);
}
```

Exercise: Implementing the same with PC2

```
DDRC=(1<<2);
{
        PORTC^=(1<<2);
        delay(t);
}
```



PB0-PB7: Port B
PA0-PA7: Port A
Reset', Vcc, AREF, Avcc

There's a pullup resistor to Vcc connected at Reset' pin. Also, it is connected to ground via push-button. If push-button is on, Reset'=0 so that the board gets reset. When it is off, Reset'=1(pull up) and no reset happens.
AREF – sets the maximum voltage of ADC output. (AREF mapped to AREF 1023)

#define F_CPU 16000000UL
UL means unsigned long. If the above line is not included, then the program considers a default value of 1 MHz. But, in reality, the frequency is 16 MHz. So, the same task(say delay) will get completed 16 times faster. This is as time is maintained by counting clock cycles. Say t ms map to x clock cycles at 1 MHz. Then at 16 MHz, t/16 ms map to x clock cycles. Then, if the program perceives it as 1 MHz, it will wait for x clock cycles and say that it has waited for t ms, whereas actually, it has only waited for t/16 ms.

TCNT- Main timer register- adds one at every clock high- 8 bit- overflows after some time and time gets reset- (16*10^6)/256 overflows per second – Normal Mode. This is a lot. How do we reduce this?
Last 3 bits of TCCR0- Prescalers- divide the clock pulse by some value(scaling)- the clock frequency remains same, but the timer updates only after n clock pulses, where n is the prescaler value.

__--__--__--__--__--__ Clock Pulse

__--_____--_____ What TCNT observes

Now, how do we use timer to generate PWM?
TCNT0 vs t graph – Ramp wave.
Set some OCR value in OCR0 register(say 200). Then, till the OCR value(200), high output is given and after OCR is crossed, low output is given. At overflow, it is reset

to high. In this way, we will have a PWM of value (OCR/256)*maxvalue. Duty cycle is OCR/256.

CTC- Frequency increased as TOP is decreased.

Day 3

3 Timers – Timer/0 Timer/2 – 8 bit timers Timer/1 – 16 bit timer

WGM00,WGM01 – set the mode – 4 possibilities
0 0 – Normal – 0-255 – overflow
1 1 – Fast PWM – brings OCR into picture
1 0 – CTC – count till OCR and overflow – frequency increases
0 1 – PWM, Phase Correct – counts till 255, but doesn't directly overflow. Instead, it comes back in a similar way(triangular wave formed)

COM01,COM00 – Used to set the nature of output in given mode – 4 possibilities
TCCR0 – gives output at PB3. When PWM enabled – it is called OC0..
OC0 – pin where we get PWM

Note: Always put loose conditions in program, as there may be an error in comparision.

When does TCNT start? – when CS00 becomes 1(some prescaler is set)

```
DDRB=(1<<3);
TCCR0=(1<<WGM01)|(1<<WGM00)|(1<<COM01)|(1<<CS00)
```

Why do we not use pulldown resistor? Because the fluctuating voltage at pin may become 5V and at that moment, the microcontroller resets. But the other way round, in pullup mode, the fluctuating voltage can never become zero unless and until it is physically connected to ground.

ADC (10 bit)
ADC works at 200 kHz
Aref – sets maximum of conversion(5V - 255)
Data path Bit Width – 8 bit. So, all 16 bit registers are divided into low and high ADCL,ADCH

Two control registers – ADMUX, ADSCRA

ADMUX- REFS – sets the mode

- **ADLAR – 0 – ADCL is full and only first two bits of ADCH are used. 1 – all the bits are moved into ADCH and only 2 remain in highest bits of ADCL.**

Used when say writing new values wipes out one of the registers, we would prefer to wipe out the LSBs – ADLAR 1 – less error

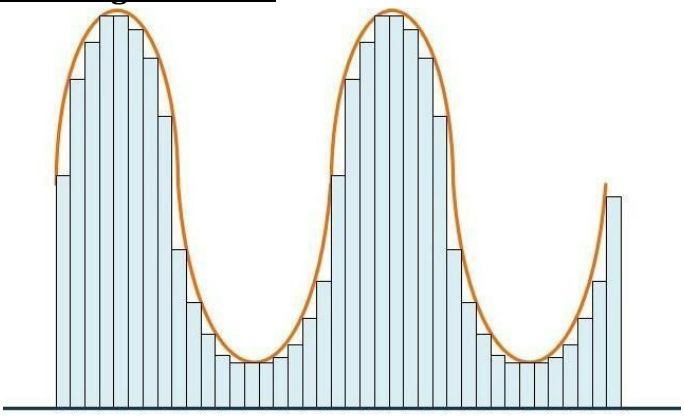
- MUX – dives command to the MUX to let the desired ADC inputs to pass to the ADC.

ADCSRA – ADEN – ADC enable

- ADSC – set 1 to start conversation, 0 when conversation ends. 0 is set automatically by the hardware. ADATE – Auto trigger enable/disable.

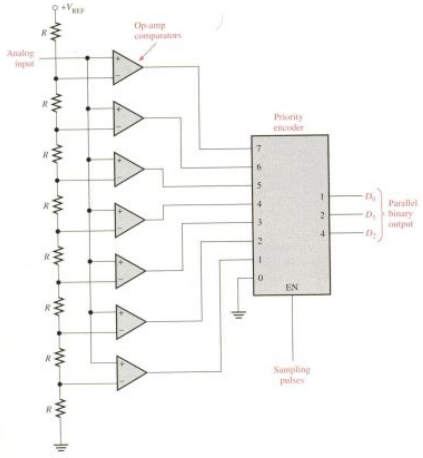
DAY 4

Working of an ADC



ADC samples the values of the input and holds it at that particular value till the next sampling.

What's inside? There's a voltage divider of 5 V using multiple resistors and from each division, a branch goes into one terminal of one comparator(1023 such comparators for a 10 bit ADC) The other terminal of all comparators gets the input voltage. So, using this, ADC checks the value that is just below the input and passes that through a priority encoder.



Now, for a typical sine-wave input, we do not want the ADC sampling rate to be too slow (prescaler too high), otherwise the output will not be a good approximation

and may not represent a sine wave. So, we generally prefer a higher sampling rate. But, if the sampling rate is too high (prescaler too low), the comparators and the encoder won't have enough time to provide the output and before they can do so, next sampling will be triggered. So, we usually settle for a middle spot – sampling rate of about 200 kHz.

However, consider the case in which we have a PWM. Then, we would take the maximum prescaler as we don't wan't the ADC to know that it's actually PWM and not an average value. Also, the problem of inaccuracy does not arise as with PWM, we WANT to output the average (inaccurate) value.

External Interrupts

#include<avr/interrupt.h>

Interrupt stops the execution of the code then and there, and the flow then jumps to the interrupt, services the interrupt, and then starts running the code back from where it stopped.

For external interrupt, we have three pins – INT0, INT1 and INT2.

MCUCR – MCU Control Register.

ISC – Interrupt Sense Control bits – we only have these for INT0, INT1, but not for INT2 as it only configures itself with the edge change (only two options) and it does not have the other options. Hence, it is less versatile and less used.

MCUCSR – MCU Control and Status Register

ISC2 bit – only one bit to set rising edge or falling edge interrupt.

GICR – General Interrrupt Control Register - 3 enable bits for the 3 interrupts.

GIFR - General Interrrupt Flag Register – 3 bits to trigger interrupt request – this is where the actual stuff happens – as soon as flag set to one, Interrupt Service Routine (ISR) is called with Interrupt Vectors as arguments.

ISR – overloaded function.

A vector is basically a pointer.

Interrupt vectors –

Note: RESET is also an interrupt vector. It takes the flow to the beginning of the code.

GICR - IVCE bit – set 1 to disable interrupts

General Algorithm to follow:

- Disable Interrupt.
- Set Bits.
- Re-enable.
- ISR.

Need not manually set the bits. sei() – enables interrupt (enables ALL the interrupts). cli() – disables interrupt.

Timer Interrupt

Is generated as soon as TCNT overflows, or when there is compare match. So, we need not check for overflows everytime.

TIMSK register- Timer/Counter Interrupt Mask Register

OCIE0 – Interrupt at output compare match – by virtue of OCR0 register.

TOIE0 – Interrupt at MAX.

TIFR – actually records if the interrupt has taken place – just like GIFR.

For decalring global variables, instead of int, we use volatile uint8_t <variable name>

Servo Motor

We can control alignment - +90 to -90 degrees.

3 pins – Ground, Vcc, PWM.

Depending upon PWM, direction of Servo motor is decided.

For PWM – T=20ms.

At Ton = 1.5 ms, points straight ahead (0^0), 1 ms – (-90^0), 2 ms – ($+90^0$). Rest is linearly graded.

So, for us, the PWM is effectively reduced to 1-2 ms.

Note: we will have to use 16 bit timer as with 8 bit, even the maximum prescaler won't give 20ms as the time period.

Day 5

SONAR

SONAR works by transmitting a pulse and when it reflects from an object and is received by the receiver. This time is measured and hence distance of the target can be calculated.

SONAR has 4 pins – Vcc, GND, TRIGGER, ECHO.

TRIGGER – Sends out the pulse to be reflected back from surrounding objects. Time period - 10 microseconds. 8 pulses in succession.

ECHO – As soon as falling edge of 8^{th} pulse occurs, echo is set and as soon as the pulse is received back after reflection, echo is cleared (by external interrupt). The duration of echo is the time taken for pulse to go and come back (we use timer interrupt to find this time).

Communication Protocols

1.USART – Universal Synchronous/Asynchronous Receiving/Transmission more used form UART (Asynchronous mode) – generally slow.

Full Duplex – Transmission, Receiving is independent. Both may happen simultaneoulsy.

Serial Communication Protocols.

Two pins – Tx – Transmission, Rx – Receiving.

Start Bit, Eight Bits of Data, Stop bit.
(low), anything, (high)

(_)_-_-_--_(-) : Bit sequence.

We prefer UART as it is easy to implement and can be connected via USB to computer – RS232 protocol for USB to UART: done using a chip called an FTDI (AtMega16U2 in Arduino)

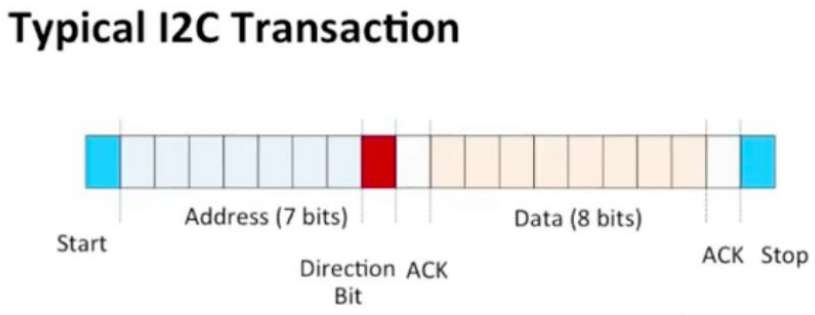
Shift register – 8 bits. Whatever data needs to be written is first parallelly written into the shift register and then, it is serially transmitted from there.
Buffer popped into -> Shift Register.
The start and stop bits are generated by the microcontroller.

2. SPI – Serial Peripheral Interface
Used to program microcontroller. High transmission speeds.
Master-Slave concept – Synchronised clock.
Three wire communication – MISO, MOSI, $(SS_x)'$
We normally don't prefer connecting multiple devices as it takes up more pins on the master.
MISO – Master In Slave Out - Read
MOSI – Master Out Slave In - Write
CLK – shorted for all GND – shorted for all
$(SS_x)'$ – normally pulled up. To activate device X, pull down $(SS_x)'$ and then transmission can be done depending on MISO or MOSI.

3. I2C – Inter-Integrated Circuit
Two wire communication – SDA, SCL – Bus formed - in open drain condition (ie. pulled up)
SDA – Serial Data Line, SCL – Serial Clock Line (since it is synchronous)
Slaves can only receive and transmit information and give the acknowledgement bit.
Master sends the start condition to indicate the beginning of a transaction, then it sends the address (7 bits – has the address of some register inside the slave. 8^{th} bit - direction ie. read vs. write – all this is sent one bit per clock cycle) first to the bus, which is then transmitted to all slaves. The receiver corresponding to the address acknowledges this by sending an acknowledgement bit (for error checking). Now, the master sends another request, this time either transmitting or receiving information. The slave complies again and caters to the request, again terminating with an acknowledgement bit. If the master does not receive the acknowledgement bit, it keeps sending the request again and again in order to establish a connection with the slave.
Start condition: SDA falls from 1 to 0 when clock is high.
Stop condition: SDA rises from 0 to 1 when clock is high.

In ALL other cases, SDA is constant when clock is high and only changes value in clock low.

Due to all the tedious processes involved, and also because of serial communciation, I2C is generally very slow.
SDA is sampled by receiver on the rising edge of SCL.
Direction bit – 0 for write, 1 for read.
For acknowledgement, receiver pulls SDA low for one clock cycle – ACK happens after every byte.



IMU – Intertial Measurement Unit.

Accelerometer, Gyroscope, Magneto/Digital compass.

ADXL 335 – Accelerometer – 5 pins – Vcc, GND, a_x, a_y, a_z
All of the above are in the form of voltages. We pass the last 3 pin values through 3 pins of ADC and get their values one by one. Now, we can easily find the inclination(s).

Gyroscope – MPU9150 – Theta+=Omega (delta(t))

But over the course of time, there may be some dW which we may be adding due to latent vibrations. This adds up over time to cause a significant error in theta – about 2 deg/min.
So, to rectify this, we use a filter – Complementary filter.
C.F. = (1-A)(accelerometer data) + A(gyroscope data)
We choose a small A as after a while, only error in gyro data remains and we don't want to add up the errors. Also, for a short period of time, gyro data is more accurate – generally 0.02.
Other filters – DCM, Kalman.

Roll, Pitch, Yaw.
Yaw will give the maximum error as for Yaw, we need ax and ay, which are zero and we will be getting a garbage value.

Day 6

Any variable that is to be accessed by both ISR and main has to declared as volatile. The volatile keyword signifies that the value of the variable can be changed by external factors apart from the main code.

Day 7

Sonar code:

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include <avr/interrupt.h>

volatile uint8_t ovf = 0;
volatile uint8_t i = 0;
volatile uint8_t current = 0;

ISR(INT1_vect)
{
   int flag = 0;
   if (i >= 100)
   {
      current /= 100;
      if(current < 74)
         PORTC = (1<<6);
      else if(current < 147)
         PORTC = (1<<7);
      else
         PORTC = 0;
      i = 0;
      current = 0;
      flag = 1;
   }
   if (flag == 0)
      current += ovf;

   PORTC |= (1<<4);
   _delay_us(10);
   ovf = 0;
   PORTC &= ~(1<<4);
   TCNT0 = 0;
   i++;
}

ISR(TIMER0_OVF_vect)
```

```
{
   ovf++;
}

int main()
{
   DDRC = (1<<4) | (1<<6) | (1<<7);
   cli();
   GICR = (1<<INT1);
   MCUCR = (1<<ISC11);
   TIMSK = (1<<TOIE0);
   sei();
   TCCR0 = (1<<CS00);
   TCNT0=0;

   PORTC |= (1<<4);
   _delay_us(10);
   ovf = 0;
   PORTC &= ~(1<<4);
   TCNT0 = 0;

   while(1)
   {
   }
}
```

Servomotor:

```
#include <avr/io.h>
#define F_CPU 16000000UL

void servoturn(float deg)
{
        if(x<0)
                float num = (90+deg)*25/24;
        if(x>0)
                float num = (150+deg)*5/8;
        OCR1A = int(num);
}

int main()
{
        DDRD=(1<<4);
        ICR1=1250;
        TCCR1A=|(1<<WGM11)|(1<<COM1A1);
        TCCR1B=(1<<WGM12)|(1<<WGM13)|(1<<CS12);
```

```
        while(1)
        {
                servoturn(0);
        }
}
```

Final Code:

```
#include <NewPing.h>
 #include <Servo.h>

#define SONAR_NUM     3 // Number or sensors.
#define MAX_DISTANCE 400 // Max distance in cm.
#define PING_INTERVAL 33 // Milliseconds between pings.
#define y 0.434
#define x 4.962
float angle(float ,float,int ,int);
 Servo myservo;
unsigned long pingTimer[SONAR_NUM]; // When each pings.
unsigned int cm[SONAR_NUM]; // Store ping distances.
uint8_t currentSensor = 0; // Which sensor is active.

NewPing sonar[SONAR_NUM] = { // Sensor object array.
  NewPing(12, 11, MAX_DISTANCE),
  NewPing(10, 9, MAX_DISTANCE),
  NewPing(8, 7, MAX_DISTANCE),

};

void setup() {
  Serial.begin(115200);
  pingTimer[0] = millis() + 75; // First ping start in ms.
  for (uint8_t i = 1; i < SONAR_NUM; i++)
    pingTimer[i] = pingTimer[i - 1] + PING_INTERVAL;
    myservo.attach(6);

}

void loop() {
  for (uint8_t i = 0; i < SONAR_NUM; i++) {
    if (millis() >= pingTimer[i]) {
      pingTimer[i] += PING_INTERVAL * SONAR_NUM;
      if (i == 0 && currentSensor == SONAR_NUM - 1)
        oneSensorCycle(); // Do something with results.
      sonar[currentSensor].timer_stop();
```

```
      currentSensor = i;
      cm[currentSensor] = 0;
      sonar[currentSensor].ping_timer(echoCheck);
    }
  }
  // The rest of your code would go here.
}

void echoCheck() { // If ping echo, set distance to array.
  if (sonar[currentSensor].check_timer())
    cm[currentSensor] = sonar[currentSensor].ping_result / US_ROUNDTRIP_CM;
}

void oneSensorCycle() { // Do something with the results.
  for (uint8_t i = 0; i < SONAR_NUM; i++) {
    Serial.print(i);
    Serial.print("=");
    Serial.print(cm[i]);
    Serial.print("cm ");
  }
  int p1=cm[2];int p2=cm[1];int p3=cm[0];
  Serial.println();
float an;
if(p3 == 0)
  p3 = 401;
if(p1 == 0)
  p1 = 401;
if(p3>p1)
{
  an=angle(-x,y,p2,p1);
  if(an == 360)
    return;
  myservo.write(-an);
  Serial.println("Angle");
  Serial.print(-an);
}
else
{
  an=angle(x,y,p2,p3);
  if(an == 360)
    return;
  myservo.write(180-an);
  Serial.println("Angle");
  Serial.print(180-an);
}
```

```
}
float angle (float a , float b,int d1,int d2)//a=It is The x-cordinate.b=It is the
Y_cordinate
                                //d1=distance detected by sonar at center.
                                //d2=distance detected by the sonar at the left or right.
 {
  float c,temp,x1,y1;

  if(d1==401||d2==401)
  return 360;
  if(d1!=0 && d2!=0)
  {
  c=(d1*d1 + a*a +b*b - d2*d2)/2;
  temp=(b*c)/(a*a + b*b);
  y1= temp + sqrt(temp*temp - (c*c - a*a*d1*d1)/(a*a +b*b));


  x1=(c -b*y1)/a;
  //if(x1 == 0)
    //return 360;

 return 180*atan(y1/x1)/(3.14);
 }
 else
 {}
 }
```