Laboratory Report

To: Dr.Randy Hoover

From: Benjamin LeBrun, Benjamin Garcia

Subject: Lab Assignment 6: TC1 and Ultrasonic Sensors

Date: April 13, 2019

Introduction

This lab tasked us with configuring the provided HC-SR04 ultrasonic range sensor using the ATMega328P's Timer/Counter1. We then had to provide a linear mapping from $[3cm, 30cm]-> [0, 255]$ and control the robot's motor speed using this mapping. The sensor output and appropriate mapping was to then be displayed to the screen with every distance sample.

Equipment

While the lab used the entire robot kit assembled together, the primary devices we used were:

- Acrylic vehicle body with screws, assembled
- Elegoo Uno (chip: Atmega328p)
- 4 DC motors with wheels, screws
- L298 H bridge module dual channel
- 2 ICR18650 batteries with battery box
- Ribbon cables
- Host laptop with AVR-gcc 8-bit toolchain
- USB 2.0 A to B cable
- HC-SR04 ultrasonic range sensor

Configuration

Our robot vehicle was assembled according to Elegoo's instructions which can be found on Elegoo's website at https://www.elegoo.com/download/. For this lab, we are using the V3.0 version of the robot kit.

The ultrasonic sensor, battery pack, and H bridge were connected to the marked locations on the Elegoo shield included in the kit. The wheel motors were connected two to a side with the H bridge in the lower portion of the robot.

While the motors require more power than the micro-controller can provide to operate, the ultrasonic sensor is capable of running off of the micro-controller's power. Both components are capable of running off of the kit-provided battery pack when it is switched on.

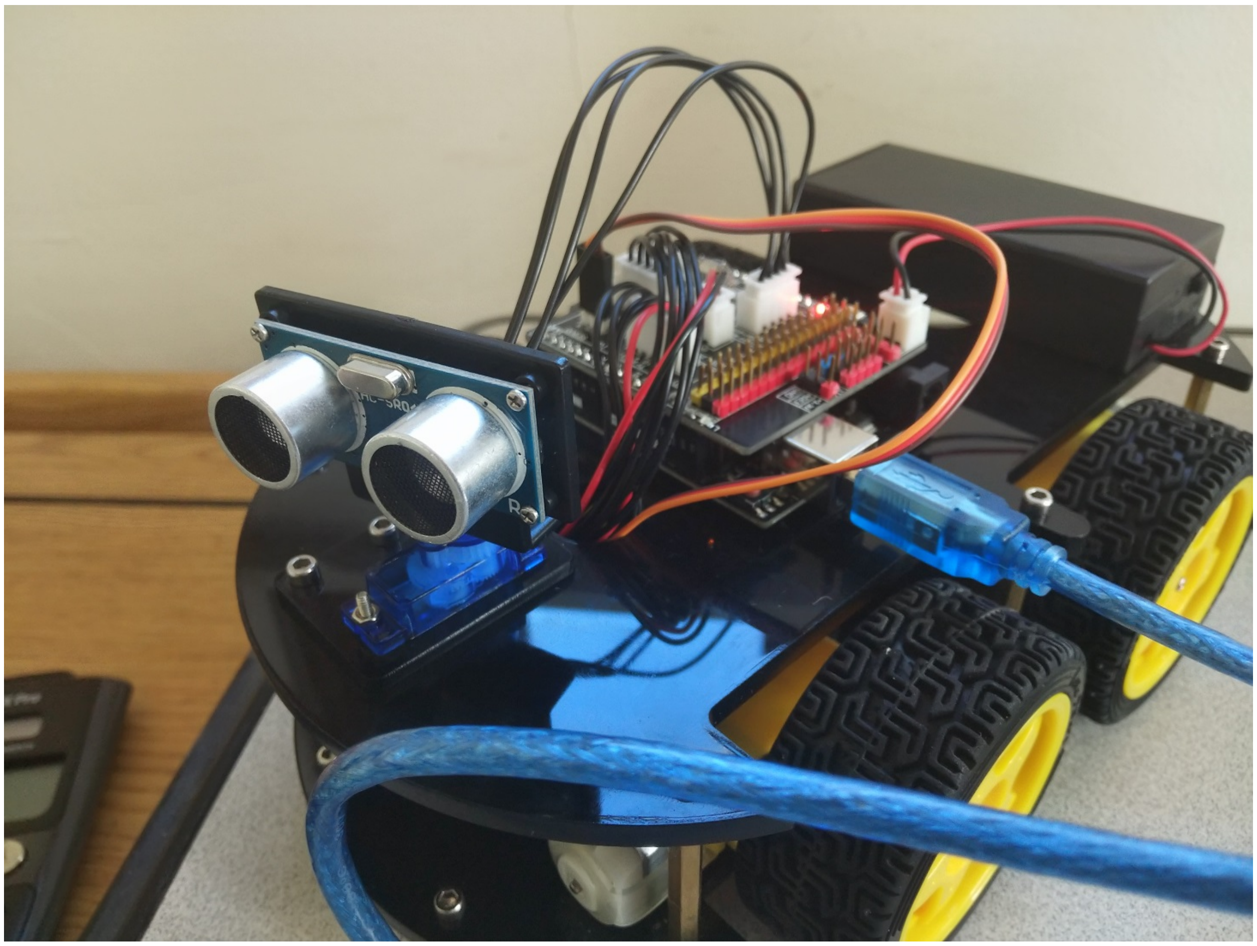

Figure 1: Robot configuration with ultrasonic sensor.

Implementation

For this assignment, we implemented some basic timer control and interfacing logic for the ultrasonic sensor. Additionally, we created the beginning of a pin-change interrupt library to help with controlling the robot later on as the complexity of the assignments increases.

In addition to the new code, the motor control code from the last lab assignment has been brought over to the current assignment, and implementation details are reproduced here from the prior assignment.

initPCINT

Initialization logic for the pin-change interrupts. Sets the pin change interrupt control registers *PCMSK1* and *PCICR* using the defined bit strings *PCMSK1_CONFIG* and *PCICR_CONFIG* provided in the file 'pcint.h'.

ISR(PCINT1_vect) (interrupt)

Interrupt handler for the second pin change interrupt vector on the board. This assignment only required the use of PCINT12 (PORTC4, pin labeled on the board as A4), which allowed the interrupt to be simplified from the more general approach of capturing the current pin state and comparing with the previous state. This also made the interrupt take less time, and therefore provide a more accurate reading for the distance.

initUltrasonic

Initialization logic for the ultrasonic sensor and TC1. Configures TC1's overflow interrupt as well as disabling its output pins. starts the timer in the 'off' (disabled) state. TC1 is run in normal mode.

turnoffTimer1

Helper function to provide a more readable mnemonic for disabling TC1. Works by assigning 0 to *TCCR1B*.

turnonTimer1

Helper function to provide a more readable mnemonic for enabling TC1. Works by assigning $0x05$ to *TCCR1B*.

triggerUltrasonic

Sends a $10\mu s$ pulse to the ultrasonic sensor to trigger a 'ping'. Waits $60ms$ after the ping is triggered as suggested in the device's documentation.

getOverflowStatus

'Getter' function to provide access to the TimerOverflow variable. This can be used to provide specialized behaviors in the event that TC1 overflows while measuring a distance.

receiveUltrasonic

Reads from TC1 and returns the distance in centimeters corresponding to the time count in TC1. The equation used for this is $(i * 64)/58$.

TIM16_ReadTCNT1

Function to read from TC1 as an atomic operation. Disables the global interrupt bit until the read is completed, stores the timer value into the 'i' local variable, re-enables interrupts, and finally returns 'i'.

TIM16_WriteTCNT1

Similar to the read operation, performs an atomic write by disabling the global interrupt bit. the unsigned integer parameter is written into the TCNT1 register, and then interrupts are re-enabled before leaving the function.

initMotor

All of our motor initializations with timer preparation and scaling, PWM modes and timer interrupt enable for turn functions.

setB

Sets the OCR0B PWM pin and one side of our motor, specifically the left side motors on our robot. Set speed and direction with included enumeration inside of this motor driver file.

setA

Sets the OCR0A PWM pin and one side of our motor, specifically the right side motors on our robot. Set speed and direction with included enumeration inside of this motor driver file.

turnLeft

Simple abstraction function, activates the proper setB and setA functions with speed and length of time to be inside the function then stops.

turnRight

Simple abstraction function, activates the proper setB and setA functions with speed and length of time to be inside the function then stops.

driveForward

Simple abstraction function, drives the wheels forward for a set amount of time, then stops.

driveBackward

Simple abstraction function, drives the whels backwards for a set amount of time, then stops.

enum WHEEL_DIRECTION

Enumeration to allow ease of programming and simple calls for our motor driver functions. Values are:

- forward
- back

ISR(TIMER0_COMPA_vect) (interrupt)

Increments the MAIC by one each time the interrupt is invoked. This allows a rough delay functionality with an accuracy based on the period of the interrupt. The program uses a 1024 prescaler, so the period of a single interrupt cycle is 16ms. We noticed possible errors with this timing method for the smallest prescaler values.

getNumInterruptsForDuration

Computes the number of PWM periods that need to pass before the specified time has passed, to the next full period. The targetCount global is set to this value and the MAIC (Motor A Interrupt Counter) variable is compared against it to determine when the delay is finished. MAIC is set to 0 at the end of the function to prevent the delay from being incorrect.

delayUntilTargetCount

After a call to getNumInterruptsForDuration, targetCount will be set and delayUntilTargetCount will spin wait until MAIC is greater than the targetCount. MAIC is incremented by the TIMER0_COMPA_vect interrupt so the loop will be escaped after the appropriate delay has gone by.

DELAY_COUNT

Macro function to simplify the computation for getNumInterruptsForDuration. Divides the requested speed by the PWM (interrupt) period and adds 1 if the period does not evenly divide the delay.

Discussion

The biggest challenge we faced with our approach to this lab was determining what was wrong with our pin change interrupts when they failed to get responses. We found after several experiments and some online research that overly long interrupt handling logic could cause us to miss the rising/falling edges of our sensor. We had written the handler in a general style so it could track the last state and handle changes to multiple pins, but this backfired by taking too much time. The second, and current, version assumed that only one pin would change, allowing us to simplify the logic significantly.

This lab provided a useful introduction to using timers and interrupts to provide semi-asynchronous sensor reading and control, which is useful for allowing the robot to move while it is reading in data. Otherwise, we would be required to have the robot stop, take measurements, and then move again.

Responses

1. In this lab we are using timer/counter 1 to generate a dedicated ”clock” for pulse measuring. The Wiring library provided with the Arduino IDE provides a function pulseIn() which uses a different timing method. Explain how this pulseIn() function works.

   In the native arduino `PulseIn()` the library is counting the pulse by having calculated the number of clock cycles in one spin of a while loop. This property is used to count the status between when the pin changed outputs. This of course depends on the number of clock cycles to remain the same in the processor's main frequency, and with some software engineering, can be adapted depending on which hardware the compiler is configured for. This is in contrast to our method of relying on the Atmega328p's internal timer 1 to count the clock cycles between high and low ends of the timer.

2. Timer/counter 1 is 16 bits, meaning TCNT1 is two bytes wide. With the ATMega328P's 8-bit data bus two separate reads must be completed to read TCNT1. In the space of these two reads, it is possible that TCNT1 changes state. How does the ATMega328P work around this potential data-integrity hazard?

   In the space between these reads the microprocessor has an 8 bit temporary register that handles the storage of the high byte while the low byte is being read. The assignment of the high byte then happens in the same clock cycle that the data is being written to the temp space. n the high byte read the arduinio will send the temp register directly to the register to be assigned. This is why it is advised to read from the lower register before the higher register when reading two from 16 bit registers.

Appendices

Table of contents:

Appendix A: main.c

```
#define F_CPU 16000000
#include "bit_macros.h"
#include "motor_driver.h"
#include "pcint.h"
#include "robotIo.h"
#include "ultrasonic.h"
#include <avr/interrupt.h>
#include <avr/io.h>
#include <util/delay.h>

/* stdout stream */
static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL, _FDEV_SETUP_WRITE);

void Init()
{
    // configure UART and I/O
    initUART();

    // configure pin change interrupts
    initPCINT();

    // configure ultrasonic range sensor
    initUltrasonic();

    // configure motors
    initMotor();

    // Enable global interrupts
    sei();
}

unsigned char motorSpeedMap(unsigned int USDistance) {
    unsigned int ans = ((USDistance - 3) * 255 / 27);
    return (ans > 255? 255 : ans);
}

int main()
{
    Init();
    while (1)
    {
        triggerUltrasonic();
        while (!responseAvailable)
        {
        }
        unsigned int dist = receiveUltrasonic();
        unsigned char motorSpeed = motorSpeedMap(dist);
        fprintf(&mystdout, "Sensor reading: %dcm , mapped value: %d\n",
            dist, motorSpeed);
        driveForward(motorSpeed);
    }
    return 1;
}
```

Appendix B: motor_driver.h

```
#ifndef _MOTOR_DRIVER_H_
#define _MOTOR_DRIVER_H_
#include "bit_macros.h"
#include "pin_map.h"
#include "util/delay.h"
#include <avr/interrupt.h>
#include <avr/io.h>

#define TURN_DELAY_CIRCLE 250
#define TURN_DELAY_SQUARE 500
#define DRIVE_DELAY 500

typedef enum WHEEL_DIRECTION
{
    FORWARD = 0,
    BACK
} wheelDirection;

void initMotor();
void setB(unsigned char speed, wheelDirection direction);
void setA(unsigned char speed, wheelDirection direction);
void stop();
void turnLeftTimed(unsigned char speed, int time_ms);
void turnRightTimed(unsigned char speed, int time_ms);
void driveFoward(unsigned char speed);
void driveBackward(unsigned char speed);
void driveForwardTimed(unsigned char speed, int time_ms);
void driveBackwardTimed(unsigned char speed, int time_ms);
void delayUntilTargetCount();
void getNumInterruptsForDuration(int duration_ms);

#endif
```

Appendix C: motor_driver.c

```
#include "motor_driver.h"

// motors use pins

/*
 * IN 1 & IN 2 & EN A
 *
 * IN 3 & IN 4 & EN B
 */

#define DELAY_COUNT(time, rate) time / rate + (time % rate == 0 ? 0 : 1)

// interrupt counter for motor A
volatile unsigned long MAIC;
unsigned long targetCount;

void setA(unsigned char speed, wheelDirection direction)
{
    switch (direction)
    {
    case FORWARD:
        setBit(PORTD, H_IN1);
        clearBit(PORTB, H_IN2);
        break;
    case BACK:
        setBit(PORTB, H_IN2);
        clearBit(PORTD, H_IN1);
        break;
    }
    OCR0A = speed;
}

void setB(unsigned char speed, wheelDirection direction)
{
    switch (direction)
    {
    case BACK:
        setBit(PORTB, H_IN3);
        clearBit(PORTB, H_IN4);
        break;
    case FORWARD:
        setBit(PORTB, H_IN4);
        clearBit(PORTB, H_IN3);
        break;
    }
    OCR0B = speed;
}

void initMotor()
{
    // Init port B for output
    DDRB = 0xFF;
    DDRD = 0xFF;
    PORTB = 0x00;
    PORTD = 0x00;

    // set both motors to start off
    OCR0A = 0;
    OCR0B = 0;

    // start TCNT0 at 0
    TCNT0 = 0x00;

    // Enable fast pwm mode for DC motor output
    TCCR0A = 0xA3;
    TCCR0B = 0x05; // 1024 prescaler
    // TCCR0B = 0x04; // 256 prescaler
    // TCCR0B = 0x03; // 64 prescaler
    // TCCR0B = 0x02; // 8 prescaler
    // TCCR0B = 0x01; // 1 prescaler

    // Enable counter match interrupt for counter A
    TIMSK0 = 0x02;
}

void turnLeftTimed(unsigned char speed, int time_ms)
{
    getNumInterruptsForDuration(time_ms);
    setB(speed, FORWARD);
    setA(speed, BACK);
    delayUntilTargetCount();
}

void turnRightTimed(unsigned char speed, int time_ms)
{
    getNumInterruptsForDuration(time_ms);
    setB(speed, BACK);
    setA(speed, FORWARD);
    delayUntilTargetCount();
}

void driveForwardTimed(unsigned char speed, int time_ms)
{
    getNumInterruptsForDuration(time_ms);
    setA(speed, FORWARD);
    setB(speed, FORWARD);
    delayUntilTargetCount();
}

void driveForward(unsigned char speed)
{
    setA(speed, FORWARD);
    setB(speed, FORWARD);
}

void driveBackward(unsigned char speed)
{
    setA(speed, BACK);
    setB(speed, BACK);
}

void driveBackwardTimed(unsigned char speed, int time_ms)
{
    getNumInterruptsForDuration(time_ms);
    setA(speed, BACK);
    setB(speed, BACK);
    delayUntilTargetCount();
}

void stop()
{
    setA(0, FORWARD);
    setB(0, FORWARD);
}

void delayUntilTargetCount()
{
    while (MAIC <= targetCount)
    {
    };
    MAIC = 0;
}

void getNumInterruptsForDuration(int duration_ms)
{
    int prescaler_choice = TCCR0B & 0x07;

    switch (prescaler_choice)
    {
    case 0: // no clock...no motor???
        // ERROR: without a clock a) the motors aren't running and
        // b) we can't wait a duration...
        targetCount = 0;
        break;
    case 1: // 1 prescaler, one interrupt ~ every 0.02ms
        targetCount = DELAY_COUNT(100 * duration_ms, 2);
        break;
    case 2: // 8 prescaler, one interrupt ~ every 0.1ms
        targetCount = DELAY_COUNT(10 * duration_ms, 1);
        break;
    case 3: // 64 prescaler, one interrupt ~ every 1ms
        targetCount = DELAY_COUNT(duration_ms, 1);
        break;
    case 4: // 256 prescaler, one interrupt ~ every 4ms
        targetCount = DELAY_COUNT(duration_ms, 4);
        break;
    case 5: // 1024 prescaler, one interrupt ~ every 16ms
        targetCount = DELAY_COUNT(duration_ms, 16);
        break;
    case 6:
    case 7:
        targetCount = 0;
        // ERROR: 6 and 7 are external clocks and we can't predict them
    }
    // reset the counter variable
    MAIC = 0;
}

ISR(TIMER0_COMPA_vect) { MAIC++; }
```

Appendix D: pcint.h

```
#ifndef _PCINT_H_
#define _PCINT_H_
#include "avr/io.h"
#include "robotIo.h"
#include "ultrasonic.h"
#include <avr/interrupt.h>
#include <stdbool.h>
#include <stdio.h>

#define PCMSK1_CONFIG                                                        \
    (0 << PCINT14) | (0 << PCINT13) | (1 << PCINT12) | (0 << PCINT11) |      \
        (0 << PCINT10) | (0 << PCINT9) | (0 << PCINT8)

#define PCICR_CONFIG (0 << PCIE2) | (1 << PCIE1) | (0 << PCIE0)

volatile unsigned int timeResponse;
volatile bool responseAvailable;

void initPCINT();

#endif
```

Appendix E: pcint.c

```
#include "pcint.h"

/*
 * stdout stream
 *
 * available for debugging but slows ISRs a lot.
 */
// static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL,
// _FDEV_SETUP_WRITE);

void initPCINT()
{
    PCMSK1 |= PCMSK1_CONFIG;
    PCICR |= PCICR_CONFIG;
}

// specific pin change interrupt handler that is less general, but
// should be faster so less time is spent in the interrupt and
// measurements are more accurate.
//
// Expects only PCINT12 active (the ultrasonic sensor input)
ISR(PCINT1_vect)
{
    // used to detect whether pcint is going high->low or low->high
    // a value of false indicates we are low->high, a value of true indicates
    // high->low.
    static bool highEdge = false;

    if (highEdge)
    {
        turnoffTimer1();
        timeResponse = receiveUltrasonic();
        responseAvailable = true;
    }
    else
    {
        TIM16_WriteTCNT1(0);
        turnonTimer1();
        responseAvailable = false;
    }

    highEdge = !highEdge;
}

/*
// Pin change interrupt handler for pins 14-8
// written with a general switch design to
// handle cases where we are listening to more
// than one pin
ISR(PCINT1_vect)
{
    // pull current state of PORTC first thing
    unsigned char currState = PINC;

    // last state the pins were in
    static unsigned char lastState = 0x00;
    // used to detect whether pcint is going high->low or low->high
    // a value of 0 indicates we are low->high, a value of 1 indicates high->low
    static bool highEdge = false;
    // changed pins
    unsigned char pcflags = currState ^ lastState;
    lastState = currState;

    // to add handling for a specific set of pin changes, add a case
    // to the switch statement with the bits corresponding to a pin combination
    // set. Note that the specific combination must be available as a case.
    switch (pcflags)
    {
    case 0x00:
        // nothing happened, do nothing
        break;

    case 0x10: // PORTC4 (PCINT12) changed
        if (highEdge)
        {
            turnoffTimer1();
            timeResponse = receiveUltrasonic();
            responseAvailable = true;
        }
        else
        {
            TIM16_WriteTCNT1(0);
            turnonTimer1();
            responseAvailable = false;
        }
        highEdge = !highEdge;

        break;
    case 0x20: // PORTC4 (PCINT13) changed
```

```
        break;
    default:
        // a case we weren't expecting occurred, do nothing
        break;
    }
}
*/
```

Appendix F: pin_map.h

```
#ifndef PIN_DEFS_H
#define PIN_DEFS_H
/*
>>> These are the ARDUINO pin mappings. <<<
#define US_RECV   12
#define US_ECHO   A4
#define US_TRIG   A5

#define H_A_EN 5
#define H_B_EN 6
#define H_IN1  7
#define H_IN2  8
#define H_IN3  9
#define H_IN4  11

#define IR_RECV 12

#define LED     13

#define LINE_R 10
#define LINE_M 4
#define LINE_L 2
*/

//<<< These are bit offsets for each pin, usable for PIN, DDR, PORT. >>>
#define US_RECV    4   // PORTB
#define US_ECHO    4   // PORTC
#define US_TRIG    5   // PORTC

// H bridge pins
#define H_A_EN     5   // PORTD PWM pin
#define H_B_EN     6   // PORTD PWM pin
#define H_IN1      7   // PORTD
#define H_IN2      0   // PORTB
#define H_IN3      1   // PORTB
#define H_IN4      3   // PORTB

#define IR_RECV    6   // PORTB

#define LED        5   // PORTB

#define LINE_R     2   // PORTB
#define LINE_M     4   // PORTD
#define LINE_L     2   // PORTD


#endif
```

Appendix G: robotio.h

```
#ifndef _ROBOT_IO_H_
#define _ROBOT_IO_H_
#include "avr/io.h"
#include <stdio.h>

/* baudrate configuration */
#define USART_BAUDRATE 9600
/* baudrate prescaler configuration */
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)

/* configurations for the UART device register B*/
#define UCSR0B_CONFIG                                                          \
    (0 << RXCIE0) | (0 << TXCIE0) | (0 << UDRIE0) | (1 << RXEN0) |             \
        (1 << TXEN0) | (0 << UCSZ02) | (0 << RXB80) | (0 << TXB80);
/* configurations for the UART device register C*/
#define UCSR0C_CONFIG                                                          \
    (0 << UMSEL01) | (0 << UMSEL00) | (0 << UPM01) | (0 << UPM00) |            \
        (0 << USBS0) | (1 << UCSZ01) | (1 << UCSZ00) | (0 << UCPOL0);

/* handles inserting characters into the output stream */
int uart_putchar(char c, FILE* stream);
/* initializes UART communications using the defines above*/
void initUART();

#endif
```

Appendix H: robotio.c

```
#include "robotIo.h"

void initUART()
{
    // init uart
    // UCSR0B |= 0x98;
    // UCSR0C |= 0x06;
    UCSR0B |= UCSR0B_CONFIG;
    UCSR0C |= UCSR0C_CONFIG;

    UBRR0L = BAUD_PRESCALE;
    UBRR0H = (BAUD_PRESCALE >> 8);
}

int uart_putchar(char c, FILE* stream)
{
    if (c == '\n')
        uart_putchar('\r', stream);
    loop_until_bit_is_set(UCSR0A, UDRE0);
    UDR0 = c;
    return 0;
}
```

Appendix I: ultasonic.h

```
#ifndef _ULTRASONIC_H_
#define _ULTRASONIC_H_
#include "bit_macros.h"
#include "pin_map.h"
#include <avr/interrupt.h>
#include <avr/io.h>
#include <stdbool.h>
#include <util/delay.h>

bool getOverflowStatus();
void initUltrasonic();
void triggerUltrasonic();
void turnoffTimer1();
void turnonTimer1();
void TIM16_WriteTCNT1(unsigned int i);
unsigned int TIM16_ReadTCNT1();
unsigned int receiveUltrasonic();

#endif
```

Appendix J: ultasonic.c

```
#include "ultrasonic.h"

/*
 * Flag for overflow:
 * false = no overflow.
 * true = overflow happened in test
 */
volatile bool TimerOverflow = false;

/*
 *
 * Ultrasonic sensor: A4 A5 (portc 4 5)
 * Servo Motor: 3 (portd 3)
 */

/*
 * Sets the pins for ultrasonic sensor,
 * then will setup the necessary timer 1
 * to properly time the sensor
 */
void initUltrasonic()
{
    // Trig on A5
    setBit(DDRC, US_TRIG);
    // Echo on A4
    clearBit(DDRC, US_ECHO);

    PORTC = 0x00;

    // setBit(PORTC, US_ECHO); // pull-up for echo

    /*
     * Timer 1
     * Pins 7:4 zeroes
     * tccr1a high byte zeroes
     * 16 us
     */
    OCR1A = 0;
    OCR1B = 0;
    // Start tcount at 1
    TCNT1 = 0x00;
    // Normal mode
    TCCR1A = 0x00;

    // ensure we start with the timer off
    turnoffTimer1();
}

// convenience function to turn the timer off
void turnoffTimer1() { TCCR1B &= 0x00; }

// convenience function to turn the timer back on with a default prescaler
void turnonTimer1()
{
    /*
     * Prescaler value goes here
     * 1 - no prescaling
     * 2 - clock /8
     * 3 - clock /64
     * 4 - clock /256 - 16us
     * 5 - clock /1024 - 64us
     * source -
     * https://sites.google.com/site/qeewiki/books/avr-guide/timers-on-the-atmega328
     */
    TCCR1B |= 0x05;
}

// Triggers ultrasonic sensor, then waits 60 ms
void triggerUltrasonic()
{
    TimerOverflow = false;
    setBit(PORTC, US_TRIG);
    _delay_us(10);
    clearBit(PORTC, US_TRIG);
    // reset counter 1
    // TIM16_WriteTCNT1(0);
    // Delay while pulse is sent
    _delay_ms(60);
}

// Getter for overflow status
bool getOverflowStatus() { return TimerOverflow; }

// Overflow vector
ISR(TIMER1_OVF_vect)
{
    // Timer 1 overflow
    TimerOverflow = true;
}
```

```
/*
 * Gets the current status of timer 1, then
 * converts it into clock/cm
 */
unsigned int receiveUltrasonic()
{
    unsigned int i = TIM16_ReadTCNT1();
    // 64 us per count in i
    // return ((58 * 64) / i);
    return (i * 64) / 58;
}

// Reads from timer 1 counter
unsigned int TIM16_ReadTCNT1()
{
    unsigned char sreg;
    unsigned int i;
    // Save global interrupt flag
    sreg = SREG;
    // Disable interrupts
    cli();
    // Read TCNT1 into i
    i = TCNT1;
    // Restore global interrupt flag
    SREG = sreg;
    return i;
}

// Sets timer 1 counter
void TIM16_WriteTCNT1(unsigned int i)
{
    unsigned char sreg;
    // Save global interrupt flag
    sreg = SREG;
    // Disable interrupts
    cli();
    // Set TCNT1  to i
    TCNT1 = i;
    // Restore global interrupt flag
    SREG = sreg;
}
```