Laboratory Report

To: Dr.Randy Hoover

From: Benjamin LeBrun, Benjamin Garcia

Subject: Lab Assignment 5: Timers and Motor Control

Date: April 5, 2019

Introduction

For this lab, we utilized the full car kit of our Elegoo robot package to explore using power width modulation (PWM) to send a controlled motor signal to an H-bridge chip and drive a set of DC motors with two separate channels. This required the use of writing to the Atmega328p's internal timers to prevent overloading or locking the CPU with delay signals or other inefficient methods.

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

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 components in the kit have a clearly marked port on the Arduino shield included in the package. This makes it easier to identify and connect together ports using the also included ribbon cables with some degree of cable management. One of the early pitfalls is that the H bridge connector is a six pin ribbon cable, which to the uncareful eye looks like the five pin cable connector between the shield board and the line following IR sensors.

Besides the ribbon cable for our ports, we also have a DC cable to power the H bridge driver. This is because the DC motors have massively larger power requirements (up to a maximum of 2 amps) than our Arduino (40 milliamps maximum, both at 5 volts). Therefore, the external batteries are required to power the DC motors via the H bridge separate from the Arduino. Wheels were also attached and secured with standard phillips head screws.

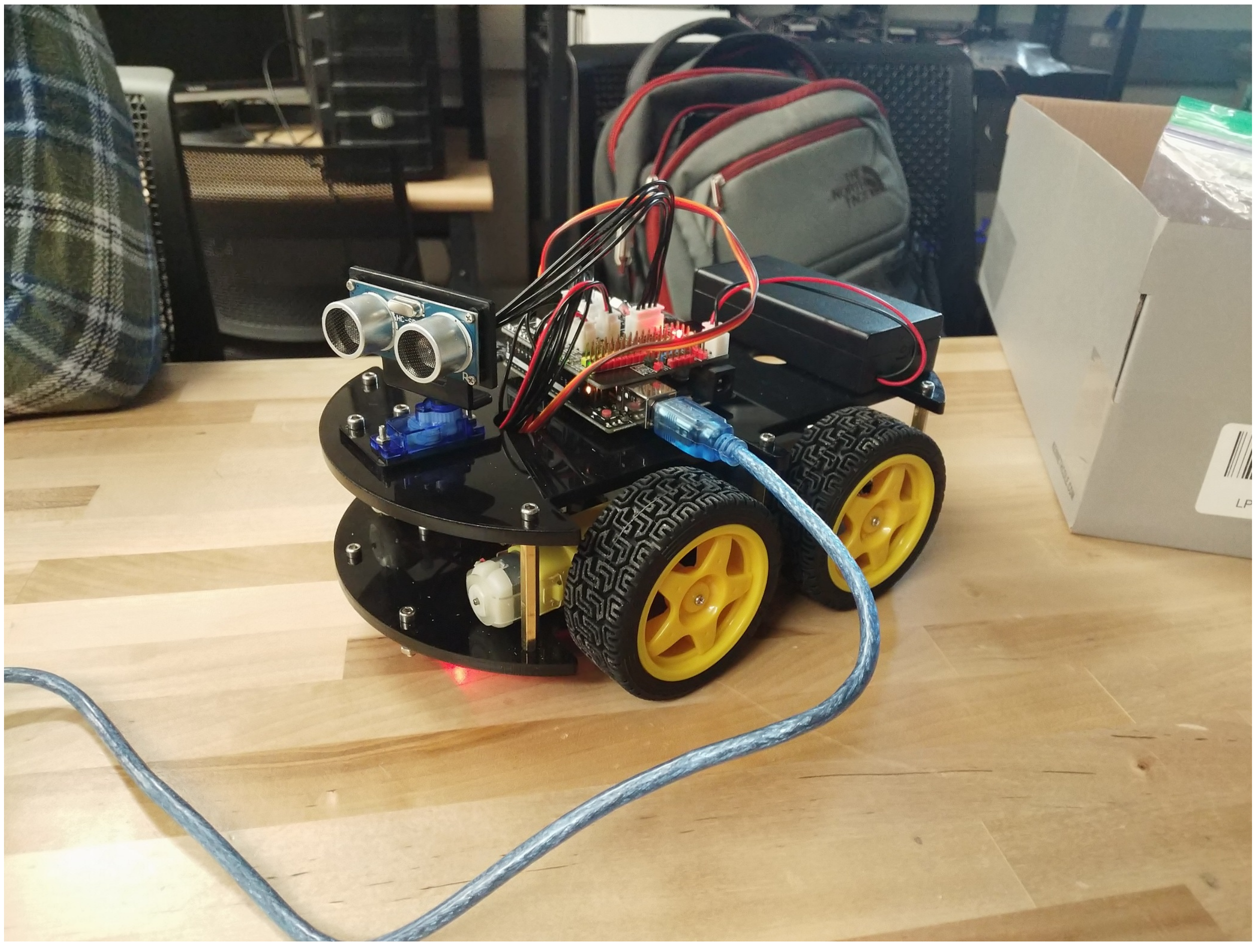

Figure 1: Fully assembled Elegoo robot car kit V3.0.

Implementation

For this lab we are writing our own motor driver header and library which should allow us to use in the future. Because it also uses a pin map header file, we can also move the motor headers around when needed.

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

Timers and motor drivers are finicky devices, especially on platforms with some degree of inaccuracy. By their inherent design DC motors are not entirely accurate but made to be fast and provide continuous drive. One of our first issues was in driving the motors and accounting for issues with the quality of the build, especially power distribution and toe, camber, and alignment of the wheels on their motors. Some degree of physical manipulation was performed but we ultimately took to software solutions to bias the wheels correctly by adjusting the PWM value written to the output compare registers.

Another issue was using the correct data type. On initial sketch we used integers, however, we later recalled that these are in fact 16 bit values being assigned to 8 bit accurate registers. To prevent overflow, we changed these to 8 bit c-character variables immediately.

This lab is fairly straightfoward, using the OCR0X registers directly to write a relative speed between 0-255 in fast PWM mode on a 1024 prescale, we can more or less come to approximate to the easy mode Arduino IDE sketch of setting the digital pins PWM of 0-255. Then it's a matter of writing corresponding functions to make easier controlling the speed and direction to the motors.

Responses

1. PWM frequency for Timer 0 is controlled by the clock-select bits in TCCR0B. Test all five PWM output frequencies while holding OCR0n static. What effects do you see on motor speed/torque? Drawing on your knowledge from Circuits II, what is happening?

   As we increase the prescaler, the motors start to whine to even higher pitches as we increase the scale of the clock until what we suspect is a barely audible noise at 16MHz which is the absolute highest clock. In addition, as we went up through the pre-scalers, there was a noticable loss of torque as the prescale value was increased. At a prescale value of 1, no prescale, we suspect there's some resolution or overflow happening, as the motors remained on full power as it ran through the lab test program.

   The internal of the motor is a coil of wire around a magnet which is attempting to either attract itself or deflect itself against the poles of the magnet and the poles of the magnetic inductive force around the driving coils. When our prescale value is too high for this reaction to take place and affects the total duty cycle available to the DC motor. In the case of the extremely high frequencies to our motors, the full duty cycle is not being executed.

2. Watchdog timers are a critical part of robust embedded systems. How do watchdog timers function? Does the ATMega328P include a watchdog timer? If so, from where does it pull its clock signal? To what value would one set WDTCSR if they wanted to reset the system after one second of inactivity?

A watchdog timer is a timer that, when hitting a timeout, will attempt to reset the entire system. This is generally used to check for faults or large values of other timers and systems to prevent hanging of the entire system. It's generally good practice to reset or refresh the watchdog timer value at the end of certain processes to ensure it doesn't erroneously reset the entire system.

The ATMega328P does have a watchdog timer which pulls from it's own independent clock generator. One would set the last four bits of the WDTCSR register to 0110 for a one second watchdog timer.

Appendices

Table of contents:

- main.c - entry, initialization and drive instructions
- bit_macros.h - bit manipulations macros
- motor_driver.h - header file for main motor driver
- motor_driver.c - main c file for motor driver
- pin_map.h - map of our motor pins

Appendix A: main.c

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

// interrupt counter for motor A
volatile unsigned long MAIC;
// number of interrupts to stop driving after
unsigned long targetCount;

void Init()
{
    // setup motor controller PWM
    initMotor();
    // Enable global interrupts
    sei();
}

void testSquare()
{
    for (int i = 0; i < 3; i++)
    {
        driveForward(128, 750);
        stop();
        turnLeft(128, 500);
        stop();
    }
    driveForward(128, 750);
}

void testCircle()
{
    for (int i = 0; i < 100; i++)
    {
        driveForward(64, 200);
        turnLeft(64, 200);
    }
}

int main()
{
    // int i = 0;
    Init();
    while (1)
    {
        // requirement 2: drive forward at full-speed for 2 seconds
        driveForward(255, 2000);
        _delay_ms(500);
        // requirement 3: drive backwards at half-speed for 4 seconds
        driveBackward(128, 4000);
        _delay_ms(500);
        // requirement 4: turn in place counter-clockwise at full-speed 2
        // seconds
        turnLeft(255, 2000);
        _delay_ms(500);
        // requirement 5: turn in place clockwise at half-speed for 4 seconds
        turnRight(128, 4000);
        _delay_ms(500);
        // requirement 6: approximate a square path
        testSquare();
        _delay_ms(500);
        // requirement 7: approximate a circular path
        testCircle();
        _delay_ms(500);
    }
    return 1;
}

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

Appendix B: bit_macros.h

```
#ifndef _BIT_MACROS_H_
#define _BIT_MACROS_H

#define bitVal(bit) (1 << bit)
#define setBit(byte, bit) ((byte) |=(bitVal(bit)))
#define clearBit(byte, bit) ((byte) &= ~(bitVal(bit)))
#define togBit(byte, bit) ((byte) ^= (bitVal(bit)))

#endif
```

Appendix C: 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/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 turnLeft(unsigned char speed, int time_ms);
void turnRight(unsigned char speed, int time_ms);
void driveForward(unsigned char speed, int time_ms);
void driveBackward(unsigned char speed, int time_ms);
void delayUntilTargetCount();
void getNumInterruptsForDuration(int duration_ms);

#endif
```

Appendix D: 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 extern unsigned long MAIC;
extern 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 turnLeft(unsigned char speed, int time_ms)
{
    getNumInterruptsForDuration(time_ms);
    setB(speed, FORWARD);
    setA(speed, BACK);
    delayUntilTargetCount();
}

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

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

void driveBackward(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;
}
```

Appendix E: 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

#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
```