

6.115 Laboratory Report #4

David Lee

March 30, 2005

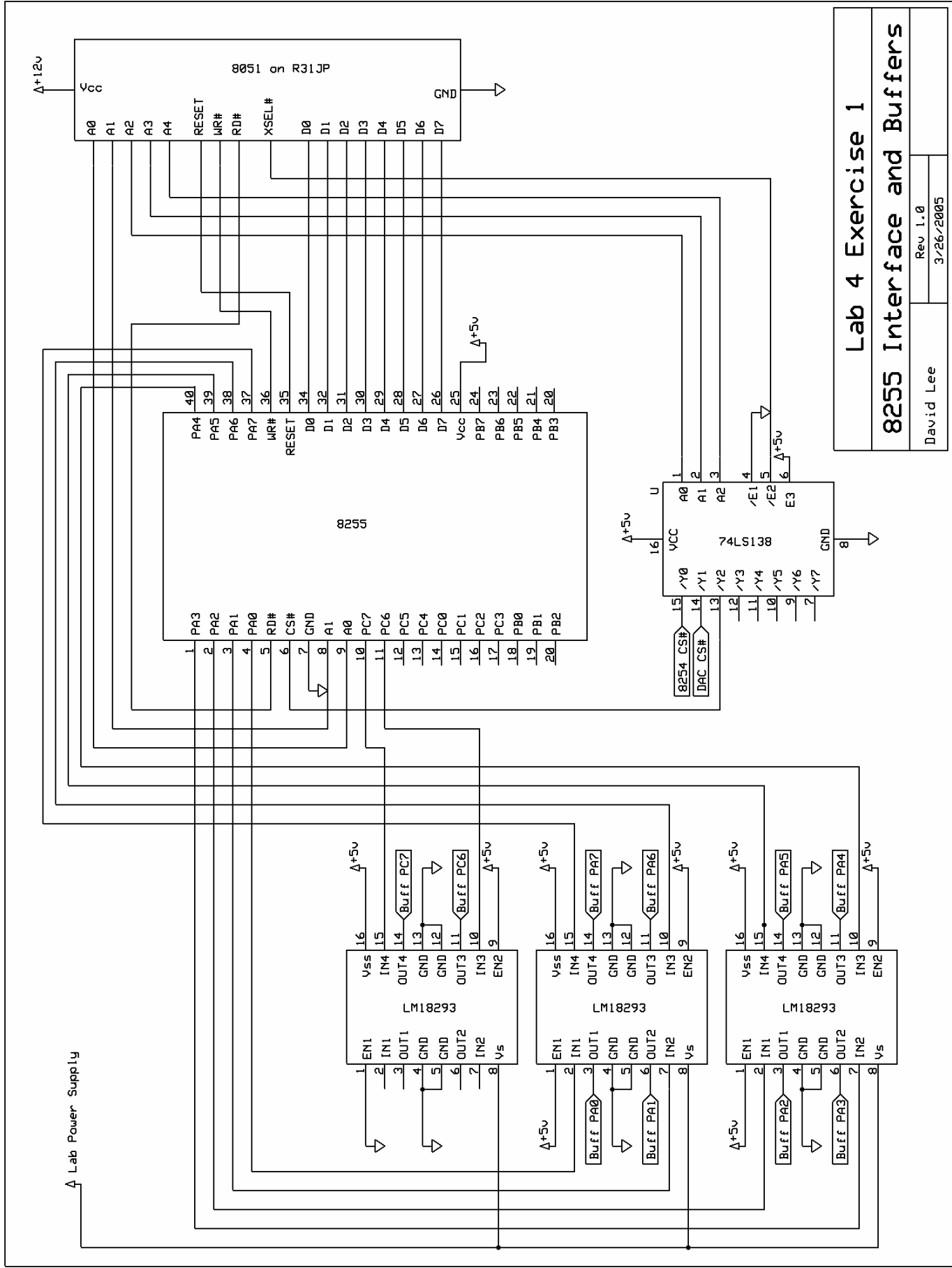
The goals of this laboratory were to connect more peripherals to the R31JP, understand, energize, and run a collection of electromagnetic actuators, use an A/D converter, explore image processing, and practice using the PIC16F628 microcontroller with C programming language. The first part of the lab involved adding an 8255 parallel interface chip to the R31JP, and buffering 10 of its output lines. The second part involved experimenting with relays and DC motors. The third part involved generating PWM waveforms to control such devices without analog conversion. The fourth part used a PWM to implement motor controlling a robot arm. The fifth part of the lab dealt with SpinDude, a stepper motor/image mapping device, and the final part involved compiling/burning another C program onto the PIC microcontroller.

Exercise 1:

In Exercise 1, we are tasked with adding an 8255 parallel interface chip to the R31JP. I memory-mapped the chip to address locations FE08h – FE0B, with 3 byte-wide ports available for input/output. Of these, I buffered 10 lines using L293 chips, using 10 separate buffers across 3 chips powered by my laboratory power supply. These lines took up all the port A pins and pins 6 and 7 of port C. These could be easily configured as output lines by writing the control word 80h to memory location FE0Bh. Writing to FE08h outputs to port A, and writing to FE0Ah outputs to port C.

Testing these pins by writing to memory locations FE08h and FE0Ah showed that the 8255 and L293 chips worked correctly. The DAC and 8254 also continued to work when accessed with the correct memory-mapped address.

My schematic for wiring the 8255 and L293 chips are as follows:



Lab 4 Exercise 1

8255 Interface and Buffers

David Lee

Rev 1.0

3/26/2005

Exercise 2:

In Exercise 2, we test out an electromagnetic relay. Upon inspection, I reasoned that the relay was a three-contact, SPDT relay. Powering the solenoid portion of the relay with my lab power supply, I found that, when starting from 0 V and increasing the voltage towards 12 volts, the relay switched to its “on” position at 6 V. With the relay on and decreasing back from 12 volts to 0, the relay switched back “off” at 3.5 V. This is because the solenoid must overcome the force of the spring to switch into its “on” position, but has less force to overcome when switching back to its neutral “off” position.

I then configured the relay to be controlled by the buffered drive pin from the 8255. There were two ways to do this: one with the relay across buffered output (from the L293) and ground, and one with the relay across the buffered output and its high voltage (in this case, 12 V). A 1N4001 diode was absolutely essential across the solenoid, because once the buffered output goes from high to low there will be a voltage across the charged solenoid that will seek to dissipate in any way possible. Without a diode, this would mean short-circuiting across the buffer, destroying the L293 chip. The diode safely prevents such short circuits from 12V to 0.

The following program drove the relay on or off upon a key press.

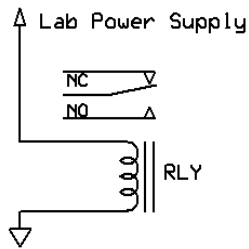
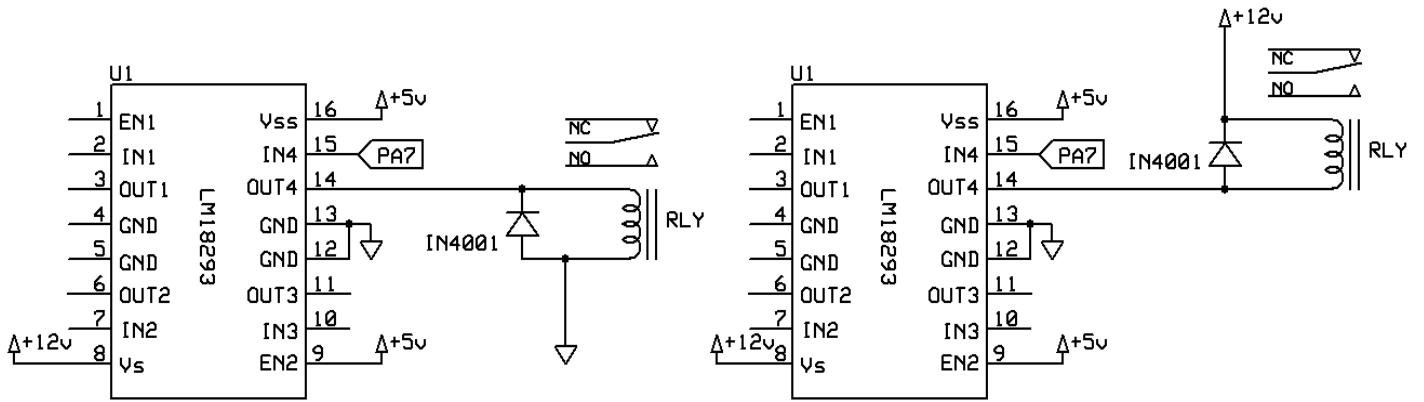
```
mov tmod, #20h      ; set timer 1 for auto reload - mode 2
mov tcon, #40h      ; run counter 1
mov th1, #0FDh      ; set 9600 baud with xtal=11.059MHz
mov scon, #50h      ; set serial control reg for 8 bit data
                   ; and mode 1

mov dptr, #0fe0bh   ; set control word register on 8255
mov a, #80h         ; for regular I/O on all ports
movx @dptr, a
mov dptr, #0fe08h   ; we will target port a, bit 7
mov a, #0

loop:
  jnb ri, loop      ; wait for key press
  cpl acc.7         ; toggle the bit
  movx @dptr, a     ; send to the 8255
  clr ri           ; clear serial "receive status" flag
  sjmp loop
```

Driving the solenoid between the buffer and ground, when the buffered output was high, the relay turned on, and a low signal turned it off. Switching the order of the relay pins did not affect this behavior. When driving the solenoid between the buffer and 12V, however, this behavior was reversed, with high turning it off and low turning it on. Again, reversing the pins did not affect the behavior.

The following are full schematics and some waveforms on the 8255 drive pin:



Lab 4 Exercise 2		
Relay Configurations		
David Lee	Rev 1.0	
	3/26/2005	

The 8255 drive pin waveforms for either relay configuration are the same. 8255 drive pin going low to high:

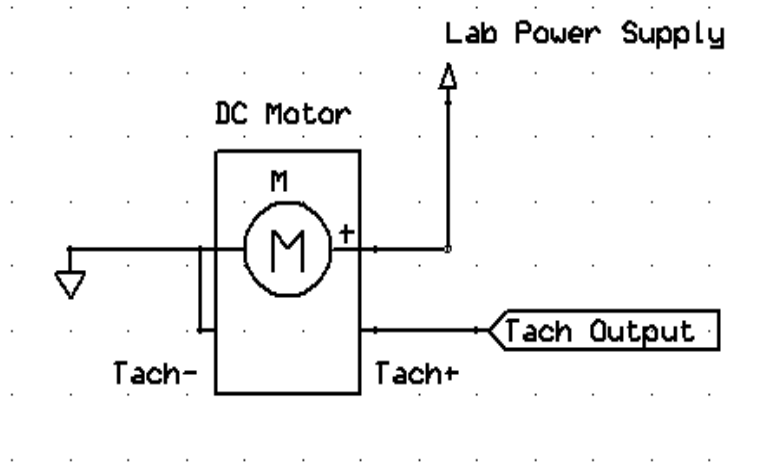


8255 drive pin going high to low:



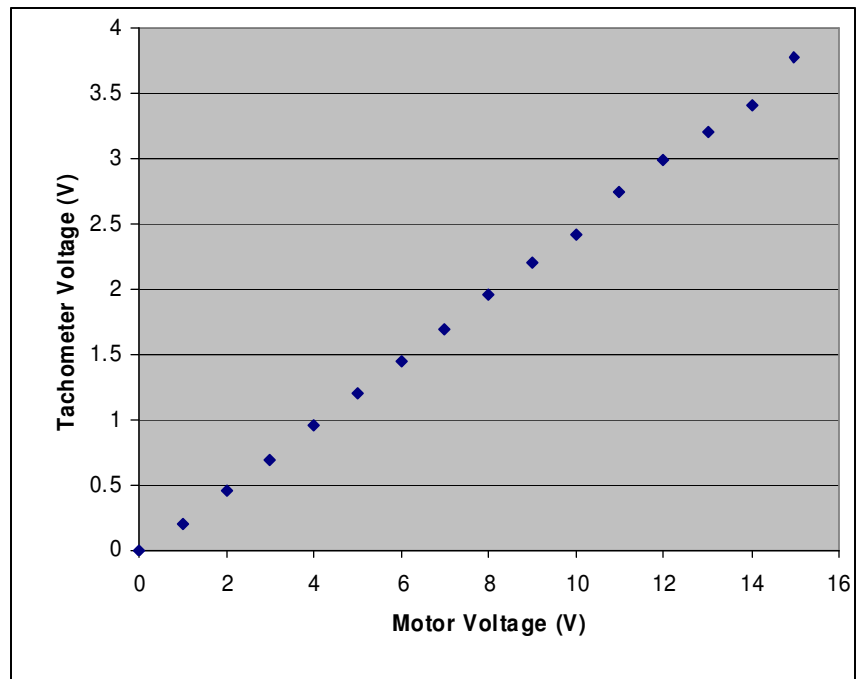
Exercise 3:

In Exercise 3, we run a motor with our laboratory power supply, as shown:



With the voltage starting at 0V and current limit as high as possible, I increased the voltage across the motor by 1V steps until reaching 15V. At each step, I recorded the tachometer voltage. Here are the readings and a plot versus motor voltage:

Motor Voltage (V)	Tachometer Voltage (V)
0	0
1	0.2
2	0.46
3	0.69
4	0.96
5	1.2
6	1.45
7	1.69
8	1.96
9	2.2
10	2.42
11	2.74
12	2.99
13	3.2
14	3.41
15	3.77



This plot makes sense to me. The DC motor's speed is proportional to its drive voltage, and the tachometer's voltage is proportional to its driving speed. Since some work is done driving the physical motor, the generated voltage on the tachometer will be proportionally less than the motor voltage.

When I grabbed the shaft of the motor as it ran on 15V, **the current into the motor jumped up as it tried to adjust to the increased load, before settling back down to a current higher than its earlier steady state.** Then, I set the current limit to its steady state voltage at 15V, and grabbed the shaft of the motor. **This time, the motor was much easier to stop, the supply signaled overload (keeping the current output from increasing), and the tachometer output voltage dropped to zero.** This makes sense because the motor cannot draw more current to counteract the increased load, and stops without a struggle.

From my observations, I would conclude that terminal voltage proportionally affects the steady state speed of the motor; in particular, **it is proportional to the rotations per second.** Terminal current, however, influences the **torque** on the motor, or how much load it can handle at that speed.

Exercise 4:

In Exercise 4, we generate PWM waveforms to approximate an analog voltage drive. The first task was to generate a 12V PWM waveform with duty cycle 50% and switch period of two seconds. This program is as follows:

```
; the low frequency pwm waveform, 2 second switch period

org 0h
ljmp main
org 0bh
ljmp t0isr

org 030h
main:
  mov dptr, #0fe0bh      ; set control word register on 8255
  mov a, #80h           ; for regular I/O on all ports
  movx @dptr, a
  mov dptr, #0fe08h     ; we will target port a, bit 7
  mov a, #0

  mov r7, #20
  mov tmod, #01h        ; run timer 0 in mode 1, with
                        ; 16-bit count
  mov ie, #82h          ; enable timer 0 interrupt
  setb tr0
loop:
  sjmp loop

t0isr:
  clr tr0
  djnz r7, skip
  cpl acc.7             ; complement the bit
  movx @dptr, a        ; send to the 8255
  mov r7, #20          ; 20 * .05 = 1 s
  ljmp exit
```

```

skip:
  mov th0, #04ch ; count out .05 s
  mov tl0, #000h
exit:
  setb tr0
  reti

```

The next task was to generate a PWM waveform with duty cycle 50% and switch period .2 milliseconds. The program is as follows:

```

; the high frequency pwm waveform, .2 ms switch period

org 0h
ljmp main
org 0bh
ljmp t0isr

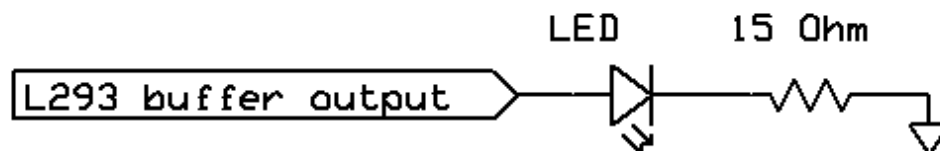
org 030h
main:
  mov dptr, #0fe0bh ; set control word register on 8255
  mov a, #80h ; for regular I/O on all ports
  movx @dptr, a
  mov dptr, #0fe08h ; we will target port a, bit 7
  mov a, #0

  mov tmod, #02h ; run timer 0 in mode 2, with
  ; 8-bit auto-reload
  mov th0, #0a5h ; count out to .2 ms
  mov ie, #82h ; enable timer 0 interrupt
  setb tr0
loop:
  sjmp loop

t0isr:
  cpl acc.7 ; toggle the bit
  movx @dptr, a ; send to the 8255
  reti

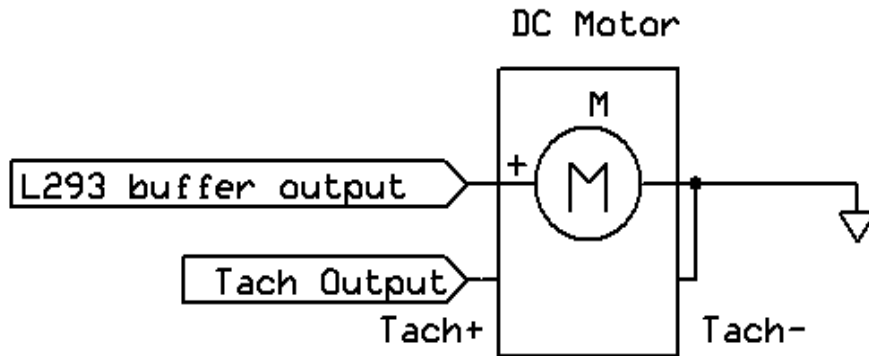
```

Running the L293 with the PA7 drive pin (from the 8255) and a 12V laboratory power supply, I powered an LED and current-limiting resistor with the buffered outputs of these two PWM waveforms.

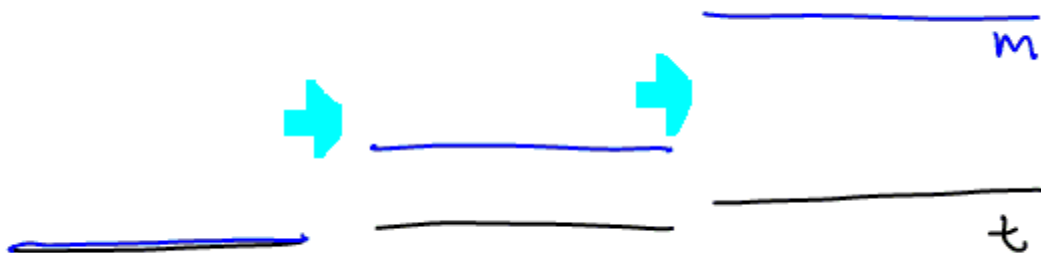


Using the low-frequency PWM, the LED turns on and off at regular intervals (half the switch period). This is because the two voltage levels that the PWM alternates are held too long to be averaged, and the low-pass filter has little effect on this waveform since it is already at a low frequency. **Using the high-frequency PWM, the LED shines constantly, although not at its brightest.** This is because it is responding to the average voltage (half the high voltage), and the low-pass filter is ignoring the high frequency component of the wave. **In this system, the diode-resistor combination act as the low-pass filter.**

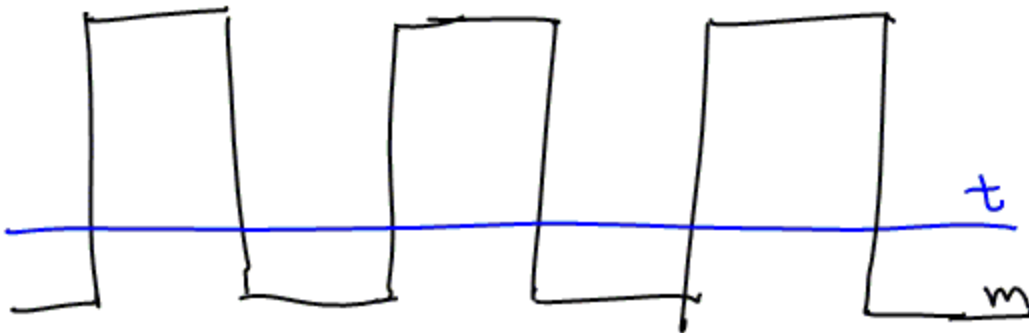
Next, I drove the motor/tachometer combination from Exercise 3 using these PWM waveforms.



For the low frequency waveform, the tachometer and motor voltages looked like this:



For the high frequency waveform, they looked like this:



As in the LED example, the motor turned on and off at regular intervals for the slow PWM waveform, and ran at constant speed for the fast one. This is because the motor is filtering out the high frequency modulation and acting on the average voltage, while the low frequency wave passes through. **The inductor inside the motor acts as its low-pass filter.**

For the high frequency program, the average tachometer speed reading was 1.24 V. This drive voltage approximately corresponds to a 5V motor drive voltage in Exercise 3. The average voltage applied to the motor (as read from the oscilloscope) was 5.25V, which is pretty close to the measured constant value.

Exercise 5:

In Exercise 5, we implement variable PWM waveforms using three different programming schemes.

Scheme 1

The first scheme used timer 0 to generate a high-frequency interrupt. A switch period consisted of 16 interrupts. A number of these interrupts drove the pin high, acting as the duty cycle, while the rest were low. The ratio of “high interrupts” to the total 16 was the duty cycle of the PWM waveform. Here is the program implementing this:

```
; pwm scheme 1

org 0h
ljmp main
org 0bh
ljmp t0isr

org 030h
main:
  mov dptr, #0fe0bh      ; set control word register on 8255
  mov a, #80h           ; for regular I/O on all ports
  movx @dptr, a
  mov dptr, #0fe08h    ; we will target port a, bit 7
  mov a, #80h          ; start high

  mov r7, #16          ; r7 should count from 15-0, repeat

  mov tmod, #02h       ; run timer 0 in mode 2, with
                       ; 8-bit auto-reload
  mov th0, #0e0h       ; use a high frequency
  mov ie, #82h         ; enable timer 0 interrupt
  setb tr0
loop:
  sjmp loop

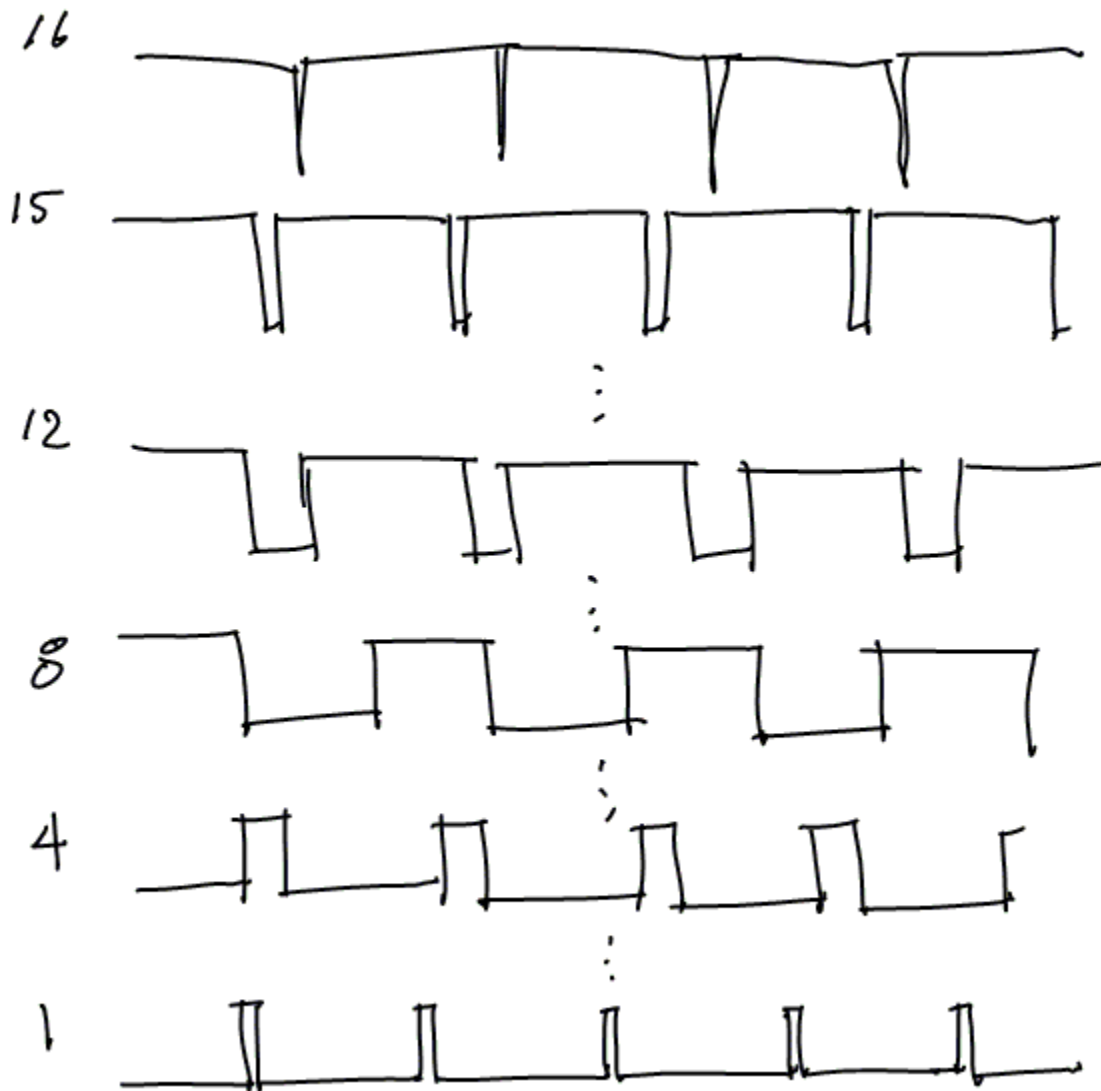
t0isr:
  djnz r7, notzero
  clr acc.7            ; set bit low
  movx @dptr, a       ; send to the 8255
  mov r7, #16
notzero:
  cjne r7, #4, exit   ; if we hit DT, go high
                       ; the absolute in this command is the duty cycle,
```

```

setb acc.7          ; i.e. 4/16 = 25%
movx @dptr, a      ; set bit high
exit:              ; send to the 8255
reti

```

This scheme, when driving the LED, produces brighter light for higher duty cycles and dimmer light for lower duty cycles. The waveforms generated look like the following:



The switch frequency is controlled by the timer 0 count. It is determined by the following equation:

$$\text{switch frequency} = \frac{11.0592 \text{ MHz}}{(256 - \text{th0}) * 12 * 16}$$

th0 = 224 switch frequency = **1.8 kHz**

The duty cycle is determined by the absolute value in the `cjne` instruction in the program, which ranges from 1-16, divided by 16. All duty cycles from 1/16 to 1 are accessible this way. 0 is inaccessible in this case because of the way I programmed the implementation, but can be made accessible through different methods.

Scheme 2

The second scheme uses timer 0 again, but alters the timer count each time it interrupts. Each time, it also complements the output bit, such that we can alter the explicit amount of time the bit goes high or low. I had to choose my timing values carefully such that the total switch frequency remained constant. Here is the program implementing this:

```

; pwm scheme 2
; note: make sure appropriate values are in the program space
; high->1000h and low->1001h

org 0h
ljmp main
org 0bh
ljmp t0isr

org 030h
main:
  mov dptr, #1000h      ; load "high" fraction count
  movx a, @dptr
  mov 40h, a
  mov dptr, #1001h     ; load "low" fraction count
  movx a, @dptr
  mov 41h, a

  mov dptr, #0fe0bh    ; set control word register on 8255
  mov a, #80h          ; for regular I/O on all ports
  movx @dptr, a
  mov dptr, #0fe08h    ; we will target port a, bit 7
  mov a, #80h          ; start high

  mov tmod, #02h       ; run timer 0 in mode 2, with
                       ; 8-bit auto-reload
  mov th0, 40h         ; load "high" fraction first
  mov ie, #82h         ; enable timer 0 interrupt
  setb tr0
loop:
  sjmp loop

t0isr:
  jb acc.7, golow      ; if we're high, should load "low" fraction
                       ; otherwise, load the high one
  mov th0, 40h         ; load "high" fraction
  setb acc.7           ; set bit high
  movx @dptr, a        ; send to the 8255
  sjmp exit
golow:
  mov th0, 41h         ; load "low" fraction
  clr acc.7            ; set bit low
  movx @dptr, a        ; send to the 8255
exit:
  reti

```

Scheme 2 requires that the two counts be placed in external memory using MINMON before it can create the PWM. The different duty cycles from D=0 to D=1 are best approximated using the following count values:

D	low fraction	high fraction
0	255	192
1/8	248	200
1/4	240	208
3/8	232	216
1/2	224	224
5/8	216	232
3/4	208	240
7/8	200	248
1	192	255

The switch period is dependant on both fractions, which should total to the same value in most cases. The equation for determining the switch period is as follows:

$$\text{switch period} = \frac{12 * (512 - \text{high fraction} - \text{low fraction})}{11.0592 \text{ MHz}}$$

In most cases, switch period = 69.4 microseconds

The counter should count out 64 counts each time, and a number of these counts will pull the waveform high. The duty cycle is determined by the following equation:

$$\text{Duty cycle} = 1 - \frac{(256 - \text{high fraction})}{64}$$

However, at D=0 and D=1 this scheme is rather inaccurate, because the minimum amount of instructions that must always be performed during the timer interrupt keep the low or high portion of the PWM waveform from ever reaching zero. The scheme is most accurate at the intermediate duty cycles. At these cycles, the behavior of scheme 2 is comparable to scheme 1 in terms of lighting the LED, with higher duty cycles creating brighter light.

Scheme 3

The third scheme uses the 8254 timers. One timer operates in mode 2 as the rate generator, pulsing during each switch period. The other timer is activated by this pulse, pulling the output pin high for the duty cycle period. Thus, the counter of the second timer divided by the counter of the first is our duty cycle. The schematic for wiring this is on a later page. The PWM can be implemented with MINMON commands.

```
WFE03 = B4      setting clock 2 to mode 2, rate generator
WFE02 = 00      count will be 1000h
WFE02 = 10
WFE03 = 72      setting clock 1 to mode 1, one-shot
WFE01 = 00      count divided by 1000h will be duty cycle
WFE01 = 08      in this case, 800h / 1000h = 50% duty cycle
```

Subsequent writes to WFE01 can alter the duty cycle. For 8 steps from D=0 to D=1, we can write 0h, 200h, 400h, 600h, 800h, A00h, C00h, E00h, and 1000h.

The switch period here is about 2.44 kHz, as shown by the following equation:

$$10 \text{ MHz} / 4096 \text{ (1000h in decimal)} = 2.44 \text{ kHz}$$

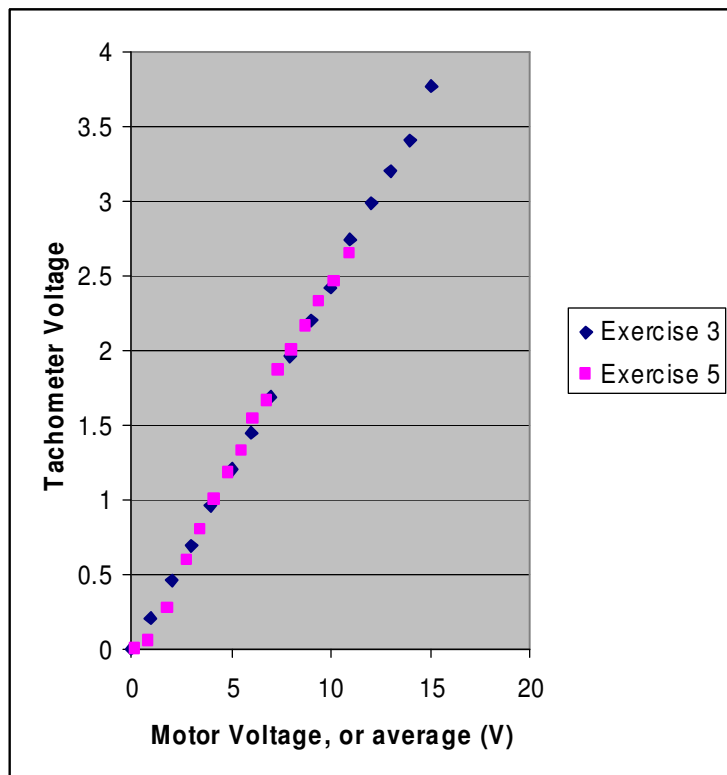
The duty cycle is determined by the duty timer count over the switch timer count:

$$\text{Duty cycle} = 1 - \frac{\text{duty timer count}}{4096}$$

The duty cycles attainable range from D=0 to D=1, give or take a 10 MHz cycle because the synchronization of the gates can potentially override the duty timer. The behavior of scheme 3 is comparable to scheme 1 in terms of lighting the LED, with higher duty cycles creating brighter light.

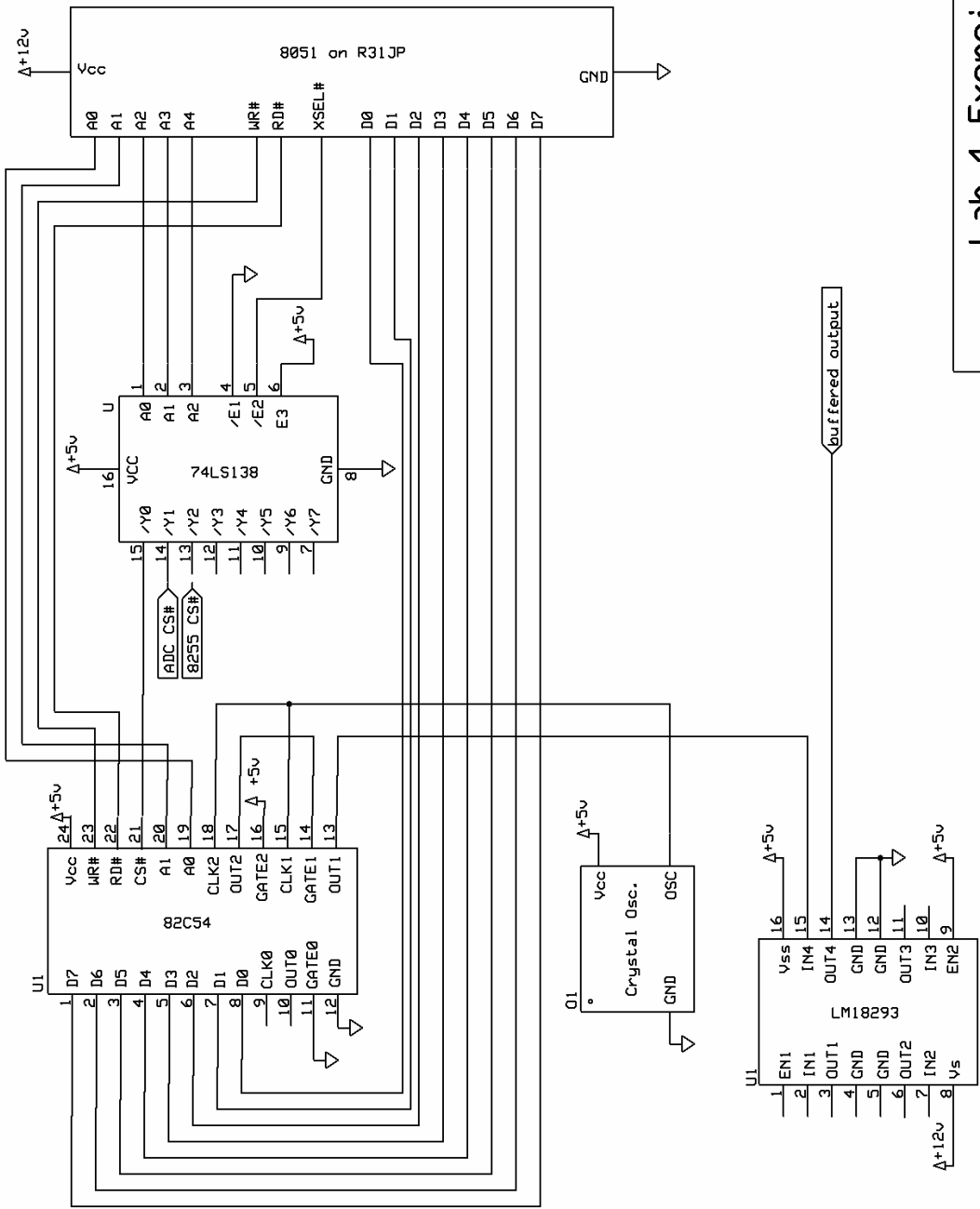
Using scheme 1, I drove my motor through the L293 chip buffer, and measured the average motor voltage (using the oscilloscope) and the tachometer voltage at 16 different duty cycles. Here are the voltages and plots:

Motor Voltage (V)	Tachometer Voltage (V)
0.24	0
0.84	0.06
1.8	0.27
2.8	0.6
3.52	0.8
4.2	1
4.86	1.18
5.55	1.33
6.16	1.54
6.81	1.66
7.42	1.86
8.01	2
8.72	2.16
9.42	2.33
10.2	2.46
11	2.65



These data points match up nicely, and show that PWM waveforms can be used to approximate intermediate voltages between the high and low voltages of digital circuits. The average voltages of the PWM waveforms result in the same motor speeds as a constant voltage at those values would.

The schematic for wiring the 8254 PWM scheme is as follows:



(Other connections not shown)

Exercise 6:

The first part of this exercise required connecting a robot arm to the buffered outputs of the 8255, and testing the arms with a simple program. The program is as follows, and the schematic is on the following page:

```
org 0h
ljmp main

org 030h
main:
    lcall init
loop:
    lcall testarm
    sjmp loop

init:
; Set some initial variables
    mov dptr, #0fe0bh    ; set control word register on 8255
    mov a, #80h         ; for regular I/O on all ports
    movx @dptr, a

    mov r5, #0
    mov r6, #0
    mov r7, #0
ret

testarm:
; Tests the arm by setting each control bit high while the
; rest are kept low. This will test each degree of the
; arm's motion. A delay routine allows for time for the arm
; to move appreciably.
    lcall wristup
    lcall delay
    lcall wristdown
    lcall delay
    lcall shoulderup
    lcall delay
    lcall shoulderdowndown
    lcall delay
    lcall baseup
    lcall delay
    lcall basedown
    lcall delay
    lcall elbowup
    lcall delay
    lcall elbowdown
    lcall delay
    lcall gripup
    lcall delay
    lcall gripdown
    lcall delay
    lcall stoparm
    lcall delay
ret

clearall:
; Clears all the control bits for the robot arm

    mov 20h, #0
    mov 21h, #0
```

ret

sendarmcommand:

```
; Send the current arm commands to the robot arm
mov a, 21h ; send bits 21.0-21.7 to port a
mov dptr, #0fe08h ; target port a
movx @dptr, a ; send to the 8255
mov a, 20h ; send bits 21.6-21.7 to port c
mov dptr, #0fe0Ah ; target port c
movx @dptr, a ; send to the 8255
ret
```

wristup:

```
; Moves the wrist one direction
lcall clearall
setb 06h
lcall sendarmcommand
ret
```

wristdown:

```
; Moves the wrist the other direction
lcall clearall
setb 07h
lcall sendarmcommand
ret
```

shoulderup:

```
; Moves the shoulder one direction
lcall clearall
setb 0fh
lcall sendarmcommand
ret
```

shoulderdown:

```
; Moves the shoulder the other direction
lcall clearall
setb 0eh
lcall sendarmcommand
ret
```

baseup:

```
; Moves the base one direction
lcall clearall
setb 08h
lcall sendarmcommand
ret
```

basedown:

```
; Moves the base the other direction
lcall clearall
setb 09h
lcall sendarmcommand
ret
```

elbowup:

```
; Moves the elbow one direction
lcall clearall
setb 0ah
lcall sendarmcommand
ret
```

elbowdown:

```
; Moves the elbow the other direction
lcall clearall
```

```
    setb 0bh
    lcall sendarmcommand
ret
```

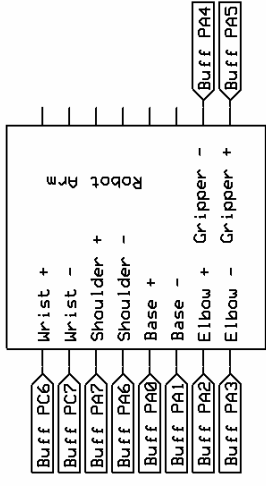
```
gripup:
; Moves the grip one direction
    lcall clearall
    setb 0dh
    lcall sendarmcommand
ret
```

```
gripdown:
; Moves the grip the other direction
    lcall clearall
    setb 0ch
    lcall sendarmcommand
ret
```

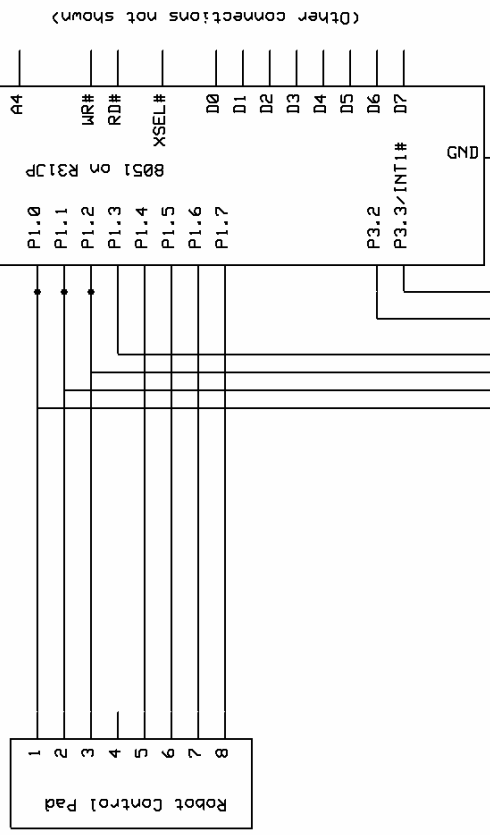
```
stoparm:
; Sends a no-move command to the arm
    lcall clearall
    lcall sendarmcommand
ret
```

```
delay:
; Delays the program while the arm has a chance to move
    djnz r5, delay
    djnz r6, delay
    dec r7
    dec r7
    dec r7
    dec r7
    dec r7
    dec r7
    dec r7
    djnz r7, delay
```

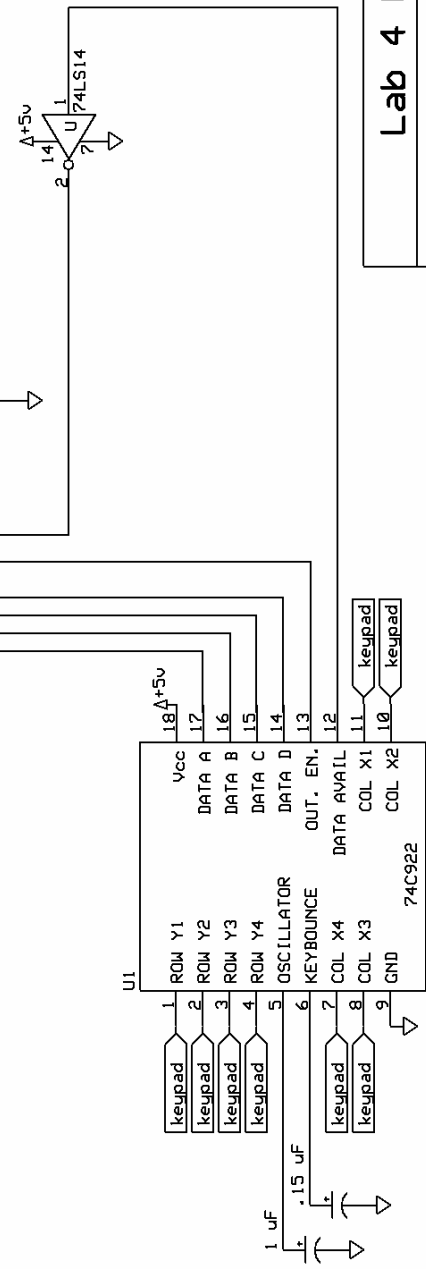
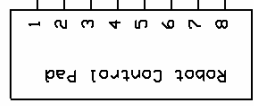
```
ret
```



These buffered signals come from the L293 chips which are powered by a 5.5V lab power supply



(Other connections not shown)



Lab 4 Exercise 6	
Robot Arm Connections	
David Lee	Rev 1.0 3/29/2005

Next, I connected the arm control pad to the R31JP. I used Port 1, tri-stating the keypad. The program I wrote supports multiple key presses, and is as follows:

```
org 0h
ljmp main
org 0bh
ljmp t0isr

org 030h
main:
  lcall init
  loop:
    sjmp loop

init:
; Set some initial variables
mov dptr, #0fe0bh ; set control word register on 8255
mov a, #80h ; for regular I/O on all ports
movx @dptr, a

mov p1, #0FFh ; prevent burnout
setb p3.2 ; tri-state the number pad

mov tmod, #02h ; run timer 0 in mode 2, with
; 8-bit auto-reload
mov th0, #0e0h ; use a low frequency
mov ie, #82h ; enable timer 0 interrupt
setb tr0

ret

sendarmcommand:
; Send the current arm commands to the robot arm
mov a, 21h ; send bits 21.0-21.7 to port a
mov dptr, #0fe08h ; target port a
movx @dptr, a ; send to the 8255
mov a, 20h ; send bits 21.6-21.7 to port c
mov dptr, #0fe0Ah ; target port c
movx @dptr, a ; send to the 8255
ret

t0isr:
; Timer zero interrupt routine
; Should scan the control pad for button presses,
; and adjust arm behavior accordingly
clr p1.0 ; pull "negative pin" low
setb p1.7 ; raise "positive pin" high
mov c, p1.1
mov 06h, c
mov c, p1.2
mov 0fh, c
mov c, p1.4
mov 08h, c
mov c, p1.5
mov 0ah, c
mov c, p1.6
mov 0dh, c
; otherwise, scan the opposite direction
clr p1.7 ; pull "positive pin" low
setb p1.0 ; raise "negative pin" high
mov c, p1.1
mov 07h, c
mov c, p1.2
mov 0eh, c
```

```

mov c, p1.4
mov 09h, c
mov c, p1.5
mov 0bh, c
mov c, p1.6
mov 0ch, c
lcall sendarmcommand
reti

```

Schematics for wiring the robot control pad are in the previous drawing.

Finally, I included a PWM waveform to drive the robot arm, using the keypad to select between duty cycles of 25%, 50%, 75%, and 100%. I kept the keypad enabled at all times, with an external interrupt to pin P3.3 signaling a change in duty cycle. Another timer interrupt generated the PWM waveforms. The keypad connections were identical to that of previous labs; the modified program is as follows:

```

org 0h
ljmp main
org 0bh
ljmp t0isr
org 13h
ljmp ex1isr
org 1bh
ljmp t1isr

org 030h
main:
lcall init
loop:
sjmp loop

init:
; Set some initial variables
mov dptr, #0fe0bh ; set control word register on 8255
mov a, #80h ; for regular I/O on all ports
movx @dptr, a

mov p1, #0FFh ; prevent burnout
clr p3.2 ; enable the number pad

mov tmod, #22h ; run timer 0 and 1 in mode 2, with
; 8-bit auto-reload
mov th0, #0c0h ; use a low frequency
mov th1, #0e0h ; use a high frequency

mov ie, #8Eh ; enable timer 0, 1, and ex1 interrupt
mov a, #4
mov 40h, #4 ; 40h should count from 3-0, repeat

setb tr0
setb tr1
ret

sendarmcommand:
; Send the current arm commands to the robot arm
push acc
mov a, 21h ; send bits 21.0-21.7 to port a
mov dptr, #0fe08h ; target port a
movx @dptr, a ; send to the 8255
mov a, 20h ; send bits 21.6-21.7 to port c
mov dptr, #0fe0Ah ; target port c

```

```

    movx @dptr, a      ; send to the 8255
    pop  acc
    ret

sendarmlow:
; Send low bits to the robot arm (PWM)
    push acc
    mov  a, #0         ; send bits 21.0-21.7 to port a
    mov  dptr, #0fe08h ; target port a
    movx @dptr, a     ; send to the 8255
    mov  a, #0         ; send bits 21.6-21.7 to port c
    mov  dptr, #0fe0Ah ; target port c
    movx @dptr, a     ; send to the 8255
    pop  acc
    ret

t0isr:
; Timer zero interrupt routine
; Should scan the control pad for button presses,
; and adjust arm behavior accordingly
    setb p3.2         ; tri-state the number pad
    mov  p1, #0ffh    ; reset p1
    clr  p1.0         ; pull "negative pin" low
    setb p1.7         ; raise "positive pin" high
    mov  c, p1.1
    mov  06h, c
    mov  c, p1.2
    mov  0fh, c
    mov  c, p1.4
    mov  08h, c
    mov  c, p1.5
    mov  0ah, c
    mov  c, p1.6
    mov  0dh, c
; otherwise, scan the opposite direction
    clr  p1.7         ; pull "positive pin" low
    setb p1.0         ; raise "negative pin" high
    mov  c, p1.1
    mov  07h, c
    mov  c, p1.2
    mov  0eh, c
    mov  c, p1.4
    mov  09h, c
    mov  c, p1.5
    mov  0bh, c
    mov  c, p1.6
    mov  0ch, c
    clr  p3.2         ; enable the number pad
    reti

ex1isr:
; This interrupt triggers with a keypad press. Depending on
; which key was pressed, the accumulator will carry either
; 1, 2, 3, or 4
    mov  a, p1         ; get value and put it in the accumulator
    anl  a, #03h      ; mask off 6 high bits
    inc  a

waitforrelease:
    jnb  p3.3, waitforrelease ; wait for key release
    reti

t1isr:
    djnz 40h, notzero ; go low

```

```

lcall sendarmlow
mov 40h, #4
notzero:
cjne a, 40h, exit ; if we hit DT, go high
lcall sendarmcommand
exit:
reti

```

Based on these experiments, I found that it was not easy to get the arm to repeat its behavior over consecutive tests. This is because there is no method of position control, only visual feedback. It was tough to position the arm correctly to pick up an item, a relatively simple task. For industrial solutions, a closed-loop feedback system would provide better control for complex tasks.

Exercise 7

This exercise involved working with the stepper motor in SpinDude. The first task was to determine what kind of stepper motor SpinDude used. I reasoned by the 6 pins (two sets of three interconnected pins) and the resistances across them that it was **unipolar**. From these resistances I also determined what each pin corresponded to on a standard unipolar stepper motor:

Red is the center tap in the inductor between yellow and orange
Green is the center tap between brown and black

This is because the resistance between red-yellow, red-orange, green-brown, and green-black was about 40 Ohms, and between yellow-orange and brown-black was about 80 Ohms, twice as high.

Running the buffered L293 output pins at 10V with current limited to 1 amp, I connected them to the stepper motor pins. The program for driving the stepper motor through the full rotation is as follows:

```

org 0h
ljmp main
org 0bh
ljmp t0isr

org 030h
main:
lcall init
loop:
lcall getchr
lcall stepmotor
lcall stepmotor
lcall stepmotor
lcall stepmotor
lcall stepmotor
lcall stepmotor
lcall stepmotor
ljmp loop

init:
; Set some initial variables
mov dptr, #0fe0bh ; set control word register on 8255
mov a, #80h ; for regular I/O on all ports
movx @dptr, a

```

```

mov dptr, #0fe08h ; target port a

mov r7, #20 ; r7 should hold 2-60, depending on delay
; r7 * .05 s = the delay period
clr 20h ; delay ending trigger

mov tmod, #21h ; set timer 1 for auto reload - mode 2
; and timer 0 for mode 1 (16 bit)
mov tcon, #40h ; run counter 1
mov th1, #0FDh ; set 9600 baud with xtal=11.059MHz
mov scon, #50h ; set serial control reg for 8 bit data
; and mode 1
mov ie, #82h ; enable timer 0 interrupt
setb tr1
ret

t0isr:
; 16-bit timer interrupt for delay count
clr tr0
djnz r7, skip
setb 20h
mov r7, #20 ; r7 should hold 2-60, depending on delay
; r7 * .05 s = the delay period
ljmp exit
skip:
mov th0, #04ch ; count out .05 s
mov tl0, #000h
exit:
reti

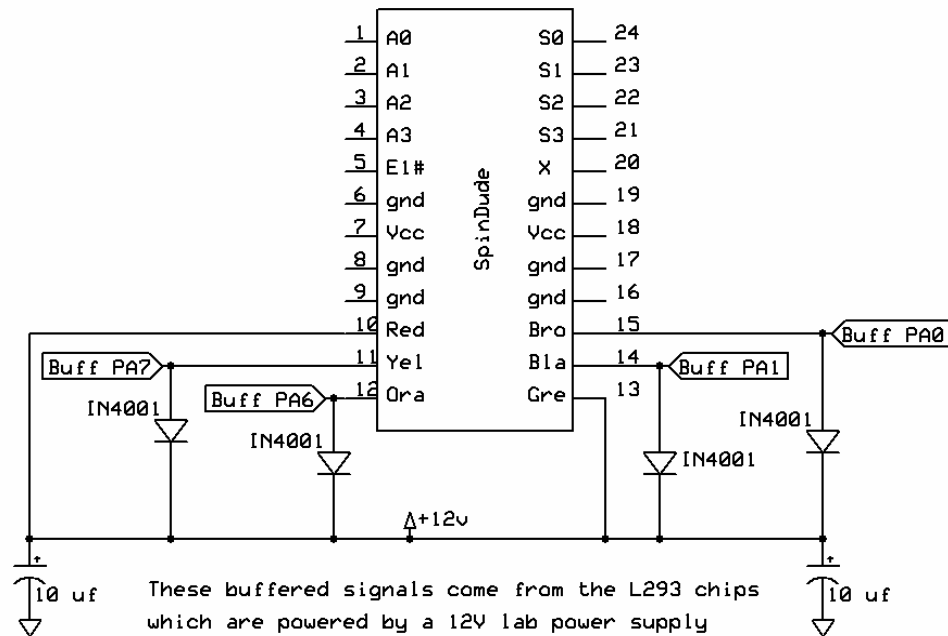
getchr:
; Receives a character from the keyboard.
; Doesn't do anything with it.
jnb ri, getchr ; wait till character received
clr ri ; clear serial "receive status" flag
ret

stepmotor:
; Step motor 4 steps
mov a, #080h ; set 1a
movx @dptr, a ; send to the 8255
lcall delay
mov a, #001h ; set 2a
movx @dptr, a ; send to the 8255
lcall delay
mov a, #040h ; set 1b
movx @dptr, a ; send to the 8255
lcall delay
mov a, #002h ; set 2b
movx @dptr, a ; send to the 8255
lcall delay
ret

delay:
; Delay for some time, from .1 to 3 seconds
setb tr0
jnb 20h, delay
clr 20h
clr tr0
ret

```

The schematic for this set-up is as follows:



Lab 4 Exercise 7		
Stepper Motor connections		
David Lee	Rev 1.0	
	3/29/2005	

Exercise 8

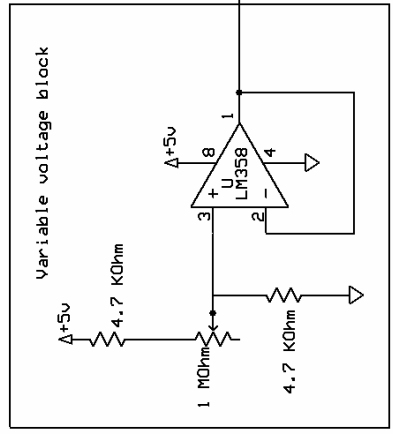
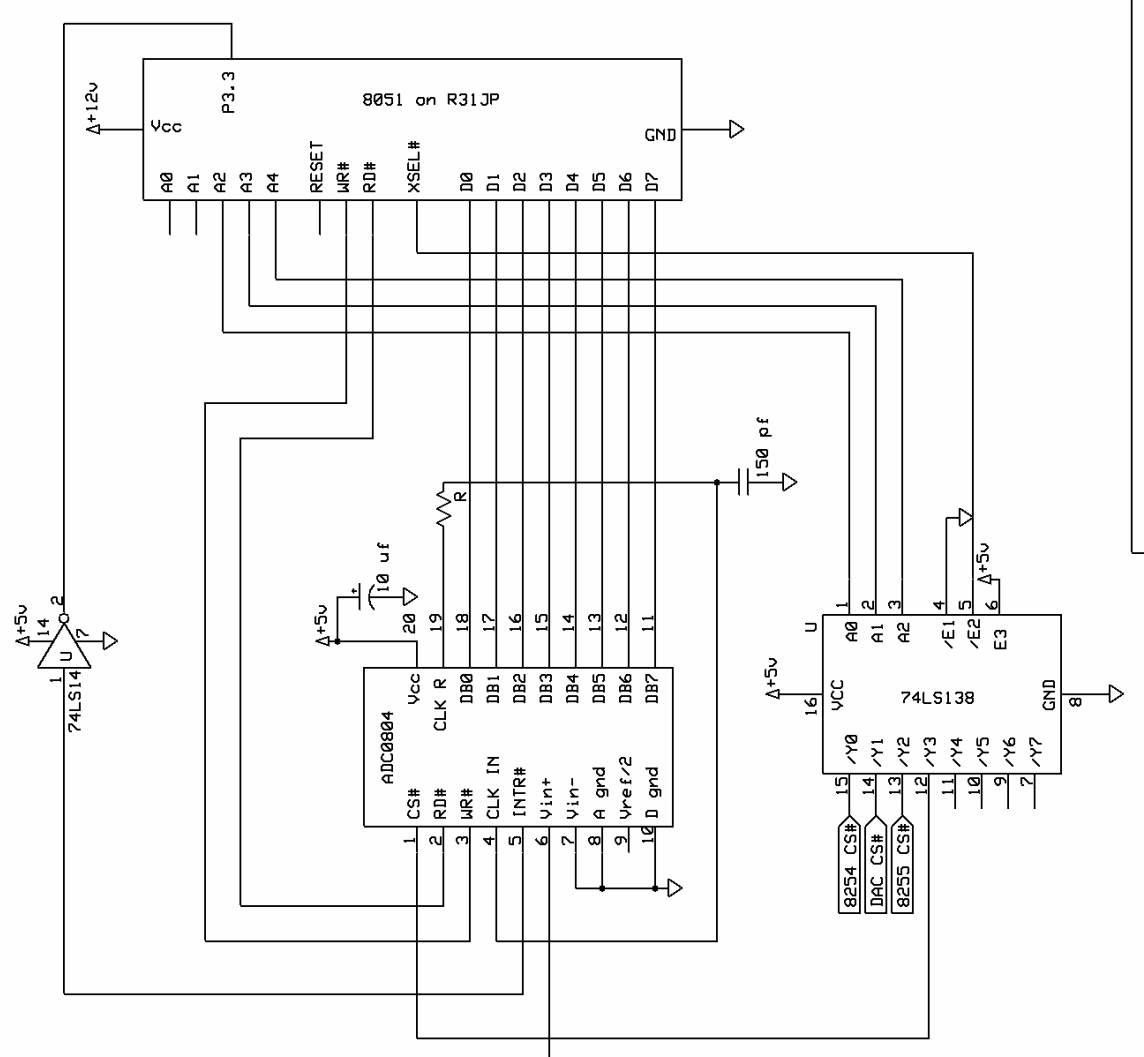
This exercise set up the ADC0804 as an additional peripheral to measure analog voltages, eventually to be used in measuring light output on SpinDude. A short program to measure the input analog voltage and display the digital output on the port 1 LED bank is as follows:

```

setb p3.2          ; tri-state the keypad
mov p1, #0ffh
mov a, #1
mov dptr, #0fe0ch ; target ADC
movx @dptr, a     ; get reading
loop:
  jnb p3.3, loop  ; wait for the data to be ready
  movx a, @dptr   ; display on p1
  mov p1, a
endloop:
  sjmp endloop

```

Following is a schematic of the ADC set-up and a variable voltage source useful for testing the ADC.



Lab 4 Exercise 8

ADC Converter

David Lee

Rev 1.0
3/29/2005

Exercise 9

This exercise combined the work from previous exercises to produce meaningful data to be used in imaging a 3-dimensional object sitting on a platform rotated by the stepper motor. The motor takes 24 steps through a full rotation, and on each step, the row of LED's and phototransistors on either side of the object mount flashes and transmits the amount of shadow back to the microcontroller (one LED-phototransistor pair at a time). This data is sent to the terminal as a table of hex values, and a MATLAB program reconstructs a cross-sectional image of the object. The following is the program for producing this projection data and the schematic for wiring the rest of SpinDude to the R31JP:

```
org 0h
ljmp main
org 0bh
ljmp t0isr

org 030h
main:
  lcall init
loop:
  lcall getchr      ; wait for the key press
  lcall stepmotor   ; each call rotates 4 steps
  lcall stepmotor
  lcall stepmotor
  lcall stepmotor
  lcall stepmotor
  lcall stepmotor
  ljmp loop

init:
; Set some initial variables
setb p3.2          ; tri-state the keypad
mov p1, #0ffh     ; prevent burnout

mov dptr, #0fe0bh ; set control word register on 8255
mov a, #80h       ; for regular I/O on all ports
movx @dptr, a

mov r5, #0
mov r6, #0
mov r7, #20       ; r7 should hold 2-60, depending on delay
                  ; r7 *.05 s = the delay period
clr 20h          ; delay ending trigger

mov tmod, #21h    ; set timer 1 for auto reload - mode 2
                  ; and timer 0 for mode 1 (16 bit)
mov tcon, #40h    ; run counter 1
mov th1, #0FDh   ; set 9600 baud with xtal=11.059MHZ
mov scon, #50h   ; set serial control reg for 8 bit data
                  ; and mode 1
mov ie, #82h     ; enable timer 0 interrupt
setb tr1
ret

t0isr:
; 16-bit timer interrupt for delay count
clr tr0
djnz r7, skip
setb 20h
```

```

    mov r7, #20      ; r7 should hold 2-60, depending on delay
                    ; r7 * .05 s = the delay period
    ljmp exit
skip:
    mov th0, #04ch  ; count out .05 s
    mov tl0, #000h
exit:
    reti

stepmotor:
; Step motor 4 steps
    mov a, #080h    ; set 1a
    mov dptr, #0fe08h ; target port a
    movx @dptr, a   ; send to the 8255
    lcall delay     ; wait to stabilize
    lcall scanshadow ; scan the shadows
    mov a, #001h    ; set 2a
    mov dptr, #0fe08h ; target port a
    movx @dptr, a   ; send to the 8255
    lcall delay
    lcall scanshadow
    mov a, #040h    ; set 1b
    mov dptr, #0fe08h ; target port a
    movx @dptr, a   ; send to the 8255
    lcall delay
    lcall scanshadow
    mov a, #002h    ; set 2b
    mov dptr, #0fe08h ; target port a
    movx @dptr, a   ; send to the 8255
    lcall delay
    lcall scanshadow
    reti

scanshadow:
; Illuminate each LED in turn, scan the shadows
    mov p1, #0fh    ; start with last light
scanloop:
    lcall delay2
    lcall scanstep
    djnz p1, scanloop ; work our way down to the first
    lcall delay2     ; make sure to do the "zero" LED
    lcall scanstep
    lcall crlf      ; carriage return
    mov p1, #10h    ; turn off the lights
    reti

scanstep:
; For a single phototransistor, sends its voltage to the terminal
    mov dptr, #0fe0ch ; target ADC
    movx @dptr, a     ; get reading
waitloop:
    jnb p3.3, waitloop ; wait for the data to be ready
    movx a, @dptr     ; send to terminal
    lcall prthex
    mov a, #32        ; send a space
    lcall sndchr
    reti

;;;;;;;;;;;;; Useful delay routines ;;;;;;;;;;;;;;

delay:
; Delay for some time, from .1 to 3 seconds
    setb tr0
    jnb 20h, delay

```

```

    clr 20h
    clr tr0
    ret

delay2:
; Delay for about 71 milliseconds, enough to get a
; phototransistor reading
    djnz r5, delay2
    djnz r6, delay2
    ret

;;;;;;;;;;;;; Useful serial port routines ;;;;;;;;;;;;;;

getchr:
; Receives a character from the keyboard.
; Doesn't do anything with it.
    jnb ri, getchr      ; wait till character received
    clr ri              ; clear serial "receive status" flag
    ret

sndchr:
; Sends a byte to the terminal (PC)
    clr scon.1         ; clear the ti complete flag
    mov sbuf, a        ; move a character from acc to the sbuf
txloop:
    jnb scon.1, txloop ; wait till chr is sent
    ret

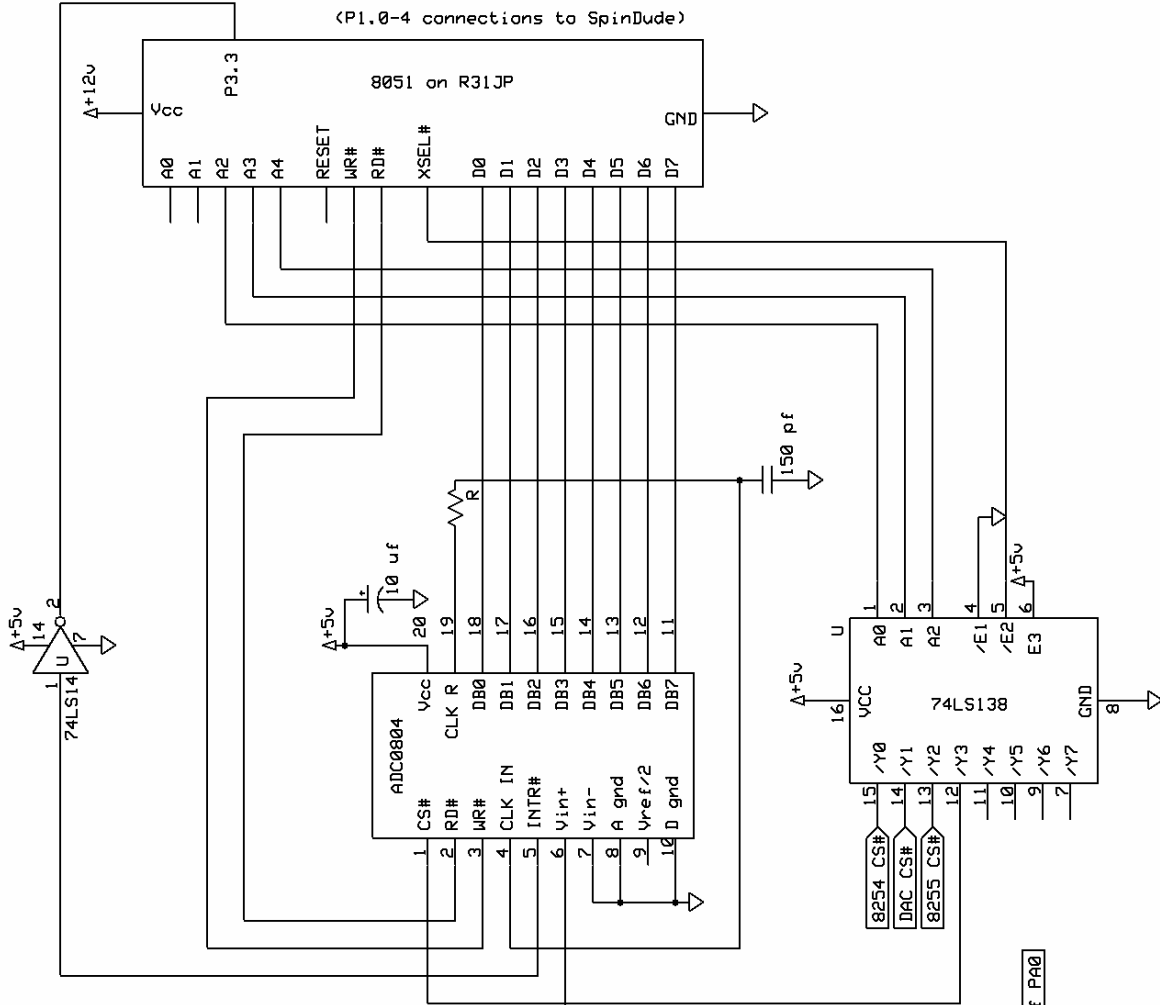
prthex:
    push acc
    lcall binasc        ; convert acc to ascii
    lcall sndchr        ; print first ascii hex digit
    mov a, r2           ; get second ascii hex digit
    lcall sndchr        ; print it
    pop acc
    ret

binasc:
    mov r2, a           ; save in r2
    anl a, #0fh         ; convert least sig digit.
    add a, #0f6h        ; adjust it
    jnc noadj1          ; if a-f then readjust
    add a, #07h
noadj1:
    add a, #3ah         ; make ascii
    xch a, r2           ; put result in reg 2
    swap a              ; convert most sig digit
    anl a, #0fh         ; look at least sig half of acc
    add a, #0f6h        ; adjust it
    jnc noadj2          ; if a-f then re-adjust
    add a, #07h
noadj2:
    add a, #3ah         ; make ascii
    ret

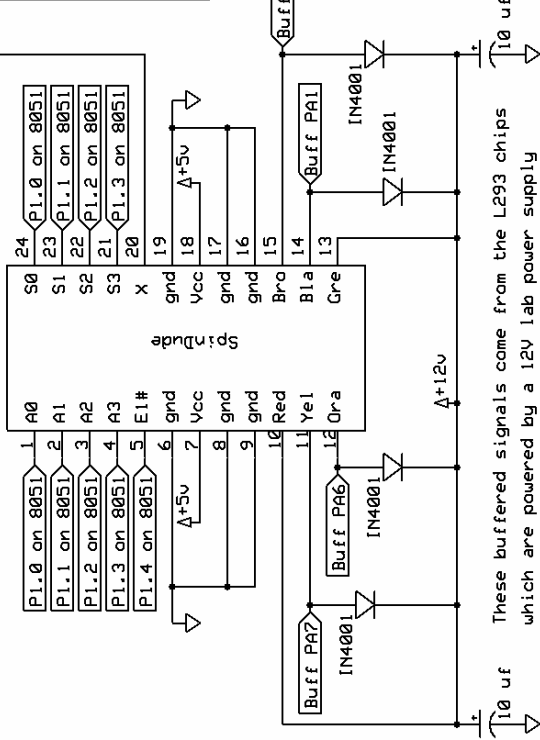
crlf:
    mov a, #0ah         ; print lf
    lcall sndchr

cret:
    mov a, #0dh         ; print cr
    lcall sndchr
    ret

```



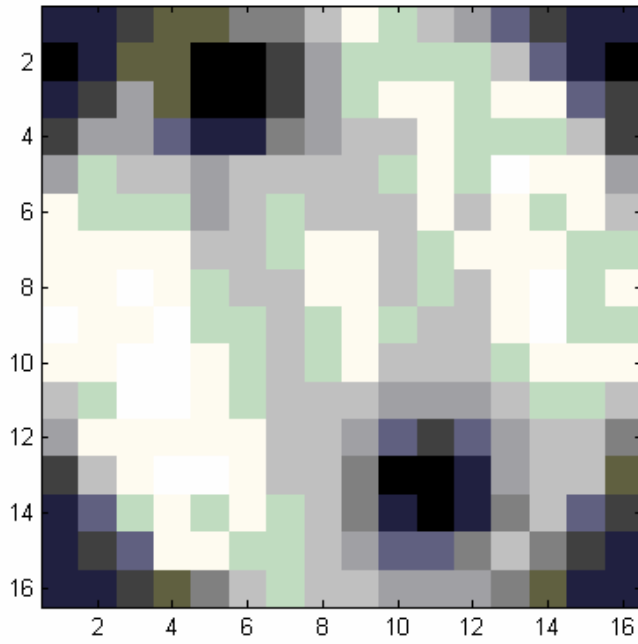
Lab 4 Exercise 9
SpinDude
David Lee
Rev 1.0
3/29/2005



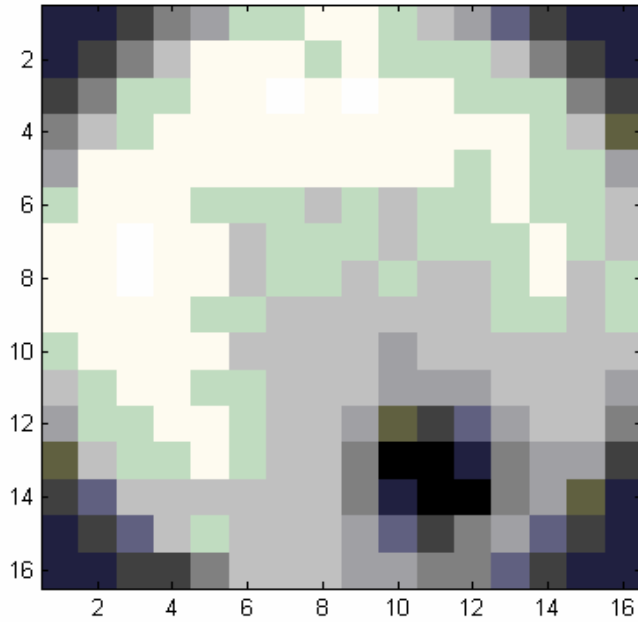
These buffered signals come from the L293 chips which are powered by a 12V lab power supply

Here are some of the images reconstructed from the SpinDude data:

Two dowels:



One dowel:



This program could provide more useful projection data if every phototransistor were scanned during each LED flash. As it is, only one angle is scanned (straight across the SpinDude by one LED and one phototransistor), while scanning all angles with one LED on could provide more useful data. To produce a 3-dimensional image, we could raise the rotating platform with each full rotation, scanning several cross-sections and putting together a solid form.

Exercise 10

This exercise continued work on the PIC microcontroller. First, I coded, compiled, and burned the provided square wave program, square.c, onto the PIC microcontroller. Then, I connected logic switches 1-4 on the lab kit to pins PB0-3 on the PIC. The program produces a square wave with period determined by which of the switches are high. Each switch adds twice as much time as the previous one, such that there are 16 different periods available. The square wave is output on pin PA1.

I wrote another program that could drive the unipolar stepper motor. It still uses the logic switches to determine overall period of waveforms, but produces 4 different waveforms, each pulsing in succession. Connected properly, this would power the stepper motor properly through each step. I used PA0-3 for this purpose. The program is as follows:

```
// square2.c

#include <pic.h>
#include <pic16f6x.h>

__CONFIG(0x3ff0);

void delay(int j);

void main(void) {
    int s1, s2, s3, s4, j;
    TRISB = 0x0F;
    TRISA = 0x00;

    PORTA = 0b00000000;
    while(1) {
        s1 = RB0;
        s2 = RB1;
        s3 = RB2;
        s4 = RB3;
        j = ((625 * s1)+(1250 * s2)+(2500 * s3)+(5000 * s4));
        if (j > 0) {
            PORTA = 0b00000000;
            delay(j);
            PORTA = 0b00000010;
            delay(j);
            PORTA = 0b00000000;
            delay(j);
            PORTA = 0b00000001;
            delay(j);
            PORTA = 0b00000000;
            delay(j);
            PORTA = 0b00000100;
            delay(j);
            PORTA = 0b00000000;
            delay(j);
        }
    }
}
```

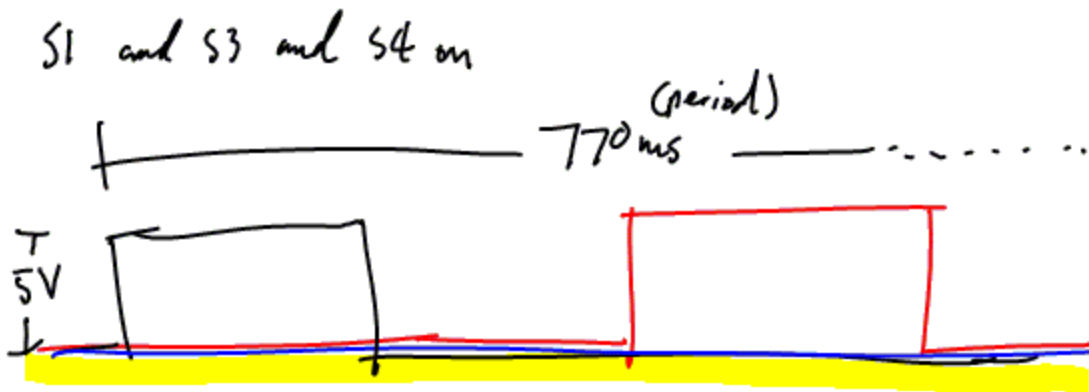
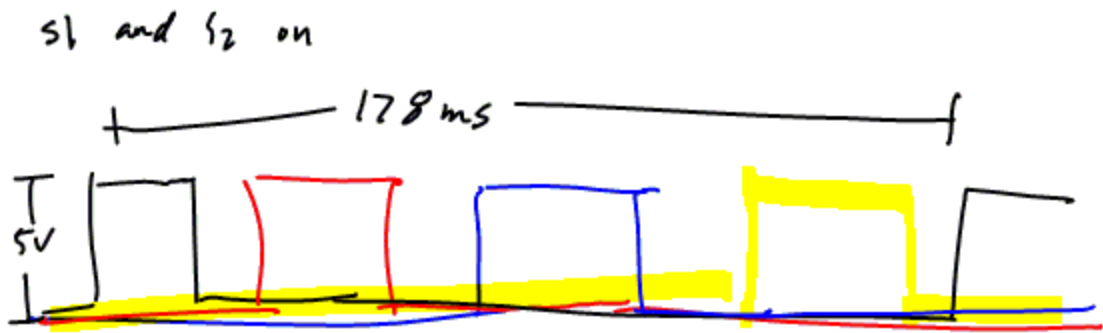
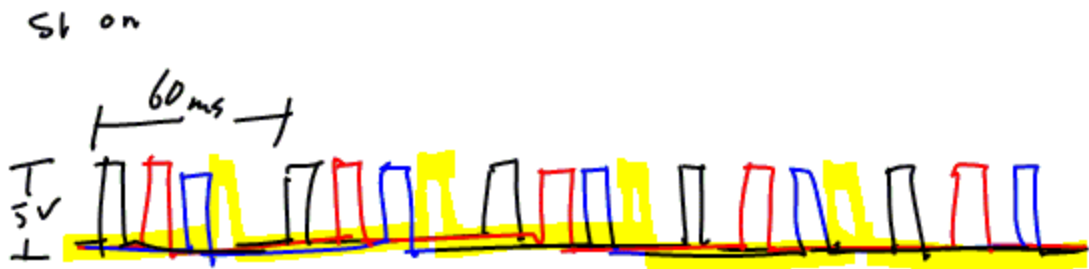
```

        PORTA = 0b00001000;
        delay(j);
    }
    else PORTA = 0b00000000;
}
}

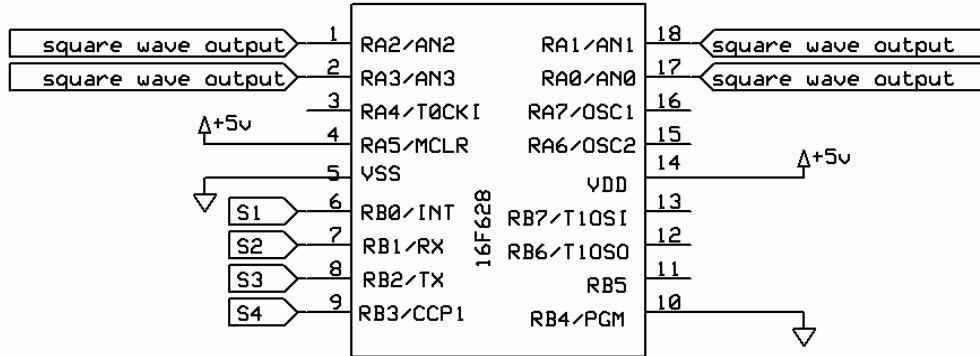
void delay(int j) {
    unsigned int i;
    for (i = 0; i < j; i++);
}

```

This produces waveforms on each of the 4 output Port A pins. Here are rough sketches under three different settings on my logic switches:



The schematic for the PIC wiring is as follows:



Lab 4 Exercise 10		
PIC Square Waves		
David Lee	Rev 1.0	
	3/30/2005	