Summary

The task of the experiment was to write a program that would control a stepper-motor based spectrometer, using the ATmega16 microcontroller. The hardware also had to be designed but this consisted of only two small circuits that allowed the microcontroller to be able to receive input from the motors, and read and external input.

Firstly a ‘start’ command is sent to the microcontroller via the serial link from the computer. This starts off the motors and measurement cycle. The motors turn in the correct sequence, stopping and starting when they should. However the results do not show up on the screen at the end. I have tried to find the source of this problem but I cannot, but I have tracked the problem down to somewhere between the input to the Analogue-to-Digital Converter (ADC) and the output to the computer, as there is a current produced by the photo-diode. Therefore, the place where the problem occurs is either in the conversion process or the conversion of the numbers to ASCII so the computer can recognize them as characters.

I have failed in the task of making the spectrometer, but I am pretty close. The problem is a software issue and I don't think it has to do with the hardware used, as it was checked and replaced, and not very complicated.

Introduction

The design task of this experiment was to construct a piece of software that can be run on a microcontroller to measure the light transmittance through different colour filters. There are two discs, one filter and one sample that are mounted on separate stepper motors. The filter disc has 2 holes cut out of it, one with no filter for a control sample. The sample disc has 3 holes cut out of it, with a red, a green and a blue filter placed in the hole. The motors could turn independently of each other. The light level would be calculated for each of the filters on both wheels. This was done by using a photodiode and an LED.

On a 'start' command, the wheels are sent to the start position and measurements begin. This involved turning the wheels so that the LED would shine light through the two filters and the photo-diode would pick up the amount of light. First, the light is light level is found for the blank filter in the sample wheel and the red filter of the filter wheel. The sample wheel is then moved so that the sample wheel and the red filter are lined up. The measurement is taken again. The sample wheel moves back to the original position, and the same thing is repeated with the green and blue filters respectively.

The output of the photodiode is a current so it was converted to a voltage and then an Analogue-to-Digital Converter (ADC) was used to so the light transmittance would be found. The light transmittance was calculated by the ratio of the light level with the filter, and without it. Once the values were calculated, there were sent over a serial link to the computer.

The wheels then return to the start position, and wait for the next start command.

Design Considerations

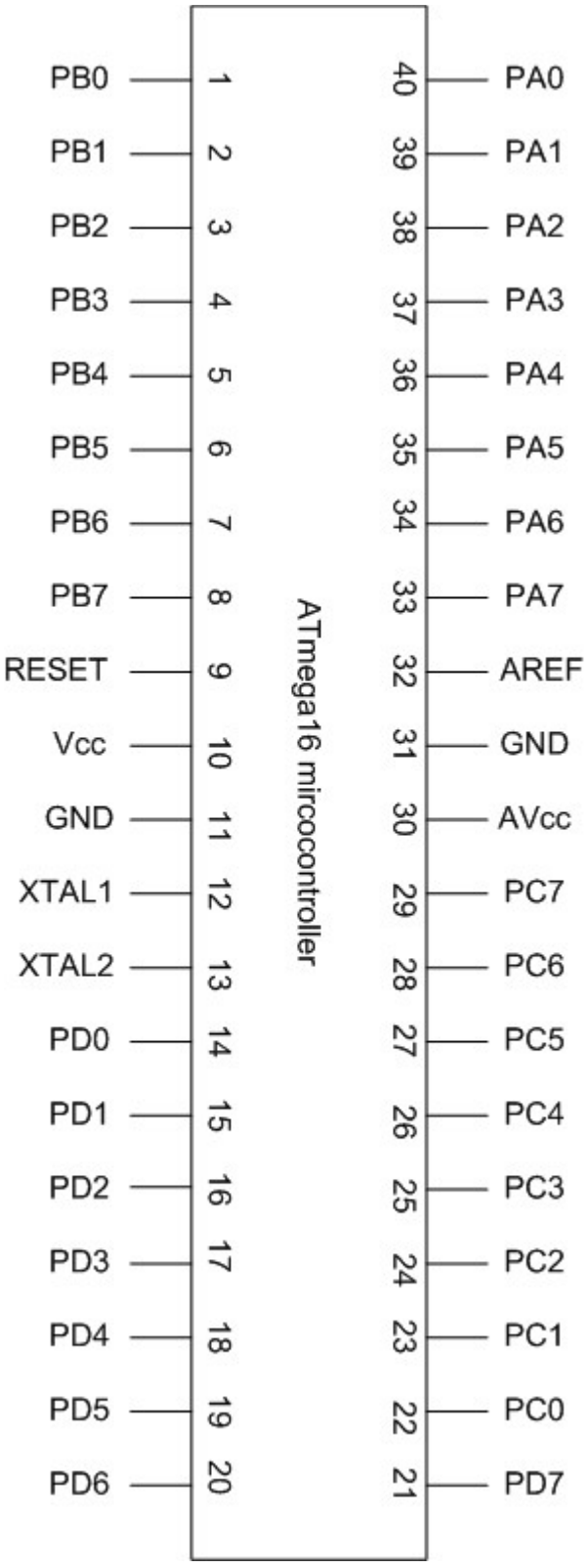
The first section task was to write software that would control the stepper motors. Each stepper motor has four windings, which may be turned on and off, to make the motor turn. When power is applied to a specific winding, it asserts a magnetic force on the rotor, pulling it towards the coil, in a circular fashion. Stepper motors are found in many everyday situations, for example floppy drives, flatbed scanners and printers. The stepper motor uses a Darlington Pair and TTL transistors to provide the 7.4V and 1.3A needed to drive the motor. The Darlington Pair is a single device that combines two bipolar junction transistors. The reason this is used is because the gain is very high and takes up less space than using two discrete transistors. The Darlington Pair is named after the inventor of the configuration, Sidney Darlington while working at Bell Labs.

The whole experiment was centred on the Atmega16, a general purpose RISC (Reduced Instruction Set Computer) microcontroller with 32 general-purpose 8-bit registers. The Atmega16 has the following specifications:

- 16kB of flash memory, where the program is stored when writing to it from the computer.
- 1kB of SRAM
- 512B EEPROM
- USART (Universal Syncronous/Asycronous Reciever/Transmitter) for connecting the device to the computer for programming.
- Analogue comparator
- Eight channel ADC
- 8-bit timer/counter module
- Two 16-bit timer/counter modules
- Four 8-bit input/output ports

To program the microcontroller to perform a specific task, the instructions are written in a program called AVR Studio. From here your program can be transferred, via a serial link to the device. To put it in programming mode, the logical switch must be LOW, and the RESET pulser pushed. To run the program once it has been transferred, set the logical switch HIGH and push the pulser again. When the RESET button is pushed, the program automatically jumps into the RESET routine of the program.

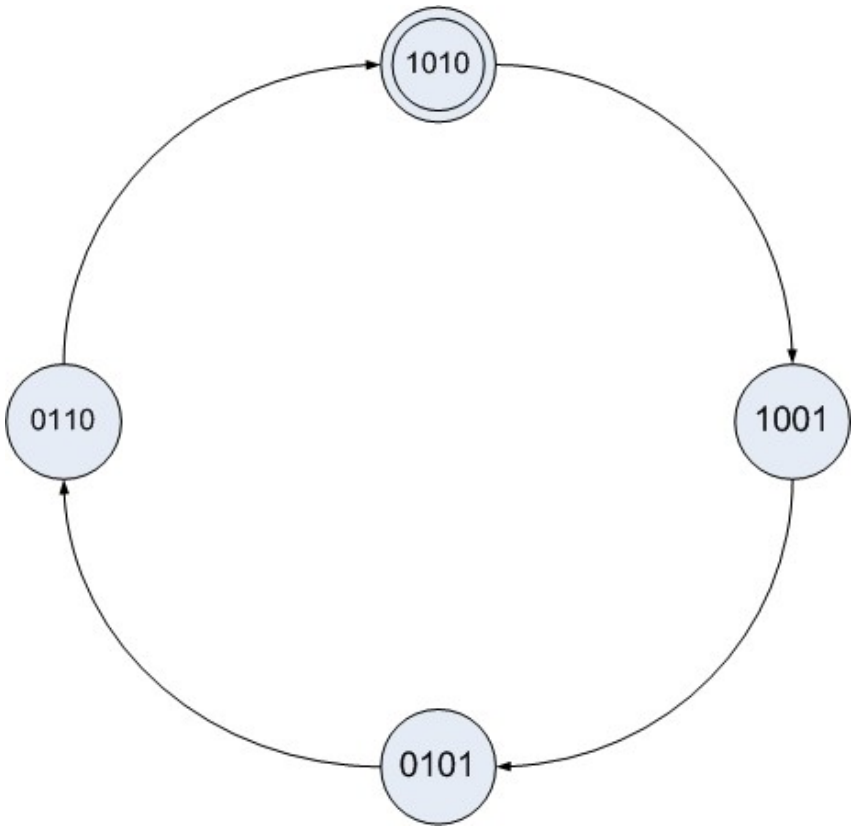
To start the program, a utility called Hyper-terminal is used, which comes with Microsoft Windows. This connects with the chip via the serial link. Below is a pinout of the Atmega16 and the functions of each pin.


ATmega16 mircocontroller
PB0 1
PB1 2
PB2 3
PB3 4
PB4 5
PB5 6
PB6 7
PB7 8
RESET 9
Vcc 10
GND 11
XTAL1 12
XTAL2 13
PD0 14
PD1 15
PD2 16
PD3 17
PD4 18
PD5 19
PD6 20
40 PA0
39 PA1
38 PA2
37 PA3
36 PA4
35 PA5
34 PA6
33 PA7
32 AREF
31 GND
30 AVcc
29 PC7
28 PC6
27 PC5
26 PC4
25 PC3
24 PC2
23 PC1
22 PC0
21 PD7

For this experiment, the motors needed to be driven in full-step mode and in only one direction. The motors needed to be able to be driven independently of each other. For a full-step sequence, that is 1.8° per turn, the signals have to be sent as follows:

Step	W1	W2	W3	W4
1	1	0	1	0
2	1	0	0	1
3	0	1	0	1
4	0	1	1	0

The state transition diagram below shows the order of the states that have to be changed on the motor's four input wires to make it turn in a clockwise, full-step fashion. Each step is 180 degrees, so to turn the sample motor around; the states have to be changed 100 times.



Once both wheels were in the start position, the measurements would begin. The first measurement would be taken at the start position; the sample wheel was then turned through 180° and the second measurement taken. The sample wheel would then return to the start position and the filter wheel turned through 120°, to the next colour. The measurement would then be taken and the same process occurs with the sample wheel. The last colour of the filter wheel would then be measured in the same way.

Once the 6 measurements were taken, the built-in Analogue to Digital converter was used to calculate the value from the photo-diode and the results printed out to the computer screen. The current output of the photo-diode had to be converted to a voltage before being fed into the ADC.

Once the digital number was found, the ratio of light transference was calculated. This was done by the following formulae:

$T_{red} = 100 * I/I_0$
$T_{green} = 100 * I/I_0$
$T_{blue} = 100 * I/I_0$

Where I_0 is the incident light level with no filter, and I is the light level with the filter in-place.

These figures were then converted to ASCII characters so the computer could recognise them as number or letters. The formatted results were then sent over the serial link to the computer.

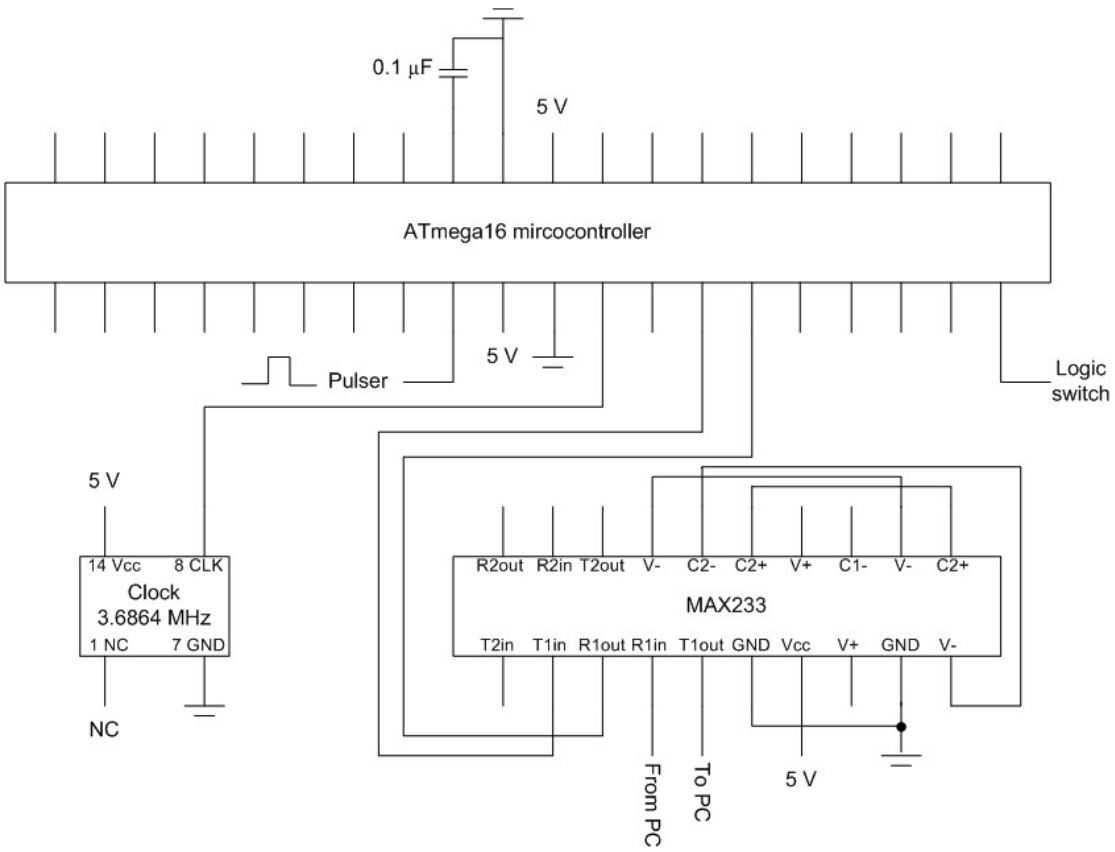
Design Details

For the start command, I decided to go for just a simple push of the 's' key. This is because as the results were going to be printed out to the computer, it should be started there as well. To do this, I set up the on-board USART in polled mode, waiting to receive the 's' command, and once it did, to go with the rest of the program. To check if the 's' key was pushed, the program kept checking to see if it had received the ASCII character 73.

With the 's' command received, the motors begin to turn and the measurements begin. First off, the wheels turn until they are at the start position. This was achieved with the photo-interupter. Each wheel has a tab on the side of it. When this tab goes through the photo-interupter, the output of it goes LOW. Normally it is HIGH. Therefore, the wheels are turned until the photo-interupter goes LOW. To set up the serial link, a MAX233 chip was used. For the microprocessor to be able to respond to the change of output of the photo-interupter, it must be connected to it somehow. This is the purpose of a port.

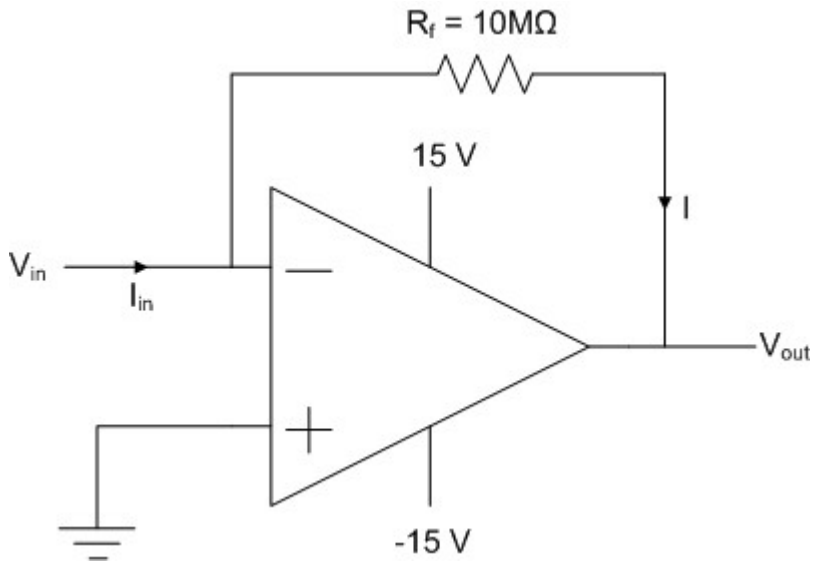
The Atmega16 has four ports that can be configured either as an input or an output. This is programmed in software by the DDRX register, where X is the port letter A to D. When the DDRX register is LOW, it is an input and an output when HIGH. Here it is easiest to use hexadecimal numbers, where LOW is 0x00 and HIGH is 0xFF. Each port also has a special function, for example PORTA is an 10-bit Analogue-to-Digital Converter which is built-in. PORTD is used for the serial link to connect it to the computer. Each port can be used either in the generic or specialised form. It can be changed with software, so during different stages of the program they can perform different functions.

To complete the microprocessor circuit, a external clock is needed. We chose a crystal oscillator that ran at 3.6864MHz. The pinout of the MAX233, the microprocessor and the clock is show below. It shows the basic set up of the ATmega16 without anything that is specific to this experiment (with the possible exception of the MAX233 serial chip).



There were two ways to make the stepper motor turn. These were to run both motors off the single eight-bit register and therefore one port. The code and logic required to make this work is more complicated, but it is a better design because less ports and registers are used, thus leading to lower cost or more functionality. To make each motor turn individually, the two nibbles of the 'state' register had to be changed one at a time. For example, if the filter wheel wanted to be turned, the upper nibble of the state register was changed with the lower nibble kept the same. This required quite a few left or right logical shifts and swap commands. The nibble that has to be modified is then sent to the getNextState method which looks at the current state, and loads in the value of the next state. Before returning back it goes through a delay sequence. This slows the program down so the motor can respond to the changes of state. If the delay loop was not there, the state would change very quickly, and so the motor would do nothing. The delay which is a nested loop delay causes a delay of about 50ms. This is enough time for the motor to respond to changes of input.

Once the wheels are both in the start position, the measurements can begin. The first measurement is taken at the start position, with the blank of the sample wheel and the green of the filter wheel. The sample wheel is then spun 180° so the LED shines through the sample and the same green filter. As each motor step is 1.8°, it is just moved 100 steps. The light that goes through both filter falls upon the photodiode. The output of the photodiode is a current. The light transmittance is calculated as the current with no sample over the current with the sample, multiplied by 100, as shown above. However to get to that stage, it first has to be converted from a current to a voltage. This is achieved by the operational amplifier circuit, as shown below.



The output of the above circuit can be shown to be:
$V_{out} = -A\ I\ R_f / (A + 1) \approx -I_{in}\ R_f$ for $A \gg 1$

Once light transmittance level was converted to a voltage, it can now be fed into the on-board ADC (see above diagram). As it now a digital number, new can now perform arithmetic on it. The value of the light level with no sample is stored, so the formula above can be calculated with the light level when the wheel has tuned 180°. When we have two numbers available, we do the calculation and store the answer in another register so we can send the results out at the end of the whole experiment.

As the Atmega16 does not have any division instructions, division has to be synthesised. This is done by a series of commands which are provided by the manufacturer because divison is a well-used calculation. Basically what it does is take away the divisor from the remainder each time the result is negative.

This process occurs with the other two filter colours in the same way, and so now we have all the answers stored in registers. We can send the numbers directly as they are but as the serial link expects ASCII characters to be sent, it will print out gibberish. So, to get around this each nibble of each number is converted to ASCII code before being sent. Thus when it arrives at the computer screen with the rest of the formatting it will be able to be read.

Once, the results have printed out to the screen, it will go back and begin to poll the ‘s’ character to be pushed again, and the process will start all over again.

Performance

As the measurement part of the experiment did not function correctly, I have nothing to report. However the motors turned correctly, and started or stopped when they should. The microprocessor correctly looked for the ‘start’ command, and acted appropriately on the input. The motors turned independently of each other, until both were at the start position. From here the sample motor turned, and then the filter wheel turned and so on. All motions were in the correct order and direction with no jumps or anything unexpected.

The part where my code was incorrect was with the ADC, I think. I think that the code to do the calculations is correct, but without testing it in real life it is hard to say. In debug mode, it seems to work correctly however. The serial link is working correctly. It receives the 'start' command as well as sending the formatting for the answers correctly.

Conclusion and Summary

I am confident that my hardware is correct, but part of my software is correct. It works up until the stage where calculations are made. It looks for the start command and starts accordingly. The motors turn to the correct angles, and stop when they should. However when it comes to reading the data in from the on-board ADC, it does not work. I have spent a long time trying to find solutions but am confident that I am not too far off.

Through testing in debug mode of AVR Studio, the ADC and calculation routines seem to be correct, but when testing in the real world, nothing is printed out to the screen. I have spent quite a time trying to figure out the problem, but I am reasonably confident that there is not something seriously wrong with my design.

This is a pity because after a lot of work, I don't have much to show for it and doesn't actually achieve the task of measuring the light levels.

Appendicies

Parts List

The parts required for this experiment are:

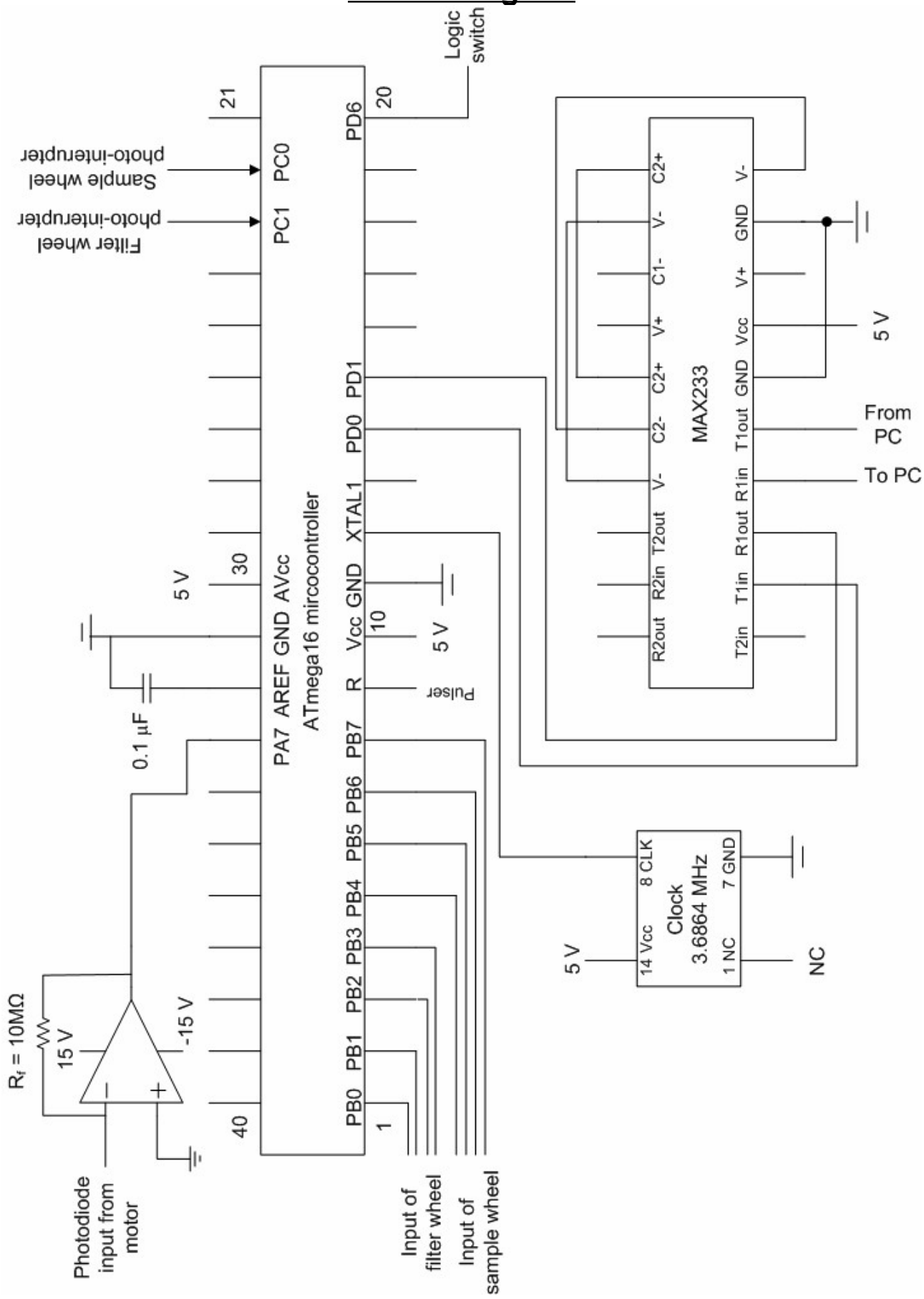
- Atmega16 microcontroller
- PC with serial link
- Spectrometer
- Clock
- MAX233 chip
- 741 Operational Amplifier
- Student Design Station

Software Listings

The only software needed for this experiment are:

- AVR Studio
- Hyper-terminal (built into Microsoft Windows)

Circuit Diagram



Program Code

```
;This file contains defs for all of the mega16's
;registers.  Also includes values for various
;constants like the size of the internal SRAM.
.include "m16def.inc"


;General registers
.def temp1 = r16
.def temp2 = r17
.def temp3 = r18
.def temp4 = r19
.def temp5 = r20

.def steps = r21

;The current state of the both motors,
;(4 bits each)
.def state = r22
.def tempState = r23

.def loop1 = r24
.def loop2 = r25
.def counter = r27

.def BB =r2
.def BS =r3
.def RS =r4
.def RB =r5
.def GB =r6
.def GS =r7
.def redresult =r8
.def blueresult =r9
.def greenresult =r10

;place these instructions at the
;start of program memory
.org 0x0000

; If a reset occurs, then jump to the start of the code
      jmp RESET

RESET:
      ;* First we need to setup the 'stack pointer' otherwise we will
have problems
      ;* when we call a subroutine.  This is usually set to the end of
the internal SRAM.

      ldi   temp1, high(RAMEND)   ; set temp1 to equal the high byte of
the address RAMEND
      out SPH, temp1              ; set the high byte of the hardware
stack pointer
      ldi temp1, low(RAMEND)      ; set temp1 to equal the low byte of the
address RAMEND
```

```
      out SPL, temp1             ; set the low byte ofthe hardware stack
pointer

      ;load state with 0xA9
      ldi state, 0b10101001
      ldi temp2, 0
      ldi temp3, 0
      ldi counter, 0

      ;Make port B and output
      ldi temp2, 0xff
      out DDRB, temp2

      ;Make ports A and C inputs
      ldi temp2, 0x00
      out DDRC, temp2
      out DDRA, temp2

main:
      call setupUSART       ;Do the initial set for USART

      call SetupADC   ;Do the initial setup for the ADC

      call StartMessage          ;Tell the user what to do


loop:
      call GetStart
      rjmp loop



;********** SUBROUTINES ***************************************

;*************************************************************
;* Sit in polling mode until an 's' is sent                    *
;*************************************************************
getStart:
      sbis UCSRA, RXC
      rjmp getchar

      in temp1, UDR
      cpi temp1, 73    ;if(temp1 == 's')
      breq moveWheels

      ret

moveWheels:
      ;The steps field is not used in this sequence because
      ;we want to stop at the photo-interuptuers, so set
      ;it to something large
      ldi steps, 200
      ;if both are in the start, then do measurements
      cpi temp2, 0b11
      rjmp doMeasurements
```

```
      ;filter is in the start position, so move sample
      cpi temp2, 0b01
      call moveSampleWheel

      ;sample is in the start position, so move filter
      cpi temp2, 0b01
      call moveFilterWheel

      ;call until both wheels are in the start position
      rjmp moveWheels

checkFilterWheelPosition:
      ;if the photo-interupt for filter is HIGH, then keep going
      ;if bit 0 of port C is set, skip next instruction
      sbis PINC, 0
      nop
      ori temp2, 0b01
      ret

checkSampleWheelPosition:
      sbis PINC, 1
      nop
      ori temp2, 0b10
      ret

moveSampleWheel:
      call checkSampleWheelPosition
      mov tempState, state
      lsr state
      lsr state
      lsr state
      lsr state
      swap state
      lsl tempState
      lsl tempState
      lsl tempState
      lsl tempState
      swap tempState
      call getNextState
      or state, tempState
      ;Now the filter part of the state register
      ;is unchanged, but the next state in sent
      ;out, so the sample wheel moves
      out PORTB, state
      ret

moveFilterWheel:
      call checkSampleWheelPosition
      mov tempState, state
      lsl tempState
      lsl tempState
      lsl tempState
      lsl tempState
      ;Find the next state
      call getNextState
      lsr state
```

```
      lsr state
      lsr state
      lsr state
      or state, tempState
      ;Now the sample part of the state register
      ;is unchanged, but the next state in sent
      ;out, so the filter wheel moves
      out PORTB, state
      ret

;*********************************************************
;* Calculates the next state given the current state in    *
;* tempState. Stops when the counter is reached                *
;*********************************************************
getNextState:
      cp steps, counter
      breq getNextState

      cpi tempState, 0b1010
      breq state1

      cpi tempState, 0b1001
      breq state2

      cpi tempState, 0b0101
      breq state3

      cpi tempState, 0b0110
      breq state0

state1:
      ldi tempState, 0b1001
      rjmp doDelay

state2:
      ldi tempState, 0b0101
      rjmp doDelay

state3:
      ldi tempState, 0b0110
      rjmp doDelay

state0:
      ldi tempState, 0b1010
      rjmp doDelay

doDelay:
      ldi loop1, 0x01
      ldi loop2, 0x01
      rjmp delay

delay:
      mov temp3, loop2

loopA:
      mov loop2, temp3
```

```
loopB:
      dec loop2
      brne loopB

      dec loop1
      brne loopA

      ret


;************************************************************
;* Main loop of the second measuring section                  *
;* calls all the methods to move the wheels and do         *
;* measurements
        *
;************************************************************
doMeasurements:
      ;steps is set here
      ;* 100 when i want sample to turn 180 deg
      ;* 67 to turn filter 120 deg
      ldi steps, 0
      call measureRedBlank       ; measure red blank
      call moveFilterWheel
      call doDelay

      ldi steps, 100
      call measureRedSample      ; measure red sample
      call moveSampleWheel
      call doDelay

      ldi steps, 67
      call measureGreenBlank          ; measure green blank
      call moveSampleWheel
      call doDelay

      ldi steps, 100
      call measureGreenSample         ; measure green sample
      call moveSampleWheel
      call doDelay

      ldi steps, 67
      call measureBlueBlank      ; measure blue blank
      call moveSampleWheel
      call doDelay

      ldi steps, 100
      call measureBlueSample          ; measure blue sample
      call moveFilterWheel
      call doDelay

      call calculate
      call transmit

      jmp loop
```

```
measureBlueBlank:
      call readADC
      mov BB, temp1
      ret

measureBlueSample:
      call readADC
      mov BS, temp1
      ret

measureRedSample:
      call readADC
      mov RS, temp1
      ret

measureRedBlank:
      call readADC
      mov RB, temp1
      ret

measureGreenBlank:
      call readADC
      mov GB, temp1
      ret

measureGreenSample:
      call readADC
      mov GS, temp1
      ret


;*************************************************************
;* Calculates the vaules, actualy does / or *                *
;*************************************************************
calculate:

      mov temp1, RS                   ;calculate red
      mov temp2, RB
      call divide
      call multiply
      mov redresult, temp4


      mov temp1, GS                   ;calculate green
      mov temp2, GB
      call divide
      call multiply
      mov greenresult, temp4


      mov temp1, BS                   ;calculate blue
      mov temp2, BB
      call divide
      call multiply
      mov blueresult, temp4
```

```
      ret

transmit:

      ldi temp1, 'R'             ;red result
      call sendChar
      ldi temp1, 'E'
      call sendChar
      ldi temp1, 'D'
      call sendChar
      ldi temp1, ':'
      call sendChar
      mov temp1, redresult
      call SendByte
      ldi temp1, 10
      call sendChar

      ldi temp1, 'G'             ;green result
      call sendChar
      ldi temp1, 'R'
      call sendChar
      ldi temp1, 'E'
      call sendChar
      ldi temp1, 'E'
      call sendChar
      ldi temp1, 'N'
      call sendChar
      ldi temp1, ':'
      call sendChar
      mov temp1, greenresult
      call SendByte
      ldi temp1, 10
      call sendChar

      ldi temp1, 'B'             ;blue result
      call sendChar
      ldi temp1, 'L'
      call sendChar
      ldi temp1, 'U'
      call sendChar
      ldi temp1, 'E'
      call sendChar
      ldi temp1, ':'
      call sendChar
      mov temp1, blueresult
      call SendByte
      ldi temp1, 10
      call sendChar

      ret



;************************************************************
;* Setup the USART, the clock divider is passed in r16:r17 *
```

```
;***********************************************************

SetupUSART:
      ;* Set the baud rate divider registers
      out   UBRRH, temp2          ; set the high byte of the serial clock
divider
      out UBRRL, temp1      ; set the low byte of the serial clock
divider

      ;* Enable the receiver and transmitter logic
      ldi   temp1, (1<<RXEN)|(1<<TXEN)  ; set the RX enable and TX enable
bits
      out   UCSRB, temp1          ;

      ;* Set the frame format to 8-bit data, 2 stop bits, no parity
      ldi temp1, (1<<URSEL)|(1<<USBS)|(3<<UCSZ0)  ; combine the required
bits
      out UCSRC, temp1

      ret                                 ; we're done (notice temp1 has
been modified in the routine)


;***********************************************************************
******
;* Sendchar: Waits until the USART is ready then sends a char (passed in
r16) *
;***********************************************************************
******

SendChar:

      ;* Wait until the USART transmitter buffer is empty
      sbis UCSRA, UDRE      ; 'skip next if bit (UDRE) in register
(UCSRA) is set (1)'
      rjmp sendchar               ; if not then go back to the start again

      ;* The USART is ready now so put the char in the output buffer
      out UDR, temp1

      ret


;***********************************************************************
******
;* Getchar: Waits until the USART has a char then get it (passed back in
r16) *
;***********************************************************************
******

GetChar:

      ;* Wait until the USART receiver has a char
      sbis UCSRA, RXC       ; 'skip next if bit (RXC) in register (UCSRA)
is set (1)'
      rjmp getchar          ; if not then go back to the start again
```

```
      ;* The USART has a char now so copy it into temp1 to pass it back
to the main program
      in temp1, UDR

      ret


;*****************************************************************
;* Convert the byte in r16 to ASCII hex and send it out the USART *
;*****************************************************************

SendByte:

      mov   temp3, temp1     ; take a copy of the byte
      ldi temp1, '0'         ; put the ASCII code for "0" in temp1 (this
is done by the assembler!)
      call SendChar
      ldi temp1, 'x'         ; put the ASCII code for "0" in temp1 (thsi
is done by the assembler)
      call SendChar

      mov temp1, temp3 ; restore temp1 from the copy in temp3
      swap temp1             ; swap the high and low nibbles of the byte
so the high one gets sent first!
      call Nibble2ASCII      ; convert the low nibble in r16 into ASCII
hex
      call SendChar          ; send the char to the USART

      mov temp1, temp3 ; restore the byte again
      call Nibble2ASCII      ; covert the low nibble this time
      call SendChar
      ret



;*************************************************************
;* Converts the lower nibble in r16 into an ASCII hex character *
;* The character is returned in r16 (uses r16 and r17)          *
;*************************************************************

Nibble2ASCII:
      andi temp1, 0x0F ; remove the high nibble
      cpi temp1, 0x0A        ; compare the nibble with 0x0A
      brsh isHex             ; branch to IsHex if nibble is A-F

      ldi temp2, 0x30        ; add 0x30 to the nibble for ASCII 0-9
      add   temp1, temp2     ;
      ret                          ; return for ASCII 0-9

isHex:
      ldi temp2, 0x37        ; add 0x37 to the value for ASCII A-F
      add temp1, temp2
      ret                          ; return for ASCII A-F

;*************************************************************
```

```
;* Setup the ADC module for 0 - 5V input, single-ended inputs *
;************************************************************

SetupADC:

      ;* Enable the ADC and set the clock signal used for the successive
      ;* approximation ADC logic to the main clock divided by 16
      ldi temp1, (1<<ADEN)+(9<<ADPS0)
      out ADCSR, temp1

      ret


;*********************************************************************
***********
;* Perform an ADC measurement on ADC channel in r16 (channel # = PORTA
bit number) *
;* An 8-bit value is returned in r16.
                              *
;*********************************************************************
***********

ReadADC:

      ;* First make sure r16 is a valid channel by truncating the high 5
bits
      andi temp1, 0b00000111           ; AND temp1 with binary 00000111
      ori temp1, (1<<REFS0)+(1<<ADLAR)      ;set the ref voltage source
and result justification
      out ADMUX, temp1           ; set the channel number and reference
voltage source

      ;* Now start the actual conversion process
      sbi   ADCSR, ADSC                ; set the 'start conversion' bit
(leave the rest as they are!)

      ;* Wait until the conversion is finished
readadcloop:
      sbic ADCSR, ADSC           ; skip the next instruction if bit ADSC
in ADCSRA regsiter is clear (the adc conversion is finished)
      rjmp readadcloop

      ;* Conversion is finished now so copy the high byte of the result
to r16
      in    temp1, ADCH                     ; even though the result is
actually 10-bit we are only using the high 8 bits

      ret


;************************************************************
;* Print out "Press s to start on the screen                        *
;************************************************************

startMessage:
      ldi temp1, 'P'
```

```
      call SendChar
      ldi temp1, 'r'
      call SendChar
      ldi temp1, 'e'
      call SendChar
      ldi temp1, 's'
      call SendChar
      ldi temp1, 's'
      call SendChar
      ldi temp1, ' '
      call SendChar
      ldi temp1, 's'
      call SendChar
      ldi temp1, ' '
      call SendChar
      ldi temp1, 't'
      call SendChar
      ldi temp1, 'o'
      call SendChar
      ldi temp1, ' '
      call SendChar
      ldi temp1, 's'
      call SendChar
      ldi temp1, 't'
      call SendChar
      ldi temp1, 'a'
      call SendChar
      ldi temp1, 'r'
      call SendChar
      ldi temp1, 't'
      call SendChar

      ;send carrage return and line feed
      ldi temp1, 10
      call SendChar
      ldi temp1, 13
      call SendChar


;************************************************************
;* Multiplies temp1 with 100, high result in r1 low in r0     *
;************************************************************
      multiply:
      ldi temp3, 0x64
      mul temp3, temp4
      movw temp4, temp3
      ret

;************************************************************
;* Divides temp1 and temp2, and puts result in temp1         *
;************************************************************

divide:

      sub   temp3,temp3           ;clear remainder and carry
      ldi   temp5,9               ;init loop counter
```

```
d8u_1:
      rol   temp1      ;shift left dividend
      dec   temp5      ;decrement counter
      brne d8u_2       ;if done
      ret                    ;return

d8u_2:
      rol   temp3      ;shift dividend into remainder
      sub   temp3,temp2      ;remainder = remainder - divisor
      brcc d8u_3       ;if result negative
      add   temp3,temp2      ;restore remainder
      clc                    ;clear carry to be shifted into result
      rjmp  d8u_1 ;else
d8u_3:
      sec                    ;set carry to be shifted into result
      rjmp  d8u_1
```