Pretty Lights
Transducing Sound into Visuals

Babatunde Falade, Michael Kane, Stacie Reynolds, Ryan Rychak
ECE-511 Microprocessors
Fall 2011
December 18, 2011

Abstract

The purpose of this project is to take in audio – specifically, music – and transduce that audio into control signals which drive a number of light-emitting diodes (LEDs). This is accomplished by feeding audio, through a line-in connection, directly to a Texas Instruments MSP430 microcontroller, which processes that audio using our developed software algorithm and subsequently generate 3.3V control signals to turn on four sets of LEDs (red, green, light green and yellow). The entirety of the system is enclosed in a project enclosure (purchased at Radio Shack), which itself is housed in a six-sided box of slightly translucent polystyrene sheets. In this manner, this translucent material allows the light to scatter (and not just appear as discrete units) and produces a visually pleasing effect. The project is truly for entertainment purposes only. Upon completion of the project, the main goal was accomplished and the project was wholly successful.

The report that follows details the work accomplished to achieve our goals, and how we went about accomplishing them. Some concessions had to be made and are expanded upon in this report.

TABLE OF CONTENTS

Motivation

The project members were inspired by work done by three co-ops working at Texas Instruments during the Summer of 2011. A summary of their project can be found here: http://e2e.ti.com/group/msp430launchpad/m/project/447779.aspx.

Given the scope of our project, and the time with which we had to complete it, we decided to scale the TI project down to complete it successfully. We decided that the required two external components we had to interface with the MSP430 would be the audio (through a line-in connection) and LEDs. We opted to use standard 3 and 5mm single-color LEDs, rather than the higher-end 4-lead RGB LEDs, to conserve power, as well as to cut costs. The design was powered by four D-cell batteries offering a 6.6VDC drop to take advantage of; the total drop would be used to power the LEDs while the drop provided by two of the four D-cells were used to provide power to the MSP430.

Music Analysis Algorithm

Pretty lights uses the MSP430 microcontroller to analyze music, and correspondingly turn on and off LED's. The algorithm used to read in the music and decide when the LEDs should be turned on lies within the microcontroller. The pretty lights team wrote and evaluated two different algorithms used to read in the music, and determine the output. These two algorithms are described below.

The first algorithm evaluated uses a simple threshold to control the output. The musical signal is first read into the microcontroller via the A/D converter. The A/D value is then tested against the threshold value. If the signal is greater than the threshold, the LED's will turn on. When the A/D value is less than the threshold, the LED's will turn off. The algorithm loops continuously, and does not use a timer to control looping/sampling frequency.

The output of the first device looks like a pulse width modulated LED output, where the LED is brighter when the energy in the music is higher. The LED is dimmer or off when the music has a low amount of energy. The reason for the PWM effect is due to the natural frequencies in the music. The music spans frequencies of around 100 Hz to 13 kHz. The microcontroller, even when looping continuously, is under-sampling the musical signal. Since the frequencies in the music are so high, when the microcontroller is sampling the music, the amplitude of any given sample is almost meaningless, but bounded by the max amplitude of the music. Sequential samples of the musical signal will cross the threshold, causing the LED's to turn on, or will be under the threshold causing the LED's to turn off. Since the threshold crossings occur so quickly, a single crossing is hardly noticeable with the eye, if at all. However, if a large percentage of some amount of samples are above the threshold, the LED will appear to be bright. Likewise, if only a few samples out of some amount are above the threshold, the LED will appear dim or off. While each sample is almost uncorrelated to the energy of the music, many samples in a given amount of time will have a higher correlation to the near instantaneous energy. In other words, if the energy in the music signal is high, then there is a better chance that a sample will be above the threshold, causing the LED to shine brighter.

This algorithm works well for a direct line fed into the A/D from a music source. It does not work will with a microphone though. Since the threshold in this algorithm is pre-programmed, it does not adjust to varying volume levels. If the volume of the music increases, the samples saturate with respect to the threshold, and the LED's are always on. If the volume decreases, the threshold is rarely or never crossed, and the LED's never come on. This problem could be solved a number of different ways, and is directly the motivation for the second algorithm.

Another issue with comparing samples of the music directly to the threshold is the flicker effect. The maximum frequency that the PWM effect of the LEDs can occur at is the sampling rate of the music, which is roughly 1.5 kHz. Modulating the LED's at 1.5 kHz can not be seen by the eye, but when the music amplitude is in a region where a very large or very small percentage of samples cross the threshold, the PWM frequency can be very low, often below 50 Hz. Frequencies below ~50 Hz can be seen by the naked eye and will appear as the LED's flickering rather than becoming brighter or dimmer. This issue is also solved in the second algorithm.

The second algorithm has the same concept as the first, with 2 new improvements which are meant to increase the robustness of the project, and potentially allow for use with a microphone. There are two main ways to increase the robustness with respect to amplitude (volume) of the input signal. The first is to use some metric to increase or decrease the threshold, such that saturation does not occur and the LED's do not become stuck in either the on or off state. The second way, and the one used in the final project, finds the long term average amplitude of the signal (previous 512 sample average) and subtracts it off of the current A/D input reading before comparing it to the threshold. If the volume of the input signal is high, the average amplitude will be somewhat large, with the positive difference being a manageable signal. If the input volume is low, the average is low, and the signal is on the same order of magnitude as the large signal difference. This allows the pre-set threshold to maintain validity for a wider range of signals, and keeps the device out of saturation. The second algorithm, like the first, runs on an uncontrolled continuous loop. The additional math causes the device to sample at approximately 1 kHz, using a 1 MHz clock.

Another improvement in the second algorithm is aimed at solving the issue with LED flickering. Simply by taking a short term average (previous 16 sample average) of the input signal, the number of threshold crossings drops dramatically, while still maintaining correlation between signal energy and LED output. The second algorithm combines these two problem solving methods by taking the difference of the short term average and long term average. The difference is then checked against the threshold to determine the output. This is essentially the smoothed variance of the input signal.

LED Hold Algorithm

The two original Pretty Lights algorithms showed very promising results. Once issue that arose after evaluating the second algorithm was simple the amount of LED action, or business of the LEDs. The LED output tracked the music well, however, with quick variations in the energy of the signal, the LED's would be turning on and off a lot, even with the smoothing to reduce flicker. A simple solution to this was to low pass filter the LED output signal itself. An algorithm which holds the LED's in their current state for some time assists with this issue. The algorithm works initialized by setting timers (which simply amount to counters) to force the LED's to stay on for a few milliseconds when the output is positive. After this few milliseconds, if the output of the threshold is negative, the LED's will turn off, and another counter will hold them off for a few milliseconds. After the hold-off time is up, the LED's are free to turn on at the next major crossing of the threshold. The hold-on and hold-off counter are respectively initialized at each LED transition. The appropriated counter is then incremented once for each cycle of the software. The value of the counters is set in the firmware.

LED Color Change Algorithm

Pretty lights uses 3 different colors of LED's, red, green/yellow, and blue. In order to change the appearance of pretty lights over time, 8 different LED color sequences are implemented with 2 different modes of operation. This is intended to give the uses the appearance of color variation over time. A single LED color sequence lasts 16 seconds before rotating to a difference color scheme. The 16 seconds is counted by one Timer_A module.

The difference color sequences are blue constant, red and blue switched, red and green constant, blue and green constant, red constant, green, red and green switched, and blue and green switched. The sequence describes which colors are active during a segment of music. The two different modes are switching and constant mode. The mode describes how the LED's act when music causes them output to turn LED's on.

In constant mode, one or both of the colors active (according to sequence) turn on whenever the output of the algorithm tells them to. If there are two colors, they come on at the same time, and are off at the same time. In switched mode, there are 2 LED colors which are indicated by the current sequence. The two colors switch turns being on, and alternate on every crossing of the threshold. The algorithm keeps track of which LED was previously used, and on the next threshold crossing, the alternate color is used. Both modes still use the hold-on counter, however, the switch mode does not use the hold-off counter as it is easier to discriminate LED colors changing than a single LED turning on and off at the same rate.

Implementing Algorithms in the MSP430

The microcontroller used is the MSP430G2553. This microcontroller is the top of the line of the G series and contains two Timer_A modules, a 10 bit A/D, 512 Bytes of RAM and 16 kB of flash memory. The 4 major features of the microcontroller used were the DCO internal clock, the 10 bit A/D, one Timer_A, and the I/O port functionality to control the LED's.

The clock source for this project is the internal DCO, calibrated for approximately 1MHz. Since we are not controlling the exact sampling frequency, precision and accuracy of the internal clock are not of much concern. The MCLK thus runs at 1MHz, the SMCLK is divided by 8 to 125 kHz, and the ACLK is not divided down and is not used in this project.

The analog to digital converter is perhaps the most critical piece to set up. The A/D is set up in single sample (non-repeat) mode. The voltage source is selected to be the 1.5 volt internal reference. As the music out of a computer or MP3 Player is on the order of 0.5-1.0 volt, the 1.5 volt internal reference allows for the best use of the dynamic range. The ADC10 contains it's own internal clock reference, which is used because it is roughly five times faster than the MCLK signal.
The Timer_A module is the next configuration hurdle for pretty lights. The function of this peripheral is to switch the LED color scheme/mode every 16 seconds roughly. The SMCLK, divided again by 8 to yield 15625 Hz. In count up mode, with TACCR0 set to 62500, the timer will fire every 4 seconds and interrupt the program. Inside the timer is a counter which will change the LED scheme every 4th timer. This results in a new color scheme every 16 seconds.

The last peripherals used within the microcontroller are the input/output ports. These ports control the LED's, turning them on and off. These ports are set up in two different ways depending on the LED mode, switching or continuous. In continuous mode, the output ports corresponding to the correct color are set to output. The output direction is then set to Vcc whenever the LED should be turned on. When the LED is to be turned off, the ouput port is set to ground. LED switch mode is slightly more complicated. In this mode, the LED ports pertaining to the 2 colors are set to low upon initialization of the color mode. As there are two colors always used in switch mode, one of the ports corresponding to one color is set to output, one is set to input. Once the threshold is crossed, setting the corresponding LED ports high will only turn on one of the colors. After this color is held on, then held off, the register P1DIR, is XOR'd with the bits corresponding to the two active colors of LED's. This switches the inputs output and output to input. The next time the music crosses the threshold, the music, the other color is turned on. This cycle repeats for the full 16 seconds, until the color scheme and mode change again.

Results

Compared to the initial goal of the design, Pretty Lights was an overall success. The goal of the project is to convert the sound of music into visual effects by blinking LED's to the beat of the music, as well as make to unit move to the sound of the music with the use of a motor. Overall, that goal is accomplished. A few underlying details have been adjusted to allow the overall goals to be met.

Microphone

The original plan of pretty lights was to make it as user friendly as possible. This original idea included using a microphone do deliver the music to the A/D converter. This would make the device portable and give it flexible placement throughout the room where the music is being played. After a few rounds of

testing and evaluated, the idea of the microphone was changed to a direct feed from a computer, mp3 player, or phone. This decision has been made for a few reasons. First, the microphone would pick up other noise in the area, not necessarily associated with the music. Pretty lights would take in this noise and the outcome would be difficult to determine or control. Second, the microphone was very sensitive to changes in distance from the music source. The amplitude of the music could vary wildly when the microphone was moved from close to the source, to far away. This change to pretty lights ultimately allowed the end product to be more robust and more predictable.

Battery versus AC Power

The original plan to use AC power for Pretty Lights was necessary in order to power a large number of LEDs. The Red LEDs alone have the smallest forward voltage of any visible color at 2V, as well as the smallest current requirement of 16-18 mA. The original plan to have possibly up to 100 LEDs of various colors (including blue and green which have a higher forward voltage and require greater current than red LEDs) would drain several D cell batteries quickly.

Due to a lack of time available to connect such a large number of LEDs as well as incorporate the AC power module and regulator, the decision was made to use batteries and reduce the number of LEDs used to prevent quick draining

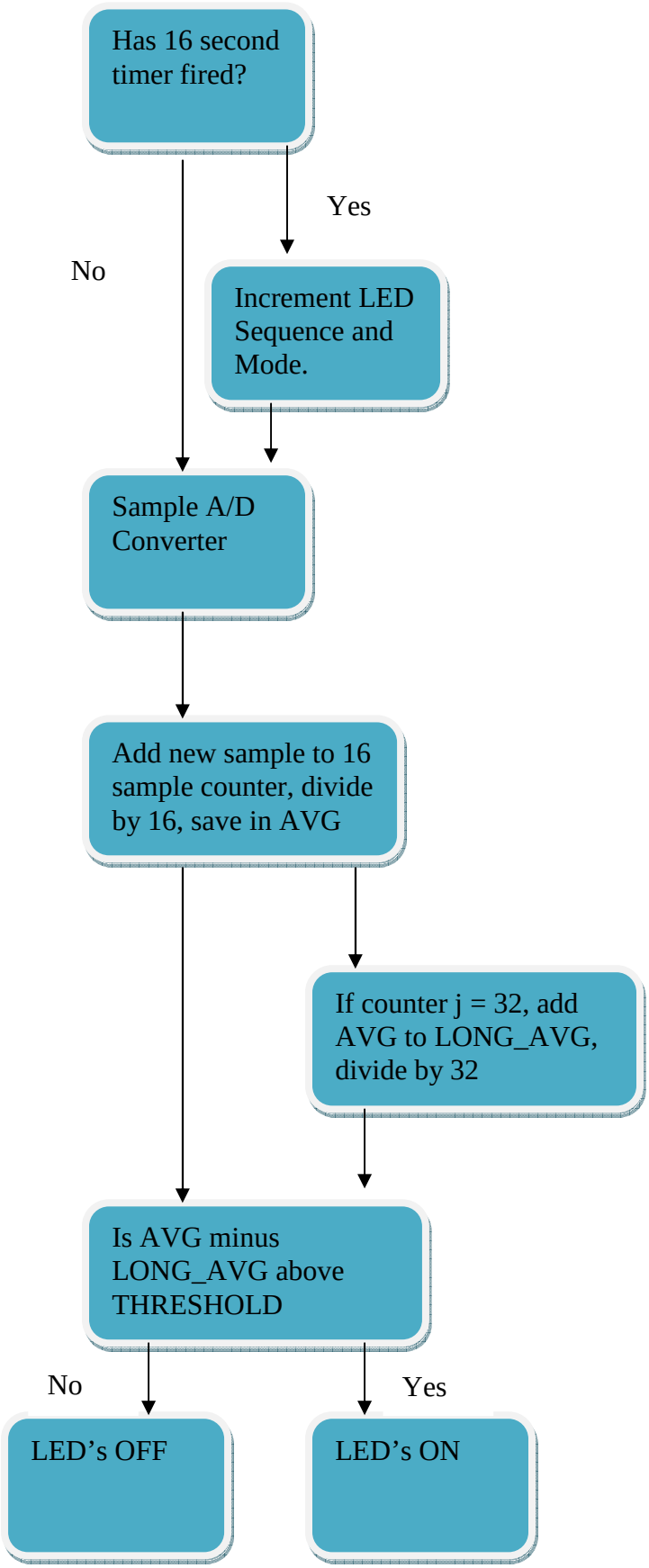
Stepper Motor

The stepper motor is used to allow the Pretty Lights to have motion at the beats of the music in addition to the lights blinking. This was a great idea, but became a challenge to integrate. The design of the Pretty Lights unit was fairly large, and possibly to large for the stepper motor to be able to rotate. Even without size being a factor, another challenge was determining where the stepper motor would fit in the unit, and how to connect the gears to the internal casing to allow the LEDs to rotate. This led the Pretty Lights design team to make another conscious decision to remove movement from the design.

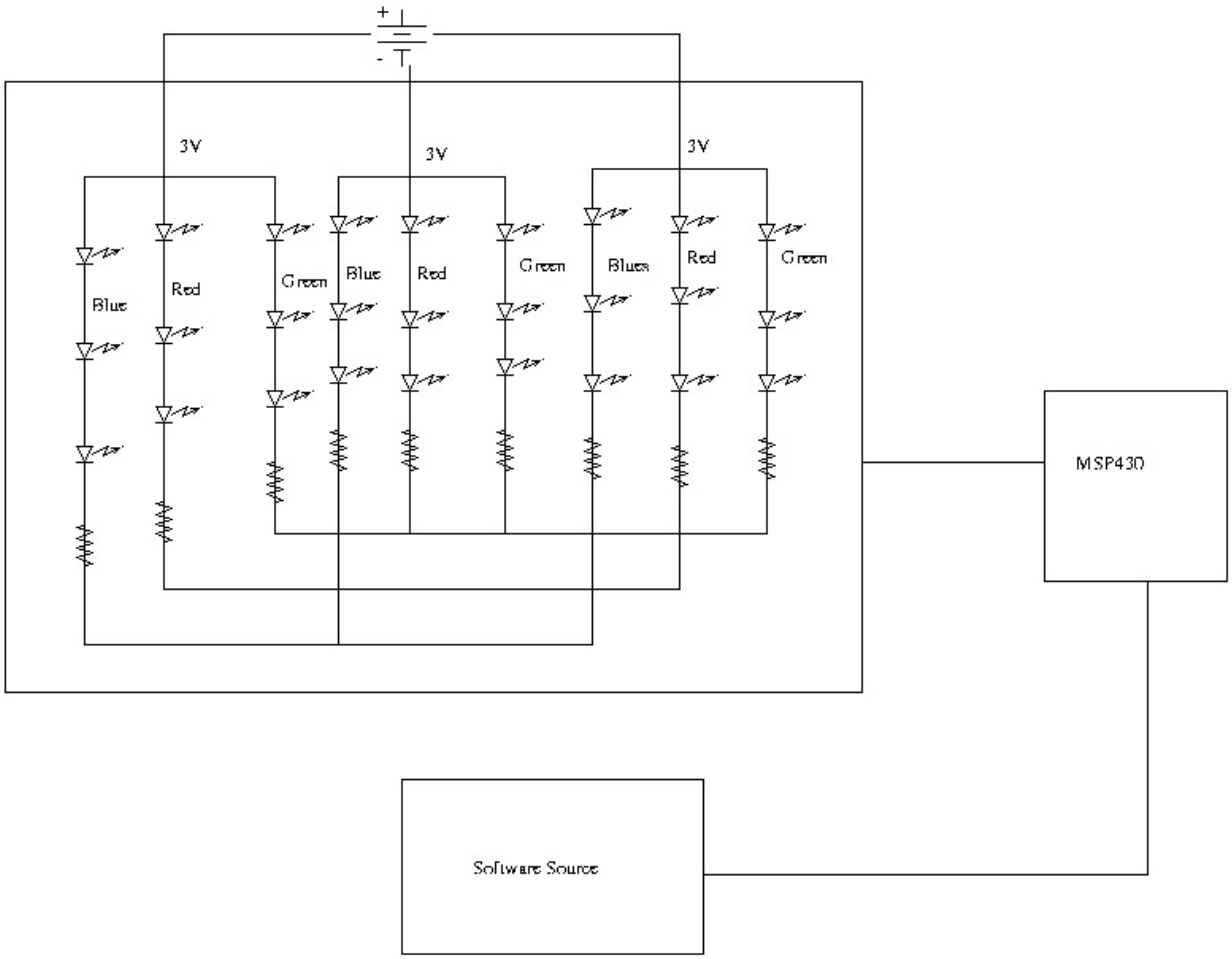
Conclusion

Overall, Pretty Lights was a successful project since it performs exactly as its title portrays: “Pretty Lights: Transducing Sound Into Visuals”. As demonstrated, music can be fed into the Pretty Lights system, and the various colors of light flash at a specific sequence according to the beat of the music. Very few projects are completed without the design team being required to make some executive decisions to the overall design in order to meet the specified goal. Pretty Lights is no exception. Even with the concessions made, the overall design was a success and should be a new edition to everyone’s home theater system.

Music Algorithm Functional Block Diagram



Schematic


+
-
3V
3V
3V
Blue
Red
Green
Blue
Red
Green
Blues
Red
Green
MSP430
Software Source

Team Members for the Project

1) Babatunde Falade
2) Michael kane
3) Ryan Rychak
4) Stacie Reynolds

Task assigned to each member of the group

- Parts Order (Michael kane and Ryan Rychak)
- Project layout (Ryan Rychak)
- Software design (Tunde Falade and Ryan Rychak)
- Hardware design and Integration (Tunde Falade, Michael Kane, Ryan Rychak and Stacie Reynolds)
- Testing and Implementation (Tunde Falade, Michael Kane, Ryan Rychak and Stacie Reynolds)

Parts List

1)	MSP430G2553	Free
2)	6x 2N6427 Transisitors	$0.66
3)	Resistors(29	Free
4)	ASMT-LX50 LEDS	
	-10x red	Free
	-4x blue	Free
	-15x green	Free
5)	3x Vector board	$1.99
6)	6x Fiber glass Pieces(for the casing)	$11.94
7)	Computer Speakers for Audio	Free

CCS C Code

```
#include <msp430g2553.h>
#include <intrinsics.h>
#include <stdbool.h>

unsigned long input;
unsigned int i,j;
char k = 0;
char l = 0;

long avg;
long sum;
int sample [32];
int total [16];
int diff;
int threshold=0;

int Mode = 8;//8

bool change_mode = true;
bool Switch = false;
int holdon,holdonsamples,holdoff,holdoffsamples;
#define Red             BIT0 // 00000001
#define Green           (BIT1|BIT2) // 01000000
#define Blue            BIT3
char Output = 0;

void main(void){

        WDTCTL = WDTPW | WDTHOLD;  // addresses WDTPW and sets hold bit to disable

        //P1DIR |= Red + Green;  // Sets P1.0 and P1.6 to output direction (high)

        //P1OUT |= LED1 ;  //  Or's p1out with 00000001
        //P1OUT &= ~(Red + Green);  // Sets LED1 and LED2 off

        BCSCTL1 = CALBC1_1MHZ;  //Calibrates DCO for 1MHz
        DCOCTL = CALDCO_1MHZ;
        BCSCTL2 = DIVS_3;   //Divide by 1 for SMCLK so 125kHz

        TACCR0=62500; /// this needs to be set for FS or 1000 times per second
        TACCTL0=CCIE;
        TACTL=TASSEL_2 | ID_3 | MC_1 | TACLR; //SMCLK, Divide by 8, Count up, Clear
TAR register

        //ADC10CTL0 = SREF_1|ADC10SHT_2|REFON|ADC10ON; //use 1.5v/VSS as ref,
Sample and hold 8 cycles
        ADC10CTL0 = SREF_0|ADC10SHT_2|ADC10ON; //use Vdd/Vss as ref, Sample and
hold 4 cycles
        ADC10CTL1 = INCH_5|ADC10DIV_0|ADC10SSEL_0;// input channel A5, divide
system clock by 0,

        _enable_interrupts();
        while(1){
                if (change_mode==true){
```

```
change_mode=false;
Mode++;
if (Mode>=9){
        Mode=1;
        Output = Red;
        P1DIR = Red;
        Switch = false;
        holdonsamples=250;
        holdoffsamples=200;
}
else if (Mode==2){
        P1DIR = Red;
        Output = Red + Green; //Blue
        Switch = true;
        holdonsamples=300;
        holdoffsamples=20;
}
        else if (Mode==3){
        Output = Green;
        P1DIR = Green;
        Switch = false;
        holdonsamples=250;
        holdoffsamples=200;
}
        else if (Mode==4){
        P1DIR = Green;
        Output = Green+ Blue;// Blue
        Switch = true;
        holdonsamples=300;
        holdoffsamples=20;
}
        else if (Mode==5){
        Output = Blue; // Blue
        P1DIR = Blue;  //Blue
        Switch = false;
        holdonsamples=250;
        holdoffsamples=200;
}
        else if (Mode==6){
        Output = Red + Green;
        P1DIR = Red;
        Switch = true;
        holdonsamples=300;
        holdoffsamples=20;
}
        else if (Mode==7){
        Output = Red + Green; //Blue
        P1DIR = Red + Green;  //Blue
        Switch = false;
        holdonsamples=250;
        holdoffsamples=200;
}
        else if (Mode==8){
        Output = Red + Blue;
        P1DIR = Red;
        Switch = true;
```

```
                                        holdonsamples=300;
                                        holdoffsamples=20;
                                }
                        }
for(j=0;j<16;j++){
for(i=0;i<32;i++){
        ADC10CTL0 |= ENC + ADC10SC;
        //delay_cycles(20);
        sample[i] = ADC10MEM;
        sum = 0;
for(l=0;l<32;l++){
sum=sum+sample[l];
}
sum >>=5;

        if(i==31){
                total[j]=sum;
                avg = 0;
                for(l=0;l<16;l++){
                        avg=avg+total[l];
                }
                avg >>=4;
        }

        if(sum>avg){
                diff=sum-avg;
        }
        else {
                diff=0;
        }

//if (avg<20){
//threshold = 2;
//}
//if (avg>40){
//threshold = 20;
//}
//if (avg >60){
//threshold = 40;
//}


if (avg<=2){
threshold = 2;
}
if ((avg>2)&&(avg<=8)){
threshold = 7;
}

if ((avg>8)&&(avg<=20)){
threshold = 12;
}
if (avg>20){
threshold = avg-3;
}
if (avg>40){
```

```
threshold = avg - 8;
}
if (avg >60){
threshold = avg - 12;
}

                        if ((sum > threshold) && (holdoff<1) && (holdon<1)){

                                P1OUT = Output;
                                holdon=holdonsamples;//250

                        }
                        if((sum<=threshold) && (P1OUT == (Output)) && (holdon<1)){
                                if((Switch== true)){
                                P1DIR ^= Output;
                                }
                                P1OUT = 0;
                                holdoff=holdoffsamples;//200
                        }
                        if (holdon>0){
                                holdon--;
                        }
                        if (holdoff>0){
                                holdoff--;
                        }

}//for i<32
}//for j<32
        }//while 1
}// main

//********** timer Interupt *****************
#pragma vector = TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR (void){

        k++;
        if (k>=4){
                k=0;
        change_mode = true;
        }

}
```