EEC 172 Lab5

Richard Szeto & Sean Ho

Monday, March 18, 2013

1 Objectives

1.1 Part 1

The objective for part 1 is to control the rpm of the motor using the IR remote and display the rpm on the OLED display.

1.2 Part 2

The objective for part 2 is to calibrate the accelerometer to 2g standards and output the vector corresponding to the tilt of the accelerometer in terms of gravity.

1.3 Part 3

The objective for part 3 is to use the fast Fourier transform to manually control the rpm of the motor.

2 Procedures

2.1 Part 1

We know that the rpm at 100% duty cycle is 2190. We want to bound the output duty cycle from 10% to 90%. By default, we want the duty cycle to be at 50%. Theoretically the rpm corresponding to these duty cycles are 219 rpm at 10% duty cycle, 1095 rpm at 50% duty cycle, and 1971 rpm for 90% duty cycle. The IR remote will be used to to increment/decrement the duty cycle within the indicated bounds to produce the corresponding rpm of the motor.

2.2 Part 2

To calibrate the accelerometer, we modified the code that was provided in the Freescale AN3468 Application Note. After calibration, we set the appropriate offsets of the x,y, and z planes to indicate that the default tilt exhibits gravity only on the z plane.

2.3 Part 3

To transform the vibrational frequency data of the motor to a rpm, the fast Fourier transform functions provided were used. The register holding the value for the x plane was used to create the vibrational frequency data. The register was constantly being polled to produce dynamic rpm outputs.

3 Implementation

We imported most of our code from previous projects, giving us the ability to read IR signals and to easily output a PWM signal. After that we added I2C capabilities in our initialization and added two functions that allowed us to read and write into registrars. We then copied the calibration code into our code to find the values to put in the offset registrars, however they were removed after we found the appropriate values.

Finally, we implemented a timer interrupt that allowed us to poll the values in the accelerometer at a rate of 125 Hz. Every time it polls it put the data in an 128 size array. When the array was full we would run a FFT on the dataset with the given code and put the calculated RPM in a global variable.

We then added the ability for the user to add requested rpm's. For that we took values from the remote and stored it into an array. After the last button was pressed, we took the string and found the rpm desired. We would then guess the pwm value based on a ratio of the theoretical maximum values. Then as the timer polls

it would check if the calculated rpm was within 10% of the desired rpm and if it wasn't it would increase or decrease the pwm accordingly.

4 Screenshots

Figure 1: An I2C transmission

We forgot to convert the register address into hexadecimal.

- ':' represents register 0x3A
- ';' represents register 0x3B

From the lecture slides, we know that:

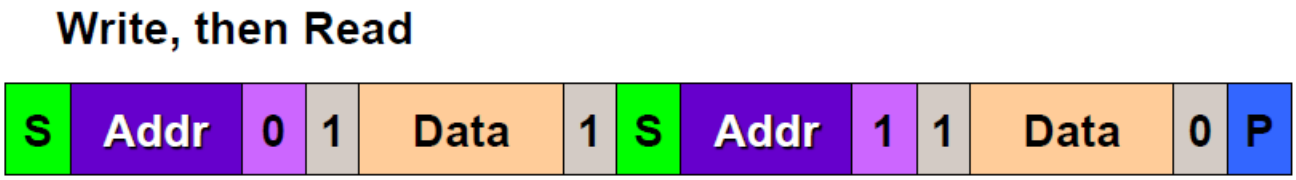

Figure 2: Write then Read in an I2C transmission

Corresponding to the logic analyzer screenshot, the first two dots represent the start bits and the last dot represents the stop bit. The write[0x3A] represents the address that is to be written to. The ACK following it represents that register 0x3A is ready. '6' is the data that is written to register 0x3A. The ACK following the data confirms that the data is written. The read[0x3B] represents the address that is to be read from. The ACK following it represents that register 0x3B is ready. '255' is the data that is read from register 0x3B. The NAK following it confirms that the data is read and it is the end of transmission.

5 Difficulties

5.1 Part 1

Initially, we did not fully grasp how to implement the bounds of the duty cycle. When our duty cycle reached 10%, it would still be allowed to decrease below 10%, making the motor generate erratic behavior. The rpm would roll around from 10% to 90% when we tried to decrement the duty cycle below 10%.

5.2 Part 2

When the calibration code was implemented, we did not fully understand that the calibration was only used to find the offsets. We were confused about why the values that were being output from the calibration code did not reflect our intended vector for the default tilt. Once we realized that the calibration code was only to be used as a tool for finding the offsets, the modified outputs fit our desired model.

5.3 Part 3

We had problems using the infinite while loop in main from previous labs that propagated to lab5. To find a work around, we used Timer to simulate an infinite while loop and wrote our code in the Timer interrupt that would have been in the infinite while loop.

We had trouble understanding what the fast Fourier transform functions did and what the inputs and outputs were for each of those functions. We did not understand exactly what needed to be done until we treated the functions as black boxes and analyzed the main.c that was provided with the functions. Once the vibrational frequency data was transformed into rpm, we had to allow manual control of the rpm. This brought a new problem where we needed to implement a self-correcting algorithm that will eventually produce a rpm that matched the rpm that was input from the IR remote.

A Code for Lab5

A.1 LM3S8962

```
//  Lab5 Code for LM3S8962
//    Sean Ho & Richard Szeto

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/pwm.h"
#include "driverlib/sysctl.h"
#include "drivers/rit128x96x4.h"
#include "inc/lm3s8962.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "drivers/rit128x96x4.h"
#include "driverlib/systick.h"
#include "driverlib/uart.h"
#include "driverlib/i2c.h"
#include "driverlib/timer.h"
#include <stdio.h>
```

```
#include <stdlib.h>

#define CODE_SIZE 16            // size of sequence from controller
typedef char byte;
//#define DATA_SIZE 128
#define bool          int
#define true          1
#define false         0

volatile int waitTime;
volatile int waitTime2;
char display[50];             // OLED display buffer
char display2[50];
int dLocx=0;                  // OLED display buffer offset
int flag = 0;                 // flag for button press delay
volatile int screenX = 0;       // OLED display x coordinate
volatile int screenY = 0;       // OLED display y coordinate
volatile int period;
volatile unsigned long maxPeriod;
int code[CODE_SIZE];          // array to hold sequence from controller
int code_offset = 0;          // placeholder for position in sequence array
char string[CODE_SIZE];       // array used to convert int to ascii
int motorOn =0;
//prototypes
char decode (char* input);      // interpret sequence data
void decodeLetter (char input);     // output corresponding character to OLED display
int getDigit (void);          // return binary number corresponding to pulse delay
volatile int dataCount =0;
#define NN 128
volatile int x[NN];  // input array
volatile int y[NN +2];  // one extra element
volatile unsigned short w[NN/2]; //first half of symetrical window 0Q16 unsigned
int rpm =0; //calculated rpm
int desiredRpm=-1; //changes with user input, -1 before user input
signed char signedByteAccelRead(int reg);
unsigned long byteAccelRead(int reg);
void changelessSpin (int);



//displays Accel values
displayAccel()
{
  unsigned char Xdata, Ydata, Zdata;
  char str[10];
    Xdata = signedByteAccelRead(0x06);
    sprintf(str, "%hhd   ", Xdata);
    RIT128x96x4StringDraw(str, 0, 17, 15);
    Ydata = signedByteAccelRead(0x07);
    sprintf(str, "%hhd   ", Ydata);
    RIT128x96x4StringDraw(str, 0, 25, 15);
    Zdata = signedByteAccelRead(0x08);
    sprintf(str, "%hhd   ", Zdata);
    RIT128x96x4StringDraw(str, 0, 33, 15);

    SysCtlDelay((SysCtlClockGet() / 3) /10);

}

//finds max value in array and returns index for fft
int findMax (int* ar)
{
```

```
  int num=-10000, i, index=0;
  for (i=0;i<NN;i++)
    if(ar[i]>num)
    {
      index = i;
      num = ar[i];
    }

    return index;
}

//polls at 125 hz, fills array and runs fft + display when full
void TimerIntHandler ()
{
  signed char c;
  char s[20];
  int i, maxIndex, f, num;
  float fl;
  if(dataCount < 128)
    x[dataCount++]= signedByteAccelRead(0x06);
  else
  {

//create windowing function 0Q16  (0.5 =32768   0.99998 =65535)
for (i=0;i<NN/2;i++)   w[i]=0xffff;   //0.99998
//clear output array
for (i=0;i<NN;i++)   y[i]=0;
    Window16to32b_real( x, w, NN); //or just expand to 32 bits with 1/2 scale
    FFT128Real_32b(y,x);
    for(i=0;i<NN;i++)
    magnitude32_32bIn(&y[2],NN/2-1);
    for (i=0;i<=NN;i+=2) y[i+1]=0; //zero imag part of original complex frequency
    for (i=0;i<=NN;i+=2) y[i]=((y[i]+16384)>>15); //2Q30 converted to 17Q15 and rounded
    dataCount = 0;
    maxIndex = findMax(y);
    fl = maxIndex;
    fl = (fl/128) * (125/2);
    if(fl*60 < 2500  && fl*60 > 219)
      rpm = fl *60;
    sprintf(s, "%d     ", rpm);
    RIT128x96x4StringDraw(s, 0, 60, 15);
    displayAccel();

    if(desiredRpm != -1)
    {
    if(rpm && desiredRpm - rpm > 219)
      changelessSpin(-100);

    if(rpm && desiredRpm - rpm < -219)
      changelessSpin(100);
    }

  }
  TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
}


//displays Error
void showError (void) {          // OLED display

    RIT128x96x4StringDraw("Error", 50, 25, 15);
```

```
}

int checkProtocol(){              // skip repeating pulse sequences

    while(GPIOPinRead(GPIO_PORTB_BASE, GPIO_PIN_1))
    {
    }//90
    if(waitTime <80)
        return 0;
    waitTime=0;
    while(!GPIOPinRead(GPIO_PORTB_BASE, GPIO_PIN_1))
    {
    }
    waitTime=0;
    while(GPIOPinRead(GPIO_PORTB_BASE, GPIO_PIN_1))
    {
    }//24
    if(waitTime <20)
        return 0;
    else
        return 1;

}

// compare last 8 bits of IR pulse sequence
int myStrCmp(char string2[], char given[])
{
    int i;
    for(i=0; (i < 8) && (string2[i] == given[i]); i++);

    if(i && (i == 8) && (given[i]=='\0'))
            return 1;
    else
            return 0;
}

//get Digit from IR
int getDigit ()                   // determine binary number from pulse delay
{
    waitTime=0;
    while(!GPIOPinRead(GPIO_PORTB_BASE, GPIO_PIN_1))
    {
    }
    if(waitTime >6)
        return 2;
    waitTime=0;
    while(GPIOPinRead(GPIO_PORTB_BASE, GPIO_PIN_1))
    {
    }
    if(waitTime>14)
        return 1;
    else
        return 0;

}

void getData()                    // collect binary numbers
{
    int i, k;
    k =getDigit();
    code_offset=0;
    for(i=0; i<16;i++)
```

```
    {
        if(i>=8)
            string[code_offset++]=k + '0';

        k=getDigit();
    }
}

void PortBIntHandler (void) // Interrupt handler called upon IR receive
{
    char str[10];

    // Reset delay counter
    waitTime=0;


    // skip repeating IR pulse sequences
    if(!checkProtocol())
    {
      GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_0,0x00);
      GPIOPinIntClear(GPIO_PORTB_BASE, GPIO_PIN_1);
        //displayAccel();
        return;
    }
    // 2 second delay between button presses
    if(waitTime2 < 20000)
    {
        flag = 1;
    }
    else
    {
        flag = 0;
    }

    // parse IR pulse sequence data
    getData();


    // concatenate corresponding character to display buffer
    decodeLetter(decode(string));
    RIT128x96x4StringDraw(display2, 0, 80, 15);

    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_0,0x01);

    // reset delay between button presses
    waitTime2=0;
    GPIOPinIntClear(GPIO_PORTB_BASE, GPIO_PIN_1);
    //displayAccel();
}



//increments waitTime and waitTime2 every period
void SysTickHandler (){
    waitTime += 1;
    waitTime2 += 1;
}

void itos4(int temp,int offset, int ub) {
    if(ub - offset + 4 >= 0){
        display[offset] = (temp/1000) + '0';
        dLocx++;
```

```
        temp = temp - (temp/1000)*1000;
        display[offset+1] = (temp/100) + '0';
        dLocx++;

        temp = temp - (temp/100)*100;
        display[offset+2] = (temp/10) + '0';
        dLocx++;

        temp = temp - (temp/10)*10;
        display[offset+3] = temp + '0';
        dLocx++;

        display[offset+4] = '\0';
    }
}

void changelessSpin (int change)//changes PWM without changing display or theoretical period
{
  int period2 = period;
  int num = period + change;
  if(change > 0 && num > maxPeriod *9/10)
    period2 = maxPeriod *9/10;
  else if(change < 0 && period + change < maxPeriod/10)
    period2 = maxPeriod/10;
  else
    period2 += change;

  PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, period2);
}

void spin (int change)
{
  float toDisplay;
  int num = period + change;
  if(change > 0 && num > maxPeriod *9/10)
    period = maxPeriod *9/10;
  else if(change < 0 && period + change < maxPeriod/10)
    period = maxPeriod/10;
  else
    period += change;
  toDisplay = period;
  toDisplay = toDisplay/maxPeriod * 2190;
  num = toDisplay;
  itos4(num, 0, 16);

  PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, period);
  RIT128x96x4StringDraw(display, 0, 0, 15);
}

void spinManual (int rpm2)
{
  float toDisplay;
  float ratio;
  int num;
  if (rpm2 > 1971) //90%
    rpm2 = 1971;
  else if (rpm2 < 219) //10*
    rpm2 = 219;

  desiredRpm= rpm2;
  ratio = rpm2/2190.0;
```

```
  period = maxPeriod * ratio;

  toDisplay = period;
  toDisplay = toDisplay/maxPeriod * 2190;
  num = toDisplay;
  itos4(num, 0, 16);

  PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, period);
  RIT128x96x4StringDraw(display, 0, 0, 15);

}

void accelWrite (int reg,char data)
{
    // Sets the slave address
    I2CMasterSlaveAddrSet(I2C_MASTER_BASE, 0x1D, false);
    // Sends the register we want
    I2CMasterDataPut(I2C_MASTER_BASE, reg);
    I2CMasterControl(I2C_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_START);
    while(I2CMasterBusy(I2C_MASTER_BASE));



    // Set the register value
    I2CMasterDataPut(I2C_MASTER_BASE, data);
    I2CMasterControl(I2C_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH);
    while(I2CMasterBusy(I2C_MASTER_BASE));
}

signed char signedByteAccelRead(int reg)
{
    short temp;
    I2CMasterSlaveAddrSet(I2C_MASTER_BASE, 0x1D, false);
    I2CMasterDataPut(I2C_MASTER_BASE, reg);
    I2CMasterControl(I2C_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_START);
    while(I2CMasterBusy(I2C_MASTER_BASE));
    I2CMasterSlaveAddrSet(I2C_MASTER_BASE, 0x1D, true);
    I2CMasterControl(I2C_MASTER_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);
    while(I2CMasterBusy(I2C_MASTER_BASE));
    temp = I2CMasterDataGet(I2C_MASTER_BASE);
    return (signed char)(temp&0x00FF);



}


unsigned long byteAccelRead(int reg)
{
    short temp;
    I2CMasterSlaveAddrSet(I2C_MASTER_BASE, 0x1D, false);
    I2CMasterDataPut(I2C_MASTER_BASE, reg);
    I2CMasterControl(I2C_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_START);
    while(I2CMasterBusy(I2C_MASTER_BASE));
    I2CMasterSlaveAddrSet(I2C_MASTER_BASE, 0x1D, true);
    I2CMasterControl(I2C_MASTER_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);
    while(I2CMasterBusy(I2C_MASTER_BASE));
    return I2CMasterDataGet(I2C_MASTER_BASE);
```

```
}

int
main(void)
{
    unsigned long ulPeriod;
    unsigned long ReadData;
    char str[10];

    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_8MHZ);
   // Initialize the OLED display and write status.
    RIT128x96x4Init(1000000);
    RIT128x96x4StringDraw("------------------------", 0, 50, 15);


    // Status
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_STRENGTH_4MA, GPIO_PIN_TYPE_STD);
    GPIODirModeSet(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_DIR_MODE_OUT);


    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    // I2C
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
    I2CMasterInitExpClk(I2C_MASTER_BASE, SysCtlClockGet(), false);


    GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_2 | GPIO_PIN_3);

    accelWrite(0x16, 0x05);
    ReadData = byteAccelRead(0x16);
    sprintf(str, "%u", ReadData);
    RIT128x96x4StringDraw(str, 0, 10, 15);


    SysCtlPWMClockSet(SYSCTL_PWMDIV_8);
    //pwm
    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
    GPIOPinTypePWM(GPIO_PORTB_BASE, GPIO_PIN_0);
    PWMGenConfigure(PWM_BASE, PWM_GEN_1,
    PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
    ulPeriod = SysCtlClockGet()/100/8;
    period = ulPeriod/2;
    maxPeriod = ulPeriod;
    PWMGenPeriodSet(PWM0_BASE, PWM_GEN_1, ulPeriod);
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, ulPeriod/2);
    //PWMPulseWidthSet(PWM_BASE, PWM_OUT_1, SysCtlClockGet() * 0.0015/8);
    PWMGenEnable(PWM0_BASE, PWM_GEN_1);
    //on/off
    //PWMOutputState(PWM0_BASE, (PWM_OUT_2_BIT | PWM_OUT_3_BIT), true);


    //PB1
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    GPIOPadConfigSet(GPIO_PORTB_BASE, GPIO_PIN_1, GPIO_STRENGTH_4MA, GPIO_PIN_TYPE_STD);
    GPIODirModeSet(GPIO_PORTB_BASE, GPIO_PIN_1, GPIO_DIR_MODE_IN);
    IntEnable(INT_GPIOB);
    GPIOIntTypeSet(GPIO_PORTB_BASE, GPIO_PIN_1, GPIO_BOTH_EDGES);
    GPIOPinIntEnable(GPIO_PORTB_BASE, GPIO_PIN_1);
    IntPrioritySet(INT_GPIOB,0x80);
```

```
    SysTickPeriodSet(SysCtlClockGet()/10000);    // 0.1ms
    SysTickIntEnable();
    waitTime = 0;
    waitTime2 = 0;
    SysTickEnable();

    //Timer
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
    TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet()/125);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    IntEnable(INT_TIMER0A);
    TimerEnable(TIMER0_BASE, TIMER_A);
    IntPrioritySet(INT_TIMER0A, 0x79);




    //x  l=2  h=0
    //y  l=44  h=0
    //z  l=21  h=1

    accelWrite(0x10, (signed char)0);
    accelWrite(0x11, (signed char)0);
    accelWrite(0x12, (signed char)44);
    accelWrite(0x13, (signed char)0);
    accelWrite(0x14, (signed char)214);//116
    accelWrite(0x15, (signed char)7);

    IntMasterEnable();

while(1)
  {


  }

}

// interpret sequence data
char decode (char* input) {

    if (myStrCmp(input,"10000100"))
    {
        return '1';
    }
    else if (myStrCmp(input,"01000100"))
    {
       return '2';
    }
    else if (myStrCmp(input,"11000100"))
    {
       return '3';
    }
    else if (myStrCmp(input,"00100100"))
    {
       return '4';
    }
    else if (myStrCmp(input,"10100100"))
    {
```

```
        return '5';
    }
    else if (myStrCmp(input,"01100100"))
    {
        return '6';
    }
    else if (myStrCmp(input,"11100100"))
    {
        return '7';
    }
    else if (myStrCmp(input,"00010100"))
    {
        return '8';
    }
    else if (myStrCmp(input,"10010100"))
    {
        return '9';
    }
    else if (myStrCmp(input,"00000100"))
    {
        return '0';
    }
    else if (myStrCmp(input,"10011000"))
    {
        return 'U';
    }
    else if (myStrCmp(input,"11111000"))
    {
        return 'L';
    }
    else if (myStrCmp(input,"01111000"))
    {
        return 'R';
    }
    else if (myStrCmp(input,"00011000"))
    {
        return 'D';
    }
    else if (myStrCmp(input,"01100111")) // Delete
    {
        return 'T';
    }
    else if (myStrCmp(input,"11001001")) // Last
    {
            return 'S';
    }
    else
    {
        return 'P';
    }
}

// convert button to character output
void decodeLetter (char input) {
    if (input == '1')
    {
        display2[dLocx++] = '1';
    }
    else if (input == 'U')
    {
        screenY--;
        if(motorOn)
```

```
        spin(1000);

    }
    else if (input == 'D')
    {
        screenY++;
        if(motorOn)
          spin(-1000);
    }
    else if (input == 'L')
    {

        if(motorOn==1)
            {

                PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, 1);
                motorOn = 0;
            }




    }
    else if (input == 'R')
    {

      PWMOutputState(PWM0_BASE, (PWM_OUT_2_BIT | PWM_OUT_3_BIT), true);
      if(motorOn==0)
      {

            PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, period);
            motorOn = 1;
      }



        PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, period);
    }
    else if (input == 'T')
    {
        if(dLocx != 0)
        {
            dLocx--;
            display2[dLocx] = '\0';
        }
    }
    else if (input == 'S')
    {

spinManual(atoi(display2));
RIT128x96x4Clear();
RIT128x96x4StringDraw("----------------------", 0, 50, 15);
RIT128x96x4StringDraw(display, 0, 0, 15);
display2[0] = '\0';
    dLocx=0;

    return;
    }
    else if(input == 'P')
    {
        showError();          // Error
    }
```

```
    if (input == '2')
        {
            display2[dLocx++] = '2';
        }
        else if (input == '3')
        {
            display2[dLocx++] = '3';
        }
        else if (input == '4')
        {
            display2[dLocx++] = '4';
        }
        else if (input == '5')
        {
            display2[dLocx++] = '5';
        }
        else if (input == '6')
        {

        }
        else if (input == '7')
        {
            display2[dLocx++] = '7';
        }
        else if (input == '8')
        {
            display2[dLocx++] = '8';
        }
        else if (input == '9')
        {
            display2[dLocx++] = '9';
        }
        else if (input == '0')
        {
            if (flag == 1)
            {
                if(dLocx != 0)
                {
                    if (display2[dLocx - 1] == '0')
                        display2[dLocx - 1] = '␣';
                    else if (display2[dLocx - 1] == '␣')
                        display2[dLocx - 1] = '0';
                    else
                        display2[dLocx++] = '0';
                 }
                 else
                    display2[dLocx++] = '0';
             }
             else
                 display2[dLocx++] = '0';
        }
    display2[dLocx] = '\0';
}
```