```
/* ===========================================*/
// Bradley Manzo and Thomas Ke
// EEC 172 SQ2023, Lab 4
/* ===========================================*/


#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdint.h>

// Driverlib includes
#include "hw_types.h"
#include "hw_ints.h"
#include "hw_memmap.h"
#include "hw_common_reg.h"
#include "interrupt.h"
#include "hw_apps_rcm.h"
#include "prcm.h"
#include "rom.h"
#include "rom_map.h"
#include "prcm.h"
#include "spi.h"
#include "gpio.h"
#include "systick.h"
#include "utils.h"
#include "timer.h"
#include "timer_if.h"

// Common interface includes
#include "gpio_if.h"
#include "Adafruit_GFX.h"
#include "Adafruit_SSD1351.h"
#include "glcdfont.h"
#include "pin_mux_config.h"


#define APPLICATION_VERSION     "1.4.0"

//**********************************************************************************
//                 GLOBAL VARIABLES -- Start
//**********************************************************************************
#if defined(ccs)
extern void (* const g_pfnVectors[])(void);
```

```
#endif
#if defined(ewarm)
extern uVectorEntry __vector_table;
#endif

// some helpful macros for systick

// the cc3200's fixed clock frequency of 80 MHz
// note the use of ULL to indicate an unsigned long long constant
#define SYSCLKFREQ 80000000ULL
// Periods defined for our delays in post processing
#define DEBOUNCE_PERIOD 100
#define REPEAT_PERIOD 1000
// Resolution for both timer modules
#define ADC_SAMPLING_PERIOD 5000
#define DELAY_PERIOD 80000  //CLK_FREQ to one MS

#define SPI_IF_BIT_RATE  400000
#define TR_BUFF_SIZE     100

#define BLACK           0x0000
#define BLUE            0x001F
#define GREEN           0x07E0
#define CYAN            0x07FF
#define RED             0xF800
#define MAGENTA         0xF81F
#define YELLOW          0xFFE0
#define WHITE           0xFFFF

// ======================================================================//
// Processing
// ======================================================================//

long int goertzel (int sample[], long int coeff, int N);
// Goertzel variables
int N = 410;
volatile int sampling_index = 0;
int samples[410];
// Delay variables
int delay_count = 0;
int prev_delay_count = 0;
// ADC interrupt handler variables
unsigned char ADCRxBuffer[2] = {0,0};
volatile unsigned int ADC_flag = 0;
```

```
// Goertzel Power Array for all 8 frequencies
int power_all[8];
// Coefficients calculated from DTMF frequencies
int coeff[8] = { 35248, 31281, 30950, 30556, 29143, 28360, 27408, 26258 };  // array to store the
calculated coefficients
int f_tone[8] = { 697, 770, 852, 941, 1209, 1336, 1477, 1633 }; // frequencies of rows & columns
// ==================================================================//
// Post-Processing
// ==================================================================//

void post_test (void);
// Confidence interval variables
char char_current = 0, char_prev = 0, char_prev2 = 0, char_prev3 = 0;
// Multi-tap decoding variables
char message = 0, prev_message = 0, character = 0, prev_char = 0;
int repetitions = 0;
// OLED position variables
int xTx = 0;
int yTx = 64;

unsigned long ulStatus = 0;


char letters3[6][3] = {{'A', 'B', 'C'},
            {'D', 'E', 'F'},
            {'G', 'H', 'I'},
            {'J', 'K', 'L'},
            {'M', 'N', 'O'},
            {'T', 'U', 'V'}};

char letters4[2][4] = {{'P', 'Q', 'R', 'S'},
            {'W', 'X', 'Y', 'Z'}};


//*****************************************************************************
//              GLOBAL VARIABLES -- End
//*****************************************************************************


//*****************************************************************************
//                  LOCAL FUNCTION PROTOTYPES
//*****************************************************************************
static void BoardInit(void);
```

```
//*****************************************************************************
//                      LOCAL FUNCTION DEFINITIONS
//*****************************************************************************

static void
BoardInit(void)
{
/* In case of TI-RTOS vector table is initialize by OS itself */
#ifndef USE_TIRTOS
    //
    // Set vector table base
    //
#if defined(ccs)
    MAP_IntVTableBaseSet((unsigned long)&g_pfnVectors[0]);
#endif
#if defined(ewarm)
    MAP_IntVTableBaseSet((unsigned long)&__vector_table);
#endif
#endif

    //
    // Enable Processor
    //
    MAP_IntMasterEnable();
    MAP_IntEnable(FAULT_SYSTICK);

    PRCMCC3200MCUInit();
}


//-------Goertzel function---------------------------------------//
long int
goertzel (int sample[], long int coeff, int N)
//-------------------------------------------------------------//
{
//initialize variables to be used in the function
  int Q, Q_prev, Q_prev2, i;
  long prod1, prod2, prod3, power;

  Q_prev = 0;           //set delay element1 Q_prev as zero
  Q_prev2 = 0;          //set delay element2 Q_prev2 as zero
  power = 0;            //set power as zero

  // loop N times and calculate Q, Q_prev, Q_prev2 at each iteration
```

```
    for (i = 0; i < N; i++)
    {
        // >>14 used as the coeff was used in Q15 format
        Q = (sample[i]) + ((coeff * Q_prev) >> 14) - (Q_prev2);
        // shuffle delay elements
        Q_prev2 = Q_prev;
        Q_prev = Q;
    }

  //calculate the three products used to calculate power
    prod1 = ((long) Q_prev * Q_prev);
    prod2 = ((long) Q_prev2 * Q_prev2);
    prod3 = ((long) Q_prev * coeff) >> 14;
    prod3 = (prod3 * Q_prev2);

    //calculate power using the three products and scale the result down
    power = ((prod1 + prod2 - prod3)) >> 8;

    return power;
}
/* Generic function to disable a timer_IF.h timer*/
void DisableTimer(int TIMER_BASE, int TIMER_LETTER, volatile unsigned int *function_flag)
{
    unsigned long ulStatus;
    ulStatus = MAP_TimerIntStatus(TIMER_BASE, false);
    MAP_TimerIntClear(TIMER_BASE, ulStatus);
    MAP_TimerDisable(TIMER_BASE, TIMER_LETTER);
    *function_flag = 0;
}

/* Generic function to enable a timer_IF.h timer*/
void EnableTimer(int TIMER_BASE, int TIMER_LETTER, int TIMER_TIMEOUT, int delay)
{
    if(delay == 5000)
        MAP_TimerLoadSet(TIMER_BASE, TIMER_LETTER, delay);
    MAP_TimerIntEnable(TIMER_BASE, TIMER_TIMEOUT);
    MAP_TimerEnable(TIMER_BASE, TIMER_LETTER);
}
/* Handler function to increment count every millisecond. Delay is calculated */
/* from recording count the start and subtracting it from the current count.  */
static void DelayHandler(void)
{
    unsigned long ulStatus;
    ulStatus = MAP_TimerIntStatus(TIMERA1_BASE, false);
    MAP_TimerIntClear(TIMERA1_BASE, ulStatus);
    delay_count++;
}
/* Handler function to collect 410 samples, and converting into  */
/* the proper format before passing into the Goertzel algorithm. */
static void ADCIntHandler(void)
{
    unsigned long ulStatus;
    // Clears interrupt
    ulStatus = MAP_TimerIntStatus(TIMERA0_BASE, false);
    MAP_TimerIntClear(TIMERA0_BASE, ulStatus);
    // Set CS Pin Low
    GPIOPinWrite(GPIOA2_BASE, 0x2, 0x0);
    // Receive 2 bytes of sample into Rx Buffer
    MAP_SPITransfer(GSPI_BASE, 0, ADCRxBuffer, 2, SPI_CS_ENABLE|SPI_CS_DISABLE);
    // Set CS Pin High
    GPIOPinWrite(GPIOA2_BASE, 0x2, 0x2);
    // Convert from big to little-endian
    int Little_End = (((unsigned int)(ADCRxBuffer[0] & 0b00011111)) << 5) | ((ADCRxBuffer[1] &
0b11111000) >> 3);
    // Store sample if less than 410
    if(sampling_index < N)
    {
        samples[sampling_index] = Little_End;
        sampling_index++;
    }
    // If 410th sample, set flag high to continue into Goertzel
    else
    {
        ADC_flag = 1;
    }
}
//-------Post-test function---------------------------------------//
void post_test (void)
//---------------------------------------------------------------//
{
    //initialize variables to be used in the function
    int i, row, col, max_power_row, max_power_col;

    // array with the order of the digits in the DTMF system
    char row_col[4][4] =
    {
        {'1', '2', '3', 'A'},
        {'4', '5', '6', 'B'},
        {'7', '8', '9', 'C'},
        {'*', '0', '#', 'D'}
    };

    // find the maximum power in the row frequencies and the row number
    max_power_row = 0;

    //loop 4 times from 0>3 (the indices of the rows)
    for (i = 0; i < 4; i++)
    {
        //if power of the current row frequency > max_power
        if (power_all[i] > max_power_row)
        {
            //set max_power as the current row frequency
            max_power_row = power_all[i];
            //update row number
            row = i;
        }
    }

    // find the maximum power in the column frequencies and the column number
    max_power_col = 0;

    //loop 4 times from 4>7 (the indices of the columns)
    for (i = 4; i < 8; i++)
    {
        //if power of the current column frequency > max_power
        if (power_all[i] > max_power_col)
        {
            //set max_power as the current column frequency
            max_power_col = power_all[i];
            //update column number
            col = i;
        }
    }
    if((max_power_col > 10000) && (max_power_row > 10000))
    {
        // Confidence interval of 4 similar repeated inputs
        char_prev3 = char_prev2;
        char_prev2 = char_prev;
        char_prev = char_current;
        char_current = row_col[row][col - 4];
        // Enters logic if confidence interval detects four identical consecutive readings
        if((char_current == char_prev) && (char_current == char_prev2) && (char_current ==
    char_prev3))
        {
          message = row_col[row][col - 4];

          // If longer than a second between inputs, discard possibility of repeat
          if(delay_count - prev_delay_count > 1000)
          {
            repetitions = 0;
            prev_message = 0;
            prev_char = 0;
          }
          // If in between 0.1 and 1 second, decode message
          if((delay_count - prev_delay_count < REPEAT_PERIOD) && (delay_count -
    prev_delay_count > DEBOUNCE_PERIOD))
          {
            // Increment position if position is new character and not a backspace
            if(message != prev_message && prev_char != '!' && message != '*')
            {
              // If at edge of screen, move to new line
              if(xTx >= 120)
              {
                xTx = 0;
                if(yTx < 120)
                  yTx += 8;
                else
                  yTx = 64;
              }
              // Otherwise increment by width of character
              else
              {
                xTx += 6;
              }
            }
            // 0 button pressed
            if(message == '0')
            {
              character = ' ';
            }
            // 2 button pressed
            else if(message == '2')
            {
              if(prev_message == message)
                repetitions++;
        else
            repetitions = 0;
        //letters = {'A', 'B', 'C'};
        if (repetitions > (sizeof(letters3[0]) - 1))
            repetitions = repetitions - (sizeof(letters3[0]));
        character = letters3[0][repetitions];
    }
    // 3 button pressed
    else if(message == '3')
    {
        if(prev_message == message)
            repetitions++;
        else
            repetitions = 0;
        //letters = {'D', 'E', 'F'};
        if (repetitions > (sizeof(letters3[0]) - 1))
            repetitions = repetitions - (sizeof(letters3[0]));
        character = letters3[1][repetitions];
    }
    // 4 button pressed
    else if(message == '4')
    {
        if(prev_message == message)
            repetitions++;
        else
            repetitions = 0;
        //letters = {'G', 'H', 'I'};
        if (repetitions > (sizeof(letters3[0]) - 1))
            repetitions = repetitions - (sizeof(letters3[0]));
        character = letters3[2][repetitions];
    }
    // 5 button pressed
    else if(message == '5')
    {
        if(prev_message == message)
            repetitions++;
        else
            repetitions = 0;
        //letters = {'J', 'K', 'L'};
        if (repetitions > (sizeof(letters3[0]) - 1))
            repetitions = repetitions - (sizeof(letters3[0]));
        character = letters3[3][repetitions];
    }
    // 6 button pressed
```

```
else if(message == '6')
{
    if(prev_message == message)
        repetitions++;
    else
        repetitions = 0;
    //letters = {'M', 'N', 'O'};
    if (repetitions > (sizeof(letters3[0]) - 1))
        repetitions = repetitions - (sizeof(letters3[0]));
    character = letters3[4][repetitions];
}
// 7 button pressed
else if(message == '7')
{
    if(prev_message == message)
        repetitions++;
    else
        repetitions = 0;
    //letters = {'P', 'Q', 'R', 'S'};
    if (repetitions > (sizeof(letters4[0]) - 1))
        repetitions = repetitions - (sizeof(letters4[0]));
    character = letters4[0][repetitions];
}
// 8 button pressed
else if(message == '8')
{
    if(prev_message == message)
        repetitions++;
    else
        repetitions = 0;
    //letters = {'T', 'U', 'V'};
    if (repetitions > (sizeof(letters3[0]) - 1))
        repetitions = repetitions - (sizeof(letters3[0]));
    character = letters3[5][repetitions];
}
// 9 button pressed
else if(message == '9')
{
    if(prev_message == message)
        repetitions++;
    else
        repetitions = 0;
    //letters = {'W', 'X', 'Y', 'Z'};
    if (repetitions > (sizeof(letters4[0]) - 1))
```

```
                        repetitions = repetitions - (sizeof(letters4[0]));
                    character = letters4[1][repetitions];
                }
                else if(message == '*')
                {
                    character = '*';
                    fillRect(xTx,yTx,6, 8,BLACK);
                    xTx -= 6;
                }
                prev_message = message;
                prev_char = character;
                // Draw new character, (overlaps valid repeats before incrementing position)
                if(character != '1' && character != '*')
                {
                    drawChar(xTx, yTx, character, colors[font_count], BLACK, 1);
                }
            }
            // "Start Timer"
            prev_delay_count = delay_count;
        }
    }
}
/* Generic function to initialize a timer_IF.h timer*/
void TimerInit(int PRCM_TIMER, int TIMER_BASE, int TIMER_LETTER, int TIMER_MODE, int delay, void Handler(void))
{
    MAP_PRCMPeripheralClkEnable(PRCM_TIMER, PRCM_RUN_MODE_CLK);
    MAP_PRCMPeripheralReset(PRCM_TIMER);
    MAP_TimerConfigure(TIMER_BASE,TIMER_MODE);
    MAP_TimerLoadSet(TIMER_BASE, TIMER_LETTER, delay);
    MAP_TimerIntRegister(TIMER_BASE, TIMER_LETTER, Handler);
    ulStatus = MAP_TimerIntStatus(TIMER_BASE, false);
    MAP_TimerIntClear(TIMER_BASE, ulStatus);
}

int main()
{
    //
    // Initialize Board configurations
    //
    BoardInit();

    //
    // Power on the corresponding GPIO port B for 9,10,11.
    // Set up the GPIO lines to mode 0 (GPIO)
    //
    PinMuxConfig();

    //
    // Enable the SPI module clock
    //
    MAP_PRCMPeripheralClkEnable(PRCM_GSPI,PRCM_RUN_MODE_CLK);

    //
    // Reset the peripheral
    //
    MAP_PRCMPeripheralReset(PRCM_GSPI);

    //
    // Reset SPI
    //
    MAP_SPIReset(GSPI_BASE);

    //
    // Configure SPI interface
    //
    MAP_SPIConfigSetExpClk(GSPI_BASE,MAP_PRCMPeripheralClockGet(PRCM_GSPI),
                SPI_IF_BIT_RATE,SPI_MODE_MASTER,SPI_SUB_MODE_0,
                (SPI_SW_CTRL_CS |
                SPI_4PIN_MODE |
                SPI_TURBO_OFF |
                SPI_CS_ACTIVELOW |
                SPI_WL_8));

    //
    // Enable SPI for communication
    //
    MAP_SPIEnable(GSPI_BASE);

    // Initialize the OLED
    Adafruit_Init();

    // Initialize the Delay and ADC timers
    TimerInit(PRCM_TIMERA0, TIMERA0_BASE, TIMER_A, TIMER_CFG_PERIODIC,
ADC_SAMPLING_PERIOD, ADCIntHandler);
    TimerInit(PRCM_TIMERA1, TIMERA1_BASE, TIMER_A, TIMER_CFG_PERIODIC,
DELAY_PERIOD, DelayHandler);
```

```
   int i;
   int samples_avg = 0;
   ADC_flag = 0;
   // Initialize OLED Screen
   setCursor(xTx, yTx);
   setTextSize(1);
   setTextColor(WHITE, BLACK);
   fillScreen(BLACK);

   // Enable the Delay and ADC timers
   EnableTimer(TIMERA0_BASE, TIMER_A, TIMER_TIMA_TIMEOUT,
ADC_SAMPLING_PERIOD);
   EnableTimer(TIMERA1_BASE, TIMER_A, TIMER_TIMA_TIMEOUT, DELAY_PERIOD);
   while(1)
   {
      // Busy-Waits until ADC flag is high
      while (ADC_flag == 0){;}
      // Disables ADC interrupts before proessing
      DisableTimer(TIMERA0_BASE, TIMER_A, &ADC_flag);
      Timer_IF_Stop(TIMERA0_BASE, TIMER_A);
      // Average Samples
      for(i = 0; i < N; i++)
         samples_avg += samples[i];
      samples_avg = samples_avg / N;
      // Subtract Average (DC Offset)
      for(i = 0; i < N; i++)
         samples[i] = samples[i] - samples_avg;
      // Flush variable
      samples_avg = 0;
      // Calculate new average
      for(i = 0; i < N; i++)
         samples_avg += samples[i];
      samples_avg = samples_avg / N;

      // wait till N samples are read in the buffer and the ADC_flag set by the ADC ISR
      for (i = 0; i < 8; i++)
      {
         power_all[i] = goertzel(samples, coeff[i], N);   // call goertzel to calculate the power at
each frequency and store it in the power_all array
      }

      post_test();       // call post test function to validate the data and display the pressed digit if
applicable
      sampling_index = 0;
```

```
        // Reenable the ADC Interupts
        EnableTimer(TIMERA0_BASE, TIMER_A, TIMER_TIMA_TIMEOUT, 5000);
    }
}
```