```
// Filename: adc_pos.h
// Authors: Jacob Mealey <jacob.mealey@maine.edu>
//          Landyn Francis <landyn.francis@maine.edu>
// This file provides a look table to convert adc values to voltages
#ifndef ADC_POS_H
#define ADC_POS_H

double adc_pos_lut[100] = {
        0.0834,
        0.082485,
        0.0819,
        0.081645,
        0.08172,
        0.082125,
        0.08286,
        0.083925,
        0.08532000000000001,
        0.087045,
        0.0891,
        0.091485,
        0.0942,
        0.097245,
        0.10062,
        0.104325,
        0.10836,
        0.11272499999999999,
        0.11742,
        0.122445,
        0.1278,
        0.133485,
        0.1395,
        0.145845,
        0.15252,
        0.159525,
        0.16686,
        0.17452499999999999,
        0.18252000000000002,
        0.190845,
        0.1995,
        0.208485,
        0.2178,
        0.227445,
        0.23742,
        0.247725,
        0.25836000000000003,
        0.26932500000000004,
        0.28062,
        0.292245,
        0.3042,
        0.316485,
        0.32909999999999995,
        0.34204500000000004,
        0.35531999999999997,
        0.36892500000000006,
        0.38286,
        0.39712500000000006,
        0.41172,
        0.42664499999999994,
        0.44189999999999996,
        0.45748500000000003,
        0.47340000000000004,
        0.489645,
        0.50622,
```

```
        0.523125,
        0.5403600000000001,
        0.557925,
        0.57582,
        0.594045,
        0.6126,
        0.631485,
        0.6507000000000001,
        0.6702450000000001,
        0.6901200000000001,
        0.710325,
        0.7308600000000001,
        0.7517250000000001,
        0.7729199999999999,
        0.794445,
        0.8163,
        0.838485,
        0.8610000000000001,
        0.883845,
        0.90702,
        0.930525,
        0.95436,
        0.978525,
        1.00302,
        1.027845,
        1.053,
        1.078485,
        1.1042999999999998,
        1.130445,
        1.15692,
        1.183725,
        1.2108599999999998,
        1.238325,
        1.26612,
        1.2942449999999999,
        1.3227,
        1.351485,
        1.3806,
        1.4100449999999998,
        1.4398199999999999,
        1.4398199999999999,
        1.4398199999999999,
        1.4398199999999999,
        1.4398199999999999,
        1.4398199999999999,
};

#endif
```

```
// Filename: adc.c
// Authors: Jacob Mealey <jacob.mealey@maine.edu>,
//          Landyn Francis <landyn.francis@maine.edu>
// This file provides function definitions for interfacing with the ADS7960
// which is a 8 bit analog to digital converter which we communicate with
// via SPI. It uses the spi interface provided the pico sdk.
//
// Note we are using many terms from the data sheet which can be found on
// the Texas Instruments website.

#include "adc.h"
#include <stdio.h>
#include "pins.h"
#include "midi.h"
#include "keys.h"

// init_adc inititalizes an adc struct on the heap and returns
// the pointer to said struct. it accepts a pointer to the spi
// bus used (spi_ins_t*) and the chip select line.
struct adc_t *init_adc(spi_inst_t *spi, uint16_t spi_cs)
{
    // The adc will work in auto-mode 2 which is defined as auto
    // incrementing the channel from channel zero to the channel
    // programmed in during the programming process.
    if (spi == NULL)
    {
        return NULL;
    }

    // allocate space for the ADC uhh err check ?
    struct adc_t *adc = malloc(sizeof(struct adc_t));

    // initialize values in the adc struct
    adc->spi = spi;
    // initialize spi bus
    spi_init(adc->spi, 3000000);
    spi_set_format(adc->spi, 16, 0, 0, SPI_MSB_FIRST);
    adc->spi_cs = spi_cs;
    adc->control_reg = ADC_MODE_RESET;

    // operating in manual mode
    for (int i = 0; i < 6; i++)
    {
        adc_write_read_blocking(adc);
        printf("0x%04x 0x%04x\n", adc->control_reg, adc->channel_val);
    }
    // auto 2 programming register
    // We are entering auto-2 programming. 12 is the amount of channels
    // we want to go up to.
    adc->control_reg = ADC_AUTO2_PROG(12u);
    adc_write_read_blocking(adc);
    printf("0x%04x 0x%04x\n", adc->control_reg, adc->channel_val);
    // set control register to continue in auto mode-2
    adc->control_reg = ADC_MODE_AUTO2;

    // Configure timer for SPI writes
    // timer must be allocated in heap so it lives beyond lifetime of init_adc
    repeating_timer_t *timer = malloc(sizeof(repeating_timer_t));
    if (add_repeating_timer_us(25, adc_write_callback, NULL, timer))
    {
        gpio_put(LED_0, 1);
    }
```

```
    // configure interrupt for spi reads
    spi_get_hw(adc->spi)->imsc = 1 << 1;
    irq_add_shared_handler(SPI1_IRQ, adc_read_irq, 1);
    irq_set_enabled(SPI1_IRQ, true);

    return adc;
}

// adc_write_callback is called by a timer and triggers the adc to perform
// a read
bool adc_write_callback(struct repeating_timer *t)
{
    (void)(t);
    spi_get_hw(adc_global->spi)->dr = adc_global->control_reg;
    return true;
}

// This function is called triggered by the spi bus when it recieves data
// it updates the entire key state of the device
void adc_read_irq(void)
{
    // Set channel value
    adc_global->channel_val = spi_get_hw(adc_global->spi)->dr;
    adc_global->prev_chanel = ADC_PRV_CHAN(adc_global->channel_val);

    uint8_t current_value = ADC_8BIT_VAL(adc_global->channel_val);
    // Most recent key - we are always one off because adc is adc
    uint8_t mrk = adc_global->prev_chanel;

    // update the current key position values
    key *current_key = &(keyboard_global->keys[mrk]);
    current_key->prev_pos = current_key->current_pos;
    current_key->current_pos = current_value;

    // Blink light if below threshold -- for debugging
    if (current_key->current_pos < KEY_THRESH)
    {
        gpio_put(PICO_DEFAULT_LED_PIN, 1);
    }
    else
    {
        gpio_put(PICO_DEFAULT_LED_PIN, 0);
    }

    // Determines if a key is pressed
    if (current_value < KEY_THRESH && current_key->pressed == 0 && current_key->prev_pos > cur
rent_key->current_pos)
    {
        current_key->pressed = 1;
        current_key->start_time = get_absolute_time();
        current_key->start_pos = current_value;
        current_key->end_pos = 0;
    }

    // A keys end position is seperate from a key release, this is strictly
    // for calculating velocity times.
    if (current_key->current_pos < 5 && current_key->end_pos == 0)
    {
        current_key->end_time = get_absolute_time();
        current_key->end_pos = current_value;
    }

    // determines if the key is released
```

```
    if (current_key->pressed == 1 && current_key->current_pos > KEY_THRESH)
    {
        current_key->pressed = 0;
    }

    // clear the interrupt
    spi_get_hw(adc_global->spi)->icr = 0;
}

// This is a method for writing / and reading to adc. this is a blocking function.
int adc_write_read_blocking(struct adc_t *adc)
{
    // while(spi_is_busy(adc->spi));
    gpio_put(adc->spi_cs, 0);
    spi_write16_blocking(adc->spi, &adc->control_reg, 1);
    gpio_put(adc->spi_cs, 1);
    sleep_ms(10);
    gpio_put(adc->spi_cs, 0);
    spi_read16_blocking(adc->spi, adc->control_reg, &adc->channel_val, 1);
    gpio_put(adc->spi_cs, 1);
    return 0;
}
```

```
// Filename: adc.h
// Authors: Jacob Mealey <jacob.mealey@maine.edu>
//          Landyn Francis <landyn.francis@maine.edu>
// Interface for ADS7960, see adc.c for more details
#ifndef SPI_ADC_H
#define SPI_ADC_H

#include <stdlib.h>
#include <stdint.h>
#include "pico/binary_info.h"
#include "pico/stdlib.h"
#include "hardware/spi.h"
#include "hardware/gpio.h"
#include "hardware/irq.h"

// ADC Modes
#define ADC_MODE_RESET 0x1000
#define ADC_MODE_AUTO1 (0x0002 << 12) | 1u << 10
#define ADC_MODE_AUTO2 0x3000

// ADC Auto Mode 1 Control Flags
#define M1_FLAG_EN_PROGRAMMING  0x0800
#define M1_FLAG_RESET_COUNTER   0x0400
#define M1_FLAG_2X_RANGE        0x0040
#define M1_FLAG_DEV_POW_DOWN    0x0020
#define M1_FLAG_EN_GPIO_MAP     0x0010

// Masks and shifts for reading DATA
#define ADC_8BIT_VAL(d) ((d >> 4) & 0xFF)
#define ADC_PRV_CHAN(d) (d >> 12)

// For entering programming ADC auto mode 1
#define ADC_AUTO1_PROG 0x8000
// ADC_AUTO2_PROG takes one parameter an 8 bit integer to
// specify the max channel to read to starting from zero
#define ADC_AUTO2_PROG(a) (0x9000 | ((a & 0xF) << 6u))
#define ADC_ALARM_PROG ADC_MODE_RESET << 12
#define ADC_GPIO_PROG  ADC_MODE_RESET << 12
// Enabling all channels for the 12 bit ADC
#define ADC_PROG_ENALL 0x0FFF



// adc_t is the structure for storing information about the current
// adc. it has 5 member variables two are for spi, and the other
// three are for channel values and data.
struct adc_t {
    spi_inst_t *spi;        // A pointer to an spi_inst_t that the adc is
                            // connected to.
    uint16_t control_reg;   // This is the value that will be written to the
                            // ADC.
    uint16_t prev_chanel;   // This is the index of the most recently read
                            // channel from the ADC, for this project it
                            // ranges from 0 - 11
    uint16_t channel_val;   // The is the ADC value of prev_channel
    uint16_t spi_cs;        // The chip select line

};

// This is a global adc that can be seen by the entire project
extern struct adc_t *adc_global;

// init adc takes the spi and the CS lines, it allocates
```

```
// and adc_t on the heap which is returned at the end of
// the function. it also allocats a repeating_timer_t
struct adc_t *init_adc(spi_inst_t *spi, uint16_t spi_cs);

// thing wrapper around spi_write_read16_blocking
int adc_write_read_blocking(struct adc_t *adc);

// the callback function whenever the timer allocated in
// init elapses. TODO: add someway to overright this?
// perhaps using __weak__?
bool adc_write_callback(repeating_timer_t *t);
// the function which is called when the SPI bus recieves
// data from the ADC TODO add somway to overwrite this.
void adc_read_irq(void);

#endif
```

```
// display.c
// Authors: Jacob Mealey <jacob.mealey@maine.edu>
//          Landyn Francis <landyn.francis@maine.edu>
// The Display uses SPI, it can operate in many different
// colors modes but we will be operating in 4k color
// because we don't need that much stuff. Note that we
// use the built in spi provided by the pico sdk
//
// The display we are using is a 1.8 inch TFT display,
// the driver chip is the ST7735R, and we are using a
// breakout board from Adafruit. Much of this code is based
// off elements described in the data shhet and the work
// done for the ST7735R Arduino Library written by Adafruit,

#include "display.h"
#include "pins.h"
#include "font.h"

#include <string.h>


// disp - a variable passed in from the calling funciton. This is different
// from how we did the ADC because this *shouldn't* need any heap allocation
// spi - a pointer to the spi_inst_t that the display is connected to
// disp_dc - this the the dc line going to the spi bus
int init_disp(struct disp_t *disp, spi_inst_t *spi, uint16_t disp_dc) {
    if(disp == NULL || spi == NULL) {
        printf("Error bad values passes to init_disp\n");
        return 1;
    }

    // initialize struct values
    disp->spi = spi;
    disp->dc = disp_dc;
    disp->cs = SPI0_CS;
    disp_global = disp;

    // intialize spi bus :)
    spi_init(disp->spi, 1000 * 10000);
    spi_set_format(disp->spi, 8, 0, 0, SPI_MSB_FIRST);

    // initialize GPIO lines
    gpio_init(disp->dc);
    gpio_set_dir(disp->dc, GPIO_OUT);

    gpio_init(disp->cs);
    gpio_set_dir(disp->cs, GPIO_OUT);

    gpio_init(DISPLAY_RESET);
    gpio_set_dir(DISPLAY_RESET, GPIO_OUT);

    // hardware reset the display
    gpio_put(DISPLAY_RESET, 1);
    sleep_ms(50);
    gpio_put(DISPLAY_RESET, 0);
    sleep_ms(50);
    gpio_put(DISPLAY_RESET, 1);
    sleep_ms(50);

    // command buffer is a scratch pad for sending commands
    // to the display
    uint8_t command_buffer[16];
```

```
    // This command sequence is both from the display datasheet
    // and the adafruit library.
    disp_wr_cmd(disp_global, DISP_SWRST, NULL, 0);
    sleep_ms(50);
    disp_wr_cmd(disp_global, DISP_SLPOUT, NULL, 0);
    sleep_ms(255);
    command_buffer[0] = DISP_COL_4K;
    disp_wr_cmd(disp_global, DISP_COLMOD, command_buffer, 1);
    sleep_ms(10);
    command_buffer[0] = 0x01;
    command_buffer[1] = 0x2C;
    command_buffer[2] = 0x2D;
    disp_wr_cmd(disp_global, DISP_FRMCTR1, command_buffer, 3);
    disp_wr_cmd(disp_global, DISP_FRMCTR2, command_buffer, 3);
    command_buffer[0] = 0x01;
    command_buffer[1] = 0x2C;
    command_buffer[2] = 0x2D;
    command_buffer[3] = 0x01;
    command_buffer[4] = 0x2C;
    command_buffer[5] = 0x2D;
    disp_wr_cmd(disp_global, DISP_FRMCTR3, command_buffer, 6);
    command_buffer[0] = 0x07;
    disp_wr_cmd(disp_global, DISP_INVCTR, command_buffer, 6);
    command_buffer[0] = 0xA2;
    command_buffer[1] = 0x02;
    command_buffer[2] = 0x84;
    disp_wr_cmd(disp_global, DISP_PWRCTR1, command_buffer, 3);
    command_buffer[0] = 0xC5;
    disp_wr_cmd(disp_global, DISP_PWRCTR2, command_buffer, 1);
    command_buffer[0] = 0x0A;
    command_buffer[1] = 0x00;
    disp_wr_cmd(disp_global, DISP_PWRCTR3, command_buffer, 2);
    command_buffer[0] = 0x8A;
    command_buffer[1] = 0x2A;
    disp_wr_cmd(disp_global, DISP_PWRCTR4, command_buffer, 2);
    command_buffer[0] = 0x8A;
    command_buffer[1] = 0xEE;
    disp_wr_cmd(disp_global, DISP_PWRCTR5, command_buffer, 2);
    command_buffer[0] = 0x0E;
    disp_wr_cmd(disp_global, DISP_VMCTR1, command_buffer, 1);
    disp_wr_cmd(disp_global, DISP_INVOFF, NULL, 0);
    command_buffer[0] = 0xC8;
    disp_wr_cmd(disp_global, DISP_MADCTL, command_buffer, 1);
    command_buffer[0] = DISP_COL_4K;
    disp_wr_cmd(disp_global, DISP_COLMOD, command_buffer, 1);
    command_buffer[0] = 0x00;
    command_buffer[1] = 0x00;
    command_buffer[2] = 0x00;
    command_buffer[3] = 0x7F;
    disp_wr_cmd(disp_global, DISP_CASET, command_buffer, 4);
    command_buffer[0] = 0x00;
    command_buffer[1] = 0x00;
    command_buffer[2] = 0x00;
    command_buffer[3] = 0x9F;
    disp_wr_cmd(disp_global, DISP_RASET, command_buffer, 4);
    disp_wr_cmd(disp_global, DISP_NORON, NULL, 0);
    sleep_ms(10);
    disp_wr_cmd(disp_global, DISP_DISPON, NULL, 0);
    sleep_ms(100);

    return 0;
}
```

```
// disp_wr_cmd is a blocking function for sending commands to the display. it requires
// a pointer to the displayt struct, an individual command, an array of 8 bit integers
// which are the arguments for the given command and the amount of arguments.
int disp_wr_cmd(struct disp_t *disp, uint8_t command, uint8_t *args, unsigned int len) {
    gpio_put(disp->cs, 0);
    gpio_put(disp->dc, 0);
    spi_write_blocking(disp->spi, &command, 1);
    gpio_put(disp->dc, 1);
    if(len != 0) {
        spi_write_blocking(disp->spi, args, len);
    }
    gpio_put(disp->cs, 1);

    return 0;
}


// screen is the "easy" to interract with form of display data, each element
// in screen corresponds to 1 pixel, formattted like: 0x0RGB.
// disp is the formmatted version of screen to be written to the SPI bus
int screen_to_disp(uint16_t *screen, uint8_t *disp, int screen_len) {
    uint8_t acc = 0; // accumulator for screen translation
    int acc_set = 0; // variable to check accumlator
    int i = 0; // the current pixel in the screen array
    int bi = 0; // the current byte being set in disp

    while(i < screen_len) {
        if(acc_set  == 0) { // if acc is empty
            disp[bi] = (screen[i] >> 4) & 0xFF;
            acc = screen[i] & 0xF;
            acc_set = 1;
            bi++;
            i++;
        }
        if(acc_set == 2) { // if the top half of acc is set
           disp[bi] = acc;
           acc_set = 0;
           bi++;
        }
        if(acc_set == 1){ // final case - acc lowest half is set
            // write acc to the highest part of disp
            // and write the highest part of screen
            disp[bi] = ((acc << 4) & 0xF0) | ((screen[i] >> 8) & 0x0F);
            // save the lower 2/3 of screen to acc and set where to next
            acc = screen[i] & 0xFF;
            acc_set = 2;
            bi++;
            i++;
        }
    }
    return bi;
}

// set the x position for the display ram
void set_x(uint8_t x) {
    if(x > DISPLAY_W) x = DISPLAY_W;
    uint8_t command_buffer[4];
    command_buffer[0] = 0x00;
    command_buffer[1] = x;
    command_buffer[2] = 0x00;
    command_buffer[3] = 0x7F;
```

```
    disp_wr_cmd(disp_global, DISP_CASET, command_buffer, 4);
}

// set the y position for the display ram
void set_y(uint8_t y) {
    if(y > DISPLAY_HEIGHT) y = DISPLAY_HEIGHT;
    uint8_t command_buffer[4];
    command_buffer[0] = 0x00;
    command_buffer[1] = y;
    command_buffer[2] = 0x00;
    command_buffer[3] = 0x9F;
    disp_wr_cmd(disp_global, DISP_RASET, command_buffer, 4);
}


// draws a rectangle at location x,y with height h amd width w
// it fills it with color.
int draw_rect(uint8_t x, uint8_t y, uint8_t h, uint8_t w, uint16_t color){
   // static because they shouldn't be allocated everytime?
   static uint16_t screen[128];
   static uint8_t buffer[200];

   //screen is a single "row" of the rect, so only fill to w
   for(int i = 0; i < w + 1; i++) {
       screen[i] = color;
   }

   // convert screen to w
   screen_to_disp(screen, buffer, w);
   // go the the x position
   set_x(x);

   if(!(w % 3)) w += w%3;
   // loop through h values, draw a new line of the screen
   // at every incrementing h
   for(int i = 0; i < h; i++) {
       set_y(y+i);
       disp_wr_cmd(disp_global, DISP_RAMWR, buffer, (w*3) / 2);
   }

   // reset x and y to zero;
   set_x(0);
   set_y(0);

   return 0;

}

// Draw a string of characters to the display and location x,y
// str - a null terminatd string of characters.
void draw_string(const char *str, uint8_t x, uint8_t y, uint16_t font_bg, uint16_t font_fg){
    set_x(x);
    set_y(y);
    // loop until the the null character is met
    while(*str != '\0' || y < 5 || x < 5 || x > DISPLAY_W || y > DISPLAY_HEIGHT) {
        draw_char(*str, x, y, font_bg, font_fg);
        y -= 6;
        str++;
    }
}

// Draw a single character to the display
// c - character to draw
```

```
// x - x location to draw display
// y - y location to draw the display
void draw_char(char c, uint8_t x, uint8_t y, uint16_t font_bg, uint16_t font_fg) {
    static uint8_t character[5];
    static uint16_t screen[7];
    static uint8_t buffer[16];

    memcpy(character, font + 5*c, 5);
    if(y == 0) y = 6;
    if(x == 0) x = 1;


    draw_rect(x-7, y-5, 7, 9, font_bg);
    set_x(x - 7);
    for(int i = 0; i < 5; i++){ //loop through lines
        for(int j = 0; j < 7; j++){ //loop through pixels of current line
            screen[j] = ((character[i] >> j) & 1u) ? font_fg: font_bg;
        }
        set_y(y - i); // move to to next line
        // write current line to the display
        screen_to_disp(screen, buffer, 7);
        disp_wr_cmd(disp_global, DISP_RAMWR, buffer, 11);
    }

}

// this a test to draw the font - it uses some static variables
// so to use draw_font_test just run it in main loop with nothing else
void draw_font_test() {
    static int offset = 0;
    static int x;
    static int y;

    uint8_t character[5];
    uint16_t screen[7];
    uint8_t buffer[16];

    if(offset > 3125) return;

    // get current character
    memcpy(character, font + offset, 5);
    if(y == 0) y = 6;
    if(x == 0) x = 1;

    for(int i = 0; i < 5; i++){ //loop through lines
        for(int j = 0; j < 7; j++){ //loop through pixels
            screen[j] = ((character[i] >> j) & 1u) ? 0x000: 0xFFF;
        }
        set_y(y - i);
        screen_to_disp(screen, buffer, 7);
        disp_wr_cmd(disp_global, DISP_RAMWR, buffer, 11);
    }

    // update x and ys accordingly
    x += 8;
    offset += 5;

    if(x > DISPLAY_W - 7) {
        x = 1;
        y += 6;
    }
    set_x(x);
}
```

```
#ifndef DISPLAY_H
#define DISPLAY_H
// display.c
// Authors: Jacob Mealey <jacob.mealey@maine.edu>
//          Landyn Francis <landyn.francis@maine.edu>
// See display.c for description

#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include "pico/binary_info.h"
#include "pico/stdlib.h"
#include "hardware/spi.h"
#include "hardware/gpio.h"
#include "hardware/irq.h"

#define DISPLAY_W 128
#define DISPLAY_HEIGHT 160

// List of commands from pg. 77 of ST7735 Datasheet
#define DISP_NOP 0x00
#define DISP_SWRST 0x01
#define DISP_RDID 0x04
#define DISP_RDDST 0x09
#define DISP_RDDPM 0x0A
#define DISP_RDD_MACDCTL 0x0B
#define DISP_RDD_COLMOD 0x0C
#define DISP_RDDIM 0x0D
#define DISP_RDDSM 0x0E
#define DISP_SLPIN 0x10
#define DISP_SLPOUT 0x11
#define DISP_PTLON 0x12
#define DISP_NORON 0x13
#define DISP_INVOFF 0x20
#define DISP_ONVON 0x021
#define DISP_GAMSET 0x26
#define DISP_DISPOFF 0x28
#define DISP_DISPON 0x29
#define DISP_CASET 0x2A
#define DISP_RASET 0x2B
#define DISP_RAMWR 0x2C
#define DISP_RAMRD 0x2E
#define DISP_PLAR 0x30
#define DISP_TEOFF 0x34
#define DISP_TEON 0x35
#define DISP_MADCTL 0x36
#define DISP_IDMOFF 0x38
#define DISP_ODMON 0x39
#define DISP_COLMOD 0x3A
#define DISP_RDID1 0xDA
#define DISP_RDID2 0xDB
#define DISP_RDID3 0xDC

#define DISP_FRMCTR1 0xB1
#define DISP_FRMCTR2 0xB2
#define DISP_FRMCTR3 0xB3
#define DISP_INVCTR 0xB4
#define DISP_DISSET5 0xB6
#define DISP_PWRCTR1 0xC0
#define DISP_PWRCTR2 0xC1
#define DISP_PWRCTR3 0xC2
#define DISP_PWRCTR4 0xC3
#define DISP_PWRCTR5 0xC4
```

```
#define DISP_VMCTR1 0xC5
#define DISP_VMOFCTR 0xC7
#define DISP_WRID2 0xD1
#define DISP_WRID3 0xD2
#define DISP_PWCTR6 0xFC
#define DISP_NVCTR1 0xD9
#define DISP_NVCTR2 0xDE
#define DISP_NVCTR3 0xDF
#define DISP_GAMCTRP1 0xE0
#define DISP_GAMCTRN1 0xE1
#define DISP_EXTCTRL 0xF0
#define DISP_VCOM4L 0xFF

// Extra macros like colors and such
#define DISP_COL_4K 0x03
#define DISP_COL_65K 0x05
#define DISP_COLOR_262K 0x06

// this is the struct for keeping display state
// it has a pointer to a spi_inst_t, it also has
// dc and cs pins
struct disp_t
{
    spi_inst_t *spi;
    uint16_t dc;
    uint16_t cs;
};

enum display_colors
{
    RED = 0xF00,
    GREEN = 0x0F0,
    BLUE = 0x00F,
    PURPLE = 0xF0F,
    YELLOW = 0xFF0,
    ORANGE = 0xF70,
    PINK = 0xF88,
    BLACK = 0x000,
    WHITE = 0xFFF,
};

extern struct disp_t *disp_global;

int init_disp(struct disp_t *disp, spi_inst_t *spi, uint16_t disp_dc);
int disp_wr_cmd(struct disp_t *disp, uint8_t command, uint8_t *args, unsigned int len);

int screen_to_disp(uint16_t *screen, uint8_t *disp, int screen_len);

void set_x(uint8_t x);
void set_y(uint8_t y);

int draw_rect(uint8_t x, uint8_t y, uint8_t h, uint8_t w, uint16_t color);

void draw_font_test();
void draw_char(char c, uint8_t x, uint8_t y, uint16_t font_bg, uint16_t font_fg);
void draw_string(const char *c, uint8_t x, uint8_t y, uint16_t font_bg, uint16_t font_fg);
#endif
```

```
// This is a fork of the bitmap font system from the Adafruit_GFX font library
// Forked from https://github.com/adafruit/Adafruit-GFX-Library/blob/221bb41cd2e3d247b91260458
b491b6ba7f0238d/glcdfont.c

#ifndef FONT5X7_H
#define FONT5X7_H

// Standard ASCII 5x7 font
const unsigned char font[] = {
    0x00, 0x00, 0x00, 0x00, 0x00,
    0x3E, 0x5B, 0x4F, 0x5B, 0x3E,
    0x3E, 0x6B, 0x4F, 0x6B, 0x3E,
    0x1C, 0x3E, 0x7C, 0x3E, 0x1C,
    0x18, 0x3C, 0x7E, 0x3C, 0x18,
    0x1C, 0x57, 0x7D, 0x57, 0x1C,
    0x1C, 0x5E, 0x7F, 0x5E, 0x1C,
    0x00, 0x18, 0x3C, 0x18, 0x00,
    0xFF, 0xE7, 0xC3, 0xE7, 0xFF,
    0x00, 0x18, 0x24, 0x18, 0x00,
    0xFF, 0xE7, 0xDB, 0xE7, 0xFF,
    0x30, 0x48, 0x3A, 0x06, 0x0E,
    0x26, 0x29, 0x79, 0x29, 0x26,
    0x40, 0x7F, 0x05, 0x05, 0x07,
    0x40, 0x7F, 0x05, 0x25, 0x3F,
    0x5A, 0x3C, 0xE7, 0x3C, 0x5A,
    0x7F, 0x3E, 0x1C, 0x1C, 0x08,
    0x08, 0x1C, 0x1C, 0x3E, 0x7F,
    0x14, 0x22, 0x7F, 0x22, 0x14,
    0x5F, 0x5F, 0x00, 0x5F, 0x5F,
    0x06, 0x09, 0x7F, 0x01, 0x7F,
    0x00, 0x66, 0x89, 0x95, 0x6A,
    0x60, 0x60, 0x60, 0x60, 0x60,
    0x94, 0xA2, 0xFF, 0xA2, 0x94,
    0x08, 0x04, 0x7E, 0x04, 0x08,
    0x10, 0x20, 0x7E, 0x20, 0x10,
    0x08, 0x08, 0x2A, 0x1C, 0x08,
    0x08, 0x1C, 0x2A, 0x08, 0x08,
    0x1E, 0x10, 0x10, 0x10, 0x10,
    0x0C, 0x1E, 0x0C, 0x1E, 0x0C,
    0x30, 0x38, 0x3E, 0x38, 0x30,
    0x06, 0x0E, 0x3E, 0x0E, 0x06,
    0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x5F, 0x00, 0x00,
    0x00, 0x07, 0x00, 0x07, 0x00,
    0x14, 0x7F, 0x14, 0x7F, 0x14,
    0x24, 0x2A, 0x7F, 0x2A, 0x12,
    0x23, 0x13, 0x08, 0x64, 0x62,
    0x36, 0x49, 0x56, 0x20, 0x50,
    0x00, 0x08, 0x07, 0x03, 0x00,
    0x00, 0x1C, 0x22, 0x41, 0x00,
    0x00, 0x41, 0x22, 0x1C, 0x00,
    0x2A, 0x1C, 0x7F, 0x1C, 0x2A,
    0x08, 0x08, 0x3E, 0x08, 0x08,
    0x00, 0x80, 0x70, 0x30, 0x00,
    0x08, 0x08, 0x08, 0x08, 0x08,
    0x00, 0x00, 0x60, 0x60, 0x00,
    0x20, 0x10, 0x08, 0x04, 0x02,
    0x3E, 0x51, 0x49, 0x45, 0x3E, 0x00, 0x42, 0x7F, 0x40, 0x00, 0x72, 0x49,
    0x49, 0x49, 0x46, 0x21, 0x41, 0x49, 0x4D, 0x33, 0x18, 0x14, 0x12, 0x7F,
    0x10, 0x27, 0x45, 0x45, 0x45, 0x39, 0x3C, 0x4A, 0x49, 0x49, 0x31, 0x41,
    0x21, 0x11, 0x09, 0x07, 0x36, 0x49, 0x49, 0x49, 0x36, 0x46, 0x49, 0x49,
    0x29, 0x1E, 0x00, 0x00, 0x14, 0x00, 0x00, 0x00, 0x40, 0x34, 0x00, 0x00,
    0x00, 0x08, 0x14, 0x22, 0x41, 0x14, 0x14, 0x14, 0x14, 0x14, 0x00, 0x41,
```

```
    0x22, 0x14, 0x08, 0x02, 0x01, 0x59, 0x09, 0x06, 0x3E, 0x41, 0x5D, 0x59,
    0x4E, 0x7C, 0x12, 0x11, 0x12, 0x7C, 0x7F, 0x49, 0x49, 0x49, 0x36, 0x3E,
    0x41, 0x41, 0x41, 0x22, 0x7F, 0x41, 0x41, 0x41, 0x3E, 0x7F, 0x49, 0x49,
    0x49, 0x41, 0x7F, 0x09, 0x09, 0x09, 0x01, 0x3E, 0x41, 0x41, 0x51, 0x73,
    0x7F, 0x08, 0x08, 0x08, 0x7F, 0x00, 0x41, 0x7F, 0x41, 0x00, 0x20, 0x40,
    0x41, 0x3F, 0x01, 0x7F, 0x08, 0x14, 0x22, 0x41, 0x7F, 0x40, 0x40, 0x40,
    0x40, 0x7F, 0x02, 0x1C, 0x02, 0x7F, 0x7F, 0x04, 0x08, 0x10, 0x7F, 0x3E,
    0x41, 0x41, 0x41, 0x3E, 0x7F, 0x09, 0x09, 0x09, 0x06, 0x3E, 0x41, 0x51,
    0x21, 0x5E, 0x7F, 0x09, 0x19, 0x29, 0x46, 0x26, 0x49, 0x49, 0x49, 0x32,
    0x03, 0x01, 0x7F, 0x01, 0x03, 0x3F, 0x40, 0x40, 0x40, 0x3F, 0x1F, 0x20,
    0x40, 0x20, 0x1F, 0x3F, 0x40, 0x38, 0x40, 0x3F, 0x63, 0x14, 0x08, 0x14,
    0x63, 0x03, 0x04, 0x78, 0x04, 0x03, 0x61, 0x59, 0x49, 0x4D, 0x43, 0x00,
    0x7F, 0x41, 0x41, 0x41, 0x02, 0x04, 0x08, 0x10, 0x20, 0x00, 0x41, 0x41,
    0x41, 0x7F, 0x04, 0x02, 0x01, 0x02, 0x04, 0x40, 0x40, 0x40, 0x40, 0x40,
    0x00, 0x03, 0x07, 0x08, 0x00, 0x20, 0x54, 0x54, 0x78, 0x40, 0x7F, 0x28,
    0x44, 0x44, 0x38, 0x38, 0x44, 0x44, 0x44, 0x28, 0x38, 0x44, 0x44, 0x28,
    0x7F, 0x38, 0x54, 0x54, 0x54, 0x18, 0x00, 0x08, 0x7E, 0x09, 0x02, 0x18,
    0xA4, 0xA4, 0x9C, 0x78, 0x7F, 0x08, 0x04, 0x04, 0x78, 0x00, 0x44, 0x7D,
    0x40, 0x00, 0x20, 0x40, 0x40, 0x3D, 0x00, 0x7F, 0x10, 0x28, 0x44, 0x00,
    0x00, 0x41, 0x7F, 0x40, 0x00, 0x7C, 0x04, 0x78, 0x04, 0x78, 0x7C, 0x08,
    0x04, 0x04, 0x78, 0x38, 0x44, 0x44, 0x44, 0x38, 0xFC, 0x18, 0x24, 0x24,
    0x18, 0x18, 0x24, 0x24, 0x18, 0xFC, 0x7C, 0x08, 0x04, 0x04, 0x08, 0x48,
    0x54, 0x54, 0x54, 0x24, 0x04, 0x04, 0x3F, 0x44, 0x24, 0x3C, 0x40, 0x40,
    0x20, 0x7C, 0x1C, 0x20, 0x40, 0x20, 0x1C, 0x3C, 0x40, 0x30, 0x40, 0x3C,
    0x44, 0x28, 0x10, 0x28, 0x44, 0x4C, 0x90, 0x90, 0x90, 0x7C, 0x44, 0x64,
    0x54, 0x4C, 0x44, 0x00, 0x08, 0x36, 0x41, 0x00, 0x00, 0x00, 0x77, 0x00,
    0x00, 0x00, 0x41, 0x36, 0x08, 0x00, 0x02, 0x01, 0x02, 0x04, 0x02, 0x3C,
    0x26, 0x23, 0x26, 0x3C, 0x1E, 0xA1, 0xA1, 0x61, 0x12, 0x3A, 0x40, 0x40,
    0x20, 0x7A, 0x38, 0x54, 0x54, 0x55, 0x59, 0x21, 0x55, 0x55, 0x79, 0x41,
    0x22, 0x54, 0x54, 0x78, 0x42, // a-umlaut
    0x21, 0x55, 0x54, 0x78, 0x40, 0x20, 0x54, 0x55, 0x79, 0x40, 0x0C, 0x1E,
    0x52, 0x72, 0x12, 0x39, 0x55, 0x55, 0x55, 0x59, 0x39, 0x54, 0x54, 0x54,
    0x59, 0x39, 0x55, 0x54, 0x54, 0x58, 0x00, 0x00, 0x45, 0x7C, 0x41, 0x00,
    0x02, 0x45, 0x7D, 0x42, 0x00, 0x01, 0x45, 0x7C, 0x40, 0x7D, 0x12, 0x11,
    0x12, 0x7D, // A-umlaut
    0xF0, 0x28, 0x25, 0x28, 0xF0, 0x7C, 0x54, 0x55, 0x45, 0x00, 0x20, 0x54,
    0x54, 0x7C, 0x54, 0x7C, 0x0A, 0x09, 0x7F, 0x49, 0x32, 0x49, 0x49, 0x49,
    0x32, 0x3A, 0x44, 0x44, 0x44, 0x3A, // o-umlaut
    0x32, 0x4A, 0x48, 0x48, 0x30, 0x3A, 0x41, 0x41, 0x21, 0x7A, 0x3A, 0x42,
    0x40, 0x20, 0x78, 0x00, 0x9D, 0xA0, 0xA0, 0x7D, 0x3D, 0x42, 0x42, 0x42,
    0x3D, // O-umlaut
    0x3D, 0x40, 0x40, 0x40, 0x3D, 0x3C, 0x24, 0xFF, 0x24, 0x24, 0x48, 0x7E,
    0x49, 0x43, 0x66, 0x2B, 0x2F, 0xFC, 0x2F, 0x2B, 0xFF, 0x09, 0x29, 0xF6,
    0x20, 0xC0, 0x88, 0x7E, 0x09, 0x03, 0x20, 0x54, 0x54, 0x79, 0x41, 0x00,
    0x00, 0x44, 0x7D, 0x41, 0x30, 0x48, 0x48, 0x4A, 0x32, 0x38, 0x40, 0x40,
    0x22, 0x7A, 0x00, 0x7A, 0x0A, 0x0A, 0x72, 0x7D, 0x0D, 0x19, 0x31, 0x7D,
    0x26, 0x29, 0x29, 0x2F, 0x28, 0x26, 0x29, 0x29, 0x29, 0x26, 0x30, 0x48,
    0x4D, 0x40, 0x20, 0x38, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
    0x38, 0x2F, 0x10, 0xC8, 0xAC, 0xBA, 0x2F, 0x10, 0x28, 0x34, 0xFA, 0x00,
    0x00, 0x7B, 0x00, 0x00, 0x08, 0x14, 0x2A, 0x14, 0x22, 0x22, 0x14, 0x2A,
    0x14, 0x08, 0x55, 0x00, 0x55, 0x00, 0x55, // #176 (25% block) missing in old
                                              // code
    0xAA, 0x55, 0xAA, 0x55, 0xAA,             // 50% block
    0xFF, 0x55, 0xFF, 0x55, 0xFF,             // 75% block
    0x00, 0x00, 0x00, 0xFF, 0x00, 0x10, 0x10, 0x10, 0xFF, 0x00, 0x14, 0x14,
    0x14, 0xFF, 0x00, 0x10, 0x10, 0xFF, 0x00, 0xFF, 0x10, 0x10, 0xF0, 0x10,
    0xF0, 0x14, 0x14, 0x14, 0xFC, 0x00, 0x14, 0x14, 0xF7, 0x00, 0xFF, 0x00,
    0x00, 0xFF, 0x00, 0xFF, 0x14, 0x14, 0xF4, 0x04, 0xFC, 0x14, 0x14, 0x17,
    0x10, 0x1F, 0x10, 0x10, 0x1F, 0x10, 0x1F, 0x14, 0x14, 0x14, 0x1F, 0x00,
    0x10, 0x10, 0x10, 0xF0, 0x00, 0x00, 0x00, 0x00, 0x1F, 0x10, 0x10, 0x10,
    0x10, 0x1F, 0x10, 0x10, 0x10, 0x10, 0xF0, 0x10, 0x00, 0x00, 0x00, 0xFF,
    0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0xFF, 0x10, 0x00,
    0x00, 0x00, 0xFF, 0x14, 0x00, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0x00, 0x1F,
```

```
    0x10, 0x17, 0x00, 0x00, 0xFC, 0x04, 0xF4, 0x14, 0x14, 0x17, 0x10, 0x17,
    0x14, 0x14, 0xF4, 0x04, 0xF4, 0x00, 0x00, 0xFF, 0x00, 0xF7, 0x14, 0x14,
    0x14, 0x14, 0x14, 0x14, 0x14, 0xF7, 0x00, 0xF7, 0x14, 0x14, 0x14, 0x17,
    0x14, 0x10, 0x10, 0x1F, 0x10, 0x1F, 0x14, 0x14, 0x14, 0xF4, 0x14, 0x10,
    0x10, 0xF0, 0x10, 0xF0, 0x00, 0x00, 0x1F, 0x10, 0x1F, 0x00, 0x00, 0x00,
    0x1F, 0x14, 0x00, 0x00, 0x00, 0xFC, 0x14, 0x00, 0x00, 0xF0, 0x10, 0xF0,
    0x10, 0x10, 0xFF, 0x10, 0xFF, 0x14, 0x14, 0x14, 0xFF, 0x14, 0x10, 0x10,
    0x10, 0x1F, 0x00, 0x00, 0x00, 0x00, 0xF0, 0x10, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xF0, 0xF0, 0xF0, 0xF0, 0xF0, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00,
    0x00, 0x00, 0xFF, 0xFF, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x38, 0x44, 0x44,
    0x38, 0x44, 0xFC, 0x4A, 0x4A, 0x4A, 0x34, // sharp-s or beta
    0x7E, 0x02, 0x02, 0x06, 0x06, 0x02, 0x7E, 0x02, 0x7E, 0x02, 0x63, 0x55,
    0x49, 0x41, 0x63, 0x38, 0x44, 0x44, 0x3C, 0x04, 0x40, 0x7E, 0x20, 0x1E,
    0x20, 0x06, 0x02, 0x7E, 0x02, 0x02, 0x99, 0xA5, 0xE7, 0xA5, 0x99, 0x1C,
    0x2A, 0x49, 0x2A, 0x1C, 0x4C, 0x72, 0x01, 0x72, 0x4C, 0x30, 0x4A, 0x4D,
    0x4D, 0x30, 0x30, 0x48, 0x78, 0x48, 0x30, 0xBC, 0x62, 0x5A, 0x46, 0x3D,
    0x3E, 0x49, 0x49, 0x49, 0x00, 0x7E, 0x01, 0x01, 0x01, 0x7E, 0x2A, 0x2A,
    0x2A, 0x2A, 0x2A, 0x44, 0x44, 0x5F, 0x44, 0x44, 0x40, 0x51, 0x4A, 0x44,
    0x40, 0x40, 0x44, 0x4A, 0x51, 0x40, 0x00, 0x00, 0xFF, 0x01, 0x03, 0xE0,
    0x80, 0xFF, 0x00, 0x00, 0x08, 0x08, 0x6B, 0x6B, 0x08, 0x36, 0x12, 0x36,
    0x24, 0x36, 0x06, 0x0F, 0x09, 0x0F, 0x06, 0x00, 0x00, 0x18, 0x18, 0x00,
    0x00, 0x00, 0x10, 0x10, 0x00, 0x30, 0x40, 0xFF, 0x01, 0x01, 0x00, 0x1F,
    0x01, 0x01, 0x1E, 0x00, 0x19, 0x1D, 0x17, 0x12, 0x00, 0x3C, 0x3C, 0x3C,
    0x3C, 0x00, 0x00, 0x00, 0x00, 0x00 // #255 NBSP
};

#endif // FONT5X7_H
```

```
// File: gpio.c
// Date Created: 9/18/22
// Authors: Landyn Francis (landyn.francis@maine.edu) Jacob Mealey (jacob.mealey@maine.edu)
// Purpose: Source file containing gpio initialization and interrupt functions
#include "pins.h"
#include <stdint.h>
#include "hardware/gpio.h"
#include "pico/stdio.h"
#include <stdio.h>
#include "keys.h"
#include "midi.h"
#include "pico/sync.h"

uint8_t pin_init()
{
        // Set UART GPIO functions (Debugging)
        gpio_set_function(UART0_TX, GPIO_FUNC_UART);
        gpio_set_function(UART0_RX, GPIO_FUNC_UART);
        printf("Hello, Midi!\n");

        // Set SPI1 GPIO functions (Display)
        gpio_set_function(SPI1_SCLK, GPIO_FUNC_SPI);
        gpio_set_function(SPI1_RX, GPIO_FUNC_SPI);
        gpio_set_function(SPI1_TX, GPIO_FUNC_SPI);
        gpio_set_function(SPI1_CS, GPIO_FUNC_SPI);
        printf("alternate functions for ADC set\n");

        // Set SPI0 GPIO functions (Display)
        gpio_set_function(SPI0_SCLK, GPIO_FUNC_SPI);
        gpio_set_function(SPI0_RX, GPIO_FUNC_SPI);
        gpio_set_function(SPI0_TX, GPIO_FUNC_SPI);
        // Set SPI0 pull downs
        gpio_set_pulls(SPI0_SCLK, false, true);
        gpio_set_pulls(SPI0_TX, false, true);
        printf("alternate functions for Display set\n");

        // LED pin definitions and initialization
        gpio_init(LED_0);
        gpio_set_dir(LED_0, GPIO_OUT);

        gpio_init(LED_1);
        gpio_set_dir(LED_1, GPIO_OUT);

        gpio_init(LED_2);
        gpio_set_dir(LED_2, GPIO_OUT);

        gpio_init(LED_3);
        gpio_set_dir(LED_3, GPIO_OUT);

        // Button Initialization
        gpio_init(OCT_DOWN);
        gpio_set_dir(OCT_DOWN, GPIO_IN);

        gpio_init(OCT_UP);
        gpio_set_dir(OCT_UP, GPIO_IN);

        gpio_init(ENCODE_PRESS);
        gpio_set_pulls(ENCODE_PRESS, true, false);
        gpio_set_dir(ENCODE_PRESS, GPIO_IN);

        gpio_init(ENCODE_A);
        // Fast slew rate
        gpio_set_slew_rate(ENCODE_A, GPIO_SLEW_RATE_FAST);
```

```
        // Pull up
        gpio_set_pulls(ENCODE_A, true, false);
        // Input
        gpio_set_dir(ENCODE_A, GPIO_IN);

        gpio_init(ENCODE_B);
        // Fast slew rate
        gpio_set_slew_rate(ENCODE_B, GPIO_SLEW_RATE_FAST);
        // Pull up
        gpio_set_pulls(ENCODE_B, true, false);
        // Input
        gpio_set_dir(ENCODE_B, GPIO_IN);

        // GPIO Interrupt Setup
        // All GPIO interrupts use the same callback function, must determine which GPIO line
triggered the callback
        gpio_set_irq_enabled_with_callback(ENCODE_PRESS, GPIO_IRQ_EDGE_FALL, true, &gpio_callb
ack);
        gpio_set_irq_enabled(OCT_DOWN, GPIO_IRQ_EDGE_FALL, true);
        gpio_set_irq_enabled(OCT_UP, GPIO_IRQ_EDGE_FALL, true);
        gpio_set_irq_enabled(ENCODE_A, GPIO_IRQ_EDGE_FALL, true);
        return 0;
}

// GPIO IRQ Trigger lookup table (used for GPIO event debugging)
static const char *gpio_irq_str[] = {
        "LEVEL_LOW",  // 0x1
        "LEVEL_HIGH", // 0x2
        "EDGE_FALL",  // 0x4
        "EDGE_RISE"       // 0x8
};

// Convert GPIO event to a printable string (GPIO Event debugging)
void gpio_event_string(char *buf, uint32_t events)
{
        for (uint i = 0; i < 4; i++)
        {
                uint mask = (1 << i);
                if (events & mask)
                {
                        // Copy this event string into the user string
                        const char *event_str = gpio_irq_str[i];
                        while (*event_str != '\0')
                        {
                                *buf++ = *event_str++;
                        }
                        events &= ~mask;

                        // If more events add ", "
                        if (events)
                        {
                                *buf++ = ',';
                                *buf++ = ' ';
                        }
                }
        }
        *buf++ = '\0';
}

// GPIO Interrupt callback function
void gpio_callback(uint gpio, uint32_t events)
{
        // Disable interrupts while servicing GPIO interrupt
```

```
        int status = save_and_disable_interrupts();
        char event_str[128];
        int volume_offset;
        // Print GPIO Event
        gpio_event_string(event_str, events);
        printf("GPIO IRQ CALLBACK\n GPIO Num %d\n, Event %s\n", gpio, event_str);

        // Determine which GPIO pin triggered the interrupt
        switch (gpio)
        {
        // Rotary Encoder switch
        case ENCODE_PRESS:
                // Mute MIDI Volume (essentially sends a Note OFF for all notes)
                mute_midi_volume(keyboard_global->channel);
                break;
        case OCT_DOWN:
                // Don't allow users to go below octave 2
                if (keyboard_global->octave <= 2)
                {
                        break;
                }
                // Decrement octave
                keyboard_global->octave--;
                // Turn off all MIDI Notes (changing octave with key press can leave higher oc
tave note playing)
                mute_midi_volume(0);
                break;
        case OCT_UP:
                // Don't allow users to go above octave 8
                if (keyboard_global->octave >= 8)
                {
                        break;
                }
                // Increment octave
                keyboard_global->octave++;
                // Turn off all MIDI Notes (changing octave with key press can leave lower oct
ave note playing)
                mute_midi_volume(0);
                break;
        case ENCODE_A:
                // Amount to adjust volume
                volume_offset = 0;
                // Check other Rotary Encoder line
                if (gpio_get(ENCODE_B) == 0)
                { // Clockwise
                        // Increment offset
                        volume_offset++;
                        // Don't allow negative volume
                        if (volume_offset > keyboard_global->volume)
                        {
                                keyboard_global->volume = 0;
                        }
                        else
                        {
                                // Apply volume decrease
                                keyboard_global->volume -= volume_offset;
                        }
                }
                // Otherwise, Counter-clockwise
                else
                {
                        // Increment offset
                        volume_offset++;
```

```
                        // Don't allow volumes over 127
                        if (keyboard_global->volume + volume_offset > 127)
                        {
                                keyboard_global->volume = 127;
                        }
                        else
                        {
                                // Apply volume increase
                                keyboard_global->volume += volume_offset;
                        }
                }
                // Send volume MIDI message
                change_midi_volume(0, keyboard_global->volume);
                break;
        }
        // Re-enable interrupts
        restore_interrupts(status);
}
```

```
// File: gpio.c
// Date Created: 9/18/22
// Authors: Landyn Francis (landyn.francis@maine.edu) Jacob Mealey (jacob.mealey@maine.edu)
// Purpose: Source file containing keyboard specific functions, suh as getting velocity in cm/
s and initializing the whole keyboard.
#include "keys.h"
#include "stdlib.h"
#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "adc_pos.h"

// Pass a key and get the centimeter per second velocity
// Uses start_pos, start_time, end_pos, and end_time, stored within the key struct
double key_get_velocity_cms(key *k)
{
    // Change in distance
    double delta_x = adc_pos_lut[k->start_pos] - adc_pos_lut[k->end_pos];
    // Change in time
    double delta_t = to_us_since_boot(k->end_time) - to_us_since_boot(k->start_time);
    // Return velocity
    return (k->activation_height) / (delta_t / 1000000.0);
}

// Initialize global keyboard
struct keyboard *init_keys()
{
    // Allocate space in memory for the full keyboard
    struct keyboard *keyb = malloc(sizeof(struct keyboard));

    // Initialize default channel to 0
    keyb->channel = 0;

    // Initialize default octave to Middle 5
    keyb->octave = 5;

    // Initialize volume to half
    keyb->volume = 64;

    // Initialize last_pressed to 0
    keyb->last_pressed = 0;

    // Initialize each key in keyboard
    for (uint32_t i = 0; i < KEY_COUNT; i++)
    {
        // Set current and previous position to top
        keyb->keys[i].current_pos = 0x5C;
        keyb->keys[i].prev_pos = 0x5C;
        // Initalize to not pressed
        keyb->keys[i].pressed = 0;
        // Initialize to not active
        keyb->keys[i].active = 0;

        keyb->keys[i].midi_active = 0;
    }

    // Set activation heights in cm
    keyb->keys[0].activation_height = 0.369;
    keyb->keys[1].activation_height = 0.482;
    keyb->keys[2].activation_height = 0.460;
    keyb->keys[3].activation_height = 0.393;
    keyb->keys[4].activation_height = 0.376;
    keyb->keys[5].activation_height = 0.373;
    keyb->keys[6].activation_height = 0.327;
```

```
    keyb->keys[7].activation_height = 0.388;
    keyb->keys[8].activation_height = 0.417;
    keyb->keys[9].activation_height = 0.309;
    keyb->keys[10].activation_height = 0.391;
    keyb->keys[11].activation_height = 0.361;
    return keyb;
}
```

```
// keys.h
// Authors: Jacob Mealey & Landyn Francis
// Purpose: See keys.c
#ifndef KEYS_H
#define KEYS_H

#include <stdint.h>
#include "pico/time.h"

// amount of keys in the keyboard
#define KEY_COUNT 12U

// The point at which a key is "pressed"
#define KEY_THRESH 63

// Largest ADC Value
#define KEY_UPPER 95U

// Lowest ADC Value
#define KEY_LOWER 3U

// Individual Key Structure: A structure that holds relevant (individual) key data
// current_pos: key position from current read
// prev_pos: key position from previous read
// pressed: Falling/Rising edge flag
// active: Held down key flag
// start_time: Timestamp of beginning of keypress
// end_time: Timestamp of end of keypress
// start_pos: Position of beggining of keypress
// end_pos: Position of end of keypress
typedef struct key
{
    uint8_t current_pos;        // Current position of the key (ADC Value)
    uint8_t prev_pos;           // Previous position of the key (ADC Value)
    uint8_t pressed;            // Flag showing whether or not the key has been pressed.
    uint8_t active;             // Flag showing whether or not the key is active. (Being held
down)
    absolute_time_t start_time; // The time when a key press begins
    absolute_time_t end_time;   // The time when a key press ends
    uint8_t start_pos;          // The position from the start of a keypress (ADC Value)
    uint8_t end_pos;            // The position from the end of a keypress (ADC Value)
    absolute_time_t midi_start; // Timer start for MIDI Velocity
    absolute_time_t midi_end;   // Timer end for MIDI Velocity
    uint8_t midi_active;
    float activation_height;

} key;

// Keyboard Structure: A structure that holds relevant keyboard data
// keys[]: Array of key structs for each individual key
// volume: 8 bit MIDI volume value
// channel: MIDI Channel (stays as 0)
// last_pressed: last pressed key number (0-11)
typedef struct keyboard
{
    key keys[KEY_COUNT];
    uint8_t volume;       // MIDI Volume (0-127)
    uint8_t octave;       // Keyboard Octave (Range of 2-8 Allowed)
    uint8_t channel;      // MIDI Channel (stays as 0)
    uint8_t last_pressed; // Last pressed key number (0-11)
} keyboard;

// Declare global keyboard structure
```

```
extern struct keyboard *keyboard_global;

// Initialization function
struct keyboard *init_keys();

// Individual key functions

// takes a type k and returns the decimal calculation velocity in cm/s
double key_get_velocity_cms(key *k);

#endif
```

```
// File: main.c
// Date: 10/25/22
// Authors: Jacob Mealey & Landyn Francis
// Purpose: Main loop of program. Performs all initializations for
// the display, ADC, GPIO, and second core.

#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "bsp/board.h"
#include "tusb.h"

#include "adc.h"
#include "pins.h"
#include "midi.h"
#include "keys.h"
#include "display.h"
#include "midi_velocity_lut.h"

#include "hardware/spi.h"
#include "hardware/gpio.h"
#include "pico/stdio.h"
#include "pico/multicore.h"
#include "pico/util/queue.h"

#define DISP_SIZE 30720

int keyboard_task();
void midi_task(struct adc_t *adc);
void core1_main();

// Global ADC Structure
struct adc_t *adc_global;

// Global Keyboard Structure
struct keyboard *keyboard_global;

// Global Display Structure
struct disp_t *disp_global;

// A queue for sending the current state of the keyboard
// to the second core
queue_t key_state_q;

/*------------- MAIN -------------*/
int main(void)
{

    // Initialize UART with 9600 baud rate
    stdio_init_all();
    uart_init(uart0, 9600);

    // Initialize GPIO pins
    pin_init();

    // Initialize ADC on SPI_1
    gpio_put(LED_0, 1);
    adc_global = init_adc(spi1, SPI1_CS);
    gpio_put(LED_0, 0);
    printf("ADC initialized\n");

    // Initialize Keyboard on SPI_0
```

```
    gpio_put(LED_1, 1);
    keyboard_global = init_keys();
    gpio_put(LED_1, 0);
    printf("Keyboard Initialized");

    // Initialize USB using TinyUSB
    gpio_put(LED_2, 1);
    tusb_init();
    gpio_put(LED_2, 0);
    printf("USB initialized\n");

    // Initialize Queue for inter core comms
    queue_init(&key_state_q, sizeof(struct keyboard), 2);

    // Launch display code on second core
    gpio_put(LED_3, 1);
    multicore_launch_core1(core1_main);
    gpio_put(LED_3, 0);

    while (1)
    {
        // TinyUSB Device task (setup USB configuration)
        tud_task();

        // MIDI Task for preparing MIDI transactions
        midi_task(adc_global);

        // Read keyboard
        keyboard_task();
    }

    return 0;
}
// Code to run on Second Core
// The second core runs the display, and handles any heavy compuations, such as velocity.
void core1_main()
{
    struct disp_t disp;
    printf("initializing display");
    init_disp(&disp, spi0, TFT_DC);
    printf("Entering main loop\n");

    // Buffer that will be written to the screen
    char print_buffer[128];

    uint8_t buffer[DISP_SIZE];
    int screen_size = 128 * 160;
    uint16_t *screen = malloc(screen_size * sizeof(uint16_t));
    if (screen == NULL)
    {
        printf("SCREEN BASED\n");
    }

    for (int i = 0; i < screen_size; i++)
    {
        screen[i] = 0xFFF;
    }

    screen_to_disp(screen, buffer, screen_size);
    disp_wr_cmd(&disp, DISP_RAMWR, buffer, DISP_SIZE);
    disp_wr_cmd(&disp, DISP_NOP, NULL, 0);

    sprintf(print_buffer, "MIDI Keyboard");
```

```
    draw_string(print_buffer, 55, 125, WHITE, BLACK);

    while (1)
    {
        keyboard keystate;
        if (!queue_try_remove(&key_state_q, &keystate))
        {
            continue; // we couldn't remove so we do nothing else
        }
        double vel;

        // Get most recently pressed key
        key active_key = keystate.keys[keystate.last_pressed];
        // Determine velocity
        vel = key_get_velocity_cms(&active_key);

        // Print first voltage level read on downward press
        sprintf(print_buffer, "Voltage = %f", 2.5 * (active_key.start_pos / (255.0)));
        draw_string(print_buffer, 25, 125, WHITE, BLACK);

        // Print calculated velocity
        sprintf(print_buffer, "Velocity: %.2f cm/s", vel);
        draw_string(print_buffer, 45, 125, WHITE, BLACK);

        // Print ADC value of current key
        sprintf(print_buffer, "pos: %d ", active_key.current_pos);
        draw_string(print_buffer, 55, 125, WHITE, BLACK);

        // Clear out old data from screen
        sprintf(print_buffer, "                    ");
        draw_string(print_buffer, 35, 125, WHITE, BLACK);

        // Determine time difference between start of keypress and end of keypress
        float deltaT = to_us_since_boot(active_key.end_time) - to_us_since_boot(active_key.sta
rt_time);

        // Print delta T
        sprintf(print_buffer, "delta t (ms): %.2f", (deltaT / 1000.0));
        draw_string(print_buffer, 35, 125, WHITE, BLACK);

        // Print which key number was pressed
        sprintf(print_buffer, "Last Pressed: %d", keystate.last_pressed);
        draw_string(print_buffer, 15, 125, WHITE, BLACK);
    }
}

// Query each key and determine if they are pressed
// If a key is pressed do the following:
//      1. Send a MIDI message with the appropraite note and MIDI velocity
//      2. Gather press data, and transfer to second core to display.
int keyboard_task()
{
    static int i = 0;

    // Calculate note based on current octave and key pressed
    uint8_t note = i + (keyboard_global->octave * 12);

    if (keyboard_global->keys[i].current_pos < 75 && keyboard_global->keys[i].midi_active == 0
)
    {
        keyboard_global->keys[i].midi_active = 1;
        keyboard_global->keys[i].midi_start = get_absolute_time();
        printf("First threshold!\n");
```

```
    }

    if (keyboard_global->keys[i].current_pos > 80 && keyboard_global->keys[i].midi_active == 1
)
    {
        keyboard_global->keys[i].midi_active = 0;
    }

    // If the key has been pressed, and was previously not pressed (falling edge)
    if (keyboard_global->keys[i].pressed == 1 && keyboard_global->keys[i].active == 0)
    {
        // Key is now active
        keyboard_global->keys[i].active = 1;

        keyboard_global->keys[i].midi_end = get_absolute_time();

        int delta_t = to_ms_since_boot(keyboard_global->keys[i].midi_end) - to_ms_since_boot(k
eyboard_global->keys[i].midi_start);

        if (delta_t > 127)
        {
            delta_t = 127;
        }
        if (delta_t < 1)
        {
            delta_t = 1;
        }

        uint8_t velocity = 127 - delta_t;
        // uint8_t velocity = output_start + slope * ((delta_t) - input_start);
        printf("MIDI Delta T (%d): %d, velocty = %d\n", i, delta_t, velocity);

        // Send NOTE ON MIDI Message
        if (send_general_midi_message(NOTE_ON, keyboard_global->channel, note, velocity, 0))
        {
            printf("MIDI NOTE ON FAIL\n");
            // Continue to iterate through keys
            i++;
            if (i == KEY_COUNT)
            {
                i = 0;
            }
            return 1;
        }
        // If Key is active, but no longer pressed (Rising edge)
    }
    else if (keyboard_global->keys[i].active == 1 && keyboard_global->keys[i].pressed == 0)
    {
        // Key is no longer active
        keyboard_global->keys[i].active = 0;

        // Update most recently pressed key
        keyboard_global->last_pressed = i;

        // push new state of the global keyboard
        // note: we are not doing the blocking one because if the
        // queue is full we can just skip this as it's not "critical"
        queue_try_add(&key_state_q, keyboard_global);

        // Send NOTE OFF MIDI Message
        if (send_general_midi_message(NOTE_OFF, keyboard_global->channel, note, 0, 0))
        {
            printf("MIDI NOTE OFF FAIL\n");
```

```
            // Continue to iterate through keys
            i++;
            if (i == KEY_COUNT)
            {
                i = 0;
            }
            return 1;
        }
    }

    // Continue to iterate through keys
    i++;
    if (i == KEY_COUNT)
    {
        i = 0;
    }
    return 0;
}

//--------------------------------------------------------------------+
// MIDI Task
//--------------------------------------------------------------------+

// MIDI Task to read incoming MIDI messages from host device. Must do this
// to be able to write messages.
void midi_task(struct adc_t *adc)
{
    static uint32_t start_ms = 0;

    // uint8_t const cable_num = 0; // MIDI jack associated with USB endpoint
    if (adc == NULL)
        return;

    // The MIDI interface always creates input and output port/jack descriptors
    // regardless of these being used or not. Therefore incoming traffic should be read
    // (possibly just discarded) to avoid the sender blocking in IO
    uint8_t packet[4];
    while (tud_midi_available())
        tud_midi_packet_read(packet);

    // Wait appropriate time between sending MIDI packages
    if (board_millis() - start_ms < 286)
        return;
    start_ms += 286;
}

//--------------------------------------------------------------------+
// Device callbacks
//--------------------------------------------------------------------+

// Invoked when device is mounted
void tud_mount_cb(void)
{
}

// Invoked when device is unmounted
void tud_umount_cb(void)
{
}

// Invoked when usb bus is suspended
// remote_wakeup_en : if host allow us  to perform remote wakeup
// Within 7ms, device must draw an average of current less than 2.5 mA from bus
```

```
void tud_suspend_cb(bool remote_wakeup_en)
{
    (void)remote_wakeup_en;
}

// Invoked when usb bus is resumed
void tud_resume_cb(void)
{
}
```

```
// File: midi.c
// Date Created: 9/18/22
// Authors: Landyn Francis (landyn.francis@maine.edu) Jacob Mealey (jacob.mealey@maine.edu)
// Purpose: Source file containing MIDI packaging and sending functions

#include <stdint.h>
#include <stdio.h>
#include "midi.h"
#include "keys.h"

// Send a general MIDI Message
// A "General MIDI Message" can be any of the following
// command_num: which MIDI command to send
// channel_num: which MIDI channel to use (leave as 0)
// note_num: note value for MIDI message
// velocity: MIDI velocity of keypress
// pressure: MIDI pressure of keypress (TBD)
uint8_t send_general_midi_message(uint8_t command_num, uint8_t channel_num, uint8_t note_num,
uint8_t velocity, uint8_t pressure)
{
        // Check for appropriate command number
        if ((command_num != NOTE_OFF) && (command_num != NOTE_ON) && (command_num != KEY_PRESS
URE))
        {
                return 1;
        }

        // Pressure command (not implemented)
        if (command_num == KEY_PRESSURE)
        {
                // Prepare MIDI Packet
                uint8_t message[3] = {command_num | channel_num, note_num, pressure};
                // Send MIDI message
                tud_midi_stream_write(0, message, 3);
                return 0;
        }

        // Prepare MIDI Packet
        uint8_t message[3] = {command_num | channel_num, note_num, velocity};
        // Send MIDI message
        tud_midi_stream_write(0, message, 3);
        return 0;
}

// Send a MIDI volume message
uint8_t change_midi_volume(uint8_t channel_num, uint8_t volume)
{
        // Double check if volume value is too large
        if (volume > 127)
        {
                keyboard_global->volume = 127;
                volume = 127;
        }
        // Prepare MIDI package
        uint8_t message[3] = {CONTROL_CHANGE | channel_num, VOLUME_CONTROLLER, volume};
        // Send MIDI message
        tud_midi_stream_write(0, message, 3);
        return 0;
}

// Send MIDI Volume mute message
// This specific controller number sends a note off for each key
uint8_t mute_midi_volume(uint8_t channel_num)
```

```
{
        // Prepare MIDI Packet
        uint8_t message[3] = {CONTROL_CHANGE | channel_num, MUTE_CONTROLLER, 0x00};
        // Send MIDI message
        tud_midi_stream_write(0, message, 3);
        return 0;
}
```

```
// File: midi.h
// Date Created: 9/18/22
// Authors: Landyn Francis (landyn.francis@maine.edu) Jacob Mealey (jacob.mealey@maine.edu)
// Purpose: Header file containing MIDI command codes, and other MIDI related defines

#include <stdint.h>
#include "bsp/board.h"
#include "tusb.h"

// MIDI COMMAND SET
#define NOTE_OFF 0x80             // 2 Data Bytes: Note #, Velocity
#define NOTE_ON 0x90              // 2 Data Bytes: Note #, Velocity (Note ON with 0 Velocity =
 NOTE OFF)
#define KEY_PRESSURE 0xA0         // 2 Data Bytes: Note #, Pressure
#define CONTROL_CHANGE 0xB0       // 2 Data Bytes: Controller #, Value)
#define PROGRAM_CHANGE 0xC0       // 1 Data Byte: Program number
#define CHANNEL_PRESSURE 0xD0 // 1 Data Byte: Channel Pressure
#define PITCH_BEND 0xE0           // 2 Data Bytes: (LSB,MSB)
#define SYS_EXCL_START 0xF0       // Variable Data Bytes
#define SYS_COMMON 0xF1           // 0, 1 or 2 Bytes
#define SYS_EXCL_END 0xF7         // 0 Data Bytes
#define SYS_REAL_TIME 0xF8        // 0 Data Bytes

#define VOLUME_CONTROLLER 7      // Controller number to control volume
#define MUTE_CONTROLLER 0x78 // CMM To Mute all Channels ("All sound off")

// Packs up and sends "general" MIDI message
// i.e. NOTE OFF / NOTE ON / KEY PRESSURE /
// pressure = 0 for NOTE OFF and NOTE ON messages
// velocity = 0 for KEY PRESSURE MESSAGES
// Return Values: 0 (success) 1 (error)
uint8_t send_general_midi_message(uint8_t command_num, uint8_t channel_num, uint8_t note_num,
uint8_t velocity, uint8_t pressure);

// Changes MIDI volume
// Command number not needed, here always the same for volume changes
// Volume is a 8 bit value ranging from 0 to 127 (7 bits usable).
uint8_t change_midi_volume(uint8_t channel_num, uint8_t volume);

// Mutes MIDI volume
// 0x78 Controller Number in MIDI Spec
uint8_t mute_midi_volume(uint8_t channel_num);
```

```
// File: pins.h
// Date Created: 9/16/22
// Authors: Landyn Francis (landyn.francis@maine.edu) Jacob Mealey (jacob.mealey@maine.edu)
// Purpose: Header file containing GPIO Pin definitions for MIDI Keyboard device

#ifndef PINS_H
#define PINS_H
#include <stdint.h>
#include "hardware/gpio.h"
#include "pico/stdio.h"
#include <stdio.h>

// SPI0 is the Display
#define SPI0_RX 0
#define SPI0_CS 1
#define SPI0_SCLK 2
#define SPI0_TX 3

// Other Display GPIO
#define DISPLAY_RESET 9
#define TFT_DC 10

// Four Status/Debugging LED's
#define LED_0 4
#define LED_1 5
#define LED_2 6
#define LED_3 7

// SPI1 is the ADC
#define SPI1_TX 11
#define SPI1_RX 12
#define SPI1_CS 13
#define SPI1_SCLK 14

// UART for Printing Debug Messages
#define UART0_TX 16
#define UART0_RX 17

// I2C1
#define I2C1_SDA 18
#define I2C1_SCL 19

// I2C0
#define I2C0_SDA 20
#define I2C0_SCL 21

// User Control Buttons
#define OCT_UP 23
#define OCT_DOWN 24

// Volume Dial (Rotary Encoder)
#define ENCODE_PRESS 26
#define ENCODE_B 27
#define ENCODE_A 28

// GPIO Interrupt Callback Function
void gpio_callback(uint gpio, uint32_t events);

// Initialize GPIO Pins
uint8_t pin_init();

// GPIO Event to String
void gpio_event_string(char *buf, uint32_t events);
```

```
#endif
```

```
/*
 * The MIT License (MIT)
 *
 * Copyright (c) 2019 Ha Thach (tinyusb.org)
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 *
 */

#ifndef _TUSB_CONFIG_H_
#define _TUSB_CONFIG_H_

#ifdef __cplusplus
extern "C"
{
#endif

//--------------------------------------------------------------------
// COMMON CONFIGURATION
//--------------------------------------------------------------------

// defined by board.mk
#ifndef CFG_TUSB_MCU
#error CFG_TUSB_MCU must be defined
#endif

// RHPort number used for device can be defined by board.mk, default to port 0
#ifndef BOARD_DEVICE_RHPORT_NUM
#define BOARD_DEVICE_RHPORT_NUM 0
#endif

// RHPort max operational speed can defined by board.mk
// Default to Highspeed for MCU with internal HighSpeed PHY (can be port specific), otherwise
FullSpeed
#ifndef BOARD_DEVICE_RHPORT_SPEED
#if (CFG_TUSB_MCU == OPT_MCU_LPC18XX || CFG_TUSB_MCU == OPT_MCU_LPC43XX || CFG_TUSB_MCU == OPT
_MCU_MIMXRT10XX || \
     CFG_TUSB_MCU == OPT_MCU_NUC505 || CFG_TUSB_MCU == OPT_MCU_CXD56)
#define BOARD_DEVICE_RHPORT_SPEED OPT_MODE_HIGH_SPEED
#else
#define BOARD_DEVICE_RHPORT_SPEED OPT_MODE_FULL_SPEED
#endif
#endif

// Device mode with rhport and speed defined by board.mk
#if BOARD_DEVICE_RHPORT_NUM == 0
#define CFG_TUSB_RHPORT0_MODE (OPT_MODE_DEVICE | BOARD_DEVICE_RHPORT_SPEED)
```

```
#elif BOARD_DEVICE_RHPORT_NUM == 1
#define CFG_TUSB_RHPORT1_MODE (OPT_MODE_DEVICE | BOARD_DEVICE_RHPORT_SPEED)
#else
#error "Incorrect RHPort configuration"
#endif

#ifndef CFG_TUSB_OS
#define CFG_TUSB_OS OPT_OS_NONE
#endif

// CFG_TUSB_DEBUG is defined by compiler in DEBUG build
// #define CFG_TUSB_DEBUG           0

/* USB DMA on some MCUs can only access a specific SRAM region with restriction on alignment.
 * Tinyusb use follows macros to declare transferring memory so that they can be put
 * into those specific section.
 * e.g
 * - CFG_TUSB_MEM SECTION : __attribute__ (( section(".usb_ram") ))
 * - CFG_TUSB_MEM_ALIGN   : __attribute__ ((aligned(4)))
 */
#ifndef CFG_TUSB_MEM_SECTION
#define CFG_TUSB_MEM_SECTION
#endif

#ifndef CFG_TUSB_MEM_ALIGN
#define CFG_TUSB_MEM_ALIGN __attribute__((aligned(4)))
#endif

  //--------------------------------------------------------------------
  // DEVICE CONFIGURATION
  //--------------------------------------------------------------------

#ifndef CFG_TUD_ENDPOINT0_SIZE
#define CFG_TUD_ENDPOINT0_SIZE 64
#endif

// SET as MIDI device
#define CFG_TUD_MIDI 1

// MIDI FIFO size of TX and RX
#define CFG_TUD_MIDI_RX_BUFSIZE (TUD_OPT_HIGH_SPEED ? 512 : 64)
#define CFG_TUD_MIDI_TX_BUFSIZE (TUD_OPT_HIGH_SPEED ? 512 : 64)

#ifdef __cplusplus
}
#endif

#endif /* _TUSB_CONFIG_H_ */
```

```
#include "tusb.h"

/* A combination of interfaces must have a unique product id, since PC will save device driver
 after the first plug.
 * Same VID/PID with different interface e.g MSC (first), then CDC (later) will possibly cause
 system error on PC.
 *
 * Auto ProductID layout's Bitmap:
 *   [MSB]       MIDI | HID | MSC | CDC          [LSB]
 */
#define _PID_MAP(itf, n)  ( (CFG_TUD_##itf) << (n) )
#define USB_PID           (0x4000 | _PID_MAP(CDC, 0) | _PID_MAP(MSC, 1) | _PID_MAP(HID, 2) | \
                           _PID_MAP(MIDI, 3) | _PID_MAP(VENDOR, 4) )

//--------------------------------------------------------------------+
// Device Descriptors
//--------------------------------------------------------------------+
tusb_desc_device_t const desc_device =
{
    .bLength            = sizeof(tusb_desc_device_t),
    .bDescriptorType    = TUSB_DESC_DEVICE,
    .bcdUSB             = 0x0200,
    .bDeviceClass       = 0x00,
    .bDeviceSubClass    = 0x00,
    .bDeviceProtocol    = 0x00,
    .bMaxPacketSize0    = CFG_TUD_ENDPOINT0_SIZE,

    .idVendor           = 0xCafe,
    .idProduct          = USB_PID,
    .bcdDevice          = 0x0100,

    .iManufacturer      = 0x01,
    .iProduct           = 0x02,
    .iSerialNumber      = 0x03,

    .bNumConfigurations = 0x01
};
```

```
// Invoked when received GET DEVICE DESCRIPTOR
// Application return pointer to descriptor
uint8_t const * tud_descriptor_device_cb(void)
{
  return (uint8_t const *) &desc_device;
}


//--------------------------------------------------------------------+
// Configuration Descriptor
//--------------------------------------------------------------------+

enum
{
  ITF_NUM_MIDI = 0,
  ITF_NUM_MIDI_STREAMING,
  ITF_NUM_TOTAL
};

#define CONFIG_TOTAL_LEN  (TUD_CONFIG_DESC_LEN + TUD_MIDI_DESC_LEN)

#if CFG_TUSB_MCU == OPT_MCU_LPC175X_6X || CFG_TUSB_MCU == OPT_MCU_LPC177X_8X || CFG_TUSB_MCU =
= OPT_MCU_LPC40XX
  // LPC 17xx and 40xx endpoint type (bulk/interrupt/iso) are fixed by its number
  // 0 control, 1 In, 2 Bulk, 3 Iso, 4 In etc ...
  #define EPNUM_MIDI   0x02
#else
  #define EPNUM_MIDI   0x01
#endif

uint8_t const desc_fs_configuration[] =
{
  // Config number, interface count, string index, total length, attribute, power in mA
  TUD_CONFIG_DESCRIPTOR(1, ITF_NUM_TOTAL, 0, CONFIG_TOTAL_LEN, 0x00, 100),

  // Interface number, string index, EP Out & EP In address, EP size
  TUD_MIDI_DESCRIPTOR(ITF_NUM_MIDI, 0, EPNUM_MIDI, 0x80 | EPNUM_MIDI, 64)
};

#if TUD_OPT_HIGH_SPEED
uint8_t const desc_hs_configuration[] =
{
  // Config number, interface count, string index, total length, attribute, power in mA
  TUD_CONFIG_DESCRIPTOR(1, ITF_NUM_TOTAL, 0, CONFIG_TOTAL_LEN, 0x00, 100),

  // Interface number, string index, EP Out & EP In address, EP size
  TUD_MIDI_DESCRIPTOR(ITF_NUM_MIDI, 0, EPNUM_MIDI, 0x80 | EPNUM_MIDI, 512)
};
#endif

// Invoked when received GET CONFIGURATION DESCRIPTOR
// Application return pointer to descriptor
// Descriptor contents must exist long enough for transfer to complete
uint8_t const * tud_descriptor_configuration_cb(uint8_t index)
{
  (void) index; // for multiple configurations

#if TUD_OPT_HIGH_SPEED
  // Although we are highspeed, host may be fullspeed.
  return (tud_speed_get() == TUSB_SPEED_HIGH) ?  desc_hs_configuration : desc_fs_configuration
;
#else
  return desc_fs_configuration;
```

```
#endif
}

//--------------------------------------------------------------------+
// String Descriptors
//--------------------------------------------------------------------+

// array of pointer to string descriptors
char const* string_desc_arr [] =
{
  (const char[]) { 0x09, 0x04 },              // 0: is supported language is English (0x0409)
  "UMaine ECE Capstone",                      // 1: Manufacturer
  "Hall Effect MIDI Keyboard",                // 2: Product
  "JL2040",                                   // 3: Serials, should use chip ID
};

static uint16_t _desc_str[32];

// Invoked when received GET STRING DESCRIPTOR request
// Application return pointer to descriptor, whose contents must exist long enough for transfe
r to complete
uint16_t const* tud_descriptor_string_cb(uint8_t index, uint16_t langid)
{
  (void) langid;

  uint8_t chr_count;

  if ( index == 0)
  {
    memcpy(&_desc_str[1], string_desc_arr[0], 2);
    chr_count = 1;
  }else
  {
    // Note: the 0xEE index string is a Microsoft OS 1.0 Descriptors.
    // https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/microsoft-defined-usb-
descriptors

    if ( !(index < sizeof(string_desc_arr)/sizeof(string_desc_arr[0])) ) return NULL;

    const char* str = string_desc_arr[index];

    // Cap at max char
    chr_count = strlen(str);
    if ( chr_count > 31 ) chr_count = 31;

    // Convert ASCII string into UTF-16
    for(uint8_t i=0; i<chr_count; i++)
    {
      _desc_str[1+i] = str[i];
    }
  }

  // first byte is length (including header), second byte is string type
  _desc_str[0] = (TUSB_DESC_STRING << 8 ) | (2*chr_count + 2);

  return _desc_str;
}
```