```c
/*
In this lab we were assigned the task of creating an LCD driver, a keypad driver, and a demonstrative application of our own design. This lab helped me reaffirm my understanding of how our development board's LCD interface works, gave me a significant amount of experience working with C-style strings, and allowed me to experiment a bit on my own (which is always appreciated).

I also took a few creative liberties with the LCD and keypad API. In the LCD API, I implemented a lcd_printf() function instead of lcd_print_string() and lcd_print_num(). I repurposed the lcd_print_string() as a helper function. I built the keypad API from the ground up because I wanted an interrupt driven keypad instead of a polling one. I also took the time to make some additional functions including a function for replacing the character lookup table, wait functions, conversion tools, and more.
*/

// file: main.c
// created by: Grant Wilk
// date created: 12/17/2019
// last modified: 1/11/2020
// description: A calculator program with overflow and divide by zero protection

# include <stdio.h>
# include <stdlib.h>
# include <limits.h>
# include "delay.h"
# include "lcd_driver.h"
# include "keypad_driver.h"

int main(void) {

    // initialize peripherals
    key_init();
    lcd_init();

    // op string contains the first operand, operator, and second operand terminated with a null terminator
    char opString[33];
    volatile int opStringLength = 0;

    // operand lengths
    int firstOperandLength = 0;
    int secondOperandLength = 0;

    // operand and operator flags
```

```
    // operand and operator flags
    char operatorEntered = 0;
    char secondOperandEntered = 0;
    char resultDisplayed = 0;

    while (1) {

        // block program flow and wait for a keypress from the keypad
        int key = key_get_wait();

        // if a number key is pressed
        if ((key >= 1 && key < 4) || (key >= 5 && key < 8) || (key >= 9 &&
        key < 12) || (key == 14)) {

            // do not accept new number inputs if the result is being
            displayed
            if (!resultDisplayed) {

                // do not accept new number inputs if the respective
                operand is longer than 9 digits
                if ((!operatorEntered && firstOperandLength < 9) ||
                (operatorEntered && secondOperandLength < 9)) {

                    // if no operator has been entered, increment the
                    first operand length
                    if (!operatorEntered) firstOperandLength++;

                    // if an operator has been entered, increment the
                    second operand length and set the second operand
                    entered flag
                    if (operatorEntered) {
                        secondOperandLength++;
                        secondOperandEntered = 1;
                    }

                    // get the character associated with the key
                    char key_char = key_to_char(key);

                    // add the character to the op string
                    opString[opStringLength++] = key_char;

                    // print the character to the LCD
                    lcd_printf("%c", key_char);

                }

            }

```

```
        // if an operator key is pressed
        } else if ((key == 4) || (key == 8) || (key == 12) || (key == 16))
        {

            // as long as there is some sort of input
            if (opStringLength != 0) {

                char opChar;

                // set operator entered flag
                operatorEntered = 1;

                // clear result displayed flag
                resultDisplayed = 0;

                // add the operator to the op string and print it to the LCD
                switch (key) {

                    // add key pressed
                    case 4:
                        opChar = '+';
                        break;

                    // subtract key pressed
                    case 8:
                        opChar = '-';
                        break;

                    // multiply key pressed
                    case 12:
                        opChar = '*';
                        break;

                    // divide key pressed
                    case 16:
                        opChar = '/';
                        break;

                    // unknown key pressed (should never occur)
                    default:
                        break;

                }

                // append the operator character to the op string
                opString[opStringLength++] = opChar;

```

```
            // move the cursor to the top right corner of the LCD
            lcd_cursor_set(15, 0);

            // print the operator character to the LCD
            lcd_printf("%c", opChar);

            // move the cursor to the bottom left corner of the LCD
            lcd_cursor_set(0, 1);

        }

    // if the equals key is pressed
    } else if (key == 15) {

        // if the second operand has been entered
        if (secondOperandEntered) {

            // add a null terminator to the end of the op string
            opString[opStringLength++] = '\0';

            // parse the op string
            int firstOperand;
            char operatorChar;
            int secondOperand;

            sscanf(opString, "%d%c%d", &firstOperand, &operatorChar,
            &secondOperand);

            // overflow and underflow flags
            char overflow;
            char underflow;

            // do the calculation
            int result;

            switch (operatorChar) {

                // add operator
                case '+':;

                    // determine if addition will overflow or underflow
                    overflow = (secondOperand > 0) && (firstOperand >
                    INT_MAX - secondOperand);
                    underflow = (secondOperand < 0) && (firstOperand <
                    INT_MIN - secondOperand);

                    // default to zero if overflow/underflow,
                    otherwise complete the calculation
```

```
            otherwise complete the calculation
            if (overflow || underflow) {
                result = 0;
            } else {
                result = firstOperand + secondOperand;
            }

            break;

        // subtract operator
        case '-':;

            // determine if subtraction will overflow or underflow
            overflow = (secondOperand < 0) && (firstOperand > INT_MAX + secondOperand);
            underflow = (secondOperand > 0) && (firstOperand < INT_MIN + secondOperand);

            // default to zero if overflow/underflow, otherwise complete the calculation
            if (overflow || underflow) {
                result = 0;
            } else {
                result = firstOperand - secondOperand;
            }

            break;

        // multiply operator
        case '*':;

            // determine if multiplication will overflow or underflow
            overflow = firstOperand > INT_MAX / secondOperand;
            underflow = firstOperand < INT_MIN / secondOperand;

            // default to zero if overflow/underflow, otherwise complete the calculation
            if (overflow || underflow) {
                result = 0;
            } else {
                result = firstOperand * secondOperand;
            }

            break;

        // divide operator
```

```
                case '/':;

                    // default to zero if dividing by zero, otherwise complete the calculation
                    if (secondOperand == 0) {
                        result = 0;
                    } else {
                        result = firstOperand / secondOperand;
                    }

                    break;

                // unknown operator, default to 0
                default:
                    result = 0;

            }

            // clear the LCD
            lcd_clear();

            // print result to the LCD
            lcd_printf("%d", result);

            // copy the result back into the op string for chained calculations
            sprintf(opString, "%d", result);

            // move the op string length pointer to the end of the first operand
            opStringLength = 0;
            while (opString[opStringLength++ + 1] != '\0');

            // update operand lengths
            firstOperandLength = opStringLength;
            secondOperandLength = 0;

            // update flags
            operatorEntered = 0;
            secondOperandEntered = 0;
            resultDisplayed = 1;

        }

    // if the clear key is pressed
    } else {

        // reset op string length
```

```
            opStringLength = 0;

            // reset operand lengths
            firstOperandLength = 0;
            secondOperandLength = 0;

            // reset operand entered flags
            operatorEntered = 0;
            secondOperandEntered = 0;
            resultDisplayed = 0;

            // clear the LCD
            lcd_clear();

        }

    }

}
```

```c
// file: lcd_driver.h
// created by: Grant Wilk
// date created: 12/17/2019
// last modified: 1/5/2020
// description: Header file for lcd_driver.c

// Initializes the LCD pins and readys the LCD peripheral for use
void lcd_init(void);

// Clears all of the data from the LCD
void lcd_clear(void);

// Sets the cursor back to its home position on the LCD
void lcd_cursor_home(void);

// Hides the blinking cursor on the LCD
void lcd_cursor_set(int x, int y);

// Sets the cursor to a specific (x, y) position on the LCD
void lcd_cursor_show(void);

// Shows the blinking cursor on the LCD
void lcd_cursor_hide(void);

// Prints a formatted string to the LCD
void lcd_printf(const char * format, ...);

```

```c
// file: lcd_driver.c
// created by: Grant Wilk
// date created: 12/17/2019
// last modified: 1/5/2020
// description: Contains functions for driving the LCD on the CE development board

# include <stdio.h>
# include <stdarg.h>
# include <stdint.h>
# include "delay.h"
# include "lcd_driver.h"

// RCC Addresses
# define RCC_BASE 0x40023800
# define RCC_AHB1ENR (RCC_BASE + 0x0030)

// RCC Values
# define RCC_GPIOAEN (1 << 0)
# define RCC_GPIOCEN (1 << 2)

// GPIOA Addresses
# define GPIOA_BASE 0x40020000
# define GPIOA_MODER (GPIOA_BASE + 0x00)
# define GPIOA_ODR (GPIOA_BASE + 0x14)

// GPIOA Values
# define GPIOA_ODR_LCD_DATABUS 0xFF0
# define GPIOA_MODER_LCD_DATABUS_OUTPUT 0x555500

// GPIOC Addresses
# define GPIOC_BASE 0x40020800
# define GPIOC_MODER (GPIOC_BASE + 0x00)
# define GPIOC_ODR (GPIOC_BASE + 0x14)

// GPIOC Values
# define GPIOC_ODR_LCD_RS (1 << 8)
# define GPIOC_ODR_LCD_RW (1 << 9)
# define GPIOC_ODR_LCD_E (1 << 10)
# define GPIOC_MODER_LCD_RS_OUTPUT (1 << 16)
# define GPIOC_MODER_LCD_RW_OUTPUT (1 << 18)
# define GPIOC_MODER_LCD_E_OUTPUT (1 << 20)

// Other Values
# define DATABUS_MAX_VALUE 0xFF

// LCD Characteristics
# define LCD_ROW_LENGTH 40
```

```c
# define LCD_ROW_LENGTH 40
# define LCD_MAX_LENGTH 80

// Static Function Prototypes
static void lcd_print_string(char s[]);
static void lcd_write_instruction(int instruction);
static void lcd_write_char(char character);
static void lcd_instr_clear(void);
static void lcd_instr_return_home(void);
static void lcd_instr_entry_mode_set(int cursorDirection, int displayShift);
static void lcd_instr_display_on_off(int displayOn, int cursorOn, int cursorPosOn);
static void lcd_instr_cursor_display_shift(int shift, int direction);
static void lcd_instr_function_set(int dataInterface, int lineNumber, int fontSize);

// Global Variables
static uint32_t * const gpioaODR = (uint32_t *) GPIOA_ODR;
static uint32_t * const gpiocODR = (uint32_t *) GPIOC_ODR;

// Initializes the LCD pins and readies the LCD peripheral for use
// @ param void
// @ return void
void lcd_init(void) {
    // enable GPIOA and GPIOC in RCC
    uint32_t * rccAHB1ENR = (uint32_t *) RCC_AHB1ENR;
    * rccAHB1ENR |= RCC_GPIOAEN | RCC_GPIOCEN;

    // set LCD databus pins as outputs
    uint32_t * gpioaMODER = (uint32_t *) GPIOA_MODER;
    * gpioaMODER |= GPIOA_MODER_LCD_DATABUS_OUTPUT;

    // set LCD control pins as outputs
    uint32_t * gpiocMODER = (uint32_t *) GPIOC_MODER;
    * gpiocMODER |= GPIOC_MODER_LCD_RS_OUTPUT | GPIOC_MODER_LCD_RW_OUTPUT | GPIOC_MODER_LCD_E_OUTPUT;

    // function set 8-bit interface, 1 line, and 5x8 font size
    lcd_instr_function_set(1, 1, 0);

    // function set 8-bit interface, 1 line, and 5x8 font size
    lcd_instr_function_set(1, 1, 0);

    // turn on display and cursor position
    lcd_instr_display_on_off(1, 0, 1);

    // clear the display
```

```c
    lcd_clear();

    // entry mode set increment position, display shift off
    lcd_instr_entry_mode_set(1, 0);
}

// Clears the display of the LCD
// @ param void
// @ return void
void lcd_clear(void) {
    lcd_instr_clear();
}

// Moves the cursor back to it's home position
// @ param void
// @ return void
void lcd_cursor_home(void) {
    lcd_instr_return_home();
}

// Sets the cursor to a specific (x, y) position on the LCD
// @ param x - the zero-based x-position
// @ param y - the zero-based y-position
// @ return void
void lcd_cursor_set(int x, int y) {

    // move the cursor back to its home position
    lcd_cursor_home();

    // move the cursor a full row length for each value of y
    for (int i = 0; i < y * LCD_ROW_LENGTH; i++) {
        lcd_instr_cursor_display_shift(0, 1);
    }

    // move the cursor over a single column for each value of x
    for (int i = 0; i < x; i++) {
        lcd_instr_cursor_display_shift(0, 1);
    }
}

// Shows the blinking cursor on the LCD
// @ param void
// @ return void
void lcd_cursor_show(void) {
    lcd_instr_display_on_off(1, 0, 1);
}

// Hides the blinking cursor on the LCD
```

```c
// @ param void
// @ return void
void lcd_cursor_hide(void) {
    lcd_instr_display_on_off(1, 0, 0);
}

// Prints a formatted string to the LCD
// @ param format - a variable length argument
// @ return void
void lcd_printf(const char * format, ... ) {

    // initialize the variable arg list
    va_list args;
    va_start(args, format);

    // declare the buffer that will store the formatted string
    char buffer [LCD_MAX_LENGTH];

    // print to the string buffer
    vsprintf(buffer, format, args);

    // print the string buffer to the LCD
    lcd_print_string(buffer);

    // terminate the variable arg list
    va_end(args);

}

// Prints a string to the LCD
// @ param s - the string to print
// @ return void
static void lcd_print_string(char s[]) {

    int offset = 0;

    // print every character in the string
    while (s[offset] != '\0') {
        lcd_write_char(s[offset++]);
    }

}

// Writes an instruction to the LCD
// @ param instruction - the hexadecimal instruction to write to the LCD
// @ return void
static void lcd_write_instruction(int instruction) {
```

```
    // only write the instruction if it fits in the databus (less than or equal to 0xFF)
    if (instruction <= DATABUS_MAX_VALUE) {

        // set E
        * gpiocODR |= GPIOC_ODR_LCD_E;

        // clear RS and RW
        * gpiocODR &= ~(GPIOC_ODR_LCD_RS | GPIOC_ODR_LCD_RW);

        // clear databus pins and copy instruction to them
        * gpioaODR &= ~(GPIOA_ODR_LCD_DATABUS);
        * gpioaODR |= (instruction << 4);

        // clear E to write the instruction
        * gpiocODR &= ~(GPIOC_ODR_LCD_E);
    }
}

// Writes a character to the LCD
// @ param character - the character to write to the LCD
// @ return void
static void lcd_write_char(char character) {
    // only write the character if it fits in the databus (less than or equal to 0xFF)
    if (character <= DATABUS_MAX_VALUE) {

        // set E and RS
        * gpiocODR |= GPIOC_ODR_LCD_E | GPIOC_ODR_LCD_RS;

        // clear RW
        * gpiocODR &= ~(GPIOC_ODR_LCD_RW);

        // clear databus pins and copy character to them
        * gpioaODR &= ~(GPIOA_ODR_LCD_DATABUS);
        * gpioaODR |= (character << 4);

        // clear E to write the instruction
        * gpiocODR &= ~(GPIOC_ODR_LCD_E);

        // delay for 10us
        delay_us(37);
    }
}

// Clear display instruction for the LCD
// @ param void
```

```c
// @ return void
static void lcd_instr_clear(void) {
    // the clear display instruction
    int instruction = (1 << 0);

    // write the instruction
    lcd_write_instruction(instruction);

    // delay
    delay_us(1520);
}

// Return home instruction for the LCD
// @ param void
// @ return void
static void lcd_instr_return_home(void) {
    // the return home instruction
    int instruction = (1 << 1);

    // write the instruction
    lcd_write_instruction(instruction);

    // delay
    delay_us(1520);
}

// Entry mode set instruction for the LCD
// @ param cursorDirection - when data is written, increment the cursor position if 1, decrement the cursor position if 0
// @ param displayShift - shift enabled if 1, shift disabled if 0
// @ return void
static void lcd_instr_entry_mode_set(int cursorDirection, int displayShift) {

    // the base entry mode set instruction
    int instruction = (1 << 2);

    // set instruction parameters based on function parameters
    if (cursorDirection) instruction |= (1 << 1);
    if (displayShift) instruction |= (1 << 0);

    // write instruction
    lcd_write_instruction(instruction);

    // delay
    delay_us(37);
}

```

```c
// Display ON/OFF instruction for the LCD
// @ param displayOn - entire display on
// @ param cursorOn - show cursor (underscore)
// @ param cursorBlinkOn - show cursor blink (block)
// @ return void
static void lcd_instr_display_on_off(int displayOn, int cursorOn, int cursorBlinkOn) {

    // the base display on/off instruction
    int instruction = (1 << 3);

    // set instruction parameters based on function parameters
    if (displayOn) instruction |= (1 << 2);
    if (cursorOn) instruction |= (1 << 1);
    if (cursorBlinkOn) instruction |= (1 << 0);

    // write the instruction
    lcd_write_instruction(instruction);

    // delay
    delay_us(37);
}

// Cursor display/shift instruction for the LCD
// @ param shiftSelect - shift the display if 1, shift the cursor if 0
// @ param direction - right if 1, left if 0
// @ return void
static void lcd_instr_cursor_display_shift(int shiftSelect, int direction)
{
    // the base cursor display/shift instruction
    int instruction = (1 << 4);

    // set instruction parameters based on function parameters
    if (shiftSelect) instruction |= (1 << 3);
    if (direction) instruction |= (1 << 2);

    // write instruction
    lcd_write_instruction(instruction);

    // delay
    delay_us(37);
}

// Function set instruction for the LCD
// @ param dataInterface - 8-bit interface if 0, 4-bit interface if 1
// @ param lineNumber - line number 2 if 0, line number 1 if 1
// @ param fontSize - font size 5x11 if 0, font size 5x8 if 1
// @ return void
```

```
// @ return void
static void lcd_instr_function_set(int dataInterface, int lineNumber, int fontSize) {

    // the base function set instruction
    int instruction = (1 << 5);

    // set instruction parameters based on function parameters
    if (dataInterface) instruction |= (1 << 4);
    if (lineNumber) instruction |= (1 << 3);
    if (fontSize) instruction |= (1 << 2);

    // write the instruction
    lcd_write_instruction(instruction);

    // delay
    delay_us(37);
}
```

```c
// file: keypad_driver.h
// created by: Grant Wilk
// date created: 1/5/2020
// last modified: 1/5/2020
// description: Header file for keypad_driver.c

// Initializes the keypad pins and readies the keypad peripheral for use
void key_init(void);

// Clears the last key pressed
void key_clear(void);

// Blocks program flow and waits for a keypress
void key_wait(void);

// Gets the last key pressed and returns it
int key_get(void);

// Blocks program flow, waits for a keypress, and returns it
int key_get_wait(void);

// Gets the last key pressed, converts it to a character, and returns it
char key_get_char(void);

// Blocks program flow, waits for a keypress, converts it to a character, and returns it
char key_get_char_wait(void);

// Converts a keypress to a character and returns it
char key_to_char(int key);

// Sets a new character LUT for get character functions
void key_set_char_lut(char * newCharLUT);

```

```c
// file: keypad_driver.c
// created by: Grant Wilk
// date created: 1/5/2020
// last modified: 1/5/2020
// description: Contains functions for driving the keypad on the CE development board

# include <stdint.h>
# include "delay.h"
# include "keypad_driver.h"
# include "lcd_driver.h"

// RCC Addresses
# define RCC_BASE 0x40023800
# define RCC_AHB1ENR (RCC_BASE + 0x30)
# define RCC_APB2ENR (RCC_BASE + 0x44)

// RCC Values
# define RCC_AHB1ENR_GPIOCEN (1 << 2)
# define RCC_APB2ENR_SYSCFGEN (1 << 14)

// GPIOC Addresses
# define GPIOC_BASE 0x40020800
# define GPIOC_MODER (GPIOC_BASE + 0x00)
# define GPIOC_PUPDR (GPIOC_BASE + 0x0C)
# define GPIOC_IDR (GPIOC_BASE + 0x10)
# define GPIOC_ODR (GPIOC_BASE + 0x14)

// GPIOC Values
# define GPIOC_COLUMNS 0xFF
# define GPIOC_ROWS 0xFF00
# define GPIOC_MODER_COLUMNS_OUTPUT 0x55
# define GPIOC_MODER_ROWS_OUTPUT 0x5500
# define GPIOC_ODR_COLUMNS 0x0F
# define GPIOC_ODR_ROWS 0xF0
# define GPIOC_PUPDR_COLUMNS_PULLDOWN 0xAA
# define GPIOC_PUPDR_ROWS_PULLDOWN 0xAA00

// SYSCFG Addresses
# define SYSCFG_BASE 0x40013800
# define SYSCFG_EXTICR1 (SYSCFG_BASE + 0x08)

// SYSCFG Values
# define SYSCFG_EXTIX_TO_PIN_C (1 << 1)

// EXTI Addresses
# define EXTI_BASE 0x40013C00
# define EXTI_IMR (EXTI_BASE + 0x00)
```

```c
# define EXTI_IMR (EXTI_BASE + 0x00)
# define EXTI_RTSR (EXTI_BASE + 0x08)
# define EXTI_PR (EXTI_BASE + 0x14)

// EXTI Values
# define EXTI_0_THRU_4 0x0F

// NVIC Addresses
# define NVIC_BASE 0xE000E100
# define NVIC_ISER0 (NVIC_BASE + 0x00)
# define NVIC_ICER0 (NVIC_BASE + 0x80)

// NVIC Values
# define NVIC_6_THRU_9 (0b1111 << 6)

// Register Pointers
static uint32_t * const gpiocMODER = (uint32_t *) GPIOC_MODER;
static uint32_t * const gpiocIDR = (uint32_t *) GPIOC_IDR;

// Row Lookup Table
const static int rowLUT[9] = {0, 0, 1, 1, 2, 2, 2, 2, 3};

// Character Lookup Table
const static char defaultCharLUT[17] =
{
    '\0',
    '1', '2', '3', 'A',
    '4', '5', '6', 'B',
    '7', '8', '9', 'C',
    '*', '0', '#', 'D'
};

// Character Lookup Table Pointer
static char * charLUT = (char *) defaultCharLUT;

// Last Keypress Variable
static char lastKeypress = 0;

// Initializes the keypad pins and readies the keypad peripheral for use
// @ param void
// @ return void
void key_init(void) {

    // enable GPIOC in RCC
    uint32_t * rccAHB1ENR = (uint32_t *) RCC_AHB1ENR;
    * rccAHB1ENR |= RCC_AHB1ENR_GPIOCEN;

    // set ODR to output 1's for both rows and columns
```

```
    uint32_t * gpiocODR = (uint32_t *) GPIOC_ODR;
    * gpiocODR |= GPIOC_ODR_COLUMNS | GPIOC_ODR_ROWS;

    // set PUPDR as pull-down for both rows and columns
    uint32_t * gpiocPUPDR = (uint32_t *) GPIOC_PUPDR;
    * gpiocPUPDR &= ~(GPIOC_ROWS | GPIOC_COLUMNS);
    * gpiocPUPDR |= (GPIOC_PUPDR_COLUMNS_PULLDOWN |
    GPIOC_PUPDR_ROWS_PULLDOWN);

    // configure the columns as inputs and rows as outputs
    * gpiocMODER &= ~(GPIOC_COLUMNS | GPIOC_ROWS);
    * gpiocMODER |= (GPIOC_MODER_ROWS_OUTPUT);

    // enable SYSCFG in RCC
    uint32_t * rccAPB2ENR = (uint32_t *) RCC_APB2ENR;
    * rccAPB2ENR |= RCC_APB2ENR_SYSCFGEN;

    // map EXTI to pins on GPIOC
    uint32_t * syscfgEXTICR1 = (uint32_t *) SYSCFG_EXTICR1;
    * syscfgEXTICR1 |= (SYSCFG_EXTIX_TO_PIN_C);
    * syscfgEXTICR1 |= (SYSCFG_EXTIX_TO_PIN_C << 4);
    * syscfgEXTICR1 |= (SYSCFG_EXTIX_TO_PIN_C << 8);
    * syscfgEXTICR1 |= (SYSCFG_EXTIX_TO_PIN_C << 12);

    // unmask EXTI0-EXTI3 in EXTI IMR
    uint32_t * extiIMR = (uint32_t *) EXTI_IMR;
    * extiIMR |= EXTI_0_THRU_4;

    // set interrupts on rising edge for EXTI0-EXTI3 in EXTI RTSR
    uint32_t * extiRTSR = (uint32_t *) EXTI_RTSR;
    * extiRTSR |= EXTI_0_THRU_4;

    // enable interrupt in NVIC
    uint32_t * nvicISER0 = (uint32_t *) NVIC_ISER0;
    * nvicISER0 = NVIC_6_THRU_9;

    // clear the last keypress from memory
    key_clear();

}

// Clears the last key pressed
// @ param void
// @ return void
void key_clear(void) {
    lastKeypress = 0;
}

```

```c
// Blocks program flow and waits for a keypress
// @ param void
// @ return void
void key_wait(void) {
    key_clear();
    while (key_get() == 0);
}

// Gets the last key pressed and returns it
// @ param void
// @ return the last key pressed or 0 if no key was pressed
int key_get(void) {
    return lastKeypress;
}

// Blocks program flow, waits for a keypress, and returns it
// @ param void
// @ return the last key pressed
int key_get_wait(void) {
    key_wait();
    return key_get();
}

// Gets the last key pressed, converts it to a character, and returns it
// @ param void
// @ return the character of the last key pressed or 0 if no key was pressed
char key_get_char(void) {
    return charLUT[key_get()];
}

// Blocks program flow, waits for a keypress, converts it to a character, and returns it
// @ param void
// @ return the last key pressed
char key_get_char_wait(void) {
    return charLUT[key_get_wait()];
}


// Converts a keypress to a character and returns it
// @ param key - the number of a keypress
// @ return the character associated with the keypress if it is in the range of valid characters, otherwise return a null pointer
char key_to_char(int key) {
    if (key > 0 && key <= 16) {
        return charLUT[key];
    } else {
```

```c
    } else {
        return '\0';
    }
}

// Sets a new character LUT for get character functions
// @ param newCharLUT - the new character LUT, a 17 element character array that begins with a null terminator character
// @ return void
void key_set_char_lut(char newCharLUT[]) {
    charLUT = newCharLUT;
}

// Handles keypad interrupts
// @ param column - the column the interrupt occurred on
// @ return void
static void key_interrupt_handler(int column) {

    // mask EXTI0-EXTI3 in EXTI IMR
    uint32_t * extiIMR = (uint32_t *) EXTI_IMR;
    * extiIMR &= ~(EXTI_0_THRU_4);

    // set rows as inputs and columns as outputs
    * gpiocMODER &= ~(GPIOC_COLUMNS | GPIOC_ROWS);
    * gpiocMODER |= (GPIOC_MODER_COLUMNS_OUTPUT);

    // delay 40 milliseconds for debouncing
    delay_ms(40);

    // get the one-hot value of the row
    int row = (* gpiocIDR >> 4) & 0xF;

    // if a key is still pressed and the keypress is in a valid position
    if (row > 0 && row <= 9) {

        // get the actual value of the row from the LUT
        row = rowLUT[row];

        // update the last keypress variable
        lastKeypress = row * 4 + column + 1;
    }

    // set rows as outputs and columns as inputs
    * gpiocMODER &= ~(GPIOC_COLUMNS | GPIOC_ROWS);
    * gpiocMODER |= (GPIOC_MODER_ROWS_OUTPUT);

    // clear the pending interrupt
    uint32_t * extiPR = (uint32_t *) EXTI_PR;
```

```
    * extiPR = (1 << column);

    // unmask EXTI0-EXTI3 in EXTI IMR
    * extiIMR |= EXTI_0_THRU_4;

}

// Keypad column 0 interrupt handler
// @ param void
// @ return void
void EXTI0_IRQHandler(void) {
    key_interrupt_handler(0);
}

// Keypad column 1 interrupt handler
// @ param void
// @ return void
void EXTI1_IRQHandler(void) {
    key_interrupt_handler(1);
}

// Keypad column 2 interrupt handler
// @ param void
// @ return void
void EXTI2_IRQHandler(void) {
    key_interrupt_handler(2);
}

// Keypad column 3 interrupt handler
// @ param void
// @ return void
void EXTI3_IRQHandler(void) {
    key_interrupt_handler(3);
}
```

```c
// file: delay.h
// created by: Grant Wilk
// date created: 12/10/2019
// last modified: 12/10/2019
// description: header file for delay.c

// delays for some number of milliseconds
// @ param milliseconds - the number of milliseconds to delay for
// @ return void
void delay_ms(int milliseconds);

// delays for some number of microseconds
// @ param microseconds - the number of microseconds to delay for
// @ return void
void delay_us(int microseconds);

```

```c
// file: delay.c
// created by: Grant Wilk
// date created: 12/10/2019
// last modified: 12/10/2019
// description: contains functions for making delays using the systick timer

// STK addresses
#define STK_BASE 0xE000E010
#define STK_CTRL (STK_BASE + 0x00)
#define STK_LOAD (STK_BASE + 0x04)
#define STK_VAL  (STK_BASE + 0x08)

// STK values
#define STK_CTRL_ENABLE (1 << 0)
#define STK_CTRL_COUNTFLAG (1 << 16)
#define STK_RESET 0x0

// time values
# define CYCLES_PER_MS 2000
# define CYCLES_PER_US 2

// delays for some number of milliseconds
// @ param milliseconds - the number of milliseconds to delay for
// @ return void
void delay_ms(int milliseconds){
    int* stk_ctrl = (int*) STK_CTRL;
    int* stk_load = (int*) STK_LOAD;
    int* stk_val = (int*) STK_VAL;

    // reset systick
    *stk_val = STK_RESET;

    // set systick reload value
    *stk_load = milliseconds * CYCLES_PER_MS;

    // start systick
    *stk_ctrl |= STK_CTRL_ENABLE;

    // wait until the count flag goes high
    while (!(*stk_ctrl & STK_CTRL_COUNTFLAG));

    // stop systick
    *stk_ctrl &= ~STK_CTRL_ENABLE;
}

// delays for some number of microseconds
// @ param microseconds - the number of microseconds to delay for
// @ return void
```

```c
// @ return void
void delay_us(int microseconds){
    int* stk_ctrl = (int*) STK_CTRL;
    int* stk_load = (int*) STK_LOAD;
    int* stk_val = (int*) STK_VAL;

    // reset systick
    *stk_val = STK_RESET;

    // set systick reload value
    *stk_load = microseconds * CYCLES_PER_US;

    // start systick
    *stk_ctrl |= STK_CTRL_ENABLE;

    // wait until the count flag goes high
    while (!(*stk_ctrl & STK_CTRL_COUNTFLAG));

    // stop systick
    *stk_ctrl &= ~STK_CTRL_ENABLE;

}

```