Laboratory #4
Keypad Interfacing

Objectives:

- Write a device driver for interfacing a keypad to the PIC32 MCU.
- Make use of terminal I/O functions for message output on a terminal window.

Files provided on Blackboard:

- uart.h
- uart.c
- main.c (incomplete)

Task:

The I/O shield provided for your development has a 12-key keypad module on it. In this lab assignment you will be implementing a device driver program to interface the keypad module to your PIC32 MCU. As shown in the diagram below, seven pins of PORTB are used for the interfacing.

To detect if and which key is pressed, in the implementation of your device driver, you can sequentially set one row or column to low and then check if any of the columns or rows, respectively, returns a zero value. The intersection of a column that returns a value of zero and the currently selected row corresponds to the key pressed.

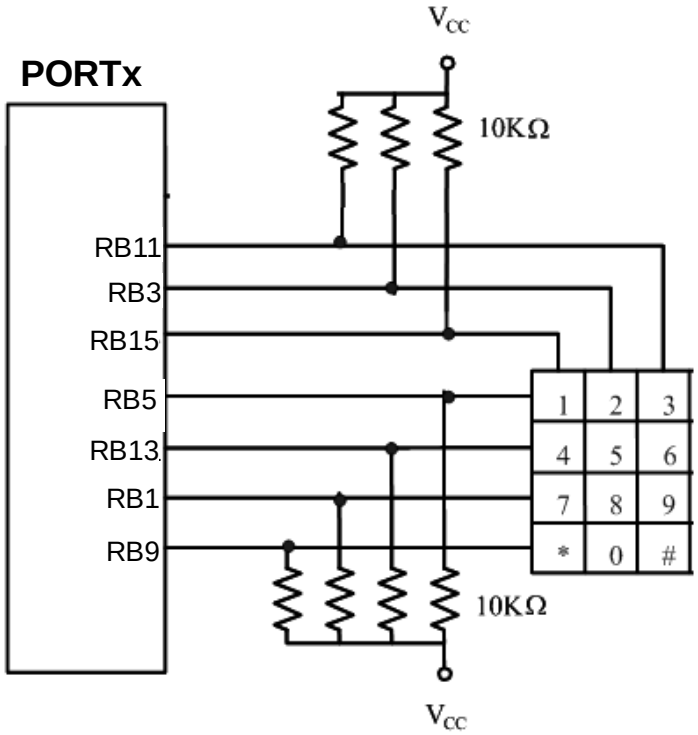

Figure 1: Keypad interfacing to the PIC32 GPIO pins

In your program, you will implement the following three functions with the given prototypes:

1. *void **initKeypad**(void)*

   As you can see in the schematic diagram of the keypad, the PIC32 pins that are used to interface to the keypad are select pins from PORTB. Since pins 11 and 13 of this port are shared with the JTAG pins, it is important to disable the JTAG module so these pins could be used for the keypad. To achieve this you need to include the following command at the beginning of your ***initKeypad()*** function:

   ```
   DDPCONbits.JTAGEN = 0; // disable the JTAG port
   ```

   Another important point to remember about PIC32's PORTB is that this port's pins support both analog and digital modes of operation. The internal analog to digital converter (ADC) module uses these pins for analog input. The actual mode for any of the PORTB pins needs to be selected using the AD1PCFG register. If a bit in this register is set to '1' the corresponding pin is configured for digital mode, otherwise a value of '0' (which is the default case) keeps the pin in analog mode.

   Once the PORTB pins are configured in digital mode, the actual direction of the pin that interface to the rows and columns of the keypad need to be properly configured, using TRISB register.

2. *unsigned char **readKey**(void)*

   Anytime a user is interested to read a character from the keypad, the user will make a call to your *readKey()* function. Your subroutine will then perform a complete scan of the keypad to see if there is any key input at that moment and will return a one byte number. If a key was found pressed during the keypad scan period, the *readKey()* function will first have to *wait* for release of the key and then return the ASCII code of the key that was pressed. However, if there was no key pressed during the scan process, the function simply returns a NULL character which is designated by a byte value of zero.

Place your *initKeypad()* and *readKey()* functions in the same source file, that you may name "keypad.c". Also, create a header file "keypad.h" that contains the port definitions, global data declarations, and function prototypes.

3. *void **main**(void)*

   A template for the *main()* function is provided in the "main.c" file on Blackboard. The code in *main()* first initializes the UART and the keypad hardware modules. Then the function displays a greeting message ("Hello, Please enter keys on the keypad:") on the user's PC monitor using the given I/O utility routine *UART1_putstr().* You need to provide the rest of the program for the *main()* function to implement an infinite loop in which you call the keypad scanning function *readKey()* to accept user inputs and displays the character entered by the user on the monitor (using *UART1_putchar()*).

   The I/O functions *UART1_putstr()* and *UART1_putchar()* that you will be using for displaying messages communicate with your PC terminal program over the USB interface (with a virtual COM port) using the UART1 serial module on the PIC32 MCU. Therefore, we need to properly initialize the UART1 module before using any of these I/O functions. For this lab

assignment you are given the code below for the initialization as well as the printing of messages on the serial monitor. The source code for the serial communication library (both uart.h and uart.c files) are provided on Blackboard. Download those files and include them in your project.

```
#include <uart.h>

void initUART1(void)
{
    U1BRG = BRATE;      //initialize the baud rate generator
    U1MODE = U_ENABLE;  //initialize the UART module
    U1STA = U_TXRX;     //enable TX & RX
}

void UART1_putchar(char c) {  //send a character to UART1
    while (U1STAbits.UTXBF == 1)
    {}            //wait until transmitter buffer becomes empty
    U1TXREG = c;  //write character to tx data register
}

void UART1_putstr(char s[]) { //send a null-terminated string to UART1
    while (s[0] != 0) {
        UART1_putcchar(s[0]);
        s++;
    }
}
```

```
int main(void)
{
      initKeypad();
      initUART1();
      UART1_putstr("Hello, please enter keys on the keypad:");

      //add your code here

}
```

When you are ready to test your program make sure that your chipKIT UNO32 board is connected to your PC using a virtual COM port via a USB cable (the MCU actually uses the UART protocol for the communication). Next, open a terminal program (such as PuTTY) on your PC. Choose the "Serial" connection type and choose the correct COM port number (by looking it up from the Windows device manager). Use the same communication parameters as in your program, i.e. 9600 baud rate and the following settings for the data frame format (8-bit data, no parity, 1 stop bit, and NO flow-control).

Note that as in your previous lab assignment you need to take care of the contact bounce problem of the switches in the keypad by using the same software time delay based strategy. You can refer

to the keypad's datasheet (available on Blackboard) for information about its internal configuration and the actual length of time that the contact bounce lasts. You can use the *msDelay()* function that was provided in lab3.

CAUTION: Debugging a program that contains a driver to the UART module or an interrupt service routine is a tricky operation. The UART module may freeze when the CPU enters debug mode. So, if you try to set a break point in your program and/or step through the code line by line, the freezing of the UART module may prevent display of characters on the terminal. Thus, you need to be careful when debugging your program; for example, by using the 'Run To Cursor' command to run your code until a line past your UART output commands, then reset your processor and repeat your debug while watching your variables of interest.

Check-off and report:

Demonstrate your working program to the lab instructor and submit, via Blackboard, a lab report that follows the format and structure guidelines given in "Laboratory and Project Report Guidelines" available on Blackboard. Your report needs to include: Title page, Table of Contents, Objectives, Hardware, Program Source Code, and Conclusions.