CoE 115 Lab 3: Keypad, Polling and Interrupts

February 25, 2019

Topics/Objectives

- Understand structure of keypad
- Implement interrupt on change as an input interface
- Interface a keypad to PIC microcontroller

PIC Ports setup

The PIC's interrupt controller can be used as a handler for the PIC's GPIO pins. For this lab exercise, we will be using the following hardware mapping between the keypad and micro controller pins.

- RA0 to RA3 (inputs) - keypad rows (top to bottom)
- RB0 to RB2 (outputs) - keypad columns (left to right)
- RB4, RB5, RB7, RB8 - LED outputs (LED1 to LED4)

Keypad

Figure 1 shows the structure of keypad. Its interface is a collection of 7 pins, each of which is assigned to a particular row or column. Whenever a button is pressed, it causes that button's corresponding row and column pins to be shorted together. (Ex. pressing the '1' key shorts col1 and row1).

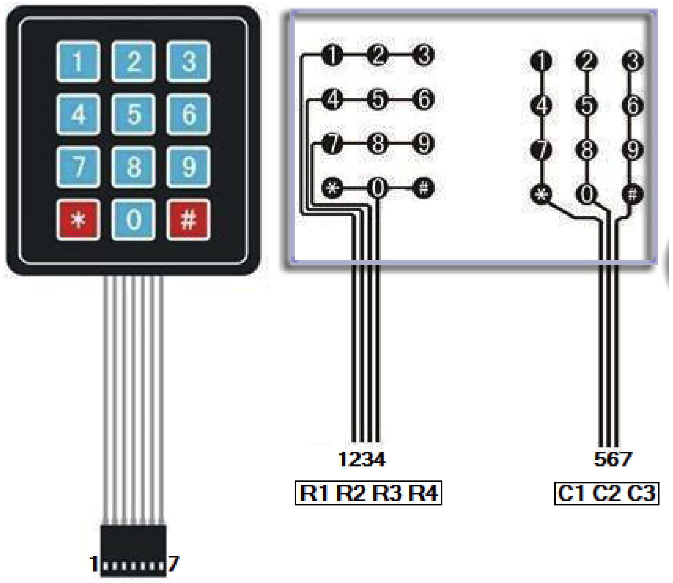

Figure 1: Port mapping of keypad

Polling a single button

Before we interface the whole keypad, let's start with a single button. In this part, you will be using polling method. Follow the steps below:

1. Connect only col1 and row1 to PIC's port RB0 and RA0, respectively.

2. Construct the LED1 output circuit by connecting an LED and a resistor through an external pull-up, such that the LED is turned on when the microcontroller pin is pulled low. The LED1 circuit should be connected to the RB4 pin of the microcontroller.

3. A sample polling program for a button '1' is shown below. The program waits for row1 to be pulled low which means that button '1' is pressed. It then lights up LED1 for each button '1' press. When button '1' is not pressed, row1 stays high because it is connected to the internal pull-up of the microcontroller's input port RA0. This is internal pull-up functionality is enabled by setting the CN2PUE bit. Table 1 shows the internal pull-up configuration bits for the rows connected to RA0 to RA3. When button '1' is pressed, row1 is shorted to col1. Since col1 is logic low, row1 is then pulled low signalling the microcontroller to light up LED1. Run the program to check if your circuit is correct.

```c
#include "xc.h"

_CONFIG1 (FWDTEN_OFF & JTAGEN_OFF)
_CONFIG2 (POSCMOD_NONE & OSCIOFNC_ON & FCKSM_CSDCMD & FNOSC_FRCPLL & PLL96MHZ_OFF &
    PLLDIV_NODIV)
_CONFIG3 (SOSCSEL_IO)

int main(void){

    /* Configure ports
     * RA0 - row1 (input)
     * RB0 - col1 (output)
     * RB4 - LED1 (output)
     */
    AD1PCFG = 0xFFFF;
    TRISA = 0x0001;
    TRISB = 0xFFEE;
    LATB = 0xFFFF;

    /* Configure internal pull-ups
     * RA0 - CN2 (CNPU1)
     */
    CNPU1 = CNPU1 | 0x0004;

    LATBbits.LATB0 = 0; // pull-down col1

    while(1){
        if(!PORTAbits.RA0) // poll row1, check if pulled low
            LATBbits.LATB4 = 0; // LED1 on
        else
            LATBbits.LATB4 = 1; // LED1 off
    }
    return 0;
}
```

Port	Register	Bit
RA0	CNPU1	CN2PUE(bit 2)
RA1	CNPU1	CN3PUE(bit 3)
RA2	CNPU2	CN30PUE(bit 14)
RA3	CNPU2	CN29PUE(bit 13)

Table 1: Internal pullup configuration bits

4. [CHECK] Modify the sample polling program to light up LED1 when the last digit of your student number is pressed. When complete, show your work to your lab instructor for checking.

Using interrupts to read a single button

Alternatively, we can use interrupts to detect a button press. Study the program below. This program implements an interrupt to detect a button press on the same hardware setup and toggles the state of LED1 for every button press. The interrupt mechanism is explained below.

```c
#include "xc.h"

_CONFIG1 (FWDTEN_OFF & JTAGEN_OFF)
_CONFIG2 (POSCMOD_NONE & OSCIOFNC_ON & FCKSM_CSDCMD & FNOSC_FRCPLL & PLL96MHZ_OFF & PLLDIV_NODIV)
_CONFIG3 (SOSCSEL_IO)

#define DEBOUNCEMAX 10
void __attribute__ ((interrupt)) _CNInterrupt(void);
void led1_toggle(void);

int row1_isPressed;

int main(void){

    AD1PCFG = 0xFFFF;
    TRISA = 0x0001;
    TRISB = 0xFFEE;
    LATB = 0xFFFF;

    CNPU1 = CNPU1 | 0x0004;

    /* Enable change notification, interrupts and clear IRQ flag
     * RA0 - CN2 (CNEN1)
    */
    CNEN1 = CNEN1 | 0x0004;
    IEC1bits.CNIE = 1;
    IFS1bits.CNIF = 0;

    LATBbits.LATB0 = 0;
    while(1){
        if(row1_isPressed){
            led1_toggle();
            row1_isPressed = 0; // clear row1 flag
        }
    }
    return 0;
}

void __attribute__ ((interrupt)) _CNInterrupt(void) {
    int debounceCounter = 0;

    if(!PORTAbits.RA0){
        while((!PORTAbits.RA0) && (debounceCounter < DEBOUNCEMAX)){
            debounceCounter++;
        }
        if(debounceCounter == DEBOUNCEMAX)
            row1_isPressed = 1; // set row1 flag
        else
            row1_isPressed = 0;
    }
    IFS1bits.CNIF = 0; // clear IRQ flag
}

void led1_toggle(void){
    LATBbits.LATB4 = !LATBbits.LATB4; // toggle LED1
    return;
}
```

Interrupts in the PIC microcontroller are controlled by a global interrupt enable bit. For this particular exercise, we will be using the Input Change Notification interrupt. The global interrupt enable bit for this interrupt can be found in the CNIE bit of control register IEC1. Input Change Notifcation interrupts are enabled by setting this bit (high). Aside from the global interrupt enable, specific pins in which we want to monitor input change notification must be specified. This is done through the CNENx register. To enable interrupt generation when a change in value asserted at GPIO pin occurs, the appropriate bit mask in CNENx should be set high. Table 2 shows the mapping of keypad row input pins (RA0-RA3) to bit fields of CNENx. In the sample program, only CN2IE is set since we are only using a key press in row1 to generate an interrupt. Whenever an interrupt is generated, a corresponding interrupt flag is set. For the Input Change Notification interrupt, this flag is the CNIF bit of the IFS1 register. Until this flag is cleared, all interrupts generated through the CNIF bit are ignored.

Port	Register	Bit
RA0	CNEN1	CN2IE(bit 2)
RA1	CNEN1	CN3IE(bit 3)
RA2	CNEN2	CN30IE(bit 14)
RA3	CNEN2	CN29IE(bit 13)

Table 2: CNENx mapping to ports

In the sample program, whenever button '1' is pressed, a change in value in row1 will occur, causing an interrupt to be generated and CNIF to be set. The microcontroller then jumps to the Interrupt Service Routine (ISR), marked by the function CNInterrupt(). The ISR in the sample program performs the following tasks:

1. Determine which pin caused the interrupt - multiple pins (identified by the CNENx bit masks) generate only one interrupt (CNIF bit of IFS1). It is up to the programmer to determine the source of the interrupt by individually checking port pins (in this case, RA0).

2. Perform a software debounce - This section prevents noise (bounce) from generating a false detection of a key press.

3. Set the key press flag - ISR's often have limited space in program memory. Thus, it is common practice for ISR's to simply modify a flag and let the main part of the program decode this flag.

4. Clear the interrupt flag - This allows future interrupt requests generated after executing the ISR to be recognized again.

In the sample program, the ISR communicates with the main function through a global variable, the row1_isPressed flag. Whenever button '1' is pressed, the ISR sets this flag to 1. The main function then recognizes this and toggles the state of LED1 by calling led1_toggle() function. It then resets the flag so that a single press only toggles the LED1 once.

[CHECK] Modify the sample interrupt program to toggle LED1 when the last digit of your student number is pressed. When complete, show your work to your lab instructor for checking.

Detecting row1 presses in different columns

Connect col2 and col3 to PIC's ports RB1 and RB2, respectively. Do the following:

1. [CHECK] Modify the code to pull-down all columns instead of just col1. What happens when a key in row1 is pressed?

2. [CHECK] Modify the code to pull-down only col2 and col3 and pull-up col1. Observe which key presses in row1 toggle LED1. Show your work to your instructor.

Detecting button presses in a single row

Construct three more LED circuits and connect them to the microcontroller as LED2, LED3, and LED4 (RB5, RB7, R8). Modify the ISR such that when a key press is made in colX, LEDX is toggled. Thus, pressing '1' toggles LED1, '2' toggles LED2, etc. Take note that you may not modify the hardware connections specified i.e. row1 must remain an input pin, while col1, col2, and col3 remain as output pins.

[CHECK] Show your work to your instructor.

Detecting button presses in single column

Remove the connections of col2 and col3 from the microcontroller, leaving only col1 connected. Then, connect all remaining rows (RA1, RA2 and RA3). You will attempt to detect key presses made in col1. Modify the program so that whenever a button press in rowX is made, LEDX is toggled. Thus, pressing '1' toggles LED1, '4' toggles LED2, etc. Here are a few pointers to achieve this:

1. Make sure to enable interrupts for each row.
2. Modify the ISR such that it correctly identifies the source of the interrupt (row1, row2, row3, or row4). You may set flags for each row.

[CHECK] Show your work to your instructor.

Full keypad

Connect all rows and columns to the microcontroller. Modify the program such that when a key press is made, the state of all LEDs are switched (and held until the next button press) to the binary equivalent of the number pressed. For example, pressing button '1' sets (LED4,LED3,LED2,LED1) to (0,0,0,1), pressing button '5' sets it to (0,1,0,1), etc. For '*', set the output to (1,0,1,0), and for '#', set the output to (1,0,1,1).

[CHECK] Show your work to your instructor.