CoE 115 Laboratory 3

Feb 21-24, 2017

1 Objectives

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

2 Port mapping

For this laboratory exercise, we will be using the following hardware mapping to the microcontroller pins:

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

3 Keypad

Figure 1 shows the structure of the 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, the corresponding row and column of the button are shorted together. For example, pressing the '1' key shorts Col1 and Row1. Keypad pinout mapping (as shown in figure 1) are shown in table 1.

Port name	Pin	Port name	Pin
Row1	b	Col1	c
Row2	g	Col2	a
Row3	f	Col3	e
Row4	d		

Table 1: Keypad port mapping

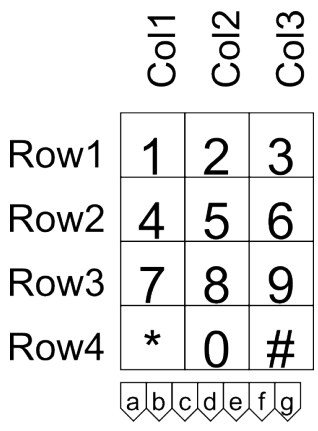

Figure 1: Keypad Schematic

4 Seatwork (10%): Polling a single button

Before we interface the whole keypad, let's start with a single button. Connect only Col1 and Row1 to the microcontroller (RB0 and RA0, respectively). Construct the LED1 output circuit by connecting an LED and a resistor through an external pullup, 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.

The program below permanently pulls Col1 low. Row1 is normally high because utilizes the internal pullup function of the microcontroller. When the '1' button is pressed, Col1 and Row1 are shorted together, pulling Row1 low as well. Row1 is continuously polled for its value, turning on LED1 when the button is pressed and off otherwise. Table 2 shows the configuration bits of the internal pullups for the ports which are mapped to the row pins of the keypad. The sample program enables the internal pullup of RA0 by setting the CN2PUE bit.

```
#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 - row 1 (input)
     * RB0 - col 1 (output)
     * RB4 - led 1 (output)
     */
    AD1PCFG = 0xffff;
    TRISA = 0x0001;
    TRISB = 0xffee;
    LATB = 0xffff;

    /* Enable internal pullups
     * RA0 - CN2 (CNPU1)
     */
    CNPU1 = CNPU1 | 0x0004;

    /* Pull down col1 */
    LATBbits.LATB0 = 0;
    while(1){
        if (!PORTAbits.RA0) //button pressed
            LATBbits.LATB4 = 0b0; //led 1 on
        else
            LATBbits.LATB4 = 0b1;
    }

    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 2: Internal pullup configuration bits

5 Seatwork (10%): Using interrupts to read a single button

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

```
#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 DEB_MAX 10

void __attribute__((interrupt)) _CNInterrupt(void);
```

```
void led1_toggle(void);

int row1_press;

int main(void) {

    /* Configure ports
     * RA0 - row 1 (input)
     * RB0 - col 1 (output)
     * RB4 - led 1 (output)
     */
    AD1PCFG = 0xffff;
    TRISA = 0x0001;
    TRISB = 0xffee;
    LATB = 0xffff;

    /* Enable internal pullups
     * RA0 - CN2 (CNPU1)
     */
    CNPU1 = CNPU1 | 0x0004;

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

    /* Pull down col1 */
    LATBbits.LATB0 = 0;
    while(1){
        if (row1_press){
            led1_toggle();
            row1_press = 0; //clear flag
        }
    }

    return 0;
}

void __attribute__((interrupt)) _CNInterrupt(void){
    int deb_ctr = 0; //debounce counter

    if (!PORTAbits.RA0){
        /* Software debounce */
        while ((!PORTAbits.RA0) && (deb_ctr < DEB_MAX)){
            deb_ctr++;
        }
        if (deb_ctr == DEB_MAX)
            row1_press = 1; //set flag
        else
            row1_press = 0;
    }

    /* Clear IRQ flag */
    IFS1bits.CNIF = 0;
}

void led1_toggle(void){
    LATBbits.LATB4 = !LATBbits.LATB4;
}
```

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 Notification 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 interrupts to be generated when a change in the value of the GPIO pin occurs, the appropriate bit mask in CNENx should be set high. Table 3 shows the mapping of keypad row 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, the 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 3: 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). Thus, 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 the row1_press flag. Whenever button '1' is pressed, the ISR sets the row1_press to 1. The main function then recognizes this and toggles the state of LED1. It then resets the row1_press flag so that every key press only toggles LED1 once.

6 Seatwork (10%): Detecting button presses in a row

Aside from Col1, now connect Col2 and Col3 to the microcontroller and perform the following:

1. Permanently pull down all columns the program (instead of just Col1). What happens when a key in row 1 of the keypad is pressed?
2. Permanently pull down only Col2 (pull up both Col1 and Col3). Observe which key presses in row 1 toggle LED1.
3. Permanently pull down only Col2 and Col3 (pull up Col1). Observe which key presses in row 1 toggle LED1

7 Seatwork (70%): Detecting button presses in a column

Remove the connections of Col2 and Col3 to the microcontroller, leaving only Col1 connected. Then, connect all remaining rows. You will attempt to detect key presses made in Col1. Construct three more LED circuits and connect them to the microcontroller as LED2, LED3, and LED4. 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).

8 Quiz (70%): Detecting button presses in a row

Remove the connections of Row2, Row3, and Row4 (leaving only Row1 connected). Then, connect both Col2 and Col3. Modify the program 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. Row1 must remain an input pin, while Col1, Col2, and Col3 remain as output pins.

9 Quiz (30%): Full keypad

Connect all rows and columns to the microcontroller. Modify the program such that when a key press is made, the state of the LED's is switched (and held until the next button press) to the binary equivalent of the number pressed. For example, pressing '1' sets (LED4,LED3,LED2,LED1) to (0,0,0,1), pressing '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).