4 Keyboard Interfacing

4.1 Introduction

A very common method of input to microprocessors from humans is a keyboard. At a low cost, a large amount of information may be input. A key works like a single pole, single throw switch. With such switches, it is better to detect the presence of a low voltage level, rather than a high. This is due to noise. Smaller keyboards up to 16 keys or of a size that fits into a palm, are known as keypads. A keyboard consists of pressure- or touch-activated switches arranged in a matrix. To detect which key has been pressed, we use a combination of hardware and/or software means.

Two basic types of keyboards are available: encoded and non-encoded. Encoded keyboards include the hardware necessary to detect which key was pressed and to hold that data until a new key stroke. Non-encoded keyboards have no hardware and must be analyzed by a software routine or by special hardware. This is the type we will be looking at here.

4.1.1 Reading from a keyboard

The complexity of the processor task when reading from a keyboard depends on the number of keys to be detected, the input ports available on the processor and the amount of processor time dedicated to this task.

4.1.1.1 Simple keyboard reading

For a small number of keys say for 1 to 8 keys, we may assign one key to each buffer or port line. To read the keyboard, it is connected directly to a buffer. Or it may be connected directly to a port. We simply read the keys, checking for a low level.

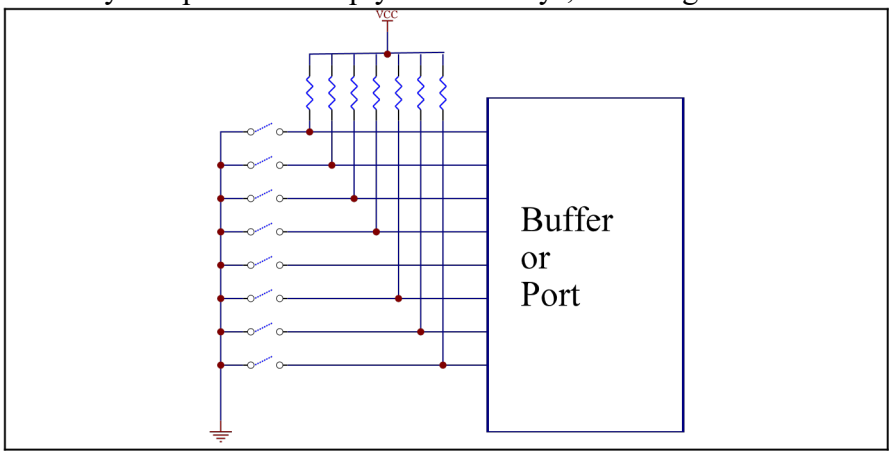

Fig 4.1 Simple key input

4.1.1.2 Matrix organisation

For larger number of keys, the keyboard may be arranged in a row and column fashion, with an n by m key organization. This is also known as a matrix organisation.

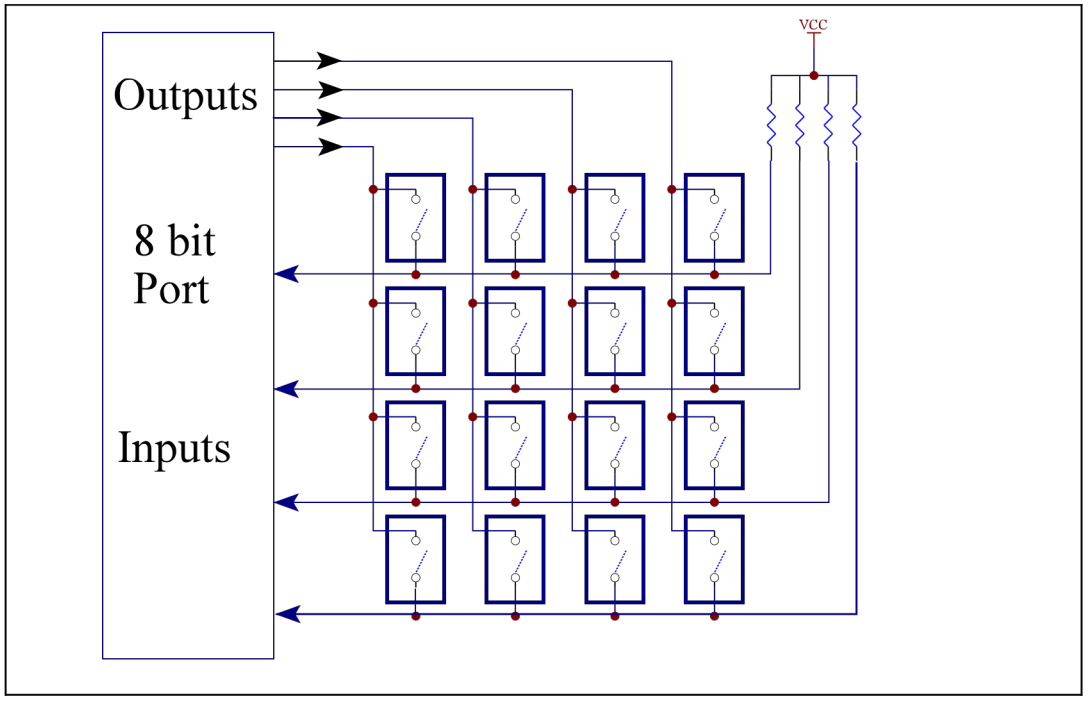

Fig 4.2 Walking Zeroes Keyboard Decode

A 16 key device can be decoded with 4 input lines and 4 output lines. This is an optimal number for an 8 bit controller as this will use one port. However, this port must be capable of having four of its bits as input, and the other four as output.
We will consider an example where the upper four bits of the port are used as output pins and the lower four as inputs. With respect to the diagram above, we denote the output lines as columns and the input lines as rows, for convenience. When the processor is not checking the keyboard, it will output all 1's. The inputs will also read all 1's, because it is tied to the pull up resistors.

Scanning the keypad

We can output to the column lines with a "walking zero" pattern and sense the row lines to see if a low voltage is detected. This key identification technique is known as "row scanning."

With respect to the diagram, we output a “0" to each of port pins 4,5,6 and 7 one at a time. We read in the input port and check for a “0" in port pins 0,1,2 and 3.

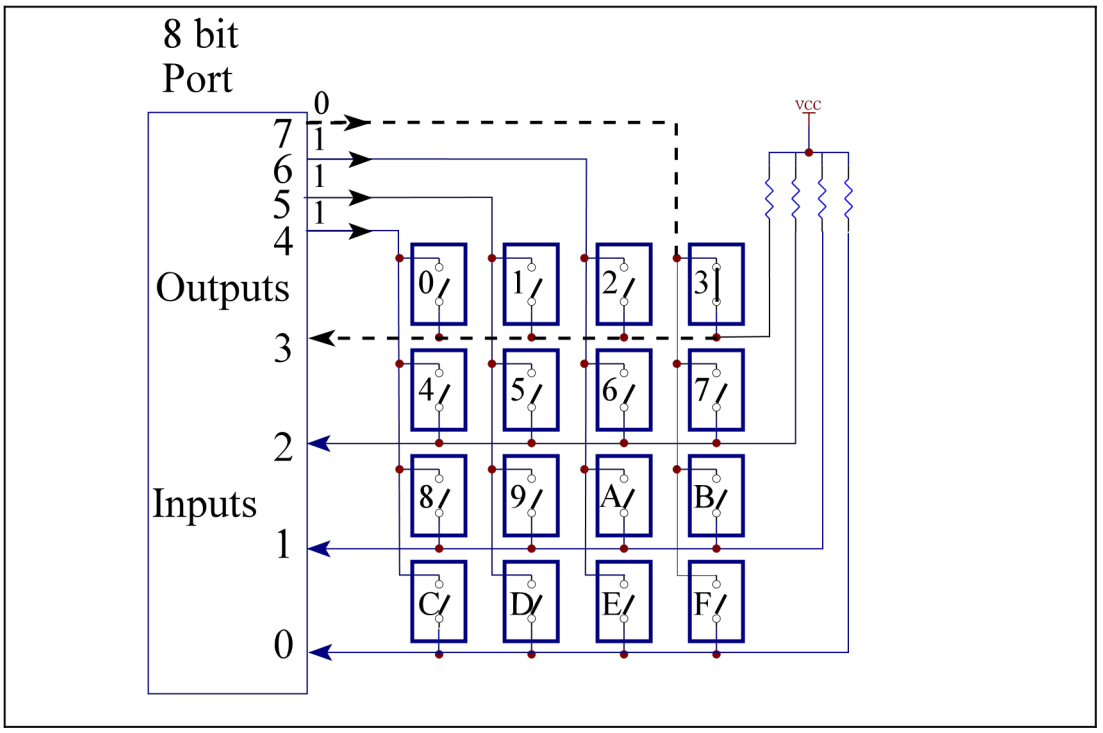

Fig 4.3 Walking “0" actual example

Let’s say the ‘3' key is pressed. If the processor did not output a “0" to pin 7, the keypress will not be registered - that is, port pins 0 to 3 will still read “1". Only when pin 7 outputs a “0" will pin 3 register a “0" as well.

Larger keyboards require more select and sense lines. First we can have latches to output the “walking zero” data to the column lines. Buffers are used to read the row lines. When selecting the key organization, we should let the smaller number be assigned to the row lines. This is because we detect a key press through these lines, the status of which is read into a register. Then software routines, most often rotates are used to detect which column has a “0". This is time consuming.

(Note: The assignment of which lines are “rows” and “columns” is entirely arbitrary. In this course we refer to the vertically oriented lines as columns purely for ease of reference)

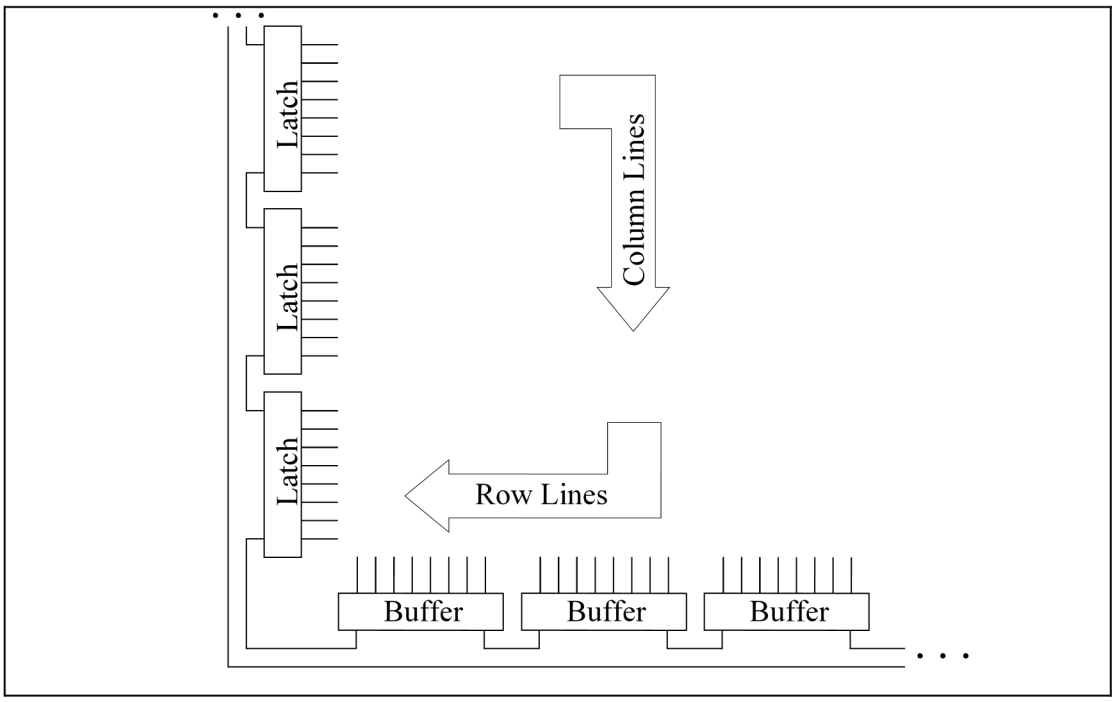

Fig 4.4 Large keyboard detection using latches

Since each column line is activated one at a time, it is possible to substitute the latches with decoders. In this case only N lines from the processor will support 2^N columns. As shown below, three 3-to-8 decoders can handle 24 column lines using 5 address lines. But up to 32 rows can be decoded.

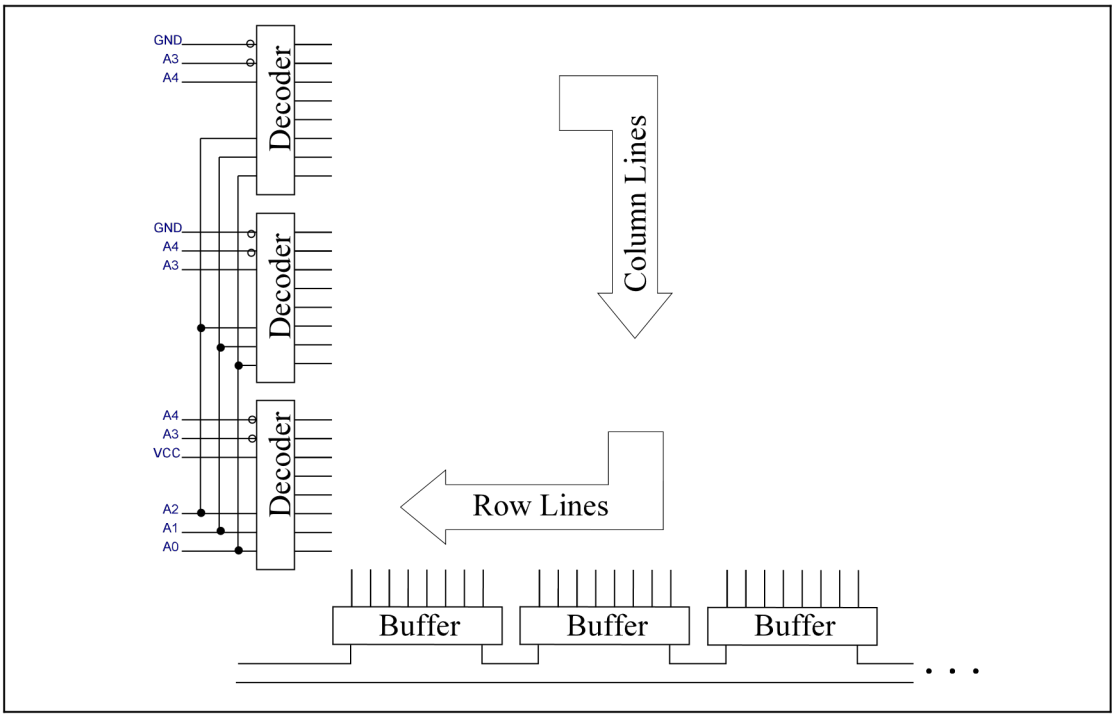

Fig 4.5 Decoder Scanning

4.1.2 Interrupt Driven Keyboard

From the previous discussion, we see that detecting a key press is a processor intensive process. We must continually select a row line and then read the column lines. If this is a matter of concern, dedicated keyboard controller hardware is available. These are normally interrupt driven and have various features built into them. But this will increase product cost.

A simple solution would be to see if we can make all the column lines go to “0" volts. If no keys are pressed, all the row lines will present a “1" level. We would only need to AND all the row lines together. A key press would then signal an interrupt, and normal scanning can proceed to find out which key is pressed.

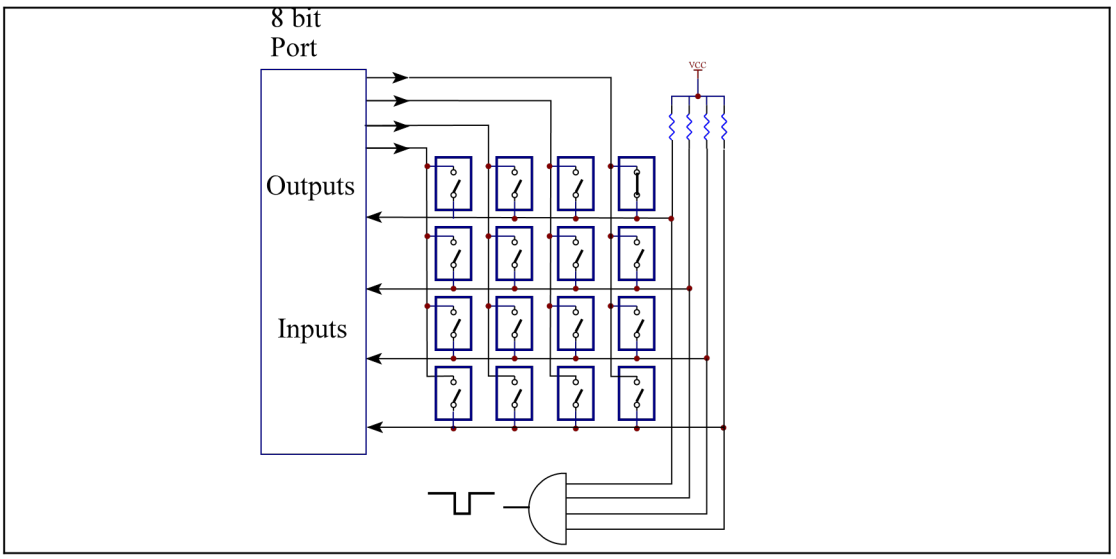

Fig 4.6 Interrupt driven keypad

4.2 Common Problems

The most common problem is that of contact bounce. In addition, there are other problems caused by certain usage patterns. In some cases, in order to increase the number of usable keys, two or more keys have to be pressed together, for example Control and Alternate keys together with another key. Or sometimes when the user presses the keys very quickly, and the processor is busy, it seems that three or more keys are depressed. These problems have to be addressed, especially when the order of digits is important. If a student’s score is 290 marks and if the scanning algorithm detects 3 simultaneous keystrokes it may read it as 092!

4.2.1 Bounce

Keybounce refers to the fact that when the contacts of a mechanical switch close, they bounce for a short time before staying together. This is also true when the switch opens. What happens is that the resistance of the switch changes, thus presenting differing voltages to circuitry for a short while. Fig. 4-7 is a time-versus-resistance plot of a typical switch contact. The bounce lasts for 10 to 20 millseconds and is denoted by the oscillatory waveform.

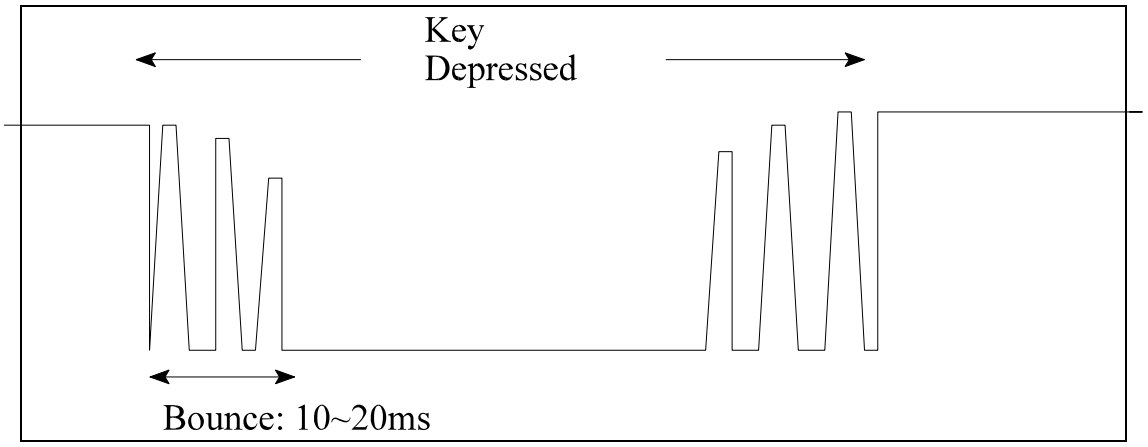

Fig 4.7 Keybounce

The solution is to wait for the status of a key to remain stable for perhaps 20 milliseconds. This may be done by hardware filtering or by a software-delay routine. The hardware solution uses an R-C filter and requires the same circuitry for each key. This is useful for if we have only a few switches in a system. In the case of a larger number of keys, software is often used.

4.2.2 Multiple key presses

When more than one key is held down at the same time, it is essential to detect this fact to prevent wrong codes from being generated. The two main techniques used to resolve this problem are the n-key rollover, and the n-key lock-out.

4.2.2.1 N-key rollover

This is again has two techniques: the software simply ignores the reading from the keyboard until only one key closure is detected. The last key to remain pressed is the correct one. This method is normally used when software routines are used to provide keyboard scanning and decoding. If the order of keys pressed is not crucial, it is possible, with more sophisticated software to store all closures in an internal buffer.

The second method is used by hardware devices. Second and later key closures are prevented from generating a strobe until the earlier ones are released. This is realized by an internal delay mechanism which is latched as long as other keys are pressed.

4.2.2.2 N-key lockout

N-key lock-out takes into account only one key pressed. Any additional keys which might have been pressed and released do not generate any codes. By convention, it may be the first key pressed which will generate the code, or else the last key pushed. The system is simplest to implement and the most often used. However, it may be objectionable to the user, as it slows down the typing: each key must be fully released before the next one is pressed down.

4.2.2.3 Phantom Key

A significant cost of n-key rollover protection is that most systems need a diode in series with every key in order to eliminate the problem created when three adjacent keys at a right angle are present. This increases the cost very significantly and is seldom used on low cost systems.

The effect is that another key seems to be pressed in addition to the three. This is the "phantom key". Consider the situation when keys 3,6 and 7 are pressed simultaneously and column 6 is scanned.

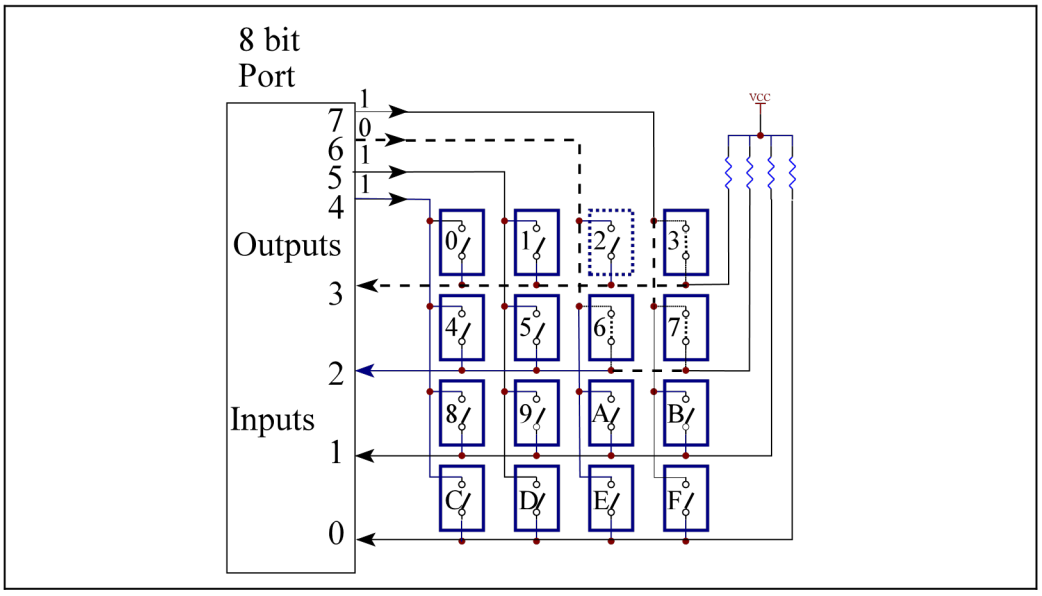

Fig 4.8 “Phantom” Key

Even though key “2" is not pressed, row 3 has a “0" value appearing there due to the flow of current in the other 3 switches. Key “2" is the phantom key. The problem can be overcome in software.

An example of a program used for keyboard scanning follows:

```
unsigned char ScanKey ();
unsigned char ProcKey ();

#define Col7Lo    0x7F                    /*  column 7 scan */
#define Col6Lo    0xBF                    /*  column 6 scan */
#define Col5Lo    0xDF                    /*  column 5 scan */
#define Col4Lo    0xEF                    /*  column 4 scan */
#define TRUE      1                       /*  neater to use! */
unsigned char ScanCode;      /* hold scan code returned     */ void
main (void)
{                          /* main entry for program      */
unsigned char i;

    while (TRUE)
    {
       i = ScanKey();
       if (i != 0xFF)  /* if key is pressed **/
                LEDPort = Bin2LED[i];   /* output to LED */


     }
}                                        /* loop forever */

/* Check for key press: if none, return 0xFF */ unsigned
char ScanKey()
{
       KbdPort = Col7Lo;               /* bit 7 low  */
       ScanCode = KbdPort;             /* Read */
       ScanCode |= 0xF0;               /* high nybble to 1 */ ScanCode
       &= Col7Lo;                     /* AND back scan value */

       if (ScanCode != Col7Lo)         /* in<>out: get key & disp */
       return ProcKey();

       KbdPort = Col6Lo;               /* bit 6 low  */
       ScanCode = KbdPort;             /* Read */
       ScanCode |= 0xF0;               /* high nybble to 1 */
       ScanCode &= Col6Lo;             /* AND back scan value */

       if (ScanCode != Col6Lo)         /* in <> out, get key */
       return ProcKey();

       KbdPort = Col5Lo;               /* bit 5 low  */
       ScanCode = KbdPort;             /* Read */
       ScanCode |= 0xF0;               /* high nybble to 1 */
       ScanCode &= Col5Lo;             /* AND back scan value */

       if (ScanCode != Col5Lo)         /* in <> out, get key */
       return ProcKey();

       KbdPort = Col4Lo;               /* bit 4 low  */
       ScanCode = KbdPort;             /* Read */
       ScanCode |= 0xF0;               /* high nybble to 1 */
       ScanCode &= Col4Lo;             /* AND back scan value */
```

```
        if (ScanCode != Col4Lo)          /* in <> out, get key  */
        return ProcKey();

        return  0xFF;                    /* no key press rtn with FF */
}   /* main */

/* Procedure here */ unsigned
char ProcKey()
{
    unsigned char i;                   /* index of scan code returned */
    for (i = 0 ; i <= 12; i++)
    if (ScanCode == ScanTable[i])              /* search in table */
       return i;                               /* exit loop if found */
    if (i == 12)
       return  0xFF;                 /* if not found, return 0xFF */
}
```