CpE 213 Digital Systems Design

Lecture 17

Thursday 10/23/2003

UMR UNIVERSITY OF MISSOURI-ROLLA
The Name. The Degree. The Difference.

Announcements

- Exam 2: Thursday Nov. 13th
- Review session: Tuesday Nov. 11th, from 7 to 9 pm. Location TBA.
- See me or send me email if you have another exam on that day.
- HW6 will be posted later today. Watch your email.

Semester Project

- You will be designing a simple, self-contained 8051-based device that plays a tune on request. Download complete description from Blackboard.
- Each group submits one report.
- Start early.
- Demos and code due during week of 12/2.
- Report due 12/6.
- Peer review due 12/19 (at the final).

Structures

- Aggregate data

```
#include <stdio.h>

struct name
{
  char*     name;
  int       age;
}; /* <== DO NOT FORGET the semicolon */

int main(int argc, char* argv[])
{
  struct name bovik;
  bovik.name = "Harry Bovik";
  bovik.age = 25;

  printf("%s is %d years old\n", bovik.name, bovik.age);
  return 0;
}
```

Structures in Keil

- In Keil, a structure is implemented as a contiguous block of memory.
- Member names serve as indices into the block for the compiler.
- Order of members in the compiler is the same as order in which they are declared.

```
struct time_str {
      unsigned char hour, min, sec;
      unsigned int days;
} time_of_day;
```

Offset	Member	Bytes
0	hour	1
1	min	1
2	sec	1
3	days	2

Explanation for Group Exercise

- Use 2 methods to set “bit y” equal to value of bit 5 of byte memory location 0x25.

```
1.  sbit x = 0x2C; //0x2C is the bit address of bvit
                   //5 of byte 0x25.
    y = x;


2.  char bdata x _at_ 0x25;
    sbit b = x^5;
    y = b;
```

Casting

```
void main(void){
   char x =1, y = 2;
   x = blah(&y);
 }

 int blah (int *v){
   *v = *v + 1;
   return *v*42;
 }
```

In the code above, we are passing a pointer to a character when the function expects a pointer to an integer.

Some compilers automatically change the function call to call-by-value (copies of arguments are passed and modified) instead of call-by-reference (pointer to arguments are passed, actual arguments are modified.)

Applications in C and ASM

Example 1: basic.c

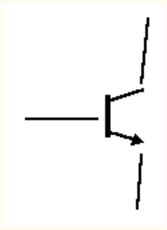
- This is a simple program for making a binary counter on Port 0. From “C for the 8051.

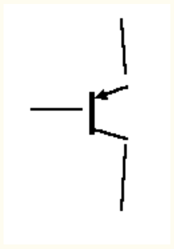
```
#include <reg51.h>
main(){
    unsigned int i;
    P0 = 0;
    while (1==1) {
          for (i = 0; i < 60000; i++) {;}
          P0 = P0 + 1;
    }
}
```

Review Material for Examples to Follow

The Transistor

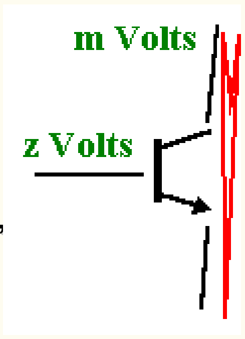
- Three parts: Collector, Base, Emitter
- Two types: NPN and PNP.
- Can be modeled as a switch controlled by voltage applied to base.

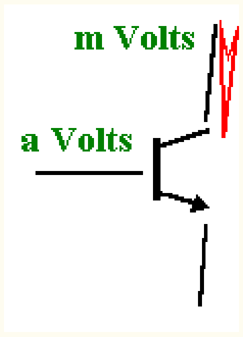


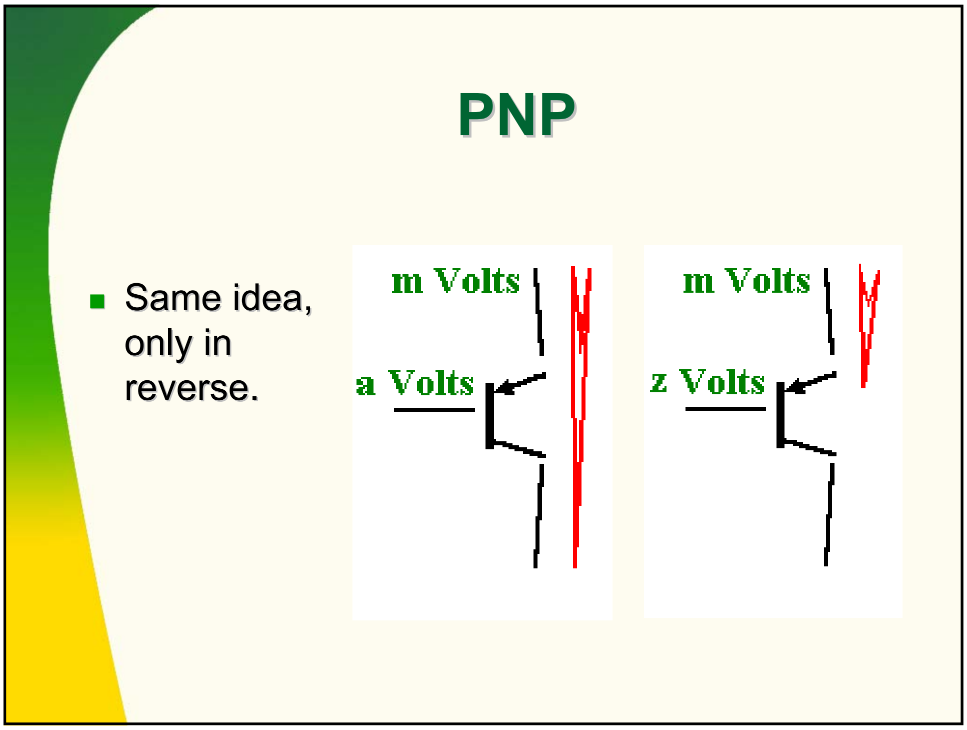


NPN

- When the base has a certain voltage, the current will flow.
- When it does not, the current does not flow.

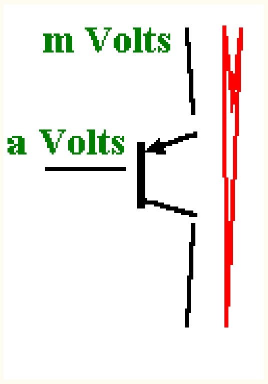


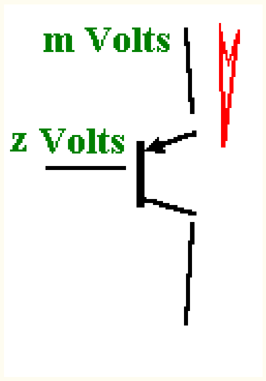


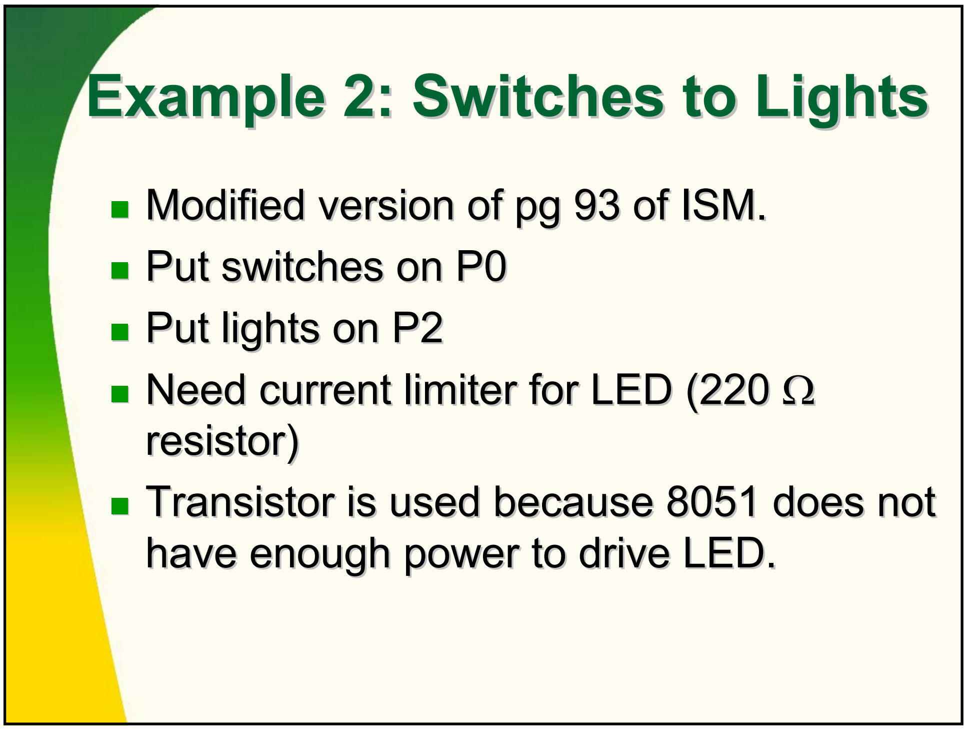


PNP

- Same idea, only in reverse.



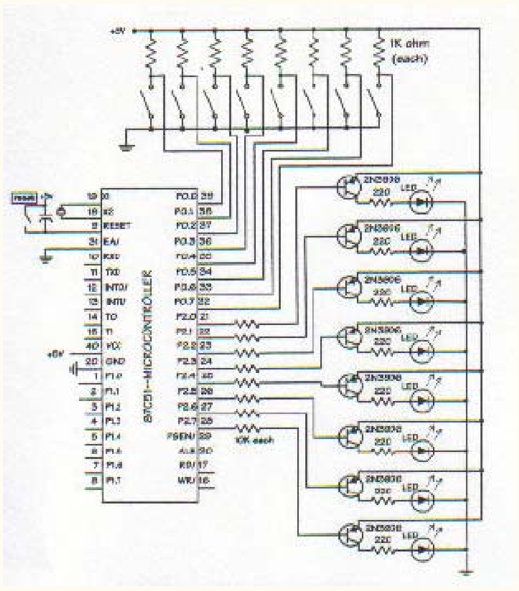




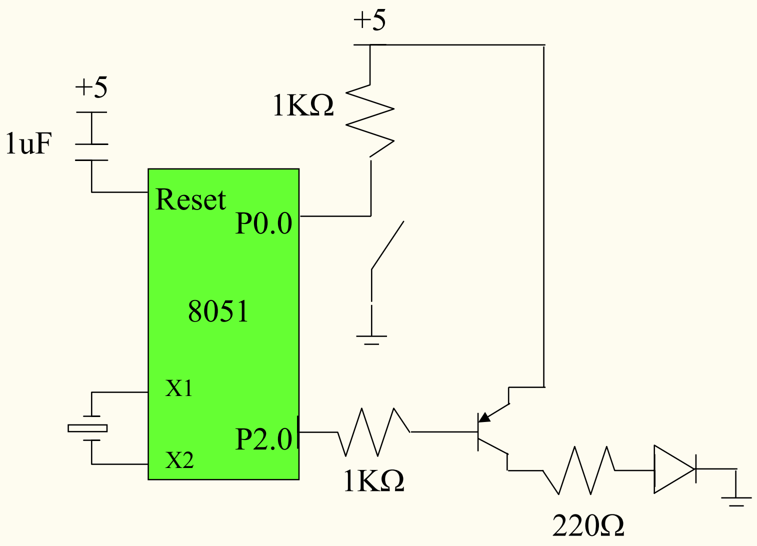
Example 2: Switches to Lights

- Modified version of pg 93 of ISM.
- Put switches on P0
- Put lights on P2
- Need current limiter for LED (220 Ω resistor)
- Transistor is used because 8051 does not have enough power to drive LED.

Switch to lights circuit



Single-pin detail



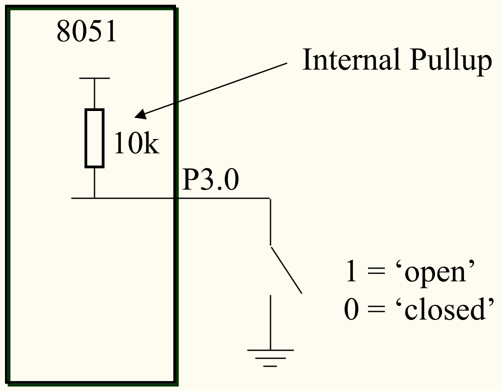
Switch to light code (C)

```
#include <reg51.h>

void msec(unsigned int);
void main (void) {
  unsigned char dat[10];
  unsigned char i;
  while(1) {
    for (i=0; i<=9; i++){
      dat[i]= P2= P0;
      msec(100);
  }
}
```

Switches to lights (A51)

```
EXTRN CODE (MSEC)
MYCODE SEGMENT CODE
MYDATA SEGMENT DATA
RSEG MYDATA
  ARRAY:  DS 10
RSEG MYCODE
  START:  MOV R0, #ARRAY
  AGAIN:  MOV ACC,P0
          MOV P2,ACC
          MOV @R0, ACC
          MOV R2,#0
         MOV R1,#100
         LCALL MSEC
         INC R0
         CJNE R0,#ARRAY+10,AGAIN
         SJMP START
END
```

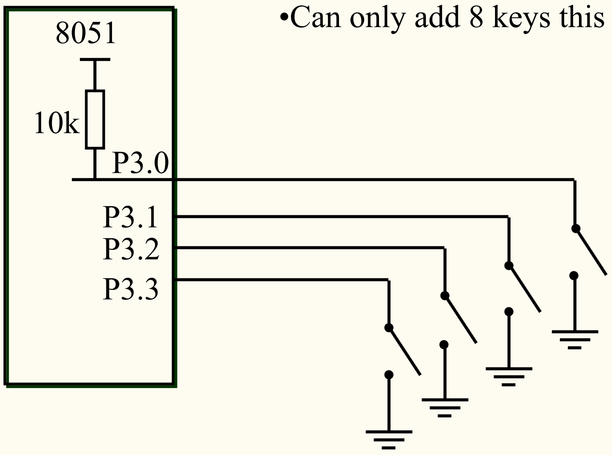
Keypad Application (1-key)



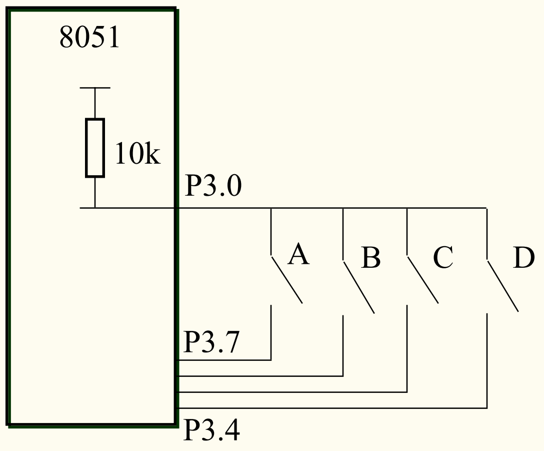
Issues:
•How to add more keys?
•How to ***debounce*** keys?

Adding more keys

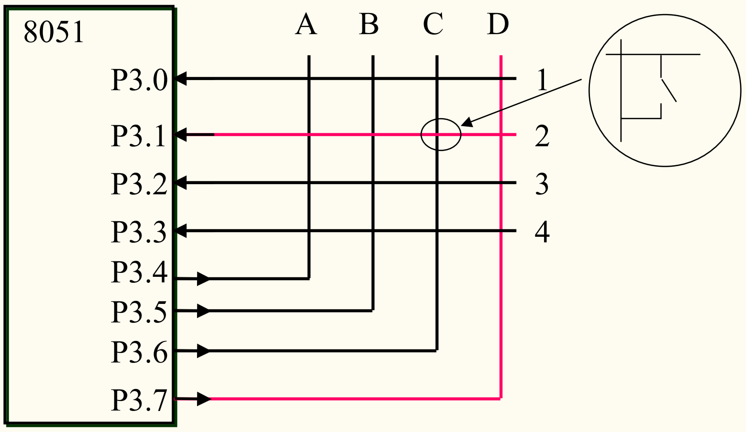
•Can only add 8 keys this way!



Use a port bit for the ground



16 key scanner



Keyscan algorithm

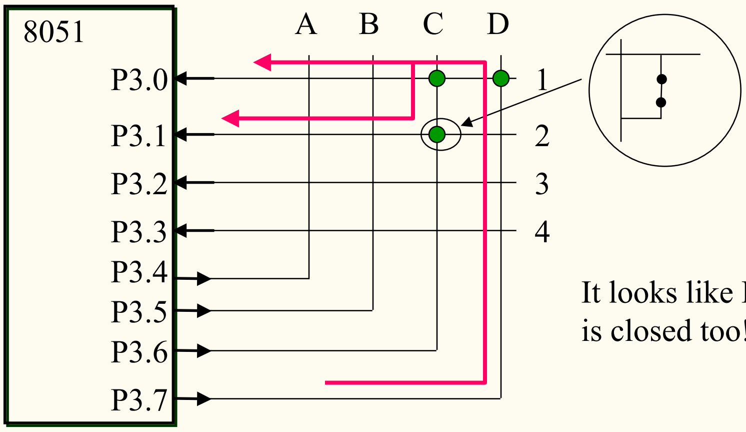
- Output 0 on one column, 1’s elsewhere
- 0’s indicate closed switches in that column
- Repeat 4 times
- 4 switches per column times 4 columns gives 16 bits (unsigned int)
- old*new’ = push (old .and. not new)
- old’*new = release (not old .and. New)
- (new .xor. old)*new = old’*new = release
- (new .xor. old)*old = old*new’ = push

Keyscan C code

```
unsigned int old,new,push,rel,temp;
unsigned char pat;

  for (pat=0x10; pat!=0; pat<<=1){
    P3= ~pat; //EF, DF, BF, 7F
    new= (new<<4) | (P3 & 0xF);}
  if ((temp=new^old)>0) { //changes!
    push=temp & old; rel= temp & new;}
  old = new;
```

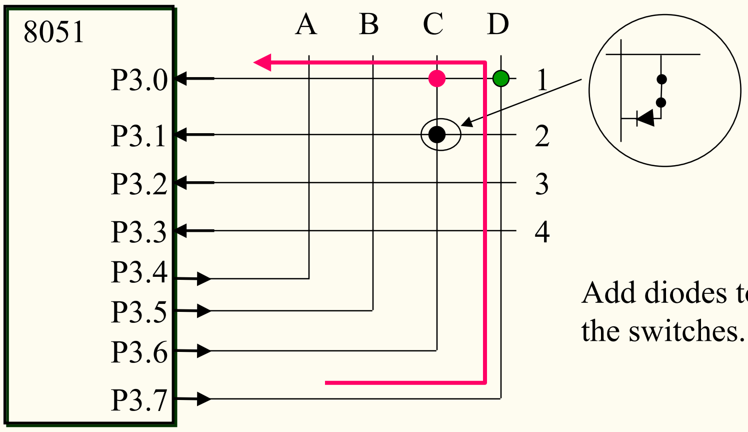
- What if new=1111_1111_1011_1111 and old=0xFFFF?

A problem! (sneak path)



It looks like D2 is closed too!

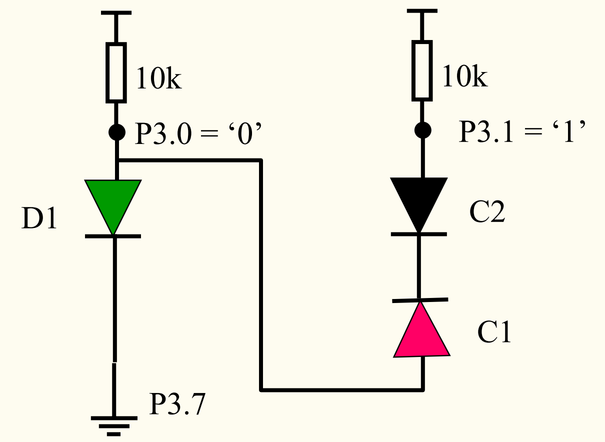
The solution



Add diodes to the switches.

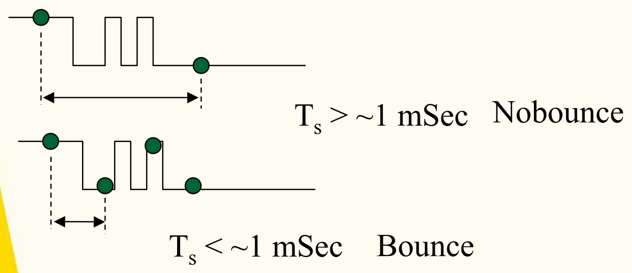
D1 is fwd biased (on), C1 is reverse biased (off).

A closer view

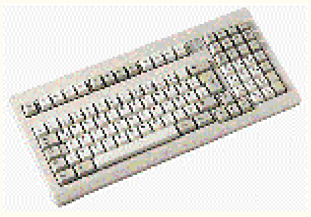


Switch bounce

- Bounce interval is typically a few milliseconds
- Sample slower than bounce interval



Some keyboards



http://www.cherrycorp.com

For next lecture

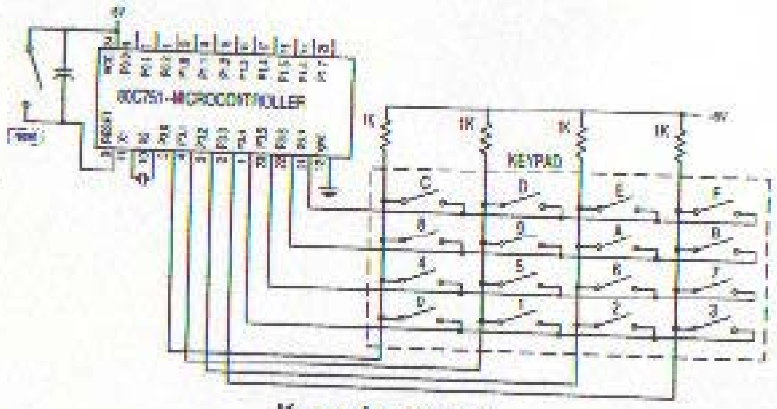
- Download the project.
- Download HW 6 after being notified that it has been posted.
- Review lecture notes and handout.

Example 1 BASIC.C (from C for the 8051)

```
/*************************************************************************
* basic.c - The basics of writing C code for the 8051 using the Keil
*  development environment.  In this case, a simple program will be
*  constructed to make a binary counter on Port 0.
*/

/*
 * As always with C, the included header files should come first.  Most
 * every project for the 8051 will want to include the file reg51.h.  This
 * header file contains the mapping of registers to names, such as setting
 * P0 (port 0) to address 0x80.  This allows the coder to use the keyword
 * "P0" in their code whenever they wish to access Port 0.  For the complete
 * list of registers named, view the file.
 */

#include <reg51.h>

/*
 * Other header files may be included after reg51.h, including any headers
 * created by the user.
 */

/*
 * The C program starts with function main().  In the case of a program
 * written using multiple .c files, main can only occur in one of them.
 * Unlike in programming user applications for a standard computer, the
 * main() function in a Keil program for the 8051 takes no inputs and
 * returns no output, thus the declaration has implied void types.
 */

/*************************************************************************
* main - Program entry point
*
* INPUT: N/A,  RETURNS: N/A
*/

main()
{
    unsigned int i;    /* will be used for a delay loop */

    /* First, Port 0 will be initialized to zero */

    P0 = 0;

    /*
     * Now the counter loop begins.  Because this program is intended
     * to run in an embedded system with no user interaction, and will
     * run forever, the rest of the program is placed in a non-exiting
     * while() loop.
     */
    while (1==1)
    {
        /*
                         * This is a very unpredictable method of implementing a delay
                         * loop, but remains the simplest.  More reliable techniques
                         * can be done using the using the built-in timers.  In this
                         * example, though, the for() loop below will run through 60000
                         * iterations before continuing on to the next instruction.
                         * The amount of time required for this loop varies with the
                         * clock frequency and compiler used.
                         */

        for (i = 0; i < 60000; i++) {;}

            /* Increment Port 0 */

            P0 = P0 + 1;
    }

}
```

IDENTIFYING BIT CHANGES

Bitwise logic operators are useful to identify *changes*.[42] Assume you have a keypad for input. You can scan the matrix one column at a time. Each time you drive a column low, read in the rows. Pushed keys come in *low*, while the others keys in the driven column come in high.



Keypad schematic

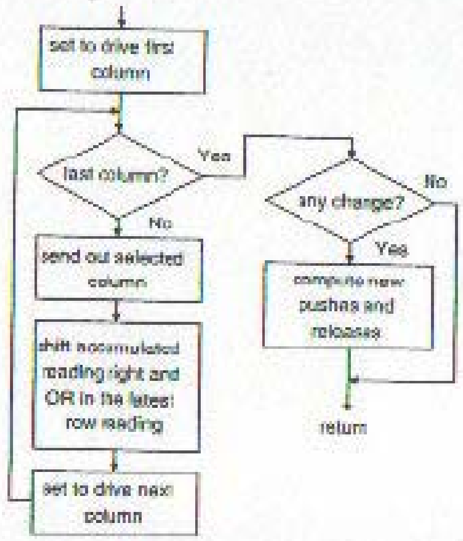
If you repeat this column scanning every 40 msec, you will give prompt recognition of user inputs and still avoid switch bounce problems. The rest of the program needs only know of *any changes* to the inputs—new buttons pushed or old buttons released. With bitwise logical operators, you can easily sort out changes, as shown next.

```
               Logic to Detect Key Changes
previous reading (old)            0 0 1 0 1 1 0 1
most recent reading (new)         0 1 0 0 1 1 0 0
old exclusive-OR new              0 1 1 0 0 0 0 1
new_pushes (old^new&old)          0 0 1 0 0 0 0 1
new_releases (old^new&new)        0 1 0 0 0 0 0 0
```

Here the bitwise logical operations easily and efficiently mark the changed keys.

[42]Suppose you decide to scan a matrix of keys directly rather than use an encoder chip. An encoder chip such as the 74922 is an alternative to scanning, saving three port bits but requiring one extra chip.

NEGATIVE LOGIC: It is easy to be confused by negative logic inputs. It is especially important to test any logic processing with a simple example—either by hand or with a program simulator.


set to drive first column
last column?
Yes
No
send out selected column
shift accumulated reading right and OR in the latest row reading
set to drive next column
any change?
No
Yes
compute new pushes and releases
return

Keypad scanning flowchart

The first part of the keyscan software is the scanning of a 4 × 4 matrix. The high nibble (4 bits) of the port sends out the column drives, and the low nibble of the port reads back the rows. The program combines the read-back results (4 bits for each column driven) into a 2-byte number. The last step involves comparing the new reading with the previous reading to test for new pushes and new releases. How short the C program is! As you see it, though, it is not intuitively understandable.[43]

[43]The flowchart may help. Details of flowcharting are given in Chapter 5.

Key Scan Program

```
   #include <absacc.h>
44 #define PORTA XBYTE[0xffc0]
   unsigned old,new,push,rel,temp;
   unsigned char clmn_pat;
   void main(void){
45  for (clmn_pat=0x10;
            clmn_pat<>0;clmn_pat<<1){
      PORTA=PORTA & clmn_pat;
      new=(new<<4) | (PORTA & 0xf);
    }
46  if (temp=new ^ old)>0){
     push=temp & old;
     rel=temp & new;
    }
  }
```

KEY SCAN PROGRAM

```
   PORTA EQU 6000H
   DSEG
     OLD: DS 2
     NEW: DS 2
     PUSH: DS 2
     REL: DS 2
   CSEG
47 SCAN:MOV DPTR,#PORTA
```

[44]This is an alternate way to define the external ports in Keil C where you do not want to link in a separate assembly module. The header file includes the definition of the *XBYTE* function.

[45]This loop scans the keys. The *for* loop is discussed in the next chapter. By looking for the pattern (*clmn_pat*) becoming zero, this will loop four times. After the fourth *shift left* (>>), the 1 bit will "fall off," and the pattern will have all 0s. A *shift* fills 0s instead of bringing bits around as an assembly language rotate would do.

[46]This block identifies the *changes* since last time using the logic just described. C permits ***embedded assignments***, so the *if* test includes the filling in of a value for *temp*. The use of *temp* could be avoided, but there might then be more code produced because the logical operation would have to be carried out three times. Notice the difference between the = here and the ==, which would only have done the test for equality without assigning any values.

[47]*SCAN* is a "complete" program but rather pointless since nothing is done with the results of the key scan. Chapter 7 goes into the details of subroutines, which would be a better use for this program. The *SHIFT* part is a *subroutine* since it ends with *RET* and is called from the main program.

```
48    MOV R0,#00010000B;COLUMN 1 LOOP:
      ANL R0,#11110000B ;4 BITS LOW
      MOV A,R0
      MOVX @DPTR,A   ;DRIVE COLUMN
      MOVX A,@DPTR   ;GET ROW READING
      MOV R2,#NEW   ;STORE LOCATION
49    LCALL SHIFT   ;STORE NIBBLE
      MOV A,R0
      RL A ;NEXT COLUMN-LEFT BY ONE
      MOV R0,A
      JNZ LOOP   ;NOT DONE SCANNING
      MOV A,NEW   ;TEST FOR CHANGES
      XRL A,OLD
      JNZ CHANGED
      MOV R0,A
      MOV A,NEW+1
      XRL A,OLD+1
      JZ DONE   ;NO CHANGES-GO ON
    CHANGED:MOV R1,A
      MOV A,OLD ;GET NEW PUSHES
50    ANL A,R0
      MOV PUSH,A
      MOV A,OLD+1
      ANL A,R1
      MOV PUSH+1,A
      MOV A,NEW ;GET NEW RELEASES
      ANL A,R0
      MOV REL,A
      MOV A,NEW+1
      ANL A,R1
```

[48]The main program, *SCAN* involves going four times around *LOOP* in order each time to drive one column and read back the four rows. The result is put into the 2-byte storage called *NEW*.

[49]This routine is used to gather the 4-bit readings into a 16-bit (2-byte) result. Notice how the *RLC* allows the low-byte bits to carry across into the high-byte. When the previous value has moved over by four, then the lower 4 bits are masked and the newest 4-bits are ORed in.

[50]The logical operations described above to sort out new pushes and releases begin here.

```
    MOV MKL+1,A
    DONE: MOV OLD,NEW ;UPDATE OLD
    MOV OLD+1,NEW+1
    LJMP SCAN ;START OVER

  SHIFT: MOV R1,#4 ;LOW 4BITS ⇒ 16BITS
  S2:CLR C
    RLC @R2+1
    RLC @R2
    DJNZ R1,S2
    ORL ACC,#0FH
    ORL @R2,ACC
    RET
  END
```

ARITHMETIC OPERATORS

Add, subtract, multiply, and divide are the math operations supported directly in the hardware of most microcontrollers. It is deceptive to say that the 8051 family *supports* all four math operations.[51] It supports them only for (unsigned) bytes. Instead, *groups* of assembly language instructions handle math for bigger variables. For *unsigned int* variables, C compilers will either code the necessary instructions in-line or add in the necessary function calls to a library function. *Complete* ANSI C compilers must support double precision, signed, and floating-point.

ACCURACY: Many operations in embedded control have very specific limits to the range and precision of numbers. Look very closely at what accuracy you *really* need. If the input is 8 bits, perhaps 8-bit math will suffice if you can be sure intermediate results won't overflow.

[51]Multiplication and division were not in the earliest machines such as the 8008 and 8080. You can get multiplication and division with successive addition or subtraction or through shift-and-add/subtract routines. Some 16- and 32-bit processors directly support unsigned integer math, but even there it has been common to rely on a math co-processor. There are at least two in the 8051 family that have an added co-processor on-chip for large variables, but even in those cases the co-processor is a separate hardware device rather than an extension to the instruction set.

MATH VS. LOOKUP: Especially with non-linear transformations (say the speed of a pump versus flow rate), it may be sufficient to use a lookup table (and linear interpolation if necessary) rather than the complex math arising out of some curve-fitting algorithm. Anything that can be pre-computed and put in a table will run much faster than a multibyte math operation.

If you are interested in writing your own math algorithms, an example in Chapter 8 (page 203) shows a set of assembly language double-precision math functions but most C programmers are quite content to rely on the libraries supplied with the compiler.

C automatically does ***type conversion*** (expanding bytes to word/integer, etc.). For example, if you add a byte to an integer, the result will be an integer. C has an operation to *force* a variable to a different type.

```
          Automatic Type Conversion
 unsigned int a,b;
 unsigned char c;
52 a=b+c;
```

```
                  Casting
 unsigned int a;
 unsigned char b,c;
53 a=b+(unsigned int)c;
```

[52]The program takes *b*, treats *c* as though there were 8 high-order zeros attached, does the math, and assigns the result to the 2 bytes of *a*. If *a* were an *unsigned char*, then the 2-byte result of the math might have the upper byte discarded (even if that included non-zero bits).
[53]Where *b* and *c* are bytes and *a* is an *int*, the math might be carried out with single-byte precision, and the carry bit might be lost *before* the result is converted to a 2-byte quantity. Some C compilers for other processor families *promote* (change—enlarge) all *char* variables to integers. Here the parenthesis ahead of C casts it to an integer before the math is done so the intermediate result must be kept in an integer.