Controlling The Real World With Computers

Experiment 2 - Expanding Switch Input Detection

This experiment will use the 8255 Programmable Peripheral Interface (PPI -- see the data sheet: 8255.PDF). An 8255 has 3 digital ports: Port A, Port B and Port C. Each port is 8 bits wide.

Here, Port A's bits will be called PA0, PA1, PA2, PA3, PA4, PA5, PA6 and PA7. Port B's bits are PB0, PB1, etc., and Port C's bits are PC0, PC1, etc.

The 8255 can be programmed in mode 0, 1 or 2.

In mode 0, Ports A and B operate as inputs or outputs, and Port C is divided into two 4-bit groups, either of which can be operated as inputs or outputs.

Mode 1 is the same as Mode 0, but Port C is used for hardware handshaking. Handshaking is defined in NightFlight's Free On-line Dictionary of Computing as "A technique for regulating the flow of data across an interface by means of signals carried on separate wires."

In mode 2, Port A is bidirectional (both input and output) and Port C is used for hardware handshaking Port B is not used.

The various modes can be set by writing a special value to the control register. The control register is at base + 0x23.

	Group A				Group B		
D7	D6	D5	D4	D3	D2	D1	D0
1 = Set Mode	Upper Mode Select		Port A	Port C Upper Nibble	Lower Mode Select	Port B	Port C Lower Nibble
	00 = Mode 0 01 = Mode 1 1X = Mode 2		1 = In 0 = Out	1 = In 0 = Out	0 = Mode 0 1 = Mode 1	1 = In 0 = Out	1 = In 0 = Out

This experiment will use mode 0. The table below shows the control register values needed to configure Ports A, B and C as inputs and outputs in mode 0:

Control Values	Bits								Port Assignments			
	D7	D6	D5	D4	D3	D2	D1	D0	Port A	Port C Upper	Port B	Port C Lower
0X80	1	0	0	0	0	0	0	0	Output	Output	Output	Output
0X81	1	0	0	0	0	0	0	1	Output	Output	Output	Input
0X82	1	0	0	0	0	0	1	0	Output	Output	Input	Output
0X83	1	0	0	0	0	0	1	1	Output	Output	Input	Input
0X88	1	0	0	0	1	0	0	0	Output	Input	Output	Output
0X89	1	0	0	0	1	0	0	1	Output	Input	Output	Input
0X8A	1	0	0	0	1	0	1	0	Output	Input	Input	Output
0X8B	1	0	0	0	1	0	1	1	Output	Input	Input	Input
Control Values	Bits								Port Assignments			
	D7	D6	D5	D4	D3	D2	D1	D0	Port A	Port C Upper	Port B	Port C Lower

0X90	1	0	0	1	0	0	0	0	Input	Output	Output	Output
0X91	1	0	0	1	0	0	0	1	Input	Output	Output	Input
0X92	1	0	0	1	0	0	1	0	Input	Output	Input	Output
0X93	1	0	0	1	0	0	1	1	Input	Output	Input	Input
0X98	1	0	0	1	1	0	0	0	Input	Input	Output	Output
0X99	1	0	0	1	1	0	0	1	Input	Input	Output	Input
0X9A	1	0	0	1	1	0	1	0	Input	Input	Input	Output
0X9B	1	0	0	1	1	0	1	1	Input	Input	Input	Input

In this experiment, the configuration code 0x82 will be used, making Port A an output, Port B an input, and both halves of Port C outputs. Only Port B will be used.

The inputs to Port B are pulled up by resistors. A bit will be high unless the input is taken to ground. Thus, Port B can be used in a manner similar to that described in Experiment 1. Doing so adds 8 more digital inputs that can be used as switch inputs. With the 3 already discussed in Experiment 1, adding Port B's 8 could provide a total of 11. That's more than enough for many applications.

In this experiment, Port B will handle switches 4 through 11, assigned in the following manner:

Switch	11	10	9	8	7	6	5	4
Port B Bit	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
HEX Bit Weight	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01

A method can be used here that is very similar to that used in Experiment 1. Let's call the Programmable Peripheral Interface Port B ppi_portb. We can return the information we need with the following:
return ((inp(ppi_portb) >> (input - 4)) & 1) ^ 1;

We again use the **right shift operator** (>>) to get the correct bit into the bit 0 (D0) position. This will cause the results of **inp(ppi_portb)** to be shifted to the right 0 to 7 bits, corresponding to switches 4 to 11 (input - 4), placing the desired bit into bit position 0 regardless of the value of the input.

Then, as in Experiment 1, the results of the above are ANDed with 1 to mask off all but bit 0, and XORed with 1 to reverse the polarity.

To outline the procedure:

1. Get the data from ppi_portb with inp(ppi_portb).

2. Shift the results of the above to the right (input - 4) bits. This will place the desired bit in the bit 0 position.

3. AND the results of the above with 1 to mask off everything but bit 0.

4. XOR the results of the above with 1 to reverse the polarity.

5. Return the resulting 1 or 0.

```
 return    the data from   shifted right    ANDed     and all of
   |         ppi_portb     input - 4 bits   with 1    this XORed
   |            |              |              |       with 1 to change polarity
   |            |              |              |          |
return ((inp(ppi_portb)   >> (input - 4))     & 1)      ^ 1;
```

The original is_switch() from Experiment 1 can be used with a few modifications, but first the ports must be declared. They are declared just above **is_switch()**. Their **scope** begins at that point in the file. Nothing above them knows about or can modify them.

```
// The following are known only to the functions that follow. They
// can't be modified or even accessed by anything above this point.
```

```
unsigned base;
unsigned switch_port;
unsigned ppi_porta;
unsigned ppi_portb;
unsigned ppi_portc;
```

Following the declarations is the new **is_switch()**:

```
int is_switch(int input)
{
  if(input < 1 || input > 11) // if the input is less than 1 or greater
    return -1;                // than 11, then return -1 showing an error

  if(input < 4) // else, we fall through the above and see if less than 4
    return ((inp(switch_port) >> (input + 3)) & 1) ^ 1; // yes, return using
switch_port

  return ((inp(ppi_portb) >> (input - 4)) & 1) ^ 1; // fell through "if(input < 4)"
                                                    // so >= 4 (greater than or
equal)
                                                    // so use PPI Port B

} // end is_switch()
```

The first **if** statement says that the input is in error if it's less than 1 or greater than 11. If the input is greater than 0 and less than or equal to 11, then we will skip the **return -1;**. Put in programming terms, the program **falls through the decision**.

The second **if** statement says, "if the input is less than 4, then the original switch_port is used to indicate an on or off condition." If it is not the case that the input is less than 4, then the program **falls through the decision** and the return will be based on input from ppi_port.

Two functions are below is_switch(). One is called **get_port()**. For now, it simply assigns the base, switch_port, ppi_porta, ppi_portb and ppi_portc their values. It will be expanded in another experiment to automatically find the base then do the assignments. The second function is **set_up_ppi()**. It is the function that places 0x82 in the control register.

```
// get the port -- this will grow later into an auto-detect function
void get_port(void)
{
  base = 0x300;
  switch_port = base + 0x18;
  ppi_porta = base + 0x20;
  ppi_portb = base + 0x21;
  ppi_portc = base + 0x22;
}
```

In Experiment 1 the 8255 Programmable Peripheral Interface (PPI) was shown to be at an offset of 0x20. That is its starting offset location. Notice in the schematic that it has address lines A0 and A1 connected to it. That's because it has several register locations in it and needs the address lines to select them. Up to 4 registers can be selected for read and 4 for write. The first three read registers are the ports that are part of the PPI. Port A is defined in the datasheet (8255.PDF) to be at the PPI's base location, which is our base plus 0x20. Port B is at the PPI's base plus 1, and Port C is at the PPI's base plus 2.

```
// set up the ppi
void set_up_ppi(void)
{
  unsigned control = base + 0x23;

  outp(control, 0x82); // a = out, b = in, c = out
```

```
}
```

Notice that the variable in set_up_ppi() called **control** is both declared and used in set_up_ppi(). Variables declared inside funtions are temporary unless the **static** keyword is added to their declaration. Adding static to the declaration causes the compiler to reserve memory space for a variable and allow it to remain in memory until the end of the program. If the line had been, **"static unsigned control = base + 0x23;"**, then **control** would have had memory reserved for it and had the value base + 0x23 assigned to it. Without an assignment, integer, unsigned, char and some other static variables are assigned a value of 0.

There was no need to make **control** static however, since it is used only once in set_up_ppi(). The value base + 0x23 is still assigned to it, but the variable is retained only as long as the function is in control. This type of variable is called an automatic variable. It is very important to remember that automatics are created for a function but are destroyed when the function terminates. They should never be expected to retain their values. Put another way, the scope of an automatic variable is limited to the function in which it is declared. Also, automatic or static variables of the same name in different functions have nothing in common except their names. Changing one does absolutely nothing to the other. Keeping these points in mind will help you avoid some common errors.

The program for testing the new is_switch() is below.

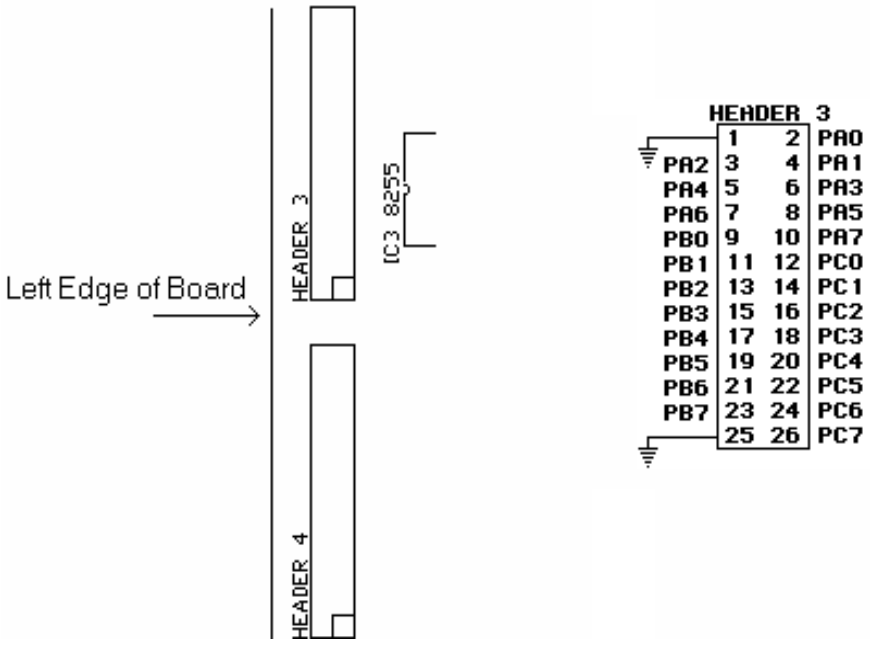
```
// experi2.c

#include <stdio.h> // contains prototypes for printf() and puts()
#include <conio.h> // contains prototypes for inp() and kbhit()

// local prototypes
int is_switch(int input);
void get_port(void);
void set_up_ppi(void);

void main(void)
{
  int x,y;

  get_port(); // get the value of the base Port Address

  set_up_ppi(); // set up ppi_porta = out, ppi_portb = in, ppi_portc = out

  switch_port = base + 0x18;

  while(1)
  {
    if(kbhit()) // "keyboard hit" -- a compiler function that sees if a key has been
      break;    // pressed.  We will break out of the while() loop if a key is
pressed.

    puts(""); // puts("") will put a new line on the screen
    for(x=0; x<7; x++) // 0 through 6 -- purposely use 0 to produce an error
    {
      y = is_switch(x);
      printf("x=%02d y=%02d|",x,y); // don't print a new line yet
    }
    puts(""); // puts("") will put a new line on the screen
              // after all 7 items have been printed

    for(x=7; x<13; x++) // 7 through 12 -- purposely use 12 to produce an error
    {
```

```
      y = is_switch(x);
      printf("x=%02d y=%02d|",x,y); // don't print a new line yet
    }
    puts(""); // puts("") will put a new line on the screen
              // after all 6 items have been printed

    sleep(1); // sleep -- compiler function -- wait one second
              // so printed lines can be seen
              // remark sleep out or delete it for full speed

  } // end while(1)

} // end main()


/*
 The following are known only to the functions that follow. They
 can't be modified or even accessed by anything above this point.
 Their scope is from here to the end of the file.
*/
unsigned base;
unsigned switch_port;
unsigned ppi_porta;
unsigned ppi_portb;
unsigned ppi_portc;


// ================================================================
//                          is_switch
// 1. Return -1 error indicator if the input
//    is less than 1 or greater than 11.
//
// 2. If there is a fall-through from the above and the input
//    is less than 4, return the status based on switch_port.
//
// 3. If there is a fall-through from both of the above, then
//    return the status based on ppi_portb.
// ================================================================
int is_switch(int input)
{
  if(input < 1 || input > 11) // if the input is less than 1 or greater
    return -1;                // than 11, then return -1 showing an error

  if(input < 4) // else, we fall through the above and see if less than 4
    return ((inp(switch_port) >> (input + 3)) & 1) ^ 1; // yes, return using
switch_port

  return ((inp(ppi_portb) >> (input - 4)) & 1) ^ 1; // fell through "if(input < 4)"
                                                    // so >= 4 (greater than or
equal)
                                                    // so use PPI Port B

} // end is_switch()


// Get the port.  This will grow into an
```

```
// auto-detect function in the future.
void get_port(void)
{
  base = 0x300;
  switch_port = base + 0x18;
  ppi_porta = base + 0x20;
  ppi_portb = base + 0x21;
  ppi_portc = base + 0x22;
} // end get_port()


// Set up the ppi in mode 0.
// Make Port A an output, Port B an input and Port C an output.
void set_up_ppi(void)
{
  unsigned control = base + 0x23;

  outp(control, 0x82); // a = out, b = in, c = out

} // end set_up_ppi()

// end experi2.c
```

NOTE: Please be sure to read the Warranty And Disclaimer before working with the hardware!

The following shows the connections to PPI Port B through Header 3. The switch inputs are on odd-numbered pins 9 through 23 of Header 3. They correspond to switch inputs 4 through 11. Switch inputs 1 through 3 are accessed on Header 1 the same way they were in Experiment 1. The location of Header 3 and its pinout are shown below. Remember that pin 1 of any of the headers is the one with the square shape on the back of the board. The view below is from the top of the board.



The following table shows the relationship between the Header 3 pins and the switches:

Switch	4	5	6	7	8	9	10	11
Port B Bit	PB0	PB1	PB2	PB3	PB4	PB5	PB6	PB7

Header 3 Pin Number To Connect To Pin 1 Or 25 To Show Indicated Switch On	9	11	13	15	17	19	21	23

What the table says is that connecting pin 9 to pin 1 or 25 of Header 3 should indicate that switch 4 has been turned on. Connecting pin 11 to pin 1 or 25 of Header 3 should indicate that switch 5 has been turned on. Switches 6 through 11 should work the same way. They use the odd-numbered pins 13 through 23. Connecting any of them to to Header 3 pin 1 or 25 should show the corresponding switch is on.

Type in the above program and save it as experi2.c. There is something wrong with it. Know what it is without compiling it? If not, compile it and note the error produced. What is the reason for the error? Can't figure it out? E-mail me and I'll let you know. A hint -- it has something to do with scope.

When you get it fixed and compiled, you can test it by connecting the appropriate lines from Header 1 and Header 3 to indicate switches 1 through 11 are on or off, and see if the program properly indicates their status.

Previous: Experiment 1 - Basic Switch Input Detection
Next: Experiment 3 - The General Purpose Digital Input/Output Module - Part 1
Problems, comments, ideas? Please e-mail me

Order | Home