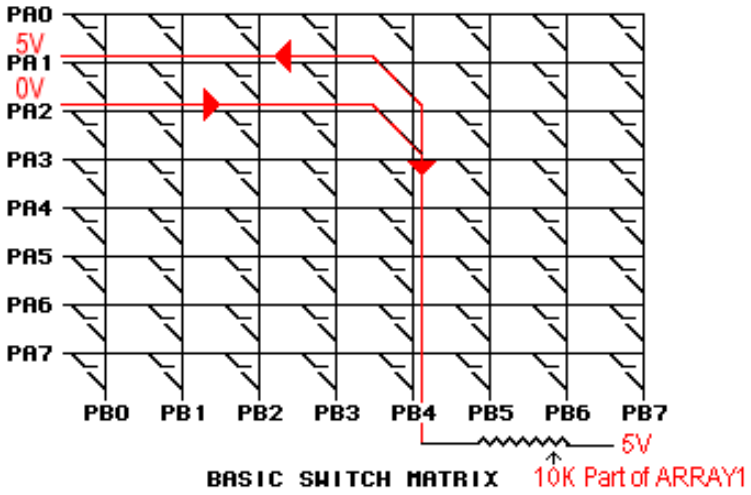
Controlling The Real World With Computers

Experiment 4 - The Multiple Closure Problem And Basic Outputs With The PPI

The matrix approach in Experiment 3 added significant closure detection capability to the system. The input portion of the tutorial will be completed in this part of our discussion of the I/O module by showing some additional approaches to the matrix, then we will work on controlling a little higher-power devices. That's probably welcome to those who are tired of all the work on closures. The reason so much is being done with closures however, is two-fold. In the first place, much of the control and embedded world deals with closures, whether they are actual, physical metalic closures or simulated by light, magnetic or other means. Secondly, it provides a good opportunity to work with bit logic, which is very important in control and embedded systems.

The first thing we have to do is find a solution to a potential problem with the matrix in its present form. If you have read the Q & A lately (advisable), you will find a question about using multiple switch closures with the matrix used in Experiment 3. Take another look at the matrix:

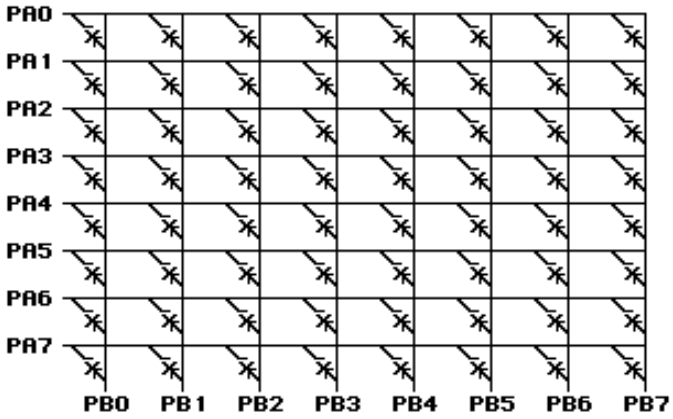


NOTE: Please be sure to read the Warranty And Disclaimer before working with the hardware!

Remember that the way a closure in the matrix was detected was to take a Port A line low in the PPI (programmable peripheral interface), which is commonly called **strobing**, then look at a column bit on Port B. The intersecting contact is closed if the Port B bit is pulled low. Let's consider a couple of closures. Let's say we take PA2 on Port A low. Port B will show 0xFF if there are no closures since all of its bits will be pulled high by the resistors in ARRAY1 (see the schematic). Let's say that a switch at the intersection of PA2 and PB4 is turned on. Since PA2 is low, it will pull PB4 low through the closure, indicating the switch is on.

So far, so good, but notice what happens if the switch at the juction of PA1 and PB4 is also turned on. Remember that the matrix is strobed by taking only one bit of Port A low at a time. That means that all of the other bits on Port A are high, so PA1 will be high. A circuit will be made through the two switches that will connect PA1 and PA2 to each other. Since one is high and the other low, they will "fight" each other, and might stress the device beyond its limits.

Now take a look at the following changes to the matrix:



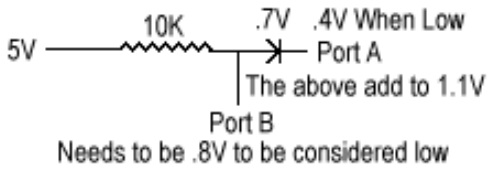
SWITCH MATRIX WITH DIODES

Notice the diode at each intersection in series with each switch (take a look at How To Read A Schematic if you don't remember what diode and/or series means). Now imagine that both PA1,PB4 and PA2,PB4 are still turned on and that PA2 is again strobed low. PB4 will still be pulled low through the switch/diode series circuit at the junction of PA2 and PB4, but the diodes will prevent PA1 and PA2 from fighting each other because there will be almost no current through the diode at PA1,PB4. Remember from How To Read A Schematic that:

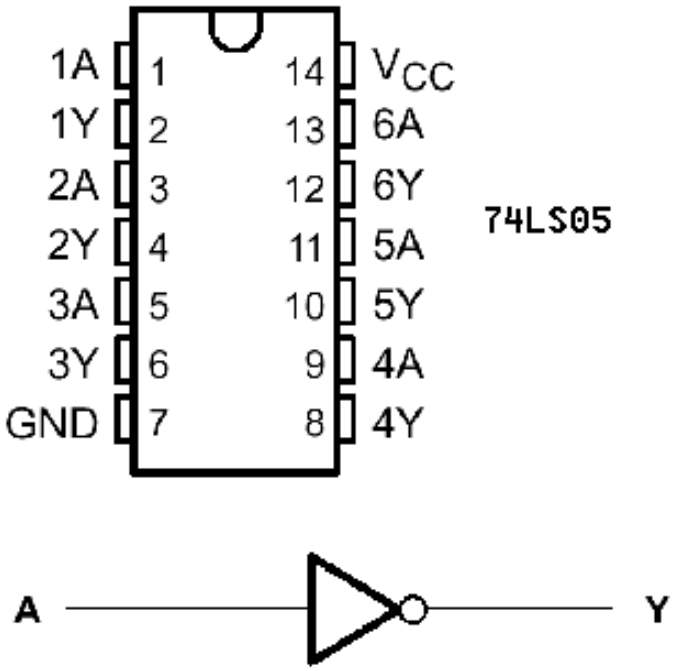
- a diode provides a one-way path for current,
- that the charge flow is generally considered in electronics to be from negative to positive
- and that the flow in a diode is, therefore, in the opposite direction of the arrow, from cathode to anode.

Remember that PA1 is near 5V and PA2 is near ground at the instant PA2 strobes the row ("near" because most digital devices do not actually go all the way to full sytem positive or all the way down to ground -- often called **rail to rail swing**, which has pert near nothing to do with 40s-style music). The charge flow would be from PA2 to PA1 (negative to positive) without a diode. The diode at PA1,PB4 is **reverse biased** however, and cannot conduct. There is a little leakage current, but it's insignificant. The diode at PA2,PB4 is **forward biased** and conducts. In this case, the charge flow is from PA2 to PB4's pullup resistor, pulling PB4 near ground. It's not going to be **at** ground due to the characteristics of the digital device, plus the fact that a silicone diode has about a .7V drop across it when it is conducting. A germanium diode is a little better, with about .3V drop.

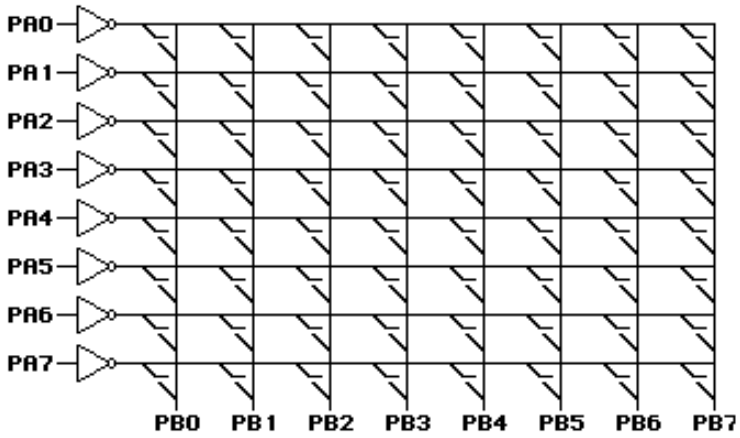
If you would like to make a matrix with diodes, you could try the common, easy-to-find 1N4148 silicone diodes. If you read the data sheet however, you will find, at least for the 82C55, that a typical maximum low input voltage is .8V. The low output has a typical maximum of .4V. There might be times when you can't pull Port B low enough to read back a low logic level. The following diagram shows the limits imposed by the silicon diode voltage drop and the low limit of the 82C55. The two effectively add, producing a 1.1V design minimum at Port B. It is considered a design minimum because worst-case parameters are used, as they should be.



Germanium diodes such as the 1N60 will work more reliably in this application with their lower .3V drop, but are a little harder to find. Another alternative is to find something to drive the rows that removes itself when not turned on. For example:



The 74LS05 (see 74LS05.PDF) and similar devices do not produce a high output. They only pull things low, commonly called **sinking**, which is what they can do with the pullups on Port B. Notice that the device is an inverter (see the boolean logic section). The Port A lines would connect to the "A" inputs, and the row lines to the "Y" outputs. A Port A output would then be taken high to strobe a row, which would cause the inverter output to pull the row low. When the Port A line is turned off, the inverter output will release the row line. It will not drive the row high, removing the need for diodes and their voltage drops. With six inverters each, two of these devices are needed to modify the matrix in this experiment:



BASIC SWITCH MATRIX

All said and done, you can use the plain matrix for most of the experiments. This information is provided just in case someone wants to experiment with multiple closure inputs.

Let's explore just a little more with the matrix, then I'll get on with powering stuff. In Experiment 3, the matrix used Ports A and B to provide 64 closure detection points and the 74LS244 added 3 more to give a total of 67. For still more capability, the 74LS244 can be used to add 3 column bits. With Port B providing 8 column bits and still using Port A to strobe the rows with its 8 bits, we have 8 * 11 = 88 closure points. That's more than enough for most applications.

There are many ways the 74LS244 can be added. For example, it and Port B can be connected to the rows rather than the columns. There is no rule that says they must be used to monitor columns. They could just as easily monitor the rows and something else used to strobe the columns. For convenience we will stay with column monitoring since we began with that arrangement.

In the real world, consideration must be given to such factors as board layout and programming complications. Consider what

would be involved if the switches were arranged in the following manner:

```
            |--------------------------------|
Port A bit 0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|
Port A bit 1|12|13|14|15|16|17|18|19|20|21|22|
Port A bit 2|23|24|25|26|27|28|29|30|31|32|33|
Port A bit 3|34|35|36|37|38|39|40|41|42|43|44|
Port A bit 4|45|46|47|48|49|50|51|52|53|54|55|
Port A bit 5|56|57|58|59|60|61|62|63|64|65|66|
Port A bit 6|67|68|69|70|71|72|73|74|75|76|77|
Port A bit 7|78|79|80|81|82|83|84|85|86|87|88|
            |--------------------------------|
              S  S  S
              W  W  W  P  P  P  P  P  P  P  P
              1  2  3  B  B  B  B  B  B  B  B
  Bit Numbers 4  5  6  0  1  2  3  4  5  6  7
```

SW1, 2 and 3 are the 74LS244 SWITCHn inputs and their bit numbers. PB[0:7] are the Port B bit numbers used ([0:7] is shorthand meaning "0 through 7"). The problem with this arrangement is that the 74LS244 is used for the first three closures in a row, then Port B must be used for the next 8, then back to the '244 for the first three places of the next row. It can be done, and should be if electrical layouts demand it, but another arrangement makes for easier programming. Consider this:

```
            |--------------------------------|
Port A bit 0| 1| 2| 3|25|26|27|28|29|30|31|32|
Port A bit 1| 4| 5| 6|33|34|35|36|37|38|39|40|
Port A bit 2| 7| 8| 9|41|42|43|44|45|46|47|48|
Port A bit 3|10|11|12|49|50|51|52|53|54|55|56|
Port A bit 4|13|14|15|57|58|59|60|61|62|63|64|
Port A bit 5|16|17|18|65|66|67|68|69|70|71|72|
Port A bit 6|19|20|21|73|74|75|76|77|78|79|80|
Port A bit 7|22|23|24|81|82|83|84|85|86|87|88|
            |--------------------------------|
              S  S  S
              W  W  W  P  P  P  P  P  P  P  P
              1  2  3  B  B  B  B  B  B  B  B
  Bit Numbers 4  5  6  0  1  2  3  4  5  6  7
```

Now the 74LS244 is used for [1:24], and Port B for [25:88]. The change of column registers is made above 24 and never switched back. Using the logic suggestions from Experiment 3 to figure out the logic for the 74LS244 section:

1. **If possible and useful, subtract a constant to make a table start at 0:**
   Subtracting 1 will change the numbers from 1 through 24 to 0 through 23.
2. **Remember that it's always possible to shift a 1 around to put it where you need it, so see if you can divide by something or do some modulus arithmetic to get a number that can be used as a left or right bit-wise shift number. In an 8-bit system, try to make things end up 0 through 7:**
   The goal is to shift the result bit down to position 0. That means we need to shift 4, 5 or 6 bits for the first 3 numbers on a row after subtracting. Dividing by 3 and neglecting the remainder won't work. That can be confirmed by trying the operation on the first row. For the first three numbers of the first row, we will get 0, 0 and 0 if we divide 0, 1 and 2 by 3. Try modulo 3. Recall from Experiment 3 that the modulo operation returns the remainder, so divide by 3 but use the remainder rather than the quotient as the result of the operation. Now we get 0,1 and 2 for the first 3 numbers. That might not be right since it gives back the same numbers, so try the last 3. The first three numbers on the last row after subtracting 1, 21, 22 and 23, will also give us 0,1 and 2. Try the middle row's first 3 numbers. The numbers 9, 10 and 11 also provide 0, 1 and 2. All that's needed after the modulo arithmetic is to add 4 which will provide the required 4, 5 and 6 shift numbers for the first 3 numbers on each row. The information from the 74LS244 buffer can be shifted the correct number of bit positions based on the input in order to move the bit into the least significant 0 position.
3. **Consider all masking possibilities, especially AND:**

Take the value returned from the 74LS244 buffer and shifted using the above procedure and AND it with 1. That will mask off all but bit 0. Now XOR with 1 to invert the result. Remember from the Boolean Logic section that the XOR operation is true if the two inputs are different. Also recall that when a closure is turned on, the resulting bit will be low. When a closure is off, the XOR will be 1 with 1, resulting in 0. When the closure is on, the XOR will be 1 with 0, resulting in 1. Thus, 0 will be returned for off, and 1 will be returned for on.

The following shows the results of the effort to get the correct shift number:

```
  Input       -1         %3         +4
 1  2  3    0  1  2    0  1  2    4  5  6
 4  5  6    3  4  5    0  1  2    4  5  6
 7  8  9    6  7  8    0  1  2    4  5  6
10 11 12    9 10 11    0  1  2    4  5  6
13 14 15   12 13 14    0  1  2    4  5  6
16 17 18   15 16 17    0  1  2    4  5  6
19 20 21   18 19 20    0  1  2    4  5  6
22 23 24   21 22 23    0  1  2    4  5  6
```

Now apply step 3 by ANDing the shifted number with 1 then XORing it with 1.

There is a multiple line way to go through the process and a one-line way. First, the multiple line way:

```
int sw_in_port_val,shiftval;  // switch in port and shift values

shiftval = input - 1;
shiftval%=3; // same as shiftval = shiftval % 3;
shiftval+=4; // same as shiftval = shiftval + 4;

sw_in_port_val = inp(switch_port); // get a number out of the 74LS244 buffer
sw_in_port_val>>=shiftval; // shift the port value to the right shiftval times
sw_in_port_val&=1; // same as sw_in_port_val = sw_in_port_val & 1;
sw_in_port_val^=1; // same as sw_in_port_val = sw_in_port_val ^ 1;
```

Then, the one-line way:

```
Return
   |      the port value
   |                |      shifted
   |                |      to the
   |                |      right
   |                |         |     by the
   |                |         |     input
   |                |         |     less 1
   |                |         |        |    modulo 3
   |                |         |        |        |   plus
   |                |         |        |        |    4.
   |                |         |        |        |     |  Then AND
   |                |         |        |        |     |  all of
   |                |         |        |        |     |  that with 1
   |                |         |        |        |     |     |  and, finally,
   |                |         |        |        |     |     |  XOR the
   |                |         |        |        |     |     |  whole mess
   |                |         |        |        |     |     |  with 1.
   |                |         |        |        |     |     |     |
return ((inp(switch_port) >> (((input - 1) % 3) + 4)) & 1) ^ 1;
       12   3           3    456         6    5    42    1
```

The numbering of the parenthesis would not, of course, be included in the actual code, although it would be handy if compilers would show it. It is included here to show what belongs to what. The raw input is from the port inside 3. This value gets shifted by what is inside 4. Everything inside 2 gets ANDed with 1. Everything inside 1 is XORed with 1, and 5 and 6 simply block things off so operations will take place as expected.

One quick check to see if there are at least the right number of parenthesis is to confirm that there are the same number of close parenthesis as opens.

Notice how the numbering works. Start numbering opens from the left. When a close is encountered, give it the same number as the last unclosed open.

Of course the whole mess can be avoided by using the multiple line method. It's a lot easier to read and understand, although it might take a little more processor time and use a bit more memory. Those are the only good reasons to use the more complex code. Never write code that's hard to understand unless it's necessary. Too many people do such things in an attempt to show how smart they are, and end up showing just how dumb they can get. It is done in this tutorial to provide you with the information you need just in case memory and/or processor time is getting critical.

All that's needed now is to figure out what needs to go in Port A to strobe the appropriate row. Maybe you have already noticed something: Any time a series of numbers has been encountered where each one was 1 greater than the previous, a modulo has been used that has been one greater than the highest number in the first series of numbers. For example, when the first series was 0, 1 and 2, modulo 3 was used. With [0:7] as the first series, it and [8:15] use modulo 8.

Any time a series of numbers has been encountered with an increment greater than 1, we have divided by the increment and neglected the remainder.

So what do we give Port A? We are going down the first column with the series 0, 3, 6, 9, 12, 15, 18 and 21 after subtracting 1. The increment is 3, not 1. We need 0 through 7 for bit numbers to strobe the rows. This is what we get when we divide by 3 and neglect the remainder:

Input			-1			divide by 3 and neglect remainder		
1	2	3	0	1	2	0	0	0
4	5	6	3	4	5	1	1	1
7	8	9	6	7	8	2	2	2
10	11	12	9	10	11	3	3	3
13	14	15	12	13	14	4	4	4
16	17	18	15	16	17	5	5	5
19	20	21	18	19	20	6	6	6
22	23	24	21	22	23	7	7	7

Now what will be stored in Port A is loaded with 0xff. Next, a 1 is shifted to the proper location using the above shift value. The result of the shift is XORed with the 0xff in the Port A value. Since this is a little different from previous approaches, consider what the results would be for the middle row.

The shift number for an input of 10, 11 or 12 is 3. Starting with 1 means only bit 0 has anything in it: 00000001. Shift it to the left 3 places and you get 00001000. The Port A value has 0xff = 11111111. Now XOR the two:

11111111
00001000
11110111 The result is the clearing of bit 3 and no others.

And this is an outline of the routine up to now:

```
// digital.c

int porta_val;
int porta_mask;

int portb_val;
int portb_mask;

int portc_val;
int portc_mask;


int is_closure(int input)
{
  int sw_in_port_val,shiftval;  // switch in port and shift values

  if(input < 1 || input > 88) // if the input is less than 1 or greater
    return -1;                // than 88, then return -1 showing an error

  // we fell through the above so see if input is less than 25
  if(input < 25)
  {
    input--; // same as input = input - 1;

    shiftval = input/3; // divide and neglect the remainder

    // A 1 is shifted to the proper location.
    // The result of the shift is XORed with
    // 0xff and stored in the Port A value,
    // turning off the desired bit and leaving
    // all of the others on.
    // Use porta_val = 1 << shiftval
    // if the 74LS05 is used, because
    // the desired operation is to turn
    // on the bit.
    porta_val = 0xff ^ (1 << shiftval); // or porta_val = 1 << shiftval with 74LS05

    // clear the appropriate Port A bit
    outp(ppi_porta, porta_val);

    shiftval = input % 3; // divide and use only the remainder
    shiftval+=4; // same as shiftval = shiftval + 4;

    sw_in_port_val = inp(switch_port); // get a number out of the 74LS244 buffer
    sw_in_port_val>>=shiftval; // shift the port value to the right shiftval times
    sw_in_port_val&=1; // same as sw_in_port_val = sw_in_port_val & 1;
    sw_in_port_val^=1; // same as sw_in_port_val = sw_in_port_val ^ 1;

    return sw_in_port_val;
  }

}// end is_closure()
// end digital.c
```

Now for the numbers greater than 24. First get the bit number for the Port A row strobe:

Input	-25	rows increment by 8 so divide by 8, neglect remainder
25 26 27 28 29 30 31 32	0 1 2 3 4 5 6 7	0 0 0 0 0 0 0 0
33 34 35 36 37 38 39 40	8 9 10 11 12 13 14 15	1 1 1 1 1 1 1 1
41 42 43 44 45 46 47 48	16 17 18 19 20 21 22 23	2 2 2 2 2 2 2 2
49 50 51 52 53 54 55 56	24 25 26 27 28 29 30 31	3 3 3 3 3 3 3 3
57 58 59 60 61 62 63 64	32 33 34 35 36 37 38 39	4 4 4 4 4 4 4 4
65 66 67 68 69 70 71 72	40 41 42 43 44 45 46 47	5 5 5 5 5 5 5 5
73 74 75 76 77 78 79 80	48 49 50 51 52 53 54 55	6 6 6 6 6 6 6 6
81 82 83 84 85 86 87 88	56 57 58 59 60 61 62 63	7 7 7 7 7 7 7 7

Then for the Port B row monitor bit shift number:

Input	-25	columns increment by 1 so modulo 8
25 26 27 28 29 30 31 32	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
33 34 35 36 37 38 39 40	8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7
41 42 43 44 45 46 47 48	16 17 18 19 20 21 22 23	0 1 2 3 4 5 6 7
49 50 51 52 53 54 55 56	24 25 26 27 28 29 30 31	0 1 2 3 4 5 6 7
57 58 59 60 61 62 63 64	32 33 34 35 36 37 38 39	0 1 2 3 4 5 6 7
65 66 67 68 69 70 71 72	40 41 42 43 44 45 46 47	0 1 2 3 4 5 6 7
73 74 75 76 77 78 79 80	48 49 50 51 52 53 54 55	0 1 2 3 4 5 6 7
81 82 83 84 85 86 87 88	56 57 58 59 60 61 62 63	0 1 2 3 4 5 6 7

See how easy it is? Notice that the divide and modulo numbers are the same.

Now just add it to the routine:

```
// digital.c

// The following are known only to the functions in this file.
// They can't be modified or even accessed by anything outside this
// file except through funtions in this file designed to provide access.
unsigned base;
unsigned switch_port;
unsigned ppi_porta;
unsigned ppi_portb;
unsigned ppi_portc;

int porta_val;
int porta_mask;

int portb_val;
int portb_mask;

int portc_val;
int portc_mask;

// ============================================================
//                          is_closure
// 1. Return -1 error indicator if the input
//    is less than 1 or greater than 88.
//
```

```
// 2. Return 1 if closure is on, 0 if it is off.

int is_closure(int input)
{
  int sw_in_port_val,shiftval;  // switch in port and shift values

  if(input < 1 || input > 88) // if the input is less than 1 or greater
    return -1;                // than 88, then return -1 showing an error

  // we fell through the above so see if the input is less than 25
  if(input < 25)
  {
    input--; // same as input = input - 1;

    shiftval = input/3; // divide and neglect the remainder

    // A 1 is shifted to the proper location.
    // The result of the shift is XORed with
    // 0xff and stored in the Port A value,
    // turning off the desired bit and leaving
    // all of the others on.
    // Use porta_val = 1 << shiftval
    // if the 74LS05 is used, because
    // the desired operation is to turn
    // on the bit.
    porta_val = 0xff ^ (1 << shiftval); // or porta_val = 1 << shiftval with 74LS05

    // clear the appropriate Port A bit
    outp(ppi_porta, porta_val);

    shiftval = input % 3;  // divide and use only the remainder
    shiftval+=4; // same as shiftval = shiftval + 4;

    sw_in_port_val = inp(switch_port); // get a number out of the '244 buffer
    sw_in_port_val>>=shiftval; // shift it to the right shiftval
                           // places to put in bit 0 position
    sw_in_port_val&=1; // same as sw_in_port_val = sw_in_port_val & 1;
    sw_in_port_val^=1; // same as sw_in_port_val = sw_in_port_val ^ 1;

    return sw_in_port_val;
  }

  // it's >= 25 if it gets this far

  input-=25; // same as input = input - 25;

  shiftval = input/8; // divide and neglect the remainder

  // A 1 is shifted to the proper location.
  // The result of the shift is XORed with
  // 0xff and stored in the Port A value,
  // turning off the desired bit and leaving
  // all of the others on.
  // Use porta_val = 1 << shiftval
  // if the 74LS05 is used, because
  // the desired operation is to turn
```

```
  // on the bit.
  porta_val = 0xff ^ (1 << shiftval); // or porta_val = 1 << shiftval with 74LS05


  // clear the appropriate Port A bit
  outp(ppi_porta, porta_val);

  shiftval = input % 8; // divide but use only the remainder

  portb_val = inp(ppi_portb); // get a number from Port B
  portb_val>>=shiftval; // shift it to the right shiftval
                        // places to put in bit 0 position
  portb_val&=1; // same as portb_val = portb_val & 1;
  portb_val^=1; // same as portb_val = portb_val ^ 1;

  return portb_val;

}// end is_closure()


// set up the ppi according to the dictates of the mode argument
void set_up_ppi(int mode)
{
  unsigned control = base + 0x23;
  int command;

  command = (mode & 0x0c) << 1; // shift bits 2 and 3 into positions 4 and 5
  command |= (mode & 3); // OR in bits 0 and 2
  command |= 0x80; // OR in bit 7 for PPI set up

  outp(control, command); // set according to mode command

} // end set_up_ppi()


// get the port -- this will grow into an auto-detect function in the future
void get_port(void)
{
  base = 0x240; // sw1 = 0 sw2 = 1 sw3 = 1
  switch_port = base + 0x18;
  ppi_porta = base + 0x20;
  ppi_portb = base + 0x21;
  ppi_portc = base + 0x22;

} // end get_port()

// end digital.c
```

The following is the test procedure:

```
// experi4a.c

#include <conio.h>
#include <stdio.h>
#include <bios.h>

// include header with constants
#include "constant.h"

// external prototypes
extern void set_up_ppi(int mode);
extern void get_port(void);
extern int is_closure(int closurenumber);

void main(void)
{
  int x,y,r,c;

  char *names[] = {"                |---------------------------------|",
                   "Header 3 Pin  2 | 1| 2| 3|25|26|27|28|29|30|31|32|",
                   "Header 3 Pin  4 | 4| 5| 6|33|34|35|36|37|38|39|40|",
                   "Header 3 Pin  3 | 7| 8| 9|41|42|43|44|45|46|47|48|",
                   "Header 3 Pin  6 |10|11|12|49|50|51|52|53|54|55|56|",
                   "Header 3 Pin  5 |13|14|15|57|58|59|60|61|62|63|64|",
                   "Header 3 Pin  8 |16|17|18|65|66|67|68|69|70|71|72|",
                   "Header 3 Pin  7 |19|20|21|73|74|75|76|77|78|79|80|",
                   "Header 3 Pin 10 |22|23|24|81|82|83|84|85|86|87|88|",
                   "                |---------------------------------|",
                   "   Header Number H1 H1 H1 H3 H3 H3 H3 H3 H3 H3 H3",
                   "      Header Pin  5  4  3  9 11 13 15 17 19 21 23",
                   NULL};

  get_port(); // get the port number and establish register locations

  // make A an output and B an input
  set_up_ppi(AOUT_CUPPERIN_BIN_CLOWERIN);

  clrscrn();  // clear the screen and automatically start printing
              // at 0,0 which is the top-left of the screen
              // (might not be available in all compilers)

  for(x=0; names[x]!=NULL; x++)
    printf("%s\n",names[x]);

  // (the following two lines might not be available in all compilers)
  r = cursrow() + 1; // get the screen row position plus 1
  c = curscol();     // get the screen column position

  while(1) // stay in loop forever
  {
    // "keyboard hit" is a function that checks
    // to see if a key has been pressed.
    if(kbhit())
      break;// A key was pressed -- break out of the while(1) loop
```

```
    // (the following might not be available in all compilers)
    poscurs(r,c); // put the cursor one line below where the phrases ended

    // Go through 88 switches with a row step of 11.  This will give
    // the first of each row, or 1, 12, 23, etc.  What the user
    // sees is reasonable for human viewing rather than reflecting
    // the matrix layout.  The y loop then runs through each row,
    // starting at 0 and going through 10.
    // The first row is x+y = 1, 2, 3 .... 11.
    // The second row is x+y = 12, 13, 14 ..... 22, and so on.
    for(x=1; x<89; x+=11)
    {
      for(y=0; y<11; y++) // each row is 11 columns wide
      {
        printf("%3d ",x+y); // print 3 characters wide with leading blanks
        if(is_closure(x+y)) // is there a closure? yes = ON, else OFF
          printf("%-3s","ON"); // "%-3s" means print string 4 characters
        else printf("%-3s","OFF"); //  wide with following blanks
      }
      puts(""); // go to next line

    } // end for(x=1; x<89; x+=11)

  } // end while(1)

} // end experi4a.c
```

This test is a little more complex than ones in the past. In the first place, it has instructions for connecting to the headers built into the program. As first mentioned in Putting It All Together, the "char *names[]" tells us that this is an array of pointers to to the phrases in quotes. In programming terms it is an array of pointers to **strings**. A string, in turn, is an **array of characters** just like a sentence on a piece of paper. For example, "This is a test" would look like this if it could be seen in memory:

T	h	i	s		i	s		a		t	e	s	t	0

The 0 at the end is called a **terminator**. It tells the software where the end of the string is located. The pointer for a string is to the memory location of the first character. Notice the NULL at the end of the array of strings. It can be considered a pointer to nowhere. The loop that reads

```
for(x=0; names[x]!=NULL; x++)
  printf("%s\n",names[x]);
```

says to start at the first string, which is names[0], and print strings until the NULL is encountered.

Please note that the clear screen and cursor location and positioning routines that are called in the program are in the MIX compiler, but might not be available with your compiler. Most DOS-based compilers have some means of clearing the screen and positioning, although it might be necessary to dig a little to find them.

There are many ways to use the various registers on the board to assist in detecting closures. There are however, some minor limits. Port B and the 74LS244 are the only registers with pullup resistors. The 74LS244 provides 3 such inputs and Port B 8. A lot of systems will do just fine with 3, 8 or 11 closure inputs. More can be had with matrix arrangements. With the 11 pulled up lines being used, there are 16 lines left on the PPI that could be used in various ways. If they were all used as strobes, 16 * 11 = 176 closures could be detected.

There will also be situations where closures would not be involved at all. For a truly universal system, all possible combinations would be needed so the programmer could select any of them. Both the basic settings for the PPI and all of the closure possibilities related to the PPI settings have to be considered.

The PPI has A, C Upper, B and C Lower that could potentially be put to use in a matrix in addition to the 74LS244. What's

more, Port B can be used as a row driver or as column register since it has pullup resistors. That means there are six things that have to be kept track of with Port B providing dual services.

The enumeration developed in Experiment 3 has already been used in this experiment to set up the PPI in the test program. Those enumeration numbers will be pushed over a little to make room for the additional information. Recall that enumerations are treated as integers. That means they are 16 bits in a DOS system. Since the PPI enumerators only take up 4 bits, they can be moved to the left 6 bits to make room for the matrix indicators with plenty of room left. The following is the PPI enumeration with the numbers shifted left 6 places:

```
  enum
  {
    Aout_CUout_Bout_CLout = 0x0,
    Aout_CUout_Bout_CLin = 0x40,
    Aout_CUout_Bin_CLout = 0x80,
    Aout_CUout_Bin_CLin = 0xC0,
    Aout_CUin_Bout_CLout = 0x100,
    Aout_CUin_Bout_CLin = 0x140,
    Aout_CUin_Bin_CLout = 0x180,
    Aout_CUin_Bin_CLin = 0x1C0,
    Ain_CUout_Bout_CLout = 0x200,
    Ain_CUout_Bout_CLin = 0x240,
    Ain_CUout_Bin_CLout = 0x280,
    Ain_CUout_Bin_CLin = 0x2C0,
    Ain_CUin_Bout_CLout = 0x300,
    Ain_CUin_Bout_CLin = 0x340,
    Ain_CUin_Bin_CLout = 0x380,
    Ain_CUin_Bin_CLin = 0x3C0
  } PPI_Values;
```

Notice "PPI_Values" at the end. It is optional. If you name an enumeration in this manner, it can only take on one of the values of the enumeration. Something such as "PPI_Values = 5;" however, would be illegal. It's a good idea to test things such as an enumeration to make sure the numbers were entered correctly. Simple:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
{
  int x,bit;

  enum
  {
    Aout_CUout_Bout_CLout = 0x0,
    Aout_CUout_Bout_CLin = 0x40,
    Aout_CUout_Bin_CLout = 0x80,
    Aout_CUout_Bin_CLin = 0xC0,
    Aout_CUin_Bout_CLout = 0x100,
    Aout_CUin_Bout_CLin = 0x140,
    Aout_CUin_Bin_CLout = 0x180,
    Aout_CUin_Bin_CLin = 0x1C0,
    Ain_CUout_Bout_CLout = 0x200,
    Ain_CUout_Bout_CLin = 0x240,
    Ain_CUout_Bin_CLout = 0x280,
    Ain_CUout_Bin_CLin = 0x2C0,
```

```
    Ain_CUin_Bout_CLout = 0x300,
    Ain_CUin_Bout_CLin = 0x340,
    Ain_CUin_Bin_CLout = 0x380,
    Ain_CUin_Bin_CLin = 0x3C0
  };

  int nums[] = {Aout_CUout_Bout_CLout,
                Aout_CUout_Bout_CLin,
                Aout_CUout_Bin_CLout,
                Aout_CUout_Bin_CLin,
                Aout_CUin_Bout_CLout,
                Aout_CUin_Bout_CLin,
                Aout_CUin_Bin_CLout,
                Aout_CUin_Bin_CLin,
                Ain_CUout_Bout_CLout,
                Ain_CUout_Bout_CLin,
                Ain_CUout_Bin_CLout,
                Ain_CUout_Bin_CLin,
                Ain_CUin_Bout_CLout,
                Ain_CUin_Bout_CLin,
                Ain_CUin_Bin_CLout,
                Ain_CUin_Bin_CLin,
                -1};

  for(x=0; nums[x]!=-1; x++)
  {
    printf("0x%-3X = ",nums[x]);
    for(bit=0x400; bit>0; bit>>1)
    {
      if(bit & nums[x])
        printf("1");
      else printf("0");
    }
    printf(" >>6 = %2d = ",nums[x]>>6);
    for(bit=128; bit>0; bit>>1)
    {
      if(bit & (nums[x]>>6))
        printf("1");
      else printf("0");
    }
    puts("");
  }
}
```

An array of integers was constructed out of the enumeration constants. Notice the lack of an asterisk. That's because nums[] is an array of integers, not pointers. The number -1 was added to the end to let the program know it has reached the end. The numbers are printed in their original HEX format, shifted to the right 6 places and printed in decimal. In addition, each is printed in binary format. The program provides the following output:

```
0x0   = 00000000000 >>6 =  0 = 00000000
0x40  = 00001000000 >>6 =  1 = 00000001
0x80  = 00010000000 >>6 =  2 = 00000010
0xC0  = 00011000000 >>6 =  3 = 00000011
0x100 = 00100000000 >>6 =  4 = 00000100
0x140 = 00101000000 >>6 =  5 = 00000101
0x180 = 00110000000 >>6 =  6 = 00000110
0x1C0 = 00111000000 >>6 =  7 = 00000111
0x200 = 01000000000 >>6 =  8 = 00001000
0x240 = 01001000000 >>6 =  9 = 00001001
0x280 = 01010000000 >>6 = 10 = 00001010
0x2C0 = 01011000000 >>6 = 11 = 00001011
0x300 = 01100000000 >>6 = 12 = 00001100
0x340 = 01101000000 >>6 = 13 = 00001101
0x380 = 01110000000 >>6 = 14 = 00001110
0x3C0 = 01111000000 >>6 = 15 = 00001111
```

This provides the opportunity to print the array in a somewhat different way:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
{
  int *ptr;

  enum
  {
    Aout_CUout_Bout_CLout = 0x0,
    Aout_CUout_Bout_CLin = 0x40,
    Aout_CUout_Bin_CLout = 0x80,
    Aout_CUout_Bin_CLin = 0xC0,
    Aout_CUin_Bout_CLout = 0x100,
    Aout_CUin_Bout_CLin = 0x140,
    Aout_CUin_Bin_CLout = 0x180,
    Aout_CUin_Bin_CLin = 0x1C0,
    Ain_CUout_Bout_CLout = 0x200,
    Ain_CUout_Bout_CLin = 0x240,
    Ain_CUout_Bin_CLout = 0x280,
    Ain_CUout_Bin_CLin = 0x2C0,
    Ain_CUin_Bout_CLout = 0x300,
    Ain_CUin_Bout_CLin = 0x340,
    Ain_CUin_Bin_CLout = 0x380,
    Ain_CUin_Bin_CLin = 0x3C0
  };

  int nums[] = {Aout_CUout_Bout_CLout,
                Aout_CUout_Bout_CLin,
                Aout_CUout_Bin_CLout,
                Aout_CUout_Bin_CLin,
                Aout_CUin_Bout_CLout,
                Aout_CUin_Bout_CLin,
                Aout_CUin_Bin_CLout,
```

```
                  Aout_CUin_Bin_CLin,
                  Ain_CUout_Bout_CLout,
                  Ain_CUout_Bout_CLin,
                  Ain_CUout_Bin_CLout,
                  Ain_CUout_Bin_CLin,
                  Ain_CUin_Bout_CLout,
                  Ain_CUin_Bout_CLin,
                  Ain_CUin_Bin_CLout,
                  Ain_CUin_Bin_CLin,
                  -1};

  ptr = &nums[0];

  while(*ptr != -1)
  {
    printf("value = 0x%-3X shifted right 6 places = %2d\n",*ptr,(*ptr)>>6);

    ptr++;
  }
}
```

Here, *ptr is a pointer to an integer. As noted, strings are already pointers. The integer array above however, is not. To get its first address location for the pointer, use the ampersand:
ptr = &nums[0];

When you want to see the value that a pointer points to, use the asterisk to **dereference** it, such as where the loop control says: while(*ptr != -1)

The same thing is done in the printf() function. The first item in the printf statement is the dereferenced HEX value before its shifted, and the second is the decimal shifted value. Remember, the only reason it's printed in decimal or HEX is because we tell it that's what we want by the %X and %d. Don't get confused by the HEX format command. The first "0x" part is a literal. The "%-3X" part tells printf() to print HEX using upper case letters and three characters and to fill with leading blanks, commanded by the "-" sign. It would provide trailing blanks without the minus sign. The parenthesis around the (*ptr) where it is shifted is there to protect it from precedence problems. It's not a problem here (take out the parenthesis and you will see no difference), but I do it anyway rather than trying to remember because, well, because I'm just a tad lazy.

The point I really wanted to make with all of this business has to do with the ptr++ statement. Remember what would happen to an integer if we did that -- it would increment by 1. Look at the table below, though. The table values are correct. Remember that a pointer points to a location in memory. We are incrementing prt with the prt++ statement just like we would an integer. That makes it move forward to point at another location in memory. There's no way however, that it can be incrementing by one and still point to the right places. Remember that integers in DOS are 16 bits, or *two*, not one byte wide. It's common these days for them to be 32 bits, or 4 bytes wide. The pointer must increment by 2 or 4 or more bytes, depending on the system. It can't increment by one or it won't point to the right place in memory. **Pointers increment or decrement by the size of the type of object they point to.** You don't have to worry about how much that is because the compiler takes care of it. The output of the program is below:

```
value = 0x0   shifted right 6 places =  0
value = 0x40  shifted right 6 places =  1
value = 0x80  shifted right 6 places =  2
value = 0xC0  shifted right 6 places =  3
value = 0x100 shifted right 6 places =  4
value = 0x140 shifted right 6 places =  5
value = 0x180 shifted right 6 places =  6
value = 0x1C0 shifted right 6 places =  7
value = 0x200 shifted right 6 places =  8
value = 0x240 shifted right 6 places =  9
value = 0x280 shifted right 6 places = 10
value = 0x2C0 shifted right 6 places = 11
value = 0x300 shifted right 6 places = 12
value = 0x340 shifted right 6 places = 13
value = 0x380 shifted right 6 places = 14
value = 0x3C0 shifted right 6 places = 15
```

Back to the original business. Repeating the first output:

```
0x0   = 00000000000 >>6 =  0 = 00000000
0x40  = 00001000000 >>6 =  1 = 00000001
0x80  = 00010000000 >>6 =  2 = 00000010
0xC0  = 00011000000 >>6 =  3 = 00000011
0x100 = 00100000000 >>6 =  4 = 00000100
0x140 = 00101000000 >>6 =  5 = 00000101
0x180 = 00110000000 >>6 =  6 = 00000110
0x1C0 = 00111000000 >>6 =  7 = 00000111
0x200 = 01000000000 >>6 =  8 = 00001000
0x240 = 01001000000 >>6 =  9 = 00001001
0x280 = 01010000000 >>6 = 10 = 00001010
0x2C0 = 01011000000 >>6 = 11 = 00001011
0x300 = 01100000000 >>6 = 12 = 00001100
0x340 = 01101000000 >>6 = 13 = 00001101
0x380 = 01110000000 >>6 = 14 = 00001110
0x3C0 = 01111000000 >>6 = 15 = 00001111
```

Notice how the bit patterns on the left are the same as the ones on the right if the left ones are shifted to the right 6 places and the leading zeros ignored. The ones on the right are the ones needed to properly set up the PPI. That says that the left-hand shifted numbers are correct. Notice how the binary values are formed in the program. The bit variable is put in a loop that begins with a weight value greater than what is anticipated to be the value of the number under consideration. It stays in the loop as long as it is greater than 0, and is shifted to the right one place each time around the loop. If ANDing the bit value with the number under consideration is not 0, "1" is printed, else "0" is printed. This is a handy test to use when checking to make sure you are getting correct binary values out of your operations.

Also notice that the right-hand, least-significant 6 bits of the unshifted enumeration constants are all 0. That's where the decision bits about the matrix configuration will go. A high bit will have the following meaning:

```
5 4 3 2 1 0
| | | | | |__ 1 = LS244 used as a column register
| | | | |____ 1 = C Lower used to strobe rows
| | | |______ 1 = B used as a column register
| | |________ 1 = B used to strobe rows
| |__________ 1 = C Upper used to strobe rows
|____________ 1 = Port A used to strobe rows
```

And finally, the new enumeration:

```
  enum
  {
    Aout_CUout_Bout_CLout = 0x0,
    Aout_CUout_Bout_CLout_244col = 0x1, // 00000000001 3 closures
    Aout_CUout_Bout_CLout_CLrow_244col = 0x3, // 00000000011 12 closures
    Aout_CUout_Bout_CLout_Brow_244col = 0x9, // 00000001001 24 closures
    Aout_CUout_Bout_CLout_Brow_CLrow_244col = 0xB, // 00000001011 36 closures
    Aout_CUout_Bout_CLout_CUrow_244col = 0x11, // 00000010001 12 closures
    Aout_CUout_Bout_CLout_CUrow_CLrow_244col = 0x13, // 00000010011 24 closures
    Aout_CUout_Bout_CLout_CUrow_Brow_244col = 0x19, // 00000011001 36 closures
    Aout_CUout_Bout_CLout_CUrow_Brow_CLrow_244col = 0x1B, // 00000011011 48 closures
    Aout_CUout_Bout_CLout_Arow_244col = 0x21, // 00000100001 24 closures
    Aout_CUout_Bout_CLout_Arow_CLrow_244col = 0x23, // 00000100011 36 closures
    Aout_CUout_Bout_CLout_Arow_Brow_244col = 0x29, // 00000101001 48 closures
    Aout_CUout_Bout_CLout_Arow_Brow_CLrow_244col = 0x2B, // 00000101011 60 closures
    Aout_CUout_Bout_CLout_Arow_CUrow_244col = 0x31, // 00000110001 36 closures
    Aout_CUout_Bout_CLout_Arow_CUrow_CLrow_244col = 0x33, // 00000110011 48 closures
    Aout_CUout_Bout_CLout_Arow_CUrow_Brow_244col = 0x39, // 00000111001 60 closures
    Aout_CUout_Bout_CLout_Arow_CUrow_Brow_CLrow_244col = 0x3B, // 00000111011 72
closures
    Aout_CUout_Bout_CLin = 0x40,
    Aout_CUout_Bout_CLin_244col = 0x41, // 00001000001 3 closures
    Aout_CUout_Bout_CLin_Brow_244col = 0x49, // 00001001001 24 closures
    Aout_CUout_Bout_CLin_CUrow_244col = 0x51, // 00001010001 12 closures
    Aout_CUout_Bout_CLin_CUrow_Brow_244col = 0x59, // 00001011001 36 closures
    Aout_CUout_Bout_CLin_Arow_244col = 0x61, // 00001100001 24 closures
    Aout_CUout_Bout_CLin_Arow_Brow_244col = 0x69, // 00001101001 48 closures
    Aout_CUout_Bout_CLin_Arow_CUrow_244col = 0x71, // 00001110001 36 closures
    Aout_CUout_Bout_CLin_Arow_CUrow_Brow_244col = 0x79, // 00001111001 60 closures
    Aout_CUout_Bin_CLout = 0x80,
    Aout_CUout_Bin_CLout_244col = 0x81, // 00010000001 3 closures
    Aout_CUout_Bin_CLout_CLrow_244col = 0x83, // 00010000011 12 closures
    Aout_CUout_Bin_CLout_Bcol = 0x84, // 00010000100 8 closures
    Aout_CUout_Bin_CLout_Bcol_244col = 0x85, // 00010000101 11 closures
    Aout_CUout_Bin_CLout_Bcol_CLrow = 0x86, // 00010000110 32 closures
    Aout_CUout_Bin_CLout_Bcol_CLrow_244col = 0x87, // 00010000111 44 closures
    Aout_CUout_Bin_CLout_CUrow_244col = 0x91, // 00010010001 12 closures
    Aout_CUout_Bin_CLout_CUrow_CLrow_244col = 0x93, // 00010010011 24 closures
    Aout_CUout_Bin_CLout_CUrow_Bcol = 0x94, // 00010010100 32 closures
    Aout_CUout_Bin_CLout_CUrow_Bcol_244col = 0x95, // 00010010101 44 closures
    Aout_CUout_Bin_CLout_CUrow_Bcol_CLrow = 0x96, // 00010010110 64 closures
    Aout_CUout_Bin_CLout_CUrow_Bcol_CLrow_244col = 0x97, // 00010010111 88 closures
    Aout_CUout_Bin_CLout_Arow_244col = 0xA1, // 00010100001 24 closures
    Aout_CUout_Bin_CLout_Arow_CLrow_244col = 0xA3, // 00010100011 36 closures
    Aout_CUout_Bin_CLout_Arow_Bcol = 0xA4, // 00010100100 64 closures
    Aout_CUout_Bin_CLout_Arow_Bcol_244col = 0xA5, // 00010100101 88 closures
    Aout_CUout_Bin_CLout_Arow_Bcol_CLrow = 0xA6, // 00010100110 96 closures
    Aout_CUout_Bin_CLout_Arow_Bcol_CLrow_244col = 0xA7, // 00010100111 132 closures
    Aout_CUout_Bin_CLout_Arow_CUrow_244col = 0xB1, // 00010110001 36 closures
    Aout_CUout_Bin_CLout_Arow_CUrow_CLrow_244col = 0xB3, // 00010110011 48 closures
    Aout_CUout_Bin_CLout_Arow_CUrow_Bcol = 0xB4, // 00010110100 96 closures
    Aout_CUout_Bin_CLout_Arow_CUrow_Bcol_244col = 0xB5, // 00010110101 132 closures
    Aout_CUout_Bin_CLout_Arow_CUrow_Bcol_CLrow = 0xB6, // 00010110110 128 closures
```

```
    Aout_CUout_Bin_CLout_Arow_CUrow_Bcol_CLrow_244col = 0xB7, // 00010110111 176
closures
    Aout_CUout_Bin_CLin = 0xC0,
    Aout_CUout_Bin_CLin_244col = 0xC1, // 00011000001 3 closures
    Aout_CUout_Bin_CLin_Bcol = 0xC4, // 00011000100 8 closures
    Aout_CUout_Bin_CLin_Bcol_244col = 0xC5, // 00011000101 11 closures
    Aout_CUout_Bin_CLin_CUrow_244col = 0xD1, // 00011010001 12 closures
    Aout_CUout_Bin_CLin_CUrow_Bcol = 0xD4, // 00011010100 32 closures
    Aout_CUout_Bin_CLin_CUrow_Bcol_244col = 0xD5, // 00011010101 44 closures
    Aout_CUout_Bin_CLin_Arow_244col = 0xE1, // 00011100001 24 closures
    Aout_CUout_Bin_CLin_Arow_Bcol = 0xE4, // 00011100100 64 closures
    Aout_CUout_Bin_CLin_Arow_Bcol_244col = 0xE5, // 00011100101 88 closures
    Aout_CUout_Bin_CLin_Arow_CUrow_244col = 0xF1, // 00011110001 36 closures
    Aout_CUout_Bin_CLin_Arow_CUrow_Bcol = 0xF4, // 00011110100 96 closures
    Aout_CUout_Bin_CLin_Arow_CUrow_Bcol_244col = 0xF5, // 00011110101 132 closures
    Aout_CUin_Bout_CLout = 0x100,
    Aout_CUin_Bout_CLout_244col = 0x101, // 00100000001 3 closures
    Aout_CUin_Bout_CLout_CLrow_244col = 0x103, // 00100000011 12 closures
    Aout_CUin_Bout_CLout_Brow_244col = 0x109, // 00100001001 24 closures
    Aout_CUin_Bout_CLout_Brow_CLrow_244col = 0x10B, // 00100001011 36 closures
    Aout_CUin_Bout_CLout_Arow_244col = 0x121, // 00100100001 24 closures
    Aout_CUin_Bout_CLout_Arow_CLrow_244col = 0x123, // 00100100011 36 closures
    Aout_CUin_Bout_CLout_Arow_Brow_244col = 0x129, // 00100101001 48 closures
    Aout_CUin_Bout_CLout_Arow_Brow_CLrow_244col = 0x12B, // 00100101011 60 closures
    Aout_CUin_Bout_CLin = 0x140,
    Aout_CUin_Bout_CLin_244col = 0x141, // 00101000001 3 closures
    Aout_CUin_Bout_CLin_Brow_244col = 0x149, // 00101001001 24 closures
    Aout_CUin_Bout_CLin_Arow_244col = 0x161, // 00101100001 24 closures
    Aout_CUin_Bout_CLin_Arow_Brow_244col = 0x169, // 00101101001 48 closures
    Aout_CUin_Bin_CLout = 0x180,
    Aout_CUin_Bin_CLout_244col = 0x181, // 00110000001 3 closures
    Aout_CUin_Bin_CLout_CLrow_244col = 0x183, // 00110000011 12 closures
    Aout_CUin_Bin_CLout_Bcol = 0x184, // 00110000100 8 closures
    Aout_CUin_Bin_CLout_Bcol_244col = 0x185, // 00110000101 11 closures
    Aout_CUin_Bin_CLout_Bcol_CLrow = 0x186, // 00110000110 32 closures
    Aout_CUin_Bin_CLout_Bcol_CLrow_244col = 0x187, // 00110000111 44 closures
    Aout_CUin_Bin_CLout_Arow_244col = 0x1A1, // 00110100001 24 closures
    Aout_CUin_Bin_CLout_Arow_CLrow_244col = 0x1A3, // 00110100011 36 closures
    Aout_CUin_Bin_CLout_Arow_Bcol = 0x1A4, // 00110100100 64 closures
    Aout_CUin_Bin_CLout_Arow_Bcol_244col = 0x1A5, // 00110100101 88 closures
    Aout_CUin_Bin_CLout_Arow_Bcol_CLrow = 0x1A6, // 00110100110 96 closures
    Aout_CUin_Bin_CLout_Arow_Bcol_CLrow_244col = 0x1A7, // 00110100111 132 closures
    Aout_CUin_Bin_CLin = 0x1C0,
    Aout_CUin_Bin_CLin_244col = 0x1C1, // 00111000001 3 closures
    Aout_CUin_Bin_CLin_Bcol = 0x1C4, // 00111000100 8 closures
    Aout_CUin_Bin_CLin_Bcol_244col = 0x1C5, // 00111000101 11 closures
    Aout_CUin_Bin_CLin_Arow_244col = 0x1E1, // 00111100001 24 closures
    Aout_CUin_Bin_CLin_Arow_Bcol = 0x1E4, // 00111100100 64 closures
    Aout_CUin_Bin_CLin_Arow_Bcol_244col = 0x1E5, // 00111100101 88 closures
    Ain_CUout_Bout_CLout = 0x200,
    Ain_CUout_Bout_CLout_244col = 0x201, // 01000000001 3 closures
    Ain_CUout_Bout_CLout_CLrow_244col = 0x203, // 01000000011 12 closures
    Ain_CUout_Bout_CLout_Brow_244col = 0x209, // 01000001001 24 closures
    Ain_CUout_Bout_CLout_Brow_CLrow_244col = 0x20B, // 01000001011 36 closures
    Ain_CUout_Bout_CLout_CUrow_244col = 0x211, // 01000010001 12 closures
    Ain_CUout_Bout_CLout_CUrow_CLrow_244col = 0x213, // 01000010011 24 closures
```

```
  Ain_CUout_Bout_CLout_CUrow_Brow_244col = 0x219, // 01000011001 36 closures
  Ain_CUout_Bout_CLout_CUrow_Brow_CLrow_244col = 0x21B, // 01000011011 48 closures
  Ain_CUout_Bout_CLin = 0x240,
  Ain_CUout_Bout_CLin_244col = 0x241, // 01001000001 3 closures
  Ain_CUout_Bout_CLin_Brow_244col = 0x249, // 01001001001 24 closures
  Ain_CUout_Bout_CLin_CUrow_244col = 0x251, // 01001010001 12 closures
  Ain_CUout_Bout_CLin_CUrow_Brow_244col = 0x259, // 01001011001 36 closures
  Ain_CUout_Bin_CLout = 0x280,
  Ain_CUout_Bin_CLout_244col = 0x281, // 01010000001 3 closures
  Ain_CUout_Bin_CLout_CLrow_244col = 0x283, // 01010000011 12 closures
  Ain_CUout_Bin_CLout_Bcol = 0x284, // 01010000100 8 closures
  Ain_CUout_Bin_CLout_Bcol_244col = 0x285, // 01010000101 11 closures
  Ain_CUout_Bin_CLout_Bcol_CLrow = 0x286, // 01010000110 32 closures
  Ain_CUout_Bin_CLout_Bcol_CLrow_244col = 0x287, // 01010000111 44 closures
  Ain_CUout_Bin_CLout_CUrow_244col = 0x291, // 01010010001 12 closures
  Ain_CUout_Bin_CLout_CUrow_CLrow_244col = 0x293, // 01010010011 24 closures
  Ain_CUout_Bin_CLout_CUrow_Bcol = 0x294, // 01010010100 32 closures
  Ain_CUout_Bin_CLout_CUrow_Bcol_244col = 0x295, // 01010010101 44 closures
  Ain_CUout_Bin_CLout_CUrow_Bcol_CLrow = 0x296, // 01010010110 64 closures
  Ain_CUout_Bin_CLout_CUrow_Bcol_CLrow_244col = 0x297, // 01010010111 88 closures
  Ain_CUout_Bin_CLin = 0x2C0,
  Ain_CUout_Bin_CLin_244col = 0x2C1, // 01011000001 3 closures
  Ain_CUout_Bin_CLin_Bcol = 0x2C4, // 01011000100 8 closures
  Ain_CUout_Bin_CLin_Bcol_244col = 0x2C5, // 01011000101 11 closures
  Ain_CUout_Bin_CLin_CUrow_244col = 0x2D1, // 01011010001 12 closures
  Ain_CUout_Bin_CLin_CUrow_Bcol = 0x2D4, // 01011010100 32 closures
  Ain_CUout_Bin_CLin_CUrow_Bcol_244col = 0x2D5, // 01011010101 44 closures
  Ain_CUin_Bout_CLout = 0x300,
  Ain_CUin_Bout_CLout_244col = 0x301, // 01100000001 3 closures
  Ain_CUin_Bout_CLout_CLrow_244col = 0x303, // 01100000011 12 closures
  Ain_CUin_Bout_CLout_Brow_244col = 0x309, // 01100001001 24 closures
  Ain_CUin_Bout_CLout_Brow_CLrow_244col = 0x30B, // 01100001011 36 closures
  Ain_CUin_Bout_CLin = 0x340,
  Ain_CUin_Bout_CLin_244col = 0x341, // 01101000001 3 closures
  Ain_CUin_Bout_CLin_Brow_244col = 0x349, // 01101001001 24 closures
  Ain_CUin_Bin_CLout = 0x380,
  Ain_CUin_Bin_CLout_244col = 0x381, // 01110000001 3 closures
  Ain_CUin_Bin_CLout_CLrow_244col = 0x383, // 01110000011 12 closures
  Ain_CUin_Bin_CLout_Bcol = 0x384, // 01110000100 8 closures
  Ain_CUin_Bin_CLout_Bcol_244col = 0x385, // 01110000101 11 closures
  Ain_CUin_Bin_CLout_Bcol_CLrow = 0x386, // 01110000110 32 closures
  Ain_CUin_Bin_CLout_Bcol_CLrow_244col = 0x387, // 01110000111 44 closures
  Ain_CUin_Bin_CLin = 0x3C0,
  Ain_CUin_Bin_CLin_244col = 0x3C1, // 01111000001 3 closures
  Ain_CUin_Bin_CLin_Bcol = 0x3C4, // 01111000100 8 closures
  Ain_CUin_Bin_CLin_Bcol_244col = 0x3C5, // 01111000101 11 closures
};
```

Note that #defines could have been used here just as well. There is no great advantage with an enumeration since the values are not sequential. It's used as much for neatness as anything.

This enumeration provides more information than most. It even has comments showing the bit pattern and the number of possible closures that could be used with each of its constants. There are no comments if no closure possibilities are provided. There are 151 possible variations with the PPI plus closure settings.

Notice that there is a comma at the end of the last enumeration (Ain_CUin_Bin_CLin_Bcol_244col). It does no harm, but it's

there for a reason. It's there because there is no way I would type in such a long thing if I could come up with a program to do it for me. In fact, all of the enumerations you have seen thus far have been generated by programs. I didn't bother to add the code to take out the last comma since there is no need to and it's easier to take it out with the editor than it is to add the code.

That's the contest. Just do what I did. Write a program that will generate the above enumeration using only logic. That means it's not legal to simply copy the above and put it in a string array or file! The prize is a free bare board or a $20 rebate if the winner has already purchased a board, a kit with a board or an assembled board (If you wish, I'll send you another board rather than the $20 if you already have a board -- those who have one let me know).

Here are the rules:

- The entry must be in the form of a single C-language source file (".c" extension).
- The source code file must be attached to an email and e-mailed to me.
- No string arrays, files or any other form of pre-formulated full-length string arrangements may be used. Short strings up to 10 characters may be used to assemble longer ones.
- Neither the target enumeration nor any form of it, such as an array or a file, may be used in the program.
- The prize will be awarded to the person who sends me the shortest source code file according to the count program that can be downloaded below, and who follows all other rules of the contest. The winner will be declared when no one sends a shorter entry for 2 weeks.
- The source file must include no C functions that are not included in the sample code below. That limits it to the following:
  - puts()
  - for() -- while() and do -- while() also allowed
  - switch()
  - if()
  - strcat() -- sprintf() also allowed
  - printf()
- All important variable names must be long enough to describe what they are for. Use the sample code below as an example.
- The body of each email must include the length of the programmer's programming experience in years and weeks (for example, 0 years 8 weeks). Preference will be given to the person with the least experience if there is a tie. All I have here is your honesty. Some long-time programmer who could't care less can cheat, but I hope not.
- I will publish the winning entry with the name of the programmer if given permission to do so. Please don't forget to include permission if you want your name published.

I will post the shortest count according to the count program so everyone can get an idea of their status. The contest will end when no one can produce a shorter file for a period of one week.

Please use your name to make up the name of your program so I can keep up with the entries. Mine, for example, might be called joereedr.c. Since count is a DOS program, it can work with file names limited to eight characters, less the ".c" extension. Use as many characters as possible to make the name unique, but no more than eight.

To use count, just type in
count myprogrm.c
then press the enter key, where myprogrm.c is the name of your source code file.

You can download count here: count.exe
If you like, you can download the source here: count.c

My source showed 1990 bytes with the count program. The count program counts only printable characters. It does not count blanks, tabs, etc. It also does not count comments that begin with "//." Thus, it does no good to crowd. Put in all of the white space you wish. Include all of the "//" comments you wish. I took all of the comments and all of the blank spaces out of my version and still got 1990 bytes. Any source that has obviously been packed in an attempt to make it shorter will be rejected. Trying to use short variable names that don't say what they are will get the same response. Reminds me of a guy I once knew who named his variables a, b, c, etc. then started with double letters when he ran out of alphabet. His subroutines went sub1, sub2, sub3 etc. As a result, no one who followed him could figure out what he was doing, not even him. Not overly smart.

Now for some clues. The following sample code is the program used to build the basic enumeration. Notice the "#pragma warning" statement. It might not be provided with your compiler, but something like it probably is. Look for the means to cause the compiler to provide you with the maximum level of warnings. The maximum warning level will cause the compiler to tell you some of the variables are declared but not used. In fact, the ones you will see listed are used in the program that generates the new enumeration, at least in my version. For example, the total variable is used to calculate the total switch closures available used in the comments at the end of the lines. And yes, the comments and binary representations are also generated by the program -- you have not finished until your program output looks like the big enumeration above. To make your program send its output to a file, do this:
myprog > outfile
then press enter. Myprog is your program, and outfile is the name of the file you want the output to go to. If your computer is connected to a local printer try this to send the output there:
myprog > lpt1

Here's the sample code:

```
// Sample Code

#include <stdio.h>
#include <conio.h>
#include <string.h>

#pragma warning

void main()
{
  int rowcount,colcount,total,matbits;
  int matbithistory[600],matbitcount=0,d;
  int ppi,matrixtest,bit,finalval;
  char PPIString[300], MatrixString[300];
  char temp[100],temp2[100];

  puts("  enum\n  {");

  for(ppi=0; ppi<16; ppi++)
  {
    PPIString[0] = 0;
    matbits = 0;

    for(bit=8; bit>0; bit/=2)
    {
      switch(bit)
      {
        case 8:
        if(!(bit & ppi))
          strcat(PPIString,"Aout");
        else if((bit & ppi))
          strcat(PPIString,"Ain");
        break;

        case 4:
        if(!(bit & ppi))
          strcat(PPIString,"_CUout");
        else if((bit & ppi))
          strcat(PPIString,"_CUin");
        break;
```

```
        case 2:
        if(!(bit & ppi))
          strcat(PPIString,"_Bout");
        else if((bit & ppi))
          strcat(PPIString,"_Bin");
        break;

        case 1:
        if(!(bit & ppi))
          strcat(PPIString,"_CLout");
        else if((bit & ppi))
          strcat(PPIString,"_CLin");
        break;

      } // end switch

    } // end for(bit..)

    printf("    %s = 0x%X,\n",PPIString,ppi<<6);

    // Hint: new code might go here -- could be
    // there's another for() loop -- maybe more
    // and another switch() -- maybe more

  } // end for(ppi=0; ppi<16; ppi++)

  puts("  };");
}
```

The matbits variable stores the 6 matrix bits noted above. A little needs to be said about matbithistory and matbitcount. They are used to avoid duplications. A check of the history is made before a line is printed. Consider the following code fragments from my program for the big enumeration:

```
 for(d=0; d<matbitcount; d++)
 {
   if(finalval == matbithistory[d])
     break;
 }

 if(d == matbitcount)
 {
   // other really important stuff goes here
 }
```

and:

```
if(matbitcount<599)
  matbithistory[matbitcount++] = finalval;

else
{
  printf("matbitcount = %d\n",matbitcount);
    exit(0);
}
```

In the first fragment, the d loop runs until it is equal to matbitcount. Thus, d will not be equal to matbitcount if a value in the history is found to be the same as the finalval variable, because the loop will be broken out of before it gets there.

In the second fragment, the final value is added to the history if matbitcount is less than 599, which is the highest storage index number that can be used in the array. The history array was declared to be be 600 integers long, which means array slots 0 through 599 can be used. Since going beyond that can cause problems, the program exits if the count is about to go over 599. The exit is preceded by printing the count for debugging purposes.

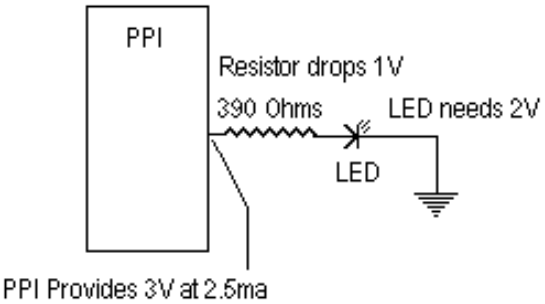
Notice what is done with the strings. For example, take note of the line, "PPIString[0] = 0;". Recall from above that a 0 terminates a string. Here it is terminated at its first character location, effectively making it appear to be empty. It can now be build it up by concatenating other strings to it. That's what strcat(...) does.

That's about it for the clues. All that's left now is for you to attach your entry to an e-mail and fire it off to me.

We are now going to see how to use the PPI to drive devices requiring more power than the rows on a matrix. If you look at 8255A.PDF you will see the capabilities of the most common 82C55 version. What we will be concerned with will be the logical one and zero output voltages and currents. The following is part of the information from the data sheet:

	LIMITS			
PARAMETER	**MIN**	**MAX**	**UNITS**	**TEST CONDITIONS**
Logical One Output Voltage	3.0		V	I_{OH} = -2.5ma
Logical Zero Output Voltage		.4	V	I_{OL} = +2.5ma

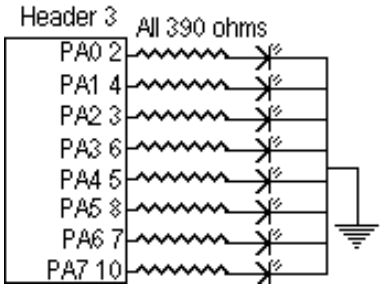
Since the flow of charge is from negative to positive, it is shown as a positive flow when the output is low, and a negative flow (flowing in) when the output is high. We will call it 2.5ma regardless of direction. Let's try something simple first -- turning on a **light emitting diode** (LED). LEDs normally need about 20ma at about 2V. Remember from How To Read A Schematic that V = I * R (Ohm's Law), and that the other relationships can be developed from this one. We find from the above table that the PPI can give us 3V at 2.5ma. We are going to limit the current with a resistor. The LED looks like a diode with lines added to indicate it provides light: 



The PPI provides 3V so the resistor needs to drop 1 volt at 2.5ma because the LED needs 2 volts. Just plug the numbers into the Ohm's Law formula for resistance that you saw in How To Read A Schematic:
R = V/I
R = 1 / 2.5ma
R = 1 / .0025 A
R = 400

The closest standard value to this is 390 ohms. The power capability requirement of the resistor must be determined in addition to its value. Power is equal to the voltage across the resistor times the current. Before the power can be calculated, the voltage drop must be recalculated once the closest standard value is found:
V = I * R = 2.5ma * 390 = .975V
P = I * V = 2.5ma * .975V = 2.4375 milliwatts

Any standard power will work -- 1/10, 1/8, 1/4, 1/2 watts, etc. Eight LEDs can be connected to Port A through header 3 in the following manner:



One short note about voltage and power. Notice what the power equation says:
P = I * V
Remember from Ohm's Law (see How To Read A Schematic) that I = V/R. It's perfectly OK to replace the I in the power equation with V/R since they are the same thing:
P = V/R * V
That says that $P = V^2/R$
If you have seen the ads about stuff running on 3.3V and have wondered what the big deal is, you have just seen the reason. Power varies with the square of voltage. If a laptop looks like 10 ohms to its batteries and runs on 5V, it takes 2.5 watts to run it. If it runs on 3.3 volts and still looks like 10 ohms, it only takes 1.089 watts -- less than half as much power for only 1.7 volts difference.

The following subroutine can be added to the end of digital.c to see if the lights will work. It accepts on-time and off-time values as its arguments. They are longs, which means they are 32 bits in DOS and therefore capable of accepting very large values. Notice that the on and off timers, y and z, are also longs so they can be used with the arguments. Notice their loops. They start with "0L," which means they start with a long zero. All the loops do is waste some time, so they end on the same line they start on, as indicated by the semicolon at the end of the line. The outside x loop goes through all eight of the bits by starting at 0x80 and shifting to the right once each time around. The x value is place in Port A, turning on the appropriate LED. The on time loop is then run, then a 0 placed in Port A to turn off all of the LEDs. The off time loop is run, then x moves to the next bit. The second x loop goes the other direction. It starts at bit 1 since the first loop ended on bit 0 and ends on bit 6 since the first loop starts on bit 7. The result is that the LED lighting moves back and forth:

```
void blinker(long on, long off)
{
  int x;
  long y,z;

  for(x=0x80; x>0; x>>=1)
  {
    outp(ppi_porta, x);
    printf("%4X",x);
    for(y=0L; y<on; y++);

    outp(ppi_porta, 0);
    for(z=0L; z<off; z++);
  }

  for(x=2; x<0x80; x<<=1)
  {
    outp(ppi_porta, x);
    for(y=0L; y<on; y++);

    outp(ppi_porta, 0);
    for(z=0L; z<off; z++);
  }
}

// end digital.c
```

The following program can be used to test the subroutine:

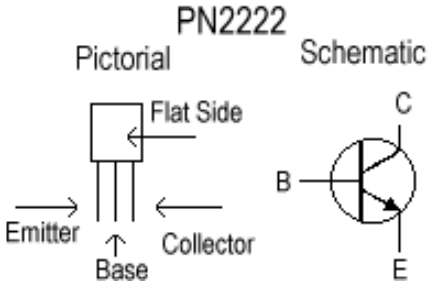
```
// experi4b.c

#include <conio.h>
#include <stdio.h>
#include <bios.h>

// include header with constants
#include "constant.h"

// external prototypes
extern void set_up_ppi(int mode);
extern void get_port(void);
extern int is_closure(int closurenumber);
extern void blinker(long on, long off);

void main(void)
{
  int x,y,r,c;

  get_port(); // get the port number and establish register locations

  // make A an output and B an input
  set_up_ppi(AOUT_CUPPERIN_BIN_CLOWERIN);

  while(1) // stay in loop forever
  {
    // "keyboard hit" is a function that checks
    // to see if a key has been pressed.
    if(kbhit())
      break;// A key was pressed -- break out of the while(1) loop

    blinker(5000L, 5000L);

  } // end while(1)

} // end experi4b.c
```

Notice that the arguments sent to blinker have L on the end to show that they are longs. You might need to experiment with different on and off times to get a reasonable display, since your computer and mine probably run at different speeds. The only reason the example uses 5000 is that it displays OK on my test machine.

Delay loops such as the ones in the subroutine should never be used where a specific delay is required. While they can sometimes be adjusted for good performance on one machine, they will often not work on another maching due to differences in speeds. In addition, different compilers might use different methods to construct loops, thereby influencing the timing. No problem here, since we are testing the circuit and are concerned about timing only to the extent that it allows us to see if the circuit works. It should if properly wired.

It will work, but it won't produce the maximum brightness out of the LEDs. Recall from above that the LEDs are rated at 20ma and 2V. That's typical. I used the Radio Shack #276-1622 variety pack and neither the voltage nor the current are specified. 20ma and 2V will be used for this discussion since they are typical values. The PPI will provide only 2.5ma, although it does give us at least 3V at that current. Mine actually provided closer to 4V. What we need is a way to increase the current. Turns out that's what a **transistor** does. It's a current amplifier. Its current gain is designated by its **hfe** parameter.

The PN2222 is a good example. It looks like a small, solid, plastic cylinder that's flat on one side. That shape is commonly designated a TO-92 case:



It's an NPN type, which means its collector is normally more positive than its emitter. The type can be determined by the emitter arrow in the schematic. It points out of the trasistor for an NPN such as the PN2222. It points inward for the PNP type. A PNP normally has its collector more negative than its emitter.

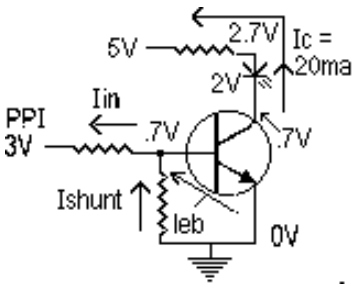
There are, of course, limits to what a transistor can handle. The following table shows part of the specifications for the PN2222:

Case	Type	BVcbo	BVceo	BVebo	Icma	hfe
TO-92	NPN	75	40	6.0	1000	50

The parameters prefixed with BV designate breakdown voltages. Each is specified with a disconnected or open base, indicated by the o. The other letters indicate the terminals being specified. Thus, cb means collector to base, ce collector to emitter and eb emitter to base. The table says that the collector to base voltage cannot exceed 75 volts, the collector to emitter voltage cannot exceed 40 volts, and the emitter to base voltage cannot exceed 6 volts. The maximum current through the collector is Icma. It's 1000ma or 1 amp (I don't care what it says, I wouldn't go over 100ma). The specs vary among the people who make them. The above is from Fairchild (see PN2222.PDF).

The gain, hfe, is 50. It is equal to Ic/Ib, or the current in the collector divided by the current in the base. That means the collector will provide 50 times the current going through the base, which is to say the transitor provides current amplification. The hfe actually varies according to several factors, but 50 is a pretty good ballpark for what we want to do here.

The PPI can supply 2.5ma, but will be required to produce only 1.25ma as insurance. The circuit will look like this:



This might look complicated, but it's not if taken one step at a time. The emitter is easy. It will be connected to ground in this experiment, so it is at 0V. It will carry the current of both the base and the collector. Let's look at the collector first. We know that the collector current, Ic, needs to be 20ma in order to power the LED. There is about .7V at the collector because the collector-emitter circuit looks like a silicon diode. There is 2V across the LED. The collector voltage adds to that, so there is 2.7V at the junction of the LED and the resistor connected to it. Points such as this are often termed to be "raised above ground" by the voltages imposed by the various devices. We are trying for a current of 20ma. The resistor is used to provide that limit. If it were not there, the LED and transistor could very likely go up in smoke. Since there is 5V at one end of the resistor and 2.7V at the other end, there is a 2.3V drop across the resistor. Since the voltage and current are known, the value of the resistor can be determined using Ohm's Law:
R = V/I = 2.3/20ma = 2.3/.02A = 115 ohms
The standard value of 120 ohms is close enough.

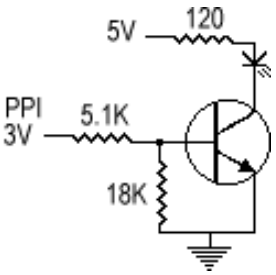
Now re-figure the current with the standard resistor value. We still have 2.3V across it, so now
I = V/R = 2.3/120 = .019166666 A
Call it .019167 amps

With an hfe current gain of 50, the base to emitter current, Ieb, needs to be
.019167/50 = .38334ma = .00038334A

The resistor from the base to ground is there to keep the transistor from picking up noise. It provides Ishunt, and its value is going to be based on the required base current. Ishunt can be about 1/10th of the base current, or .000038334A. Since there is also about .7V from the base to the emitter, the shunt resistor is
R = V/I = .7/.000038334A = 18260.55199 ohms
The standard value of 18000 ohms should work just fine.

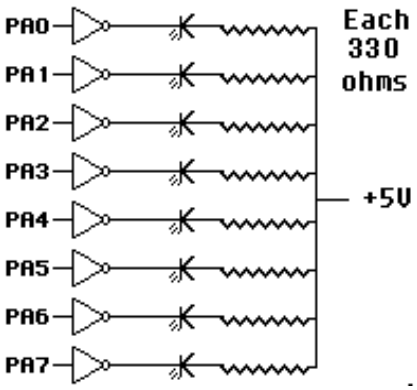
The input current needs to be the sum of the required base to emitter current of .00038334A plus the shunt current. The drop across the input resistor will be the 3V input less the .7V base to emitter voltage, or 2.3V. The input resistance is then:
R = V/I = 2.3/(.00038334A + .000038334A) = 2.3/.000421674
= 5454.45 = close to the standard 5.1K value

Here's the result:

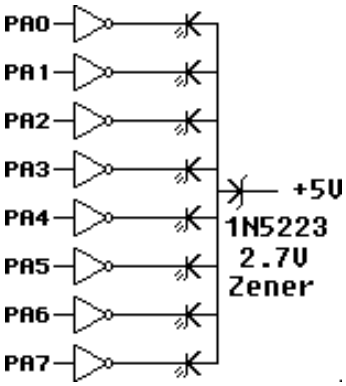


Another solution is to use the 74LS05. It provides almost everything 6 PN2222s provide, and needs only one resistor per output. Its outputs are collectors with nothing connected to them. Thus, they are commonly called **open collectors**. The emitters and bases are connected internally, so they are of no concern. The 74LS05's only limitation is that it can sink only 8ma of current (**sink** is the term used when an open collector such as this provides current). It has .5V at its collector when sinking 8ma (see 74LS05.PDF), so the drop across the resistor is

5V - (2V + .5V) = 2.5V (remember, the LED and collector voltages add)
The resistor is then
R = V/I = 2.5/8ma = 312.5 ohms -- 330 ohms should work fine:



Here is another trick you might try:



The diodes used in the matrix discussed earlier could become reverse biased and not conduct. That is true up to a point. In the real world they will conduct if their **breakdown voltage** is reached. The zener diode, designated by the bend in the line at the top of the arrow, exhibits a predictable breakdown voltage. It's 2.7V in this case. At and above 2.7V, it will always exhibit a constant drop of about 2.7V plus or minus its percent of tolerance. Since the power supply is providing 5V, with the .5V collector voltage of the 74LS05 added, the LEDs get about 1.8V, which is close enough to the specified 2V. The beauty of it is that you only need one zener for all eight LEDs. I have also tested the 1N4728 3.3V zeners. They work in the circuit and are easier to find.

Try some of these circuits and see how they work. You might consider the LEDs bright enough when powered only by the PPI. Also take note of the fact that there is nothing saying you have to use Port A. Maybe you only need 4 LEDs and could do with one of the Port C halves. Just be sure to use the resistors or a zener to limit the current or voltage. **Don't, and you are almost certain to take out the PPI, maybe the board, maybe your computer!**

Control systems of all types usually involve some sort of loop from input to output. Loops range from basic indication of closure condition to the use of complex analysis procedures before output is initiated. Add the following to digital.c:

```
void btoa(void)
{
  outp(ppi_porta, ~inp(ppi_portb));
}
```

This routine simply takes what is in Port B, inverts it and puts it in Port A. Recall from the boolean logic section what the "~" does. It takes all of the bits and inverts them. Thus, 01110011 becomes 10001100. When a closure takes a Port B line low, it turns off the bit associated with the line. To turn on the LEDs in Port A however, a bit needs to be turned on.

The routine can be tested with one of the above LED circuits as an output and the following test program after experi4c and digital are compiled and linked. The LED that is connected to Port A should turn on when the corresponding Port B line is taken low:

```
// experi4c.c

#include <conio.h>
#include <stdio.h>
#include <bios.h>

// include header with constants
#include "constant.h"

// external prototypes
extern void set_up_ppi(int mode);
extern void get_port(void);
void btoa(void);

void main(void)
{
  int x,y,r,c;

  get_port(); // get the port number and establish register locations

  // make A an output and B an input
  set_up_ppi(AOUT_CUPPERIN_BIN_CLOWERIN);

  while(1) // stay in loop forever
  {
    // "keyboard hit" is a function that checks
```

```
    // to see if a key has been pressed.
    if(kbhit())
      break;// A key was pressed -- break out of the while(1) loop

    btoa(); // get Port B, invert it and put it in Port A

  } // end while(1)

} // end experi4c.c
```

Previous: Experiment 3 - The General Purpose Digital Input/Output Module - Part 1
Next: Experiment 5 - Controlling Motors
Problems, comments, ideas? Please e-mail me

Order | Home