Controlling The Real World With Computers

Experiment 3 - The General Purpose Digital Input/Output Module - Part 1

Home | Order | Let me know what you think -- e-mail

Previous: Experiment 2 - Expanding Switch Input Detection
Next: Experiment 4 - The Multiple Closure Problem And Basic Outputs With The PPI

The general purpose digital input/output module is not a box covered with switches, lights and terminal blocks. It is not a piece of hardware at all, but a piece of software. It's called a module because its code is encapsulated in a file which will contain all of the functions and variables related to digital input and output.

It is very useful to isolate functionality in this manner when building a software project. Modulariazation makes it much easier to test and troubleshoot functions. It's a lot easier to find and fix a problem when all activities related to a particular process are to be found in one file dedicated to the process, rather than being scattered all over the place and mixed in with other processes. Modulariazation is made possible in large part by something called **file scope**. You have already read about scope in Experiment 1 and Experiment 2.

Scope was discusssed in those sections as it relates to position in the file. Remember from Experiment 2 that several variables were declared just above the is_switch() function in the file. They were known only to the functions that followed them. They could not be modified or even accessed by anything above their point of declaration. It follows that variables declared above all of the functions in a file are visible to all of the functions in the file. They are said to be **global** to the file. It is important to note however, that their scope is normally limited to the file in which they are declared. Under normal circumstances, variables are not known to functions outside the file in which they are declared. That means they are protected from being messed with by functions outside their file. It is this **encapsulation** that makes it possible to produce file-based modules that permit access to variables only through the functions declared to be access portals. That makes design and troubleshooting much easier.

Design is made easier by the fact that, once a module has been thoroughly tested and debugged, there is usually no need to worry about it again because access is limited to the tested functions. Troubleshooting is made easier by the fact that a problem can often be recognized by the characteristics it displays. For example, a program might say switch 3 is turned on when, in fact, switch 1 is turned on. It is much easier to find the source of the problem if everything having to do with switches is in one file. It is useful to view a file as an object that encapsulates a set of funtions related to one particular aspect of a project. In this experiment, the file will become an object that deals with switches. Always work with Object Oriented Programming (OOP) as a goal. C++ is designed to make OOP easier, but a lot can be done with C. After all, C++ was first implemented using a front end processor (called cfront) to turn source code into new source code that could be compiled by a standard C compiler.

The general purpose digital input/output object will be encapsulated in a file named after its purpose -- digital.c. It will be the module used to house all of the switch input procedures covered so far, plus some additional procedures that will increase the capability to permit up to 176 switches to be detected. After digital inputs will come digital outputs.

Incidentally, contact closure detection is a better, more general term than switch detection. Contact closure detection is heavily used in control and embedded systems. There is always a need to know when things pass or reach a point, have taken too long to do so, never get there at all or get there too soon. Other activities are often triggered by such information. More often than not, the information is provided as a contact or switch closure, or at least a signal that looks like one. It might actually be the detection of a light beam or a magnet or a piece of metal, but it is almost always a form of contact or contact simulation.

The 8255 Programmable Peripheral Interface (PPI -- 8255.PDF) discussed in Experiment 2 will be the main contributor to both digital input and output. The table showing how to set up the control register to configure Ports A, B and C as inputs and outputs is repeated below:

	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

Mode 0 will be used in this experiment. It would be nice to use plain English when setting up the PPI rather than HEX codes. That will be the first order of business. Notice that D7 is always high to set mode, and D6, D5 and D2 are always low for mode 0. D0, D1, D3 and D4 are the only bits that change. Notice how they work. If a bit is off, its section is an output. If a bit is on, its section is an input. D0, D1, D3 and D4 can be seen as bits of a counter just like the binary table from the Data lines section repeated below, providing we ignore the weighting normally assigned to D3 and D4:

Binary Bits D4 D3 D1 D0	Decimal	Hexadecimal	Binary Bits D4 D3 D1 D0	Decimal	Hexadecimal
0000	00	0	1000	08	8
0001	01	1	1001	09	9
0010	02	2	1010	10	A
0011	03	3	1011	11	B
0100	04	4	1100	12	C
0101	05	5	1101	13	D
0110	06	6	1110	14	E
0111	07	7	1111	15	F

A constant could be defined that says all of the ports are outputs by making it a 0 and calling it something that says to set up all four groups as outputs. Something like AOUT_CUPPEROUT_BOUT_CLOWEROUT would work. That's fairly close to English, and easier to understand than HEX. The #define directive could be used to set up a whole series of such constants:

#define AOUT_CUPPEROUT_BOUT_CLOWEROUT 0
Next would be
#define AOUT_CUPPEROUT_BOUT_CLOWERIN 1

And, at the end of the sequence,
#define AOUT_CUPPERIN_BIN_CLOWERIN 14
and
#define AIN_CUPPERIN_BIN_CLOWERIN 15

It would be nice however, if a way could be found to set up the constants without having to type in all of the #defines. There is. It's called an **enumeration**, and it looks like this:

```
enum
{
  A,
  B,
  C,
  D,
  E
};
```

Think of an enumeration as a series of constants for which the compiler automatically assigns integer values. The first constant is always given the value 0 unless it is explicitly set to something else. All constants that follow are 1 greater than the previous one. Thus, A = 0, B = 1, C = 2, D = 3 and E = 4 in the above example. The constants may be called anything as long as they are legal C language names. The numbering can also be changed in mid-stream:

```
enum
{
  A,
```

```
  B,
  C = 7,
  D,
  E
};
```

Now, A = 0, B = 1, C = 7, D = 8 and E = 9.

The following enumeration establishes the constants needed for the PPI:

```
enum
{
  AOUT_CUPPEROUT_BOUT_CLOWEROUT, // 0
  AOUT_CUPPEROUT_BOUT_CLOWERIN,  // 1
  AOUT_CUPPEROUT_BIN_CLOWEROUT,  // 2
  AOUT_CUPPEROUT_BIN_CLOWERIN,   // 3
  AOUT_CUPPERIN_BOUT_CLOWEROUT,  // 4
  AOUT_CUPPERIN_BOUT_CLOWERIN,   // 5
  AOUT_CUPPERIN_BIN_CLOWEROUT,   // 6
  AOUT_CUPPERIN_BIN_CLOWERIN,    // 7
  AIN_CUPPEROUT_BOUT_CLOWEROUT,  // 8
  AIN_CUPPEROUT_BOUT_CLOWERIN,   // 9
  AIN_CUPPEROUT_BIN_CLOWEROUT,   // 10
  AIN_CUPPEROUT_BIN_CLOWERIN,    // 11
  AIN_CUPPERIN_BOUT_CLOWEROUT,   // 12
  AIN_CUPPERIN_BOUT_CLOWERIN,    // 13
  AIN_CUPPERIN_BIN_CLOWEROUT,    // 14
  AIN_CUPPERIN_BIN_CLOWERIN      // 15
};
```

AOUT_CUPPEROUT_BOUT_CLOWEROUT will be automatically set to 0, AOUT_CUPPEROUT_BOUT_CLOWERIN to 1 and so on down to AIN_CUPPERIN_BIN_CLOWERIN which will be set to 15. The set_up_ppi() function can now be modified to accept one of the constants to configure the PPI. Since enumerations are seen as integers, set_up_ppi() will be modified to accept an integer as an argument. Now, for example, to set up A as an input, C Upper as an output, B as an output and C Lower as an input, set_up_ppi() is sent AIN_CUPPEROUT_BOUT_CLOWERIN.

The enumeration gives us 0 through 15. That's 0000 through 1111 in binary. A comparison of the binary sequence table and the 8255 set up code table will reveal that bits D0 and D1 in the enumeration are the same bits needed in the control code to properly set up the lower nibble of Port C and all of Port B. Bits D2 and D3 are the correct values needed to set up the upper nibble of Port C and all of Port A, but need to be moved to bit positions D3 and D4. For example, consider AOUT_CUPPERIN_BOUT_CLOWERIN. It's 5, or binary 0101.
D0 = 1 makes C Lower an input
D1 = 0 makes B an output
D2 = 1 makes C Upper an input if moved to the D3 position
D3 = 0 makes A an output if moved to the D4 position

We need to leave D0 and D1 alone and move D2 and D3 to positions D4 and D5. We need to turn on D7 to put the PPI in set up mode. All other bits should be turned off. Now look at the new set_up_ppi():

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

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

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

The control variable is still assigned the value base + 0x23. The **command** variable has been added, and there is now a **mode** argument being sent to the function. It's important to note that the mode variable being sent to set_up_ppi() **is not modified** by procedures inside set_up_ppi(). It would not be modified even if there were a statement that looked as though it could modify it, such as "mode = 5;". That's because a copy is made of an argument that is **passed by value**, as is mode. The copy is used in the function, not the original. **It is a common error to pass an argument by value to a function thinking it will be modified, only to find nothing happens.**

ANDing mode with 0x0c applies a **mask**, as explained in Experiment 1, that zeros out everything but bits 2 and 3 of the mode value (8 + 4 = 12 = 0x0c). Not even the copy of the mode variable is changed by such an operation however, because the compiler will write code to perform the operation and place the results in a temporary location in the microprocessor or in memory. The value from the AND operation is shifted to the left 1 bit position (<< 1) which moves bits 2 and 3 into bit positions 3 and 4. The results of the combination of the AND and shift operations is stored in the command variable.

Notice that "mode & 0x0c" is surrounded by parenthesis. That's because the bit-wise AND (**&**) operator has a lower **precedence** than the bit-wise shift (<<) operator. Operations with a higher precedence will be performed before those with a lower precedence unless the lower precedence operation is surrounded by parenthesis. The parenthesis forces the AND to take place before the shift. If the statement had been
command = mode & 0x0c << 1;
then 0x0c would have been shifted to the left one bit, the result ANDed with mode then stored in the command variable.

Another thing to consider is **associativity**. What is the answer to the following?
2 - 3 - 4 = ?
It's -5 because subtraction is left-associative. If it were right-associative, 2 - 3 - 4 would equal 2 - (3 - 4), or 3. Just about everything is left-associative. (thanks to Ambiguities and Lexical Analysis By Clifford Story).

The following table from The University of Washington Department of Chemistry lists the C language operators in order of precedence and shows the direction of associativity for each operator. The operators at the top of the list have highest precedence. The comma operator has the lowest precedence. Operators that appear in the same group have the same precedence. Don't be concerned if you don't know all of the operators yet. We'll get there. Of course you know most of them if you have read through all of the sections prior to this one. C++ operators have been left out to keep down the confusion.

Operator Name	**Associativity**	**Operators**
Primary	left to right	( ) [ ] . ->
Unary	right to left	++ -- + - ! ~ & * ( )
Multiplicative	left to right	* / %
Additive	left to right	+ -
Bitwise Shift	left to right	<< >>
Relational	left to right	< > <= >=

Equality	left to right	== !=
Bitwise AND	left to right	&
Bitwise Exclusive OR	left to right	^
Bitwise Inclusive OR	left to right	|
Logical AND	left to right	&&
Logical OR	left to right	||
Conditional	right to left	? :
Assignment	right to left	= += -= *= /= <<= >>= %= &= ^= |=
Comma	left to right	,

Back to business. After the mask and shift, the command variable is added to the mode value ANDed with 3, which causes bits D0 and D1 of the mode value to be stored, along with the shifted D2 and D3 bits. Note that "command +=" means the same thing as "command = command +".
Thus, A += 2; is the same as A = A + 2; and x+=4; is the same as x = x + 4;
Finally, command is ORed with 0x80 (using |=) to turn on bit 7 for PPI set up.

Let's look at the AOUT_CUPPERIN_BOUT_CLOWERIN example step-by-step.
AOUT_CUPPERIN_BOUT_CLOWERIN = 5 = 0101 binary.
Since an integer is being sent to set_up_ppi() as an argument, we have to make sure we deal with all 16 bits, even though we will end up using only 8. In fact, we will treat the most significant 12 bits as unknown since we know all of the information required is in the first 4 bits. X's will be used for the most significant 12 bits in the illustrations to show we consider them unknown.

XXXX XXXX XXXX 0101 mode = 5

0000 0000 0000 1100 AND with 0x0c (bits 2 and 3 on)

0000 0000 0000 0100 Result of AND

**(see the Boolean section
if you don't understand this)**

Now shift the results of the above left 1 place:

0000 0000 0000 0100 << 1 =

0000 0000 0000 1000

Hold on to this value

AND the original mode value with 3:

XXXX XXXX XXXX 0101 mode = 5

0000 0000 0000 0011 AND with 3

0000 0000 0000 0001 Result of AND with 3

Add this to the value held onto after the shift:

0000 0000 0000 1000 = 8

0000 0000 0000 0001 = 1

0000 0000 0000 1001 Result of add = 9

Finally, OR with 0x80

0000 0000 0000 1001

0000 0000 1000 0000 OR with 0X80

0000 0000 1000 1001 = 0x89, which is the correct value

But will all of the other constants work? The only way to know for sure is to build a test program. Tests consist of two halves. The **driver** is a function that sends test data to the target function. The target function is made to print or return results of its process so it can be determined if what it does is correct. The following is a driver that can be used to test set_up_ppi(). First, put the enumeration constants in a header file named constant.h:

```
// constant.h

enum
{
  AOUT_CUPPEROUT_BOUT_CLOWEROUT, // 0
  AOUT_CUPPEROUT_BOUT_CLOWERIN,  // 1
  AOUT_CUPPEROUT_BIN_CLOWEROUT,  // 2
  AOUT_CUPPEROUT_BIN_CLOWERIN,   // 3
  AOUT_CUPPERIN_BOUT_CLOWEROUT,  // 4
  AOUT_CUPPERIN_BOUT_CLOWERIN,   // 5
  AOUT_CUPPERIN_BIN_CLOWEROUT,   // 6
  AOUT_CUPPERIN_BIN_CLOWERIN,    // 7
  AIN_CUPPEROUT_BOUT_CLOWEROUT,  // 8
  AIN_CUPPEROUT_BOUT_CLOWERIN,   // 9
  AIN_CUPPEROUT_BIN_CLOWEROUT,   // 10
  AIN_CUPPEROUT_BIN_CLOWERIN,    // 11
  AIN_CUPPERIN_BOUT_CLOWEROUT,   // 12
  AIN_CUPPERIN_BOUT_CLOWERIN,    // 13
  AIN_CUPPERIN_BIN_CLOWEROUT,    // 14
  AIN_CUPPERIN_BIN_CLOWERIN      // 15
};

// end constant.h
```

Note in the following that the "extern int set_up_ppi(int mode);" prototype for set_up_ppi() reflects the fact that it is **external** to the test file. The only way the driver can access any variables in the digital.c module is through set_up_ppi() and get_port(). Place the following in experi3a.c:

```
// experi3a.c

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

// change the set_up_ppi() prototype to a void return for normal operation
extern int set_up_ppi(int mode);
extern void get_port(void);

void main(void)
{
  int x,y;

  // the following will tell us what we should get in normal operation
  char *mode_type[] = {"A = out, C Upper = out, B = out, C Lower = out",
                       "A = out, C Upper = out, B = out, C Lower = in",
                       "A = out, C Upper = out, B = in, C Lower = out",
                       "A = out, C Upper = out, B = in, C Lower = in",
                       "A = out, C Upper = in, B = out, C Lower = out",
                       "A = out, C Upper = in, B = out, C Lower = in",
                       "A = out, C Upper = in, B = in, C Lower = out",
                       "A = out, C Upper = in, B = in, C Lower = in",
                       "A = in, C Upper = out, B = out, C Lower = out",
                       "A = in, C Upper = out, B = out, C Lower = in",
                       "A = in, C Upper = out, B = in, C Lower = out",
                       "A = in, C Upper = out, B = in, C Lower = in",
                       "A = in, C Upper = in, B = out, C Lower = out",
                       "A = in, C Upper = in, B = out, C Lower = in",
                       "A = in, C Upper = in, B = in, C Lower = out",
                       "A = in, C Upper = in, B = in, C Lower = in"};

  for(x=AOUT_CUPPEROUT_BOUT_CLOWEROUT; x<=AIN_CUPPERIN_BIN_CLOWERIN; x++)
  {
    y = set_up_ppi(x);
    printf("mode = %02d command = %02X %s\n",x,y,mode_type[x]);
  }

}

// end experi3a.c
```

The "char *mode_type[]" business above provides an opportunity for a little discussion of pointers. A pointer simply points to a location in memory. The asterisk says this is a pointer. The "[]" part tells us that this is an array, which is to say it's like a list. The first item in the list is [0], the second one [1], and so on. The "char *mode_type[]" term then, tells us that this is an array of pointers to to the phrases in quotes.
Thus, mode_type[0] = "A = out, C Upper = out, B = out, C Lower = out"
and mode_type[5] = "A = out, C Upper = in, B = out, C Lower = in"

Now make the following changes to set_up_ppi() in the digital.c file. Notice that get_port() has been included at the bottom:

```
// digital.c

// The following are known only to the functions that follow them.
// 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;

// set up the ppi -- change back to a void return for normal operation
int set_up_ppi(int mode)
{
  unsigned control = base + 0x23;
  int command;

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

  // remove the following for normal operation
  return command;

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

} // end set_up_ppi(..)

// get the port -- in the future, this will grow 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;

} // end get_port()


// end digital.c
```

The return value of set_up_ppi() has been temporarily changed to an integer and the value of command is returned for purposes of testing. Both experi3a.c and digital.c are compiled individually to produce what are called object files. The two object files are then linked together by the linker to produce the executable file. The linker will know to look in other than the experi3a module for set_up_ppi() because of the extern directive.

To compile and link using POWERC, do the following:
pc digital <enter>
pc experi3a <enter>
pcl experi3a digital <enter>

Put the three commands in a batch file if you like, then change the "experi3_" part each time.

The two pc commands will compile digital and experi3a and generate object files (which have a .mix extension for the PowerC compiler -- most compilers produce .obj files). The pcl command will link the two object files and produce an executable called experi3a.exe, provided there were no errors. The executable is named after the first module, which is usually the one with main() in it.

Now simply type in experi3a <enter>

The following will be the output if everything was entered correctly:

mode = 00 command = 80 A = out, C Upper = out, B = out, C Lower = out
mode = 01 command = 81 A = out, C Upper = out, B = out, C Lower = in
mode = 02 command = 82 A = out, C Upper = out, B = in, C Lower = out
mode = 03 command = 83 A = out, C Upper = out, B = in, C Lower = in
mode = 04 command = 88 A = out, C Upper = in, B = out, C Lower = out
mode = 05 command = 89 A = out, C Upper = in, B = out, C Lower = in
mode = 06 command = 8A A = out, C Upper = in, B = in, C Lower = out
mode = 07 command = 8B A = out, C Upper = in, B = in, C Lower = in
mode = 08 command = 90 A = in, C Upper = out, B = out, C Lower = out
mode = 09 command = 91 A = in, C Upper = out, B = out, C Lower = in
mode = 10 command = 92 A = in, C Upper = out, B = in, C Lower = out
mode = 11 command = 93 A = in, C Upper = out, B = in, C Lower = in
mode = 12 command = 98 A = in, C Upper = in, B = out, C Lower = out
mode = 13 command = 99 A = in, C Upper = in, B = out, C Lower = in
mode = 14 command = 9A A = in, C Upper = in, B = in, C Lower = out
mode = 15 command = 9B A = in, C Upper = in, B = in, C Lower = in

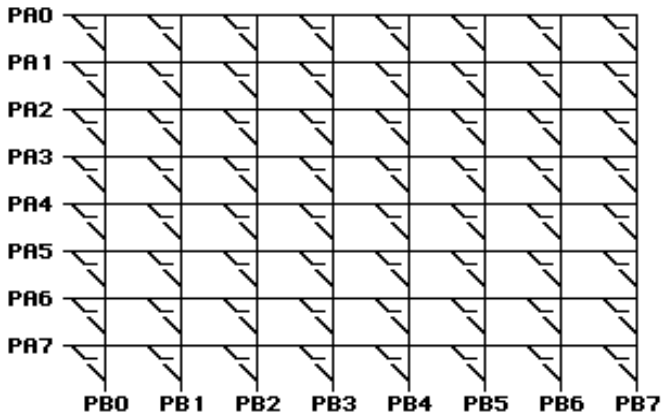
The following table, repeated from Experiment 2, confirms that the codes are correct:

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

Now change set_up_ppi() back to the form it will be in normal operation in the digital.c file:

```
// digital.c

// The following are known only to the functions that follow them.
// 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;

// 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 3 and 4
  command += (mode & 3); // add in bits 0 and 1
  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 -- in the future, this will grow 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;
} // end get_port()


// end digital.c
```

A very common way to detect a lot of closures with few inputs and outputs is to use a **switch matrix**. This is shown schematically as a grid of wires with switches at the intersections. A matrix is most commonly used in devices such as keyboards and numeric keypads. The only way a row can connect to a column in a matrix is through one of the switches:



BASIC SWITCH MATRIX

In this part of the experiment, Port A is connected to the rows and Port B is connected to the columns. You will recall from Experiment 2 that Port B has pull-up resistors connected to it. That fact was taken advantage of to add 8 switch inputs. An input would be grounded by a switch when it was turned on, producing a low on its bit in Port B. The switches no longer have one side connected to ground in the matrix, but to rows that are connected to the lines of Port A. If a Port A bit is taken low and a switch that is connected to Port A is turned on however, the Port B line connected to the same switch will be taken low. Say, for example, that PA2 is taken low. Now turn on the switch at the intersection of PA2 and PB4. PB4 will be taken low. All that's necessary to determine if a particular switch is turned on is to take the correct bit of Port A low, then check the correct bit of Port B to see if it has been pulled low. The procedure permits 64 unique closures to be detected.

To make the exercise interesting, let's use the 74LS244 for switch closures 1 through 3 as in Experiment 1, but use the matrix to add 64 more. Switches 4 through 67 will be considered to be at the matrix intersections shown below. Thus, in the example above, switch 24 will be at the intersection of PA2 and PB4.

Port A Bit 0	4	5	6	7	8	9	10	11
Port A Bit 1	12	13	14	15	16	17	18	19
Port A Bit 2	20	21	22	23	24	25	26	27
Port A Bit 3	28	29	30	31	32	33	34	35
Port A Bit 4	36	37	38	39	40	41	42	43
Port A Bit 5	44	45	46	47	48	49	50	51
Port A Bit 6	52	53	54	55	56	57	58	59
Port A Bit 7	60	61	62	63	64	65	66	67
	Port B Bit 0	Port B Bit 1	Port B Bit 2	Port B Bit 3	Port B Bit 4	Port B Bit 5	Port B Bit 6	Port B bit 7

We need to take the appropriate bit low in Port A, then look at the appropriate bit in Port B to see if a particular switch is on. Port A bit 0 should be taken low to look at switches 4 through 11, Port A bit 1 should be taken low to look at switches 12 through 19, and so on. We know how to shift a 1 by the number of bits to get things in the right place, so let's start by trying to figure out how to get 0 through 7 out of 4 through 67. Since we want 0 when we get 4 as an input, start by subtracting 4, which gives us the following:

```
     input        input-4
  4 through 11     0 - 7
 12 through 19     8 - 15
```

```
20 through 27   16 - 23
28 through 35   24 - 31
36 through 43   32 - 39
44 through 51   40 - 47
52 through 59   48 - 55
60 through 67   56 - 63
```

Since we want all of the first row to be 0, divide by 8 and neglect the remainder. Turns out, this provides the 0 through 7 that's needed:

```
input-4   (input-4)/8
 0 - 7        0
 8 - 15       1
16 - 23       2
24 - 31       3
32 - 39       4
40 - 47       5
48 - 55       6
   56 - 63       7
```

Now shift 1 to the left by the result of the above and negate to turn 1s into 0s:

```
1 << ((input - 4)/8)      ~(1 << ((input - 4)/8))
          00000001                  11111110
          00000010                  11111101
          00000100                  11111011
          00001000                  11110111
          00010000                  11101111
          00100000                  11011111
          01000000                  10111111
          10000000                  01111111
```

Thus, the value given to port a is:
porta_val = ~(1 << ((input - 4) / 8));

Since porta_val will be an integer and we only need the lower 8 bits, mask off the upper 8 bits for good measure:
porta_val = (~(1 << ((input - 4) / 8))) & 0xff;

Loading the Port A register with porta_val will take the proper row bit low to test for the indicated switch. The next task is to look at the proper column bit after reading Port B. Bit 0 needs to be used for switches 4, 12, 20, etc., bit 3 for 7, 15, 23, etc. The values change with a step of 8 since movement is down a column:

Port A Bit 0	4	5	6	7	8	9	10	11
Port A Bit 1	12	13	14	15	16	17	18	19
Port A Bit 2	20	21	22	23	24	25	26	27
Port A Bit 3	28	29	30	31	32	33	34	35
Port A Bit 4	36	37	38	39	40	41	42	43
Port A Bit 5	44	45	46	47	48	49	50	51
Port A Bit 6	52	53	54	55	56	57	58	59
Port A Bit 7	60	61	62	63	64	65	66	67

	Port B Bit 0	Port B Bit 1	Port B Bit 2	Port B Bit 3	Port B Bit 4	Port B Bit 5	Port B Bit 6	Port B bit 7

We can start by subtracting 4 again. This time however, dividing by 8 and ignoring the remainer won't work. Let's see what would happen if we divided some examples by 8 and kept **only** the remainder. The R below means remainder. Note that 4 has already been subtracted:

0 / 8 = 0 R 0	40 / 8 = 5 R 0	27 / 8 = 3 R 3	21 / 8 = 2 R 5	38 / 8 = 4 R 6	55 / 8 = 6 R 7

Looks like that might work. As it turns out, something called **modulo arithmetic** will provide us with just the remainder of an integer division. The symbol is %:

0 % 8 = 0	40 % 8 = 0	27 % 8 = 3	21 % 8 = 5	38 % 8 = 6	55 % 8 = 7

```
                        |----------------------|
                        | 0| 1| 2| 3| 4| 5| 6| 7|
                        | 8| 9|10|11|12|13|14|15|
                        |16|17|18|19|20|21|22|23|
                        |24|25|26|27|28|29|30|31|
                        |32|33|34|35|36|37|38|39|
                        |40|41|42|43|44|45|46|47|
                        |48|49|50|51|52|53|54|55|
                        |56|57|58|59|60|61|62|63|
                        |----------------------|
            (input - 4) % 8    0  1  2  3  4  5  6  7

              Input, Step 8,        Input - 4, Step 8,
going down the column   going down the column   1 << ((input - 4) % 8)
          4 ~ 60                  0 ~ 56                  00000001
          5 ~ 61                  1 ~ 57                  00000010
          6 ~ 62                  2 ~ 58                  00000100
          7 ~ 63                  3 ~ 59                  00001000
          8 ~ 64                  4 ~ 60                  00010000
          9 ~ 65                  5 ~ 61                  00100000
         10 ~ 66                  6 ~ 62                  01000000
         11 ~ 67                  7 ~ 63                  10000000
```

The Port B mask, making sure we only use the lower 8 bits, is:
portb_mask = (1 << ((input - 4) % 8)) & 0xff;

The considerable work done to produce a few lines of code illustrates an important premise of programming, and, for that matter, just about everything that's worth doing: the important part is not in the doing, but in the planning. The first impulse is to start writing code. We could have done that here. We could also have ended up with 67 if() statements or 67 case statements inside a switch(). Such inefficient code can be avoided if the up-front work is done. Efficiency, both in terms of code size and speed, is important, especially in control and embedded systems. Multiple if() statements or a long switch() take a lot more time and eat up a lot more space than does the code designed here.

A very large part of control and embedded systems work has to do with checking bits to see if they are on or off. It is a very good idea to get out a piece of paper and write down required outputs or probable inputs in tabular form as was done above, all the way down to the bit level if need be. Once the information is written down it can be analyzed using the techniques illustrated above. Here are a few possibilities:

- If possible and useful, subtract a constant to make a table start at 0.
- 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.
- Consider all masking possibilities, especially AND.

The following is the is_closure() function, including the changes to increase capability to 67:

```
// 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 67.
//
// 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 the matrix:

//             |-----------------------|
// Port A bit 0| 4| 5| 6| 7| 8| 9|10|11|
// Port A bit 1|12|13|14|15|16|17|18|19|
// Port A bit 2|20|21|22|23|24|25|26|27|
// Port A bit 3|28|29|30|31|32|33|34|35|
// Port A bit 4|36|37|38|39|40|41|42|43|
// Port A bit 5|44|45|46|47|48|49|50|51|
// Port A bit 6|52|53|54|55|56|57|58|59|
// Port A bit 7|60|61|62|63|64|65|66|67|
//             |-----------------------|
//  Port B bits  0  1  2  3  4  5  6  7

// ================================================================
int is_closure(int input)
{

  if(input < 1 || input > 67) // if the input is less than 1 or greater
    return -1;                // than 67, then return -1 showing an error

  // we fell through the above so see if input is less than 4
  if(input < 4)
    return ((inp(switch_port) >> (input + 3)) & 1) ^ 1; // yes, return using
switch_port
```

```
  // input is >= 4 so look at the matrix
  // by first setting up Port A to take the appropriate row bit low
  porta_val = (~(1 << ( (input - 4) / 8) )) & 0xff;

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

  // determine what column bit to look at for this input
  portb_mask = (1 << ((input - 4) % 8)) & 0xff;

  // a closure will cause a low, so invert the return
  if(!(inp(ppi_portb) & portb_mask))
    return 1;

  return 0;

}// 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 3 and 4
  command += (mode & 3); // add in bits 0 and 1
  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 = 0x300;
  switch_port = base + 0x18;
  ppi_porta = base + 0x20;
  ppi_portb = base + 0x21;
  ppi_portc = base + 0x22;

} // end get_port()

// end digital.c
```

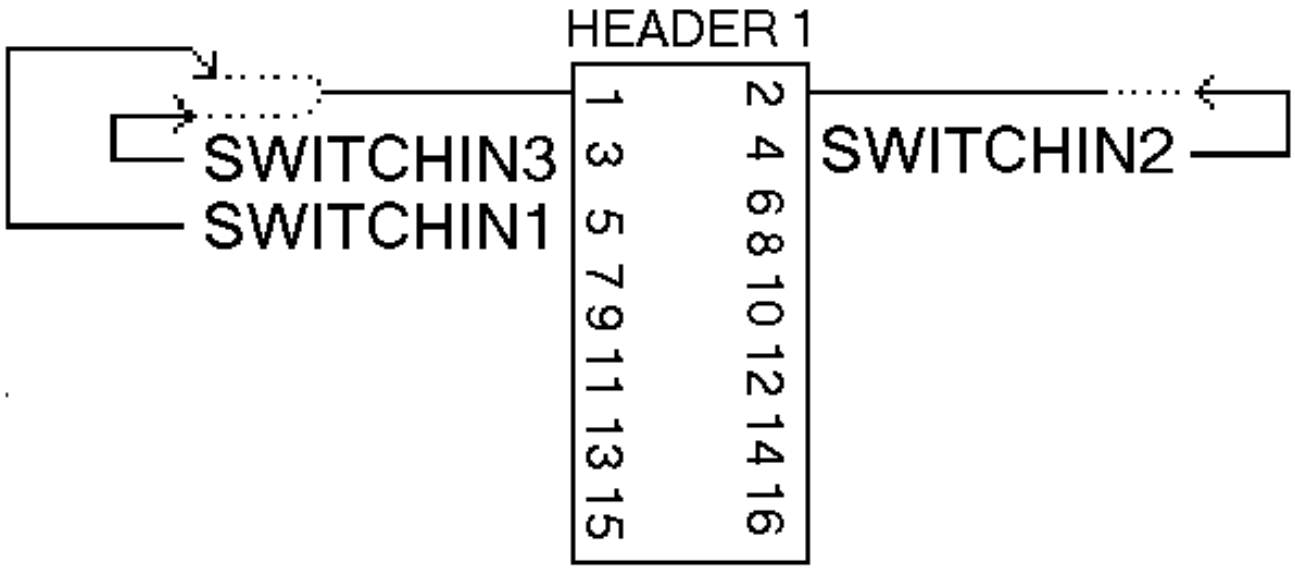
The following is the test procedure:

```
// experi3b.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;

  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

  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

    poscurs(0,0); // put the cursor at the top left

    for(x=1; x<4; x++) // first 3 switch numbers
      printf("%4d",x); // printed 4 characters wide with leading blanks
    puts(""); // go to next line

    for(x=1; x<4; x++)
    {
      if(is_closure(x))     // is there a closure -- if so, print ON, else print OFF
        printf("%4s","ON"); // "%4s" means print string 4 characters wide with
leading blanks
      else printf("%4s","OFF");
    }
    puts(""); // go to next line

    // go through matrix, starting at 4, as long as less than 68, step 8 (4, 12, 20,
etc.)
    for(x=4; x<68; x+=8)
    {
      for(y=0; y<8; y++) // go through 8 places on a matrix row -- x + y will be the
closure #
        printf("%4d",x+y); // print the numbers
      puts(""); // go to next line
```

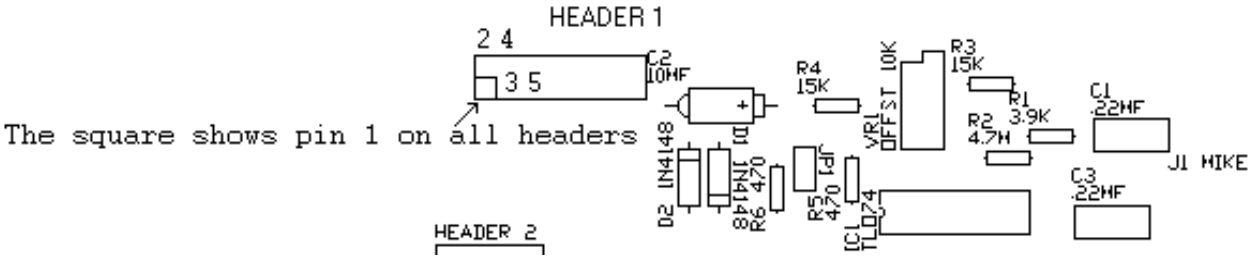
```
      for(y=0; y<8; y++)
      {
        if(is_closure(x+y)) // closure number = x + y
          printf("%4s","ON"); // is there a closure -- if so, print ON, else print
OFF
        else printf("%4s","OFF");
      }
      puts(""); // go to next line
    }
    // sleep(1); // optional wait 1 second before doing it again so results can be
seen
  }
} // end experi3b.c
```

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

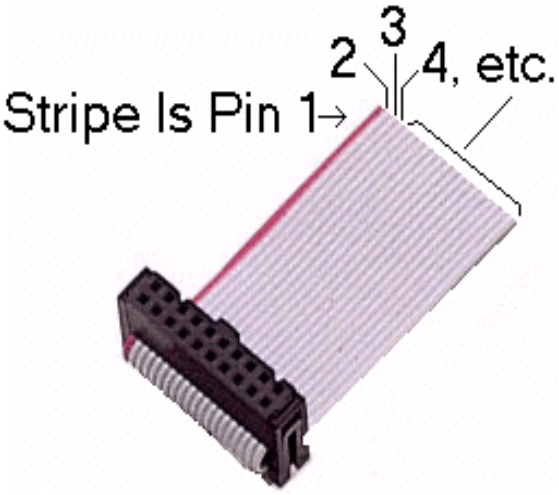
The method used to connect to the first three switch inputs is repeated from Experiment 1. The inputs are designated SWITCHIN1, SWITCHIN2 and SWITCHIN3 on the schematic and can be accessed on pins 5, 4 and 3 respectively of HEADER 1. Connecting one of the inputs to pin 1 or 2 of HEADER 1 will ground the input and pull it low to indicate that the switch has been turned on:



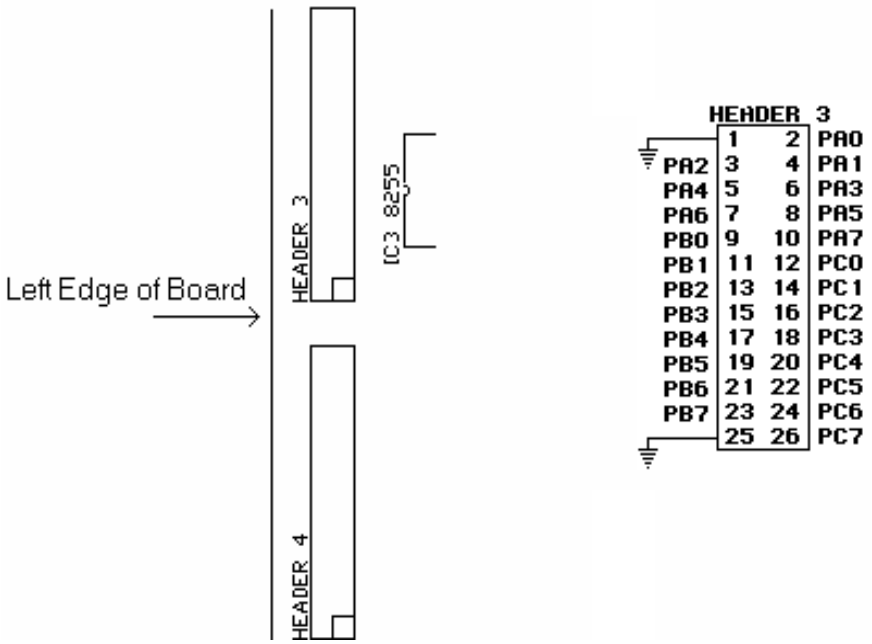
The following is part of the board layout showing the location of HEADER 1 at the top of the board:



The best way to connect to Header 1 is by means of a 16-pin 2-row female header connector with 8 pins on each row and with a .1" spacing between pins and a .1" spacing between rows. Strip the ends of wires 1, 2, 3, 4 and 5. Look for the square pin on the back of the board. It is pin 1. Plug the header socket in so that the wire with the stripe is on the same end as the square pin. It's OK to use a header with more than 16 pins on header 1. Just let the end opposite pin 1 rest on C2.

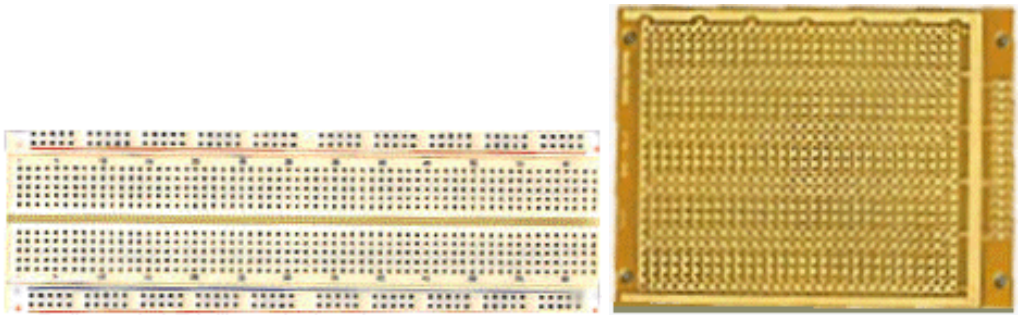


The following shows the connections to PPI Ports A and B through Header 3. Port A lines go to pins 2 through 8 and pin 10. Port B lines are on odd-numbered pins 9 through 23 of Header 3. The location of Header 3 and its pinout are shown below. Remember that pin 1 of any of the headers is the one with the square shape on the back of the board. The view below-left is from the top of the board. On the right is the schematic representation of Header 3.



The best way to connect to Header 3 is by means of a 26-pin 2-row female header connector with 13 pins on each row and with a .1" spacing between pins and a .1" spacing between rows. Strip the ends of wires 2 through 10 and the odd wires 11 through 23. Look for the square pin on the back of the board. It's pin 1. Plug the header socket in so that the wire with the stripe is on the same end as the square pin.

It might be easier to plug this many wires into a solderless or solderable breadboard such as the ones shown below. They can be purchased at just about any electronics parts supply house:

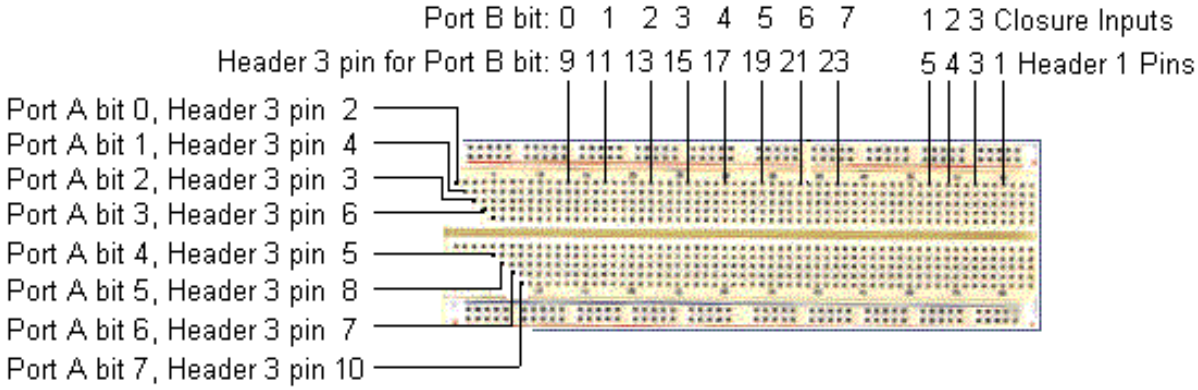


Whatever you use, it will make the testing a lot easier if you try to arrange the wires to look as close to a row/column arrangement as possible. If somebody comes up with a neat test setup, scan me a picture and I'll post it.

The following shows the relationship among inputs, port bits and Header 3:

```
                    Port B bit: 0  1  2  3  4  5  6  7
  Header 3 pin for Port B bit: 9 11 13 15 17 19 21 23
                               |-----------------------|
Port A bit 0, Header 3 pin  2| 4| 5| 6| 7| 8| 9|10|11|
Port A bit 1, Header 3 pin  4|12|13|14|15|16|17|18|19|
Port A bit 2, Header 3 pin  3|20|21|22|23|24|25|26|27|
Port A bit 3, Header 3 pin  6|28|29|30|31|32|33|34|35|
Port A bit 4, Header 3 pin  5|36|37|38|39|40|41|42|43|
Port A bit 5, Header 3 pin  8|44|45|46|47|48|49|50|51|
Port A bit 6, Header 3 pin  7|52|53|54|55|56|57|58|59|
Port A bit 7, Header 3 pin 10|60|61|62|63|64|65|66|67|
                               |-----------------------|
```

The test wiring might look like this. Notice that the Port A lines are staggered due to the fact that the board's vertical rows are connected:



Please note that only one switch at a time can be turned on without causing problems under some circumstances. Know why?

To compile and link experi3b using POWERC, do the following:
pc digital <enter>
pc experi3b <enter>
pcl experi3b digital <enter>

As noted above, you could put the three commands in a batch file if you like, then change the "experi3_" part each time.

The executible experi3b.exe will be generated if there were no errors.

Now simply type in experi3b <enter> to test is_closure().

Previous: Experiment 2 - Expanding Switch Input Detection
Next: Experiment 4 - The Multiple Closure Problem And Basic Outputs With The PPI
Problems, comments, ideas? Please e-mail me

Order | Home