Mark McKelvy
CMPS499: Embedded Software Systems
HW02

1. I installed the keil compiler and went through the steps of setting up, compiling and running the program. It seems like there are a lot of steps involved just to get a simple demo going, but it is neat to see the data that is simulated on the ports. Also I find it pretty annoying that each time a change is made you have to stop execution, stop debug mode, rebuild, enter debug mode, start execution. It'd be much simpler to just stop execution, recompile, start execution.

2.

```
/*
Mark McKelvy
CMPS499: Embedded Software Systems
HW2 Problem 2
        Writes 3 different bytes to port 2 periodically, checks a read pin and
        based upon if the read pin's value has changed since the last iteration
        writes a 0 or 1 to a write pin
*/

// keil header
#include <reg52.h>

// setup to read from port 2 pin 6
sbit read_pin = P2^6;
// setup to write to port 3 pin 4
sbit write_pin = P3^4;

// dummy delay loop for 100,000 iterations
void delay_unspecified(){
        unsigned int x,y;
        for(x = 0; x <= 1000; x++){
                for(y = 0; y <= 100; y++){
                }
        }
}

void main(){
        unsigned char byte1 = 0x08; // 0000 1000
        unsigned char byte2 = 0x40; // 0100 0000
        unsigned char byte3 = 0xFF; // 1111 1111
```

```
	int toggle = 0;				// toggle between byte values
	bit last_value = 0;			// last value of read pin
	bit current_value = 0;		// current value of read pin

	// go super loop!!
	while(1){
		// based upon toggle value will write byte 1,2, or 3 to port 2
		if(toggle == 0){
			P2 = byte1;
			toggle = 1;
		}
		else if(toggle == 1){
			P2 = byte2;
			toggle = 2;
		}
		else if(toggle == 2){
			P2 = byte3;
			toggle = 0;
		}

		// get current value of read pin
		current_value = read_pin;

		// if current value is same as last value write a 0 to write pin
		if(current_value == last_value){
			write_pin = 0;
		}
		// otherwise we'll write a 1 to it
		else{
			write_pin = 1;
		}
		// update what the last value of read pin was
		last_value = read_pin;

		// dummy delay
		delay_unspecified();
	}
}
```

3.

```
/*
 *Mark McKelvy
 *CMPS499: Embedded Software Systems
 *HW2 Problem 3
 *
 *Reads two byte strings from the input
 *and outputs the byte strings along with
 *their and, or, and xor operations.
 */
#include <stdio.h>

//outputs a byte to the screen and a newline after it
void output_byte (unsigned char byte)
{
		// 0000 0001
		unsigned char mask = 1;

		// shift one over 7 places, 1000 0000
		mask <<= 7;

		// gradually check each bit before shifting it off the end
		for(int i = 1; i <= 8; i++)
		{
			if((mask & byte) > 0)
			{
				printf ("1");
			}
			else
			{
				printf ("0");
			}
			byte <<= 1;
		}
		printf ("\n");
}

//takes a character array of size 8 and converts it to an unsigned char (byte)
unsigned char convert_char_array_to_byte (char array[])
{
		unsigned char retval = 0;
```

```
		unsigned char mask = 1;

		//check each element in the array
		for(int i = 0; i <= 7; i++)
		{
			if(array[i] == '1')
			{
				retval = (retval | mask);
			}
			if(i != 7)
			{
				retval <<= 1;
			}
		}
		return retval;
}

int main ()
{
		unsigned char x = 0xFE; // dummy value
		unsigned char y = 0x03; // dummy value
		char x_array[8];		// character array from input
		char y_array[8];		// character array from input
		char dummy;			// catch for newline

		// get x byte from user
		printf ("x = ");
		scanf ("%8c", &x_array);

		// get newline character and do nothing with it
		scanf ("%c", &dummy);

		// get y byte from user
		printf ("y = ");
		scanf ("%8c", &y_array);

		// convert the char arrays to byte strings
		x = convert_char_array_to_byte (x_array);
		y = convert_char_array_to_byte (y_array);

		// output x,y,x&y,x|y,x^y
```

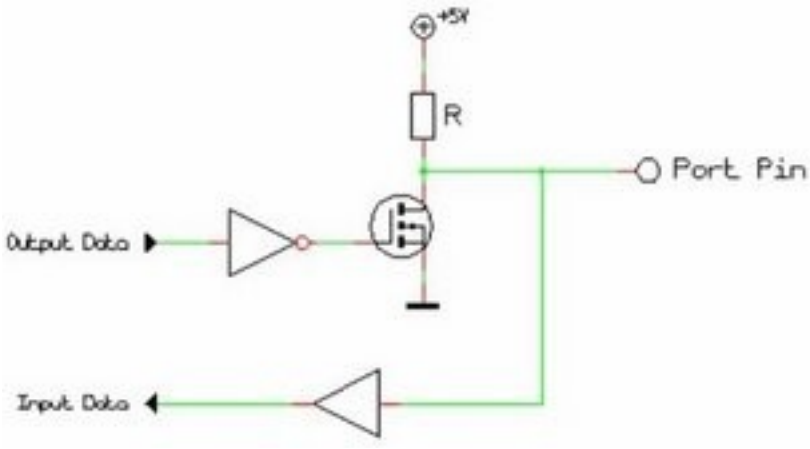
```
        printf ("x:\t");
        output_byte (x);
        printf ("y:\t");
        output_byte (y);
        printf ("x&y:\t");
        output_byte (x&y);
        printf ("x|y:\t");
        output_byte (x|y);
        printf ("x^y:\t");
        output_byte (x^y);

        return 0;
}
```

Output

```
$ make; ./bitwise_operators
g++    -c -o bitwise_operators.o bitwise_operators.C
g++ -o bitwise_operators bitwise_operators.o -I/usr/include -L/usr/lib
x = 00001111
y = 11101100
x:      00001111
y:      11101100
x&y:    00001100
x|y:    11101111
x^y:    11100011
```

4. There is not really a read/write mode for the 8051 pins. As can be seen from the simplified view below, there is some circuitry inside the chip to enable the ports to be bidirectional. What enables this is a transistor inside that is “enabled” and “disabled” at certain times. When a '1' is written to the port then the transistor switches from being grounded to sending a brief charge further in the chip. In a way this lets the port on the chip side know that he can drive the line now and so the user will be able to read from it. When the user suddenly writes anything but 0xFF to the port, the transistor enables again grounding out and “notifies” the port side that he can no longer be the driver and must accept inputs.



5. Below is the chip diagram of the standard 8051 micro controller. This chip has 4 bidirectional ports. Pins 1-8 are port 1, pins 10-17 are port 3, pins 21-28 are port 2, and pins 32-40 are port 0. Ports 1 and 2 are regular ports, port 3 is like ports 1 and 2 but with extra functionality, and port 0 is lacking "pull-up" resisters. If you wanted to use port 0, you would have to connect external "pull-up" resisters to the port to be able to use it. These resisters are what make the other ports bidirectional in nature. Port 3 doubles as a serial interface as well as a few other things. On pins 18 and 19 an external crystal is connected so regulate the clock of the chip. The speed at which the crystal "ticks" controls how fast instructions are executed within the chip. The chip may take several cycles of the crystal "tick" to complete the processing of a single instruction, so the faster the crystal "ticks" the faster the entire chip operates. There is one significant drawback to operating at a faster clock frequency. The power consumption of the chip is directly related to the clock frequency of the crystal. In an embedded system, it is often important to concern yourself with battery usage, so it may be more economical to operate at a lower clock rate to save power.