

<b>Title:</b>	<b>An Introduction to I-O v6</b>
<b>Author:</b>	<b>Ian Johnson</b> <i>(derived from PC74 version)</i> <i>(derived from Rob Williams NT/FAB version)</i>
<b>Prerequisites:</b>	<b>A Linux based computer in 2N24, complete with Amplicon PCI230 ADC/DAC interface card and MARCO rack</b>
<b>Module:</b>	<b>UFCETS-20-1</b>
<b>Awards:</b>	<b>CSI/EE/EEE/Robotics</b>
<b>References:</b>	<i>UFCETS-20-1 Lecture Notes, PCI230 manual, Comedi manual</i>

### ***Introduction:***

This worksheet introduces the basic methods of programming i/o ports using a high-level language and Unix device based interface. Switches are polled and lights illuminated.

The second assignment after Christmas will build on the techniques developed here.

### ***Learning Objectives:***

You will learn how to implement simple polling techniques. There are also some less common bitwise operations required using C operators.

### ***Commercial relevance:***

Direct i/o to ports is still a common requirement when interfacing devices to computers: indicator lights, binary sensors, switches, motors, solenoids.

### ***Unix I-O***

Like all proper operating systems, linux prevents users from directly accessing hardware. Instead access to hardware is granted by the operating system and provided through device drivers.

The device driver for the PCI230 does not makes the cards ports directly available. Instead it provides a higher level of access, namely the comedi API.

The comedi device driver for the amplicon PCI230 in 2N24 is /dev/comedi0

This device can be treated just like a file it can be opened, closed, read and written to. However in order to make sense of the data we read and write, we will use it at a higher level of abstraction. What we will do is use the comedi API library to read and write data to and from the rack.

### **What to do:**

1. Make sure the rack is plugged in & switched on!
2. Type in the following program:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <getopt.h>
#include <ctype.h>
#include <comedi.h>
#include <comedilib.h>

int main(int argc, char *argv[])
{
    int stype;
    unsigned int numval;
    int num, I, retval;
    unsigned int mask;
    comedi_t *device;
    unsigned int subdevice = 2;
    char *filename = "/dev/comedi0";

    /* open the comedi device, exit with error message if
    something goes wrong. */

    device=comedi_open(filename);
    if(!device)
    {
        comedi_perror(filename);
        exit(0);
    }

    /* check we have a valid bidirection DIO port subdevice*/

    stype = comedi_get_subdevice_type(device,subdevice);
    if(stype!=COMEDI_SUBD_DIO)
    {
```

```

        printf("%d is not a digital I/O subdevice\n",subdevice);
        exit(0);
    }
/* configure ports for output */

    for(i=0;i<8;i++)
        retval=comedi_dio_config(device,subdevice,i,COMEDI_OUTPUT);

    printf("Please enter a number [0 - 255] \n (negative to quit)\n");
    scanf("%d", &num);
    getchar();
    while ( num >= 0)
    {

        numval = (unsigned int) num;
        mask = 0xff;
        comedi_dio_bitfield(device,subdevice,mask, &numval);
        scanf("%d", &num);
        getchar();
    }

    return 0;
}

```

Refer to your notes and previous worksheets if you are unsure as to how to compile and link this program. As well as the normal C header file, `stdio.h`, please note you need to include several others!

You will also need to link the above program with the `comedi` library. In order to do this you'll need to include `-lcomedi` and `-lm` on the command line to compile, e.g.

```
gcc -Wall -lcomedi -lm myprog.c
```

The example C program presented above requires you to enter a number [0 . . . 255] at the keyboard, which is sent to the digital IO port for display on the eight discrete LEDs. Why is the value limited in range? What happens if you exceed 255?

Make sure you understand the relationship between the number you type in, and the binary pattern on the LED lamps. Experiment with the following input values:

1; 2; 3; 4; 8; 16; 32; 64; 65; 66; 127; 128; 255; 0; 256; 300

Alter the code so that all the LEDs are off when you exit with a negative number. If you are running a program which includes a keyboard input, the normal `^c` break will also work here.

3. Now modify the code to accept two decimal digits from the keyboard, add them, and display the result in binary on the LEDs.

4. Write the complementary example program which reads the binary pattern from the bit switches and converts the value to ASCII for the screen.

We have opened the DIO port read/write since it is bi-directional, so all we need to do is read rather than write. Using the `comedi_dio_bitfield()` function lets us do this easily however whilst the LED's and switches are only 8 bits, the comedi driver supports 32 bits in a 32 bit variable.

### **READ THE COMEDI\_DIO\_BITFIELD() MANUAL PAGE**

You should now be able to see how by using the `comedi_dio_bitfield` functions you can gain access to the functions of the I/O subsystem without you needing to know the squalid details of exactly how it is achieved. Once the file descriptors have been opened with the `open` function, you can simply use the `read` and `write` system calls provided by the operating system for low level file I-O. **Remember!** you'll need to be careful to ensure you only use `read` & `write` **NOT** `fread` and `fwrite`.

Note also that writable registers in I/O devices are often not readable; this is a fairly common situation for this kind of I/O hardware. The binary value corresponding to the signals on the pins of the PC74 digital port is not retrievable once written. This means that if your program needs to know the value you have written, it will need to keep a copy in a variable.

**Since the DIO port is bi-directional you can read the switches or write the LEDs through it. If you are going to do both, you'll need a delay between these. The `usleep()` function will let you create one!**

5. Write a program, using the functions provided that loops continuously reading the switches and displaying their value on the LEDs. Remember you'll need a delay between the read and write!

The program should also test the switches' value; if it is exactly 01111110 binary (outermost switches down, all others up) then the program should exit gracefully.

6. Modify your program to count the number of switches in the down position and display the binary value on the LEDs and on the screen in decimal.

7. Modify your program to show the number of switches in the down position as a "thermometer" bar-graph display on the discrete LEDs.

To isolate the least significant bit (lsb) from an integer requires either modulus division by 2 (`n % 2`) to extract the least significant bit, or the use of logical `&` operator with 01 to extract the least-sig-bit as a TRUE or FALSE value.

The shift operators, << and >>, can be useful, too.

8. Write a program to implement a **Duck Shoot** game.

In this game, you start with every alternate LED illuminated ( 01010101 ). Your program must shift this pattern on the LEDs one position at a time, at a speed of about three shifts per second; the bit of the pattern shifted off one end is stuffed back into the other end, making the pattern loop around continuously.

However, at each shift the bottom switch of the bank of eight must be examined. If the switch is up, the bit shifted off the end is inserted at the other end; if the switch is down, the bit is inverted (0 changed to 1 and vice versa) before being inserted at the other end.

In this way, by proper manipulation of the end switch, you can eliminate all the lights / ducks. When all eight lights are off, the player has won and the game terminates. If all eight lights are on, the ducks have won and the game terminates.

You should make some provision for varying the speed of shifting: ideally you should use some of the other switches to accomplish this, so that the program needs no keyboard input whatever. The direction that the ducks fly should also be determined by a switch.

## HINTS:

To get the time delay required between shifts, use a FOR loop with nothing inside it to keep the machine busy by counting up to a few thousand.

**THIS IS UNACCEPTABLE FOR THE ASSIGNMENT**

Alternatively, we can do this more efficiently using the `sleep(seconds)` or `usleep(microseconds)` function calls. Read the Friendly Manual pages for more information on these.

Note that you cannot read back the value you outputted on the DIO port. Do all your manipulations on a shadow variable, and copy it out to the DIO as necessary.

## Reading List

Comedi Manual	(online)
Lecture notes for UFCETS-20-1	(module webpage)
Any good C book!	