ECE471: Embedded Systems – Homework 5

i2c interface and LED Display

Due: Thursday, 8 October 2015, 9:30AM EDT

1. Use your Raspberry Pi for this homework. In addition you will need one 4x7 segment LED display that was handed out in class (if you missed class, stop by my office to pick one up). More info on the display can be found here: `http://www.adafruit.com/products/880`

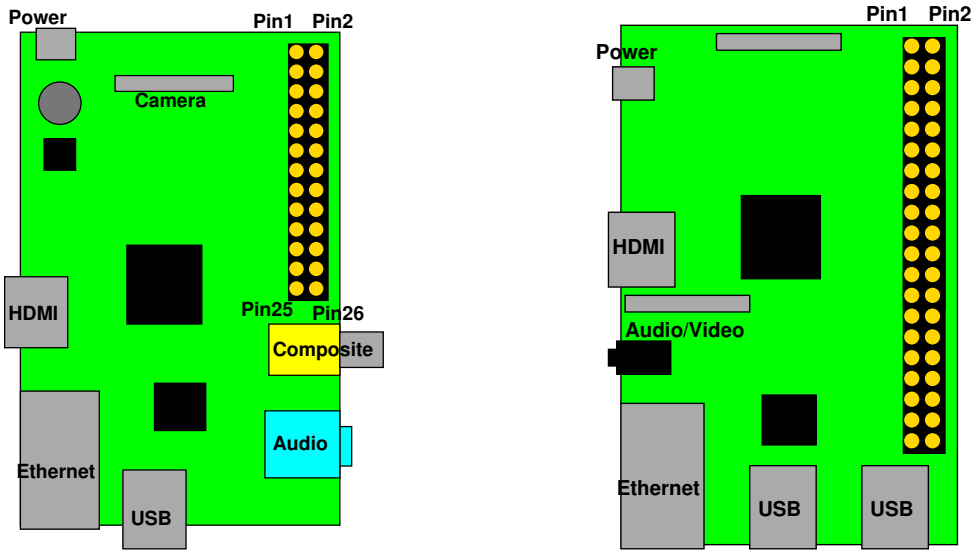


Figure 1: Location of header on Raspberry Pi Model B and B+/2

2. First you will need to hook up the LED display. You can use Figure 1 and Table 1 for guidance.

   (a) Hook the display to your pi on a breadboard:
   Connect `3.3V` on the Pi to `+` on the LED Display.
   Connect `GND` on the Pi to `-` on the LED Display.
   Connect `SDA` on the Pi to `D` on the LED Display.
   Connect `SCL` on the Pi to `C` on the LED Display.

   (b) Enable i2c support in Linux running on your pi.
   There may be multiple ways to do this.

   - On newer versions of Raspbian you can run `sudo raspi-config` pick "Advanced Options" then "I2C" then say yes, and say yes to load it be default.
   - On older versions you can install the driver (kernel module) yourself:
     `sudo modprobe i2c_bcm2708`
   - In both cases you will probably need to install the i2c-dev interface:
     `sudo modprobe i2c-dev`
   - You can have these kernel modules loaded automatically at boot time by editing `/etc/modules` and putting into the file:
     `i2c_bcm2708`
     `i2c-dev`
     You may also have to comment out (by putting a # at the start of the line) the `i2c_bcm2708` line in the file `/etc/modprobe.d/raspi-blacklist.conf`

Table 1: Raspberry Pi Header Pinout

3.3V	1	2	5V
GPIO2 (SDA)	3	4	5V
GPIO3 (SCL)	5	6	GND
GPIO4	7	8	GPIO14 (UART_TXD)
GND	9	10	GPIO15 (UART_RXD)
GPIO17	11	12	GPIO18 (PCM_CLK)
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10 (MOSI)	19	20	GND
GPIO9 (MISO)	21	22	GPIO25
GPIO11 (SCLK)	23	24	GPIO8 (CE0)
GND	25	26	GPIO7 (CE1)
ID_SD (EEPROM)	27	28	ID_SC (EEPROM)
GPIO5	29	30	GND
GPIO6	31	32	GPIO12
GPIO13	33	34	GND
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
GND	39	40	GPIO21

(c) Sanity check your i2c setup. If you have your pi on a network you can run `apt-get install i2c-tools`; if you can't, then skip this step and hopefully your display will work when you first program it.

Run `sudo i2cdetect -y -r 1` and if things are working you should see a "70" at address 70. This means Linux can see your slave device at address 0x70.

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- UU -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: 70 -- -- -- -- -- -- --
```

3. **Getting the Code**

The display is run by a ht16k33 chip. You can get the datasheet here:
`http://www.adafruit.com/datasheets/ht16K33v110.pdf`

Download the template code from the ECE471 website:
`http://web.eece.maine.edu/~vweaver/classes/ece471_2015f/ece471_hw5_code.tar.gz`

Uncompress it with `tar -xzvf ece471_hw5_code.tar.gz`

4. **Light up the Entire Display** (3 points)

   Modify the provided `display_test.c` file. Running `make` should build your code. If you get clock skew errors, either set your clock to current time or else run `make clean` before running `make`.

   Comment your code!

   Make sure you check any function calls for errors and report them back to the user.

   Here are the steps needed to talk to the device:

   (a) Open the device using the `open()` function. This returns an integer file descriptor for the device, or -1 on error. The call will look something like

   ```
   fd = open("/dev/i2c-1", O_RDWR);
   ```

   where `O_RDWR` means open for reading and writing.

   (b) Set the slave address. An IOCTL is used for this.

   ```
   result=ioctl(fd, I2C_SLAVE, 0x70);
   ```

   where `I2C_SLAVE` is defined in the `linux/i2c-dev.h` header, fd is the file descriptor from earlier, and 0x70 is the slave address. A negative value is returned on error.

   (c) Commands to the device are described starting on page 10 of the data sheet. Commands are an 8-bit value, with the command type in the top 4 bits and the data in the low 4 bits.

   (d) Send a command to activate the oscillator on the device by modifying the "System Setup Register". The high 4 command bits should be 0x2 and the low 4 bits should be 0x1. Write this 8-bit value to the device. The code will look something like this:

   ```
   unsigned char buffer[17];

   buffer[0]=(0x2<<4)|(0x1);
   result=write(fd, buffer, 1);
   ```

   This says to write to file descriptor `fd` 1 byte from a pointer pointing to the beginning of the buffer array. The return value is how many bytes were successfully written.

   Feel free to use C pre-processor defines to make the constants for the commands and data easier to read.

   (e) Next turn on the display, with blinking disabled. Do this via the "Display Setup Register"

   (f) Next set the brightness. A value from 10 to 15 is probably best. Do this via the "Display Dimming Data Input". You will need to get the proper values from the data sheet.

   (g) Finally write out what to display. For this simple test case we will write all 1s to make the display completely light up.

   To do this, write the "Display Data Address Pointer" which for our case is 0, then followed by 16-bytes holding the two 8-bit values for each of the 8 rows. It is easiest to just write all 17 bytes at once.

   ```
   unsigned char buffer[17];
   int i;

   buffer[0]=0x0;
   ```

```
for(i=0;i<16;i++) buffer[1+i]=0xff;
write(fd,buffer,17);
```

(h) Now close the file descriptor with `close(fd);` and exit the program. The display will keep displaying the last thing written to it.

5. **Display ECE 471** (3 points)

   For this part of the homework, modify `display_final.c`. First copy your `display_test.c` file over as a starting point:

   `cp display_test.c display_final.c`.

   Then work on the `display_final.c` code.

   The goal is to make the display show the following pattern:

   - `ECE`
     pause for 1 second
   - `471`
     pause for 1 second
   - Then loop back to the beginning to display ECE again, and repeat forever (you can stop it by pressing Control-C).

   The display mapping for each LED segment is shown in Figure 2.

   As an example, to display the letter 'E' in the far left column, you would do:

```
unsigned char buffer[17];

buffer[0]=0x00;  // offset pointer
buffer[1]=0x79;  // Column 1, Segments ADEFG
buffer[2]=0x00;  //  next 8 bits of column 1, not connected
buffer[3]=0x00;  // Column 2, empty
buffer[4]=0x00;  //  next 8 bits of column 2, not connected
...
write(fd,buffer,17);
```

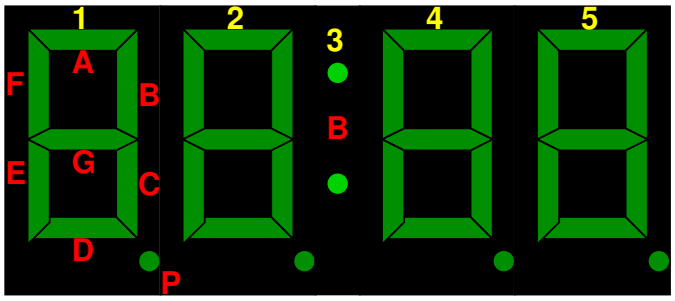
   The Linux/C function to pause (sleep) for a period of time is `usleep()`. The parameter is the amount of time to sleep in micro-seconds.

6. **Something Cool** (1 point)

   Copy your `display_final.c` code over to `display_cool.c` and modify it to do something extra.

   Some possible suggestions:

   - Display ECE 471 but have it scroll left or right.
   - Make a little animation on the screen, bouncing ball, etc.
   - Display your name, or as much of your name as you can with 7-segment displays.
   - Display the time. (the `time()` and `localtime()` routines will be helpful here).



byte 0 = 0x00 (display pointer offset)
byte 1 = (1P, 1G ,1F, 1E, 1D, 1C, 1B, 1A)
byte 2 = 0x00
byte 3 = (2P, 2G, 2F, 2E, 2D, 2C, 2B, 2A)
byte 4 = 0x00
byte 5 = (X, X, X, X, X, X, X, 3:, X)
byte 6 = 0x00
byte 7 = (4P, 4G, 4F, 4E, 4D, 4C, 4B, 4A)
byte 8 = 0x00
byte 9 = (5P, 5G, 5F, 5E, 5D, 5C, 5B, 5A)
byte10–byte16 = 0x00

Figure 2: LED Display Segment Mapping

Put into the README some notes about what your code does.

7. **Questions (2 points)**

   Answer the following in the README file:

   (a) What part of the Raspberry Pi (hardware) is in charge of the initial boot process? Is this normal for an ARM board?

   (b) What is the code called that is responsible for finishing the boot process and loading the operating system?

   (c) Why is the `/boot` partition on the Pi a FAT32 filesystem?

8. **Linux Fun (1 point)**

   (a) You can use the `wc` word-count program to count the lines, words, and bytes in a file.
   Use `wc -l display_final.c` (that's a lower-case L) to count the number of lines in your `display_final.c` file. Report how many lines it is.

   (b) You can use the `diff` command to compare two files and see the differences between them. Run
   `diff -u display_test.c display_final.c`
   Pipe the output of diff into wc and report how big the diff file is for the above command:
   `diff -u display_test.c display_final.c | wc -l`

9. **Submitting your work**

- Run `make submit` which will create a `hw5_submit.tar.gz` file containing `Makefile`, `README`, `display_test.c`, `display_final.c`, and `display_cool.c`
  You can verify the contents with `tar -tzvf hw5_submit.tar.gz`
- e-mail the `hw5_submit.tar.gz` file to me by the homework deadline. Be sure to send the proper file!