ECE471: Embedded Systems – Homework 6

Direct (bit-bang) i2c Access

Due: Friday, 25 October 2019, 1:00pm EDT

1. Use your Raspberry Pi. In addition you will need the 4x7 segment LED display used in Homework 5.
2. Hook up the display just as you did in Homework 5.

   You can use Figure 1 and Table 1 for guidance.
   As a reminder: `3.3V` to `+`, `GND` to `-`, `SDA` to `D`, and `SCL` to `C`.

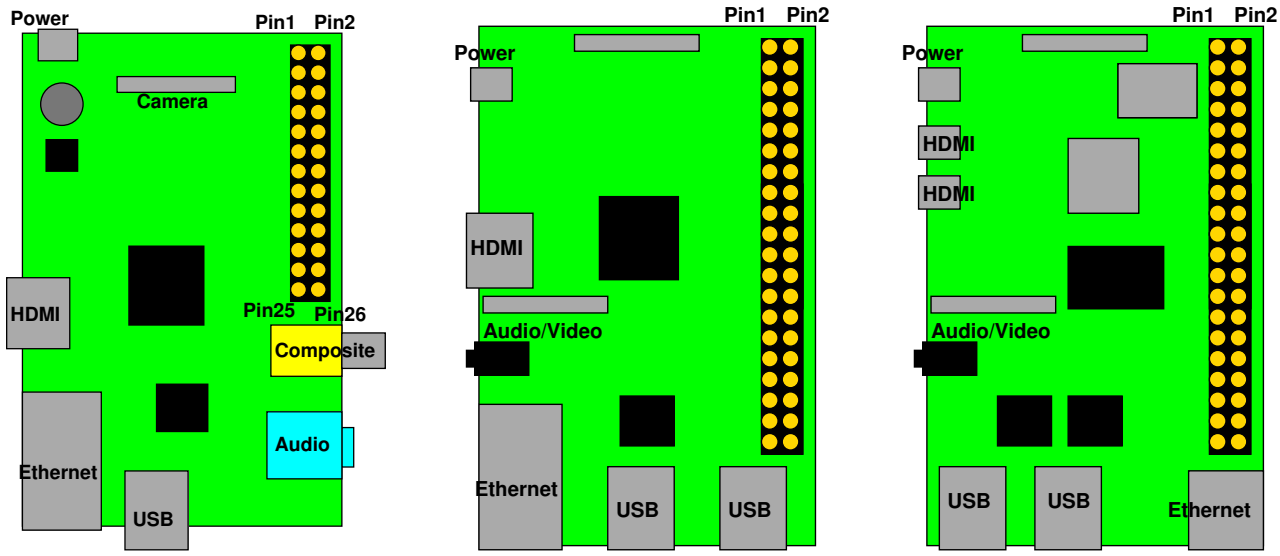

Figure 1: Location of header on Raspberry Pi Model 1B, 1B+/2/3, 4B

Table 1: Raspberry Pi Header Pinout

3.3V	1	2	5V
GPIO2 (SDA)	3	4	5V
GPIO3 (SCL)	5	6	GND
GPIO4 (1-wire)	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

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/ece471_hw6_code.tar.gz`

   Uncompress it with `tar -xzvf ece471_hw6_code.tar.gz`

   Some notes on the file layout:

   Note that there are multiple code and header files provided this time. In software engineering, it is often considered bad form to just have one large source code file with everything in it. This can be hard to navigate and also difficult to collaborate with others when everyone is working on one large file.

   To split things up, one splits the source into multiple C files, usually by purpose. In this case a `gpio.c` file is provided which does gpio setup, and `i2c-bitbang.c` includes the i2c routines. A header file for each is provided which details the interfaces provided by the files. These are the public interfaces provided by each file, and a file wanting to use them will include this header file and then can call the functions listed. When compiling the C compiler links all of the files together and makes sure that calls you do to functions in other C files get linked together properly.

   If you have code in a C file that should remain local to the file and not be called by external code, you can add the `static` identifier in front and that function or variable will not be visible outside the source code file.

4. **Enable bit-bang i2c support** (6 points)

   Modify the provided `i2c-bitbang.c` file. Running `make` should build your code. It will create the `i2c-bitbang.o` object file, then automatically link this against the `i2c-test` and `i2c-cool` executables.

   Comment your code!

   The code in `gpio.c` already does much of the GPIO configuration so you don't have to. You will need to fill in various functions in the `i2c-bitbang.c` file before your display will start to work.

   (a) **i2c SDA/SCL helper functions**

      i. `SDA_gpio_pull_low()`
      This routine should change the SDA GPIO write direction to output and then write a '0' to the (already open) `sda_fd` value file descriptor.
      You can use the provided `gpio_set_output()` routine to change the write direction: `gpio_set_output(SDA_GPIO);`

      ii. `SCL_gpio_pull_low()`
      This routine should be like the previous one but pull SCL low instead of SDA.

      iii. `SDA_gpio_float_high()`
      On an open collector bus with resistor pullups you do not want to force a pin high. Instead, you want to let it float high by not driving the output at all. An easy way to do that is to set the pin to input mode (you can use `gpio_set_input()`)

      iv. `SCL_gpio_float_high()`
      This routine should be like the previous one but letting SCL float high instead of SDA.

v. `read_SDA()`
This routine should read the SDA line and return the value.
First change the GPIO direction to input (you can use the provided `gpio_set_input(SDA_GPIO);`). Be sure to rewind `sda_fd` with `lseek(sda_fd,0,SEEK_SET);` before reading. Read the value and return 0 or 1.
Remember to convert from ASCII to decimal!

vi. `read_SCL()`
This routine should be like `read_SDA()` but for SCL instead. Be careful when cut-and-pasting that you convert all the sda to scl.

(b) **i2c protocol bit functions**

i. `i2c_start_bit()`
This routine should send an i2c start bit. If you recall, this means `SDA` goes from high to low while `SCL` is high. Ensure that `SCL` stays high for a long enough time (you can use the provided `I2C_delay()` routine.

ii. `i2c_stop_bit()`
This routine should send an i2c stop bit. This is much like the previous start bit routine, only `SDA` goes from low to high while `SCL` is high.

iii. `i2c_read_bit()`
Let `SDA` go high and wait a delay.
Let `SCL` go high and wait a delay.
At this point the slave device should have set `SDA` so read the value of SDA.
Delay, then pull `SCL` low.

iv. `i2c_write_bit()`
Pull `SCL` low.
Set `SDA` to the value you want.
Delay. Then let `SCL` go high.
Delay. Then pull `SCL` back low.

(c) **i2c protocol byte functions**

i. `i2c_write_byte()`
This routine writes an i2c byte.
To do this you will need to write all 8 bits in the passed in byte value.
One way to do this is have a loop that iterates over all 8 bits (remember, most significant bit first) and calls `i2c_write_bit()` on each.
After sending all 8-bits, read the NACK bit with `i2c_read_bit()` and make sure it is 0 signifying the device received the data OK.

Once you have all of the above implemented, running `./i2c-test` should blink "ECE".

What happens is the code in main writes a series of i2c transactions that configure the display much like in HW#5.

The provided `write_i2c()` call sends a start bit, writes all bytes using the `i2c_write_byte()` function, and sends a stop bit. The byte array sent in has the address as the first byte (7-bits plus direction) followed by the commands to send to the device.

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

   Copy your `i2c-test.c` code over to `i2c-cool.c` and modify it to do something extra. It can just be the same blinking ECE 471 or whatever you did for HW5, or you can try out something different this time. Put into the README some notes about what your code does.

   Alternately, hook up the i2c bus to a logic analyzer (an analog discovery board) and send a plot of an i2c transaction along with your homework submission. Does the i2c waveform look anything like the one shown in the lecture notes? How does it compare with the results generated by the Pi in HW#5?

6. **i2c Questions (1 point)**

   Answer the following in the README file:

   (a) Does your code in this assignment implement the full i2c protocol? What is missing?

   (b) The assignment does not specifically ask you to implement robust error handling (so you won't be graded on that). Given that, does your code handle all possible error conditions? List at least one error condition your current code would not handle well.

7. **Other Questions (1 point)**

   (a) How do you indicate that a function in C should only visible within its own `.c` file?

   (b) How does Linux on your Pi know that the board has an i2c bus? How does it know what address this i2c bus lives at?

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

   (a) You can use the `cat` command to dump a text file to the screen.
   Run `cat /proc/interrupts` which will give you status on the interrupts that have happened on your Pi since it was turned on.

   List a name of one of the interrupt sources.

   (b) There is a command called `yes`. Run it and see what it does. Why do you think a utility like this exists?

9. **Submitting your work**

   - Run `make submit` which will create a `hw6_submit.tar.gz` file containing `Makefile`, `README`, `i2c-bitbang.c`, and `i2c-cool.c`
     You can verify the contents with `tar -tzvf hw6_submit.tar.gz`
   - e-mail the `hw6_submit.tar.gz` file to me by the homework deadline. Be sure to send the proper file!