ECE471: Embedded Systems – Homework 6

Realtime Linux

Due: Friday 24 October 2025, 5:00pm

For this assignment there is no coding. You will need to download the hw6 source to build and run the tests, but please put all of your question answers into either the README or else a separate text, pdf, or word document which you then e-mail to me.

1. **Real Time Linux**

   For this you will need to connect GPIO24 to GPIO25 on your Pi. For extra safety, do this via a 1K resistor (brown-black-red). This isn't strictly necessary, but if somehow by accident GPIO24 was configured for output 0 and GPIO25 was configured to output 1 it would be a short of power to ground which could damage the Pi (having a resistor there limits the current that would flow in such a situation). See Figure 1 and Table 1 to locate the relative positions of GPIO24 and GPIO25.

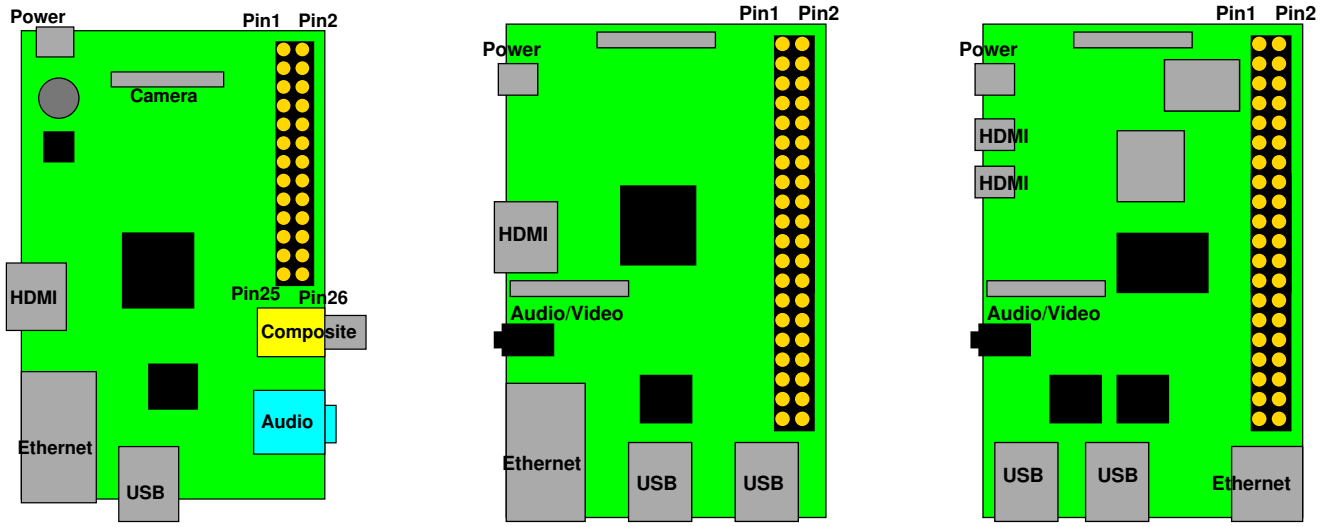

Figure 1: Location of header on Raspberry Pi Model B, Models B+/2/3, Model 4B

2. **Measure Idle System GPIO latency measurements** (2 points)

   (a) Download the template code from the ECE471 website:
   `https://web.eece.maine.edu/~vweaver/classes/ece471/ece471_hw6_code.tar.gz`
   Uncompress it with `tar -xzvf ece471_hw6_code.tar.gz`

   (b) Run `make` to compile the code.

   (c) The code works by starting two threads using the Linux pthreads library. One thread every 100ms sets GPIO24 high, records the time, then sets things low again. The other thread spins in a tight loop reading GPIO25 waiting for the line to go high, and when it does it records the time.
   The time is recorded using the `clock_gettime(CLOCK_REALTIME, &timespec);` high-resolution timer on Linux. In theory this timer can have down to 1ns resolution but in practice it's not quite that good. In an ideal world the difference between the two logged times is roughly the response time it took to notice the GPIO change.
   The `measure` program runs the experiment multiple times (10 by default), then reports the high, low, and average latency. It takes a command line argument specifying how many times to run.

   (d) Run `./measure 100` to gather results for 100 runs.

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

(e) **Questions:**

i. What is the min/max/average?

ii. If you were designing a real-time embedded system using your Pi, based on the results you measured what would be a "safe" value you could pick for how quickly GPIO25 could respond to the line going from low to high? Explain why you would pick that value.

3. **Measure Busy System GPIO latency measurements** (2 points)

(a) Now run the `./load` program. This will start up 10 threads of busy work, which should put a heavy load on all the cores in the system.

Let this program run in the background. One way of doing this is after running it, pressing control-Z, then typing `bg` to put the job in the background. Alternately you can open another connection/terminal on your pi and just do the next step in another window.

Run the `top` tool and verify that `load` is running. It should be shown as taking 400% of the CPU (As your pi probably has 4 cores) and the "load average" on the system should be gradually approaching 10 (meaning 10 things are running at once). (you can use Q to quit top)

(b) Now run `./measure 100` again.

(c) **Questions**:

i. Report the min/max/average.

ii. How are the results different from last time? How might this change your worst-case latency plans on a real-time system?

4. **Giving higher real-time priority** (2 points)

   (a) Keep `load` running in the background, but this time run
   `sudo chrt -r 70 ./measure 100`
   This tells the Linux scheduler you want to run the measure program with a real-time priority of 70 (higher is better).

   Unexpectedly you still might get bad results here, with a few high outliers. (Try running this a few times to see if the results improve).

   Try running instead:
   `sudo taskset -c 0 chrt -r 70 ./measure 100`
   which tells Linux to always run your code on CPU core #0 and this seems to help get more consistent results.

   (b) **Questions**:

      i. What is the min/max/average in this case?

      ii. Did using the real-time priority improve the latencies at all?

      iii. When we ran the `chrt` command we needed to do it as root (with `sudo`). This is because normal users are not allowed to set real time permissions by default. Why might a system administrator not want to let regular users give high-priority real-time priority to their programs?

   (c) Once you are done, you can kill the load program either with control-C, or if you put it in the background, use `fg` to bring it back to the foreground then press control-C.

5. **Giving the other job higher priority** (1 point)

   NOTE: I've tested this on my Pi with no troubles while logging in over the network with ssh. Last year some people running locally with a GUI had issues where the graphics interface would lock up. If you encounter that problem, just note in your writeup that you couldn't do the experiment due to the computer locking up.

   (a) One last experiment, this time run the load program with real-time priority 70:
   `sudo chrt -r 70 ./load` and put it in the background again.

   (b) Try measuring with regular `./measure 100`
   This might take forever. After a bit you can kill the load process with control-C so the measure finishes up.

   (c) With the load program with real-time priority 70 still in the background, now run measure with a higher priority than the load
   `sudo chrt -r 75 ./measure 100`

   (d) **Questions**:

      i. What is the min/max/average for these two cases?

6. **Real Time Classification Questions** (3 points)

   (a) You are designing an embedded system for a car that controls the anti-lock brakes. The specification says that to work properly the brakes needed to start pulsing within 10ms of detecting a skid. Would this be a hard, firm, or soft real-time task? Why?

   (b) You are designing another part of the car. The specification says that if you push the "tune" button on the stereo that it should switch stations within 1s. Would this be a hard, firm, or soft real-time task? Why?

   (c) You are working on the "info-tainment" system for the car, and it has a movie player for the backseat. The specification calls for the video decoder to be able to maintain a framerate of 60Hz. Is this a hard, firm, or soft real-time task? Why?

7. **Something Cool** (optional)

   The something cool is optional this time, but you can get extra credit for completing it.

   - (Medium) Modify the code so in addition to average it also calculates the standard deviation. Report your results.
   - (Hard) Modify the code to print all 100 values, then plot a frequency graph (using some sort of plotting program, excel, matlab, etc) showing the timing delays. Include the graph with your submission.

Submitting the Assignment

Please put your answers to Questions at 2e, 3c, 4b, 5d, 6, and 7 in some sort of document. This can be the README (in this case you can use "make submit" and submit it like previous homeworks) or instead just put it in some other document (text, pdf, doc) and ***e-mail*** it to me by the deadline.