ECE471: Embedded Systems – Homework 10
Realtime Linux / Power

Due: Friday 3 December 2021, 11:00am

For this assignment there is no coding. You will need to download the hw10 source to build and run the tests, but please put all of your question answers into a 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. If you have a female-female connector you can use that, alternately use a breadboad and the wires from previous assignments. See Figure 1 and Table 1 to locate the relative positions of GPIO24 and GPIO25.

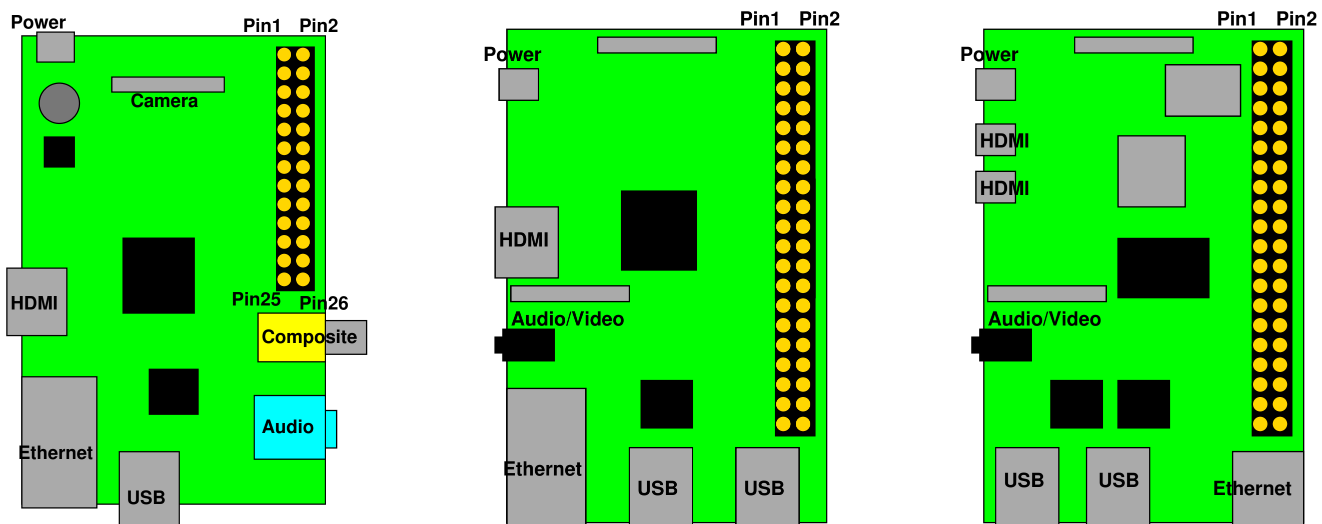

Figure 1: Location of header on Raspberry Pi Model B, Models B+/2/3, Model 4B

2. **Get the GPIO latency measurements** (5 points)

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

   (b) The code works by starting two threads using the Linux pthreads library. One thread every 100ms sets GPIO24 high then 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, ×pec);` high-resolution timer on Linux. In theory this timer can have down to 1ns resolution but in practice it's not quite that good.

   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.

   (c) Run `./measure 100` to gather results from 100 runs.

   Questions: What is the min/max/average?

   If you were designing an embedded system, what would be a "safe" value you could pick for how quickly GPIO25 could respond to the line going from low to high?

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

(d) 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 pressing control-Z, then typing `bg`. 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.

Now run `./measure 100` again.

Questions: Report the min/max/average. How is it different than last time? How might this change your worst-case latency plans on a real-time system?

(e) 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).

Question: What is the min/max/average in this case?

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 you not want regular users to give high-priority real-time permissions to their programs?

(f) 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 then press control-C.

3. **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 showing the timing delays. Include the graph with your submission.

Power and Energy (5pts)

Table 2: OpenBLAS HPL N=10000 (Matrix Multiply)

Machine	Processor	Cores	Frequency	Idle Power	Load Power	Time	Total Energy
Raspberry Pi 2	Cortex-A7	4	900MHz	1.8W	3.4W	454s	1543J
Dragonboard	Cortex-A53	4	1.2GHz	2.4W	4.7W	241s	1133J
Raspberry Pi 3	Cortex-A53	4	1.2GHz	1.8W	4.3W	178s	765J
Jetson-TX1	Cortex-A57	4	1.9GHz	2.1W	13.4W	47s	629J
Macbook Air	Broadwell	2	1.6GHz	10.0W	29.1W	14s	407J

4. Table 2 shows the energy use of various machines when doing a large Matrix-Matrix multiply.

   (a) Which machine has the lowest under-load power draw?

   (b) Which machine consumes the least amount of energy?

   (c) Which machine computes the result fastest?

5. Consider a use case with an embedded board taking a picture once every 60 seconds and then performing a matrix-multiply similar to the one in the benchmark (perhaps for image-recognition purposes). Could all of the boards listed meet this deadline?

6. Assume a workload where a device takes a picture once a minute then does a large matrix multiply (as seen in Table 1). The device is idle when not multiplying, but under full load when it is.

   (a) Over an hour, what is the total energy usage of the Jetson TX-1?

   (b) Over an hour, what is the total energy usage of the Macbook Air?

7. Given your answer in the previous question, which device would you choose if you were running this project off of a battery?

Submitting the Assignment

Please put your answers to questions 2 - 7 in some sort of document (text, pdf, doc) and ***e-mail*** it to me by the deadline.