Assignment 2: Light Dip Detector

- Submit deliverables to CourSys: https://coursys.sfu.ca/
- This assignment may be **done individually or in pairs**; marked identically. Do not give your work to students in other groups, do not copy code found online.
  - Post general questions to the course discussion forum on Piazza.
  - Ask questions specific to your solution as private messages on Piazza
- See the marking guide for details on how each part will be marked.
- Do not give your work to another student, do not copy code found online, and do not post questions about the assignment online other than the course forum. See website for guidance on using AI tools.
- If you have previously taken the course, you may *not* re-use your previous solution.

1. Write "`light_sampler`" Program

Write a C, C++, or Rust program named `light_sampler` which runs on the target to read the current intensity of light in the room. It will:

- Use a light sensor to read the ambient light level in the room
- Use the Zen cape's 2-character (14-segment display) to display the number of dips in the light intensity seen in the previous second.
- Use the Zen cape's potentiometer to control how fast an LED blink (using PWM). This LED shines on the light-level sensor.
- Display some information each second to the terminal.
- Listen to a UDP socket and respond to commands

Your program must:

- Build using CMake (or similar if using Rust).
- Use good modular design with at least four (4) modules (i.e., modules with a `.h` and `.c` file and a coherent interface to its functionality)[1]. You may have significantly more! For C, all functions should be internal linkage or external linkage as best fits their purpose; public/private in C++ or Rust.
- Have a HAL layer or directory which separates lower-level hardware access modules.

Think of this assignment as a number of mini-tasks put together:

- Wiring up some discrete parts to the BeagleBone: light-level sensor and an LED
- Analog to digital (A2D) reading (potentiometer and light sensor)
- Output on a 14-segment display using I2C
- PWM to control an LED
- UDP networking
- Use a provided module to analyze the sampling speed of your program.
- Threads and thread synchronization

See last page of assignment for suggestions on getting started, and checking your progress.

1 Assignment directions are written for C. If using Rust/C++ you must adapt requirements the best way you can.

1.1 Sampling Light Levels

In a separate thread (using `pthread`s), continually sample the light level in the room:

- Use the provided guide and electronics in your kit to wire up the light sensor to the BBG.
- Use the A2D to read the light sensor's voltage on `in_voltage1_raw` (P9, pin 40).
- All samples taken during one second are saved into a history.
- The program must store, analyze, and provide access to the previous second's samples and statistics[1].
- Sleep for 1ms between samples.
- Count the total number of light samples the program has read since it started.

Independent of the history, compute the current *average* light level.

- Compute this average using exponential smoothing over all light samples.
  - Weights the previous average at 99.9%.
- *Hint: Recompute this value each time a new sample point is read. You only need the previously calculated average value and the latest sample. It has nothing to do with the samples stored in the history. Make sure you set it correctly on the first sample.*

Hints (optional, but recommended):

Your module which samples the light levels should likely have functions to:

- Start the sampling thread (done once at startup, such as an `init()` function).
- Stop the sampling thread (done once at program end, such as a `cleanup()` function).
- Encapsulate the sample buffer and all changes to it inside the module.
  - You can decide how to represent light values (raw A2D readings, or voltages).
  - You do *not* need to convert to lux (luminosity).
- Provide access to the history. The history is the samples taken during the previous 1s.
  - This is used by the UDP networking code and light level analysis code (later).
- Provide the current average (filtered with exponential smoothing) light level.
- Make sure that all parts of your code are thread-safe. Consider what data will be used outside of the sampling thread (and how). How can you synchronize access?
  - Consider which modules and threads will be reading/changing the memory and variables.
  - For modules that need a thread, have that module create a thread and manages interactions with its data via accessor/mutator functions. Make these functions lock/unlock the necessary mutex.
  - Big Hint: Here is a possible .h file interface you may use. You may use this exactly, ignore it completely, or modify it in any way. Notice how it internalizes all use of the background thread, isolating the rest of the code from having to worry about threads.

- Use the provided hardware checking app to ensure your light sensor is wired correctly.

1 The idea is to store the *previous* second's data in the history, rather than to require that the application always provide access to all samples between the present moment and one second ago. Since the app can work with the "previous" second, you can once per second copy/move the samples that have been collected from the current buffer into the history buffer. All analysis and print-out features can then use this previous second of data.
This feature should reduce the amount of work you need to do because you don't have to write a circular buffer.

```c
// sampler.h
// Module to sample light levels in the background (uses a thread).
//
// It continuously samples the light level, and stores it internally.
// It provides access to the samples it recorded during the _previous_
// complete second.
//
// The application will do a number of actions each second which must
// be synchronized (such as computing dips and printing to the screen).
// To make easy to work with the data, the app must call
// Sampler_moveCurrentDataToHistory() each second to trigger this
// module to move the current samples into the history.

#ifndef _SAMPLER_H_
#define _SAMPLER_H_

// Begin/end the background thread which samples light levels.
void Sampler_init(void);
void Sampler_cleanup(void);

// Must be called once every 1s.
// Moves the samples that it has been collecting this second into
// the history, which makes the samples available for reads (below).
void Sampler_moveCurrentDataToHistory(void);

// Get the number of samples collected during the previous complete second.
int Sampler_getHistorySize(void);

// Get a copy of the samples in the sample history.
// Returns a newly allocated array and sets `size` to be the
// number of elements in the returned array (output-only parameter).
// The calling code must call free() on the returned pointer.
// Note: It provides both data and size to ensure consistency.
double* Sampler_getHistory(int *size);

// Get the average light level (not tied to the history).
double Sampler_getAverageReading(void);

// Get the total number of light level samples taken so far.
long long Sampler_getNumSamplesTaken(void);

#endif
```

1.2 Listening to UDP

Listen to port 12345 for incoming UDP packets (use a thread). Later sections described some values. Treat each packet as a command to respond to: reply back to the sender with one or more UDP packets containing the "return" message (plain text).

Accepted commands

- `help`
  - Return a brief summary/list of supported commands.
- `?`
  - Same as `help`
- `count`
  - Return the total number of light samples take so far (may be huge, like > 10 billion).
- `length`
  - Return how many samples were captured during the previous second.
- `dips`
  - Return how many dips were detected during the previous second's samples.
- `history`
  - Return all the data samples from the previous second.
  - Values must be the voltage of the sample, displayed to 3 decimal places.
  - Values must be comma separated, and display 10 numbers per line.
  - Send multiple return packets if the history is too big for one packet.
    - You can assume that 1,500 bytes of data will fit into a UDP packet. This works across Ethernet over USB.
    - No single sample may have its digits split across two packets.
- `stop`
  - Exit the program.
  - Must shutdown gracefully: close all open sockets, files, pipes, threads, and free all dynamically allocated memory.
- <enter>
  - A blank input (which will actually be a line-feed) should repeat the previous command. If sent as the first command, treat as an unknown command.
- All unknown commands return a message indicating it's unknown.

Error Handling

- You do not need to do extensive error checking on the commands. For example, it is fine to return the help message for the command "`help me now!`"
- Lower case commands must be accepted; optional to make it case insensitive.
- No command should be able to crash your program ("`stop`" will stop it; not crash it).

Testing

- Use the `netcat` (`nc`) utility from your host:
  `(host)$ netcat -u 192.168.7.2 12345`
- To exit `netcat` on the host, you'll have to press Control-C ("`stop`" only kills the program on the target). Or, press enter a couple times when it is not connected to a server (the target) for `netcat` to exit.

Sample output on the host via `netcat`

- Commands sent from host are shown here in bold-underlined for ease of reading. The history output was trimmed to fit the page.
- The "just pressed enter" is added here; it's not part of the actual output.
- The history output has 20 numbers per line (word-wraps in this document).
- Your output need not exactly match the sample, but it must have the same elements.

```
brian@PC-debian:~$ netcat -u 192.168.7.2 12345
help

Accepted command examples:
count       -- get the total number of samples taken.
length      -- get the number of samples taken in the previously completed
second.
dips        -- get the number of dips in the previously completed second.
history     -- get all the samples in the previously completed second.
stop        -- cause the server program to end.
<enter>     -- repeat last command.
?

Accepted command examples:
count       -- get the total number of samples taken.
length      -- get the number of samples taken in the previously completed
second.
dips        -- get the number of dips in the previously completed second.
history     -- get all the samples in the previously completed second.
stop        -- cause the server program to end.
<enter>     -- repeat last command.
count
# samples taken total: 9670
count
# samples taken total: 11589
                                                    <just pressed enter>
# samples taken total: 12343
length
# samples taken last second: 549
                                                    <just pressed enter>
# samples taken last second: 552
dips
# Dips: 21
history
1.383, 1.362, 1.342, 1.311, 1.298, 1.288, 1.280, 1.265, 1.347, 1.491,
1.531, 1.556, 1.567, 1.577, 1.584, 1.587, 1.588, 1.589, 1.591, 1.593,
1.594, 1.593, 1.559, 1.515, 1.470, 1.440, 1.405, 1.380, 1.360, 1.343,
1.318, 1.300, 1.291, 1.285, 1.272, 1.252, 1.466, 1.519, 1.550, 1.564,
1.573, 1.580, 1.585, 1.587, 1.589, 1.590, 1.593, 1.593, 1.593, 1.586,
1.537, 1.497, 1.453, 1.418, 1.390, 1.365, 1.349, 1.328, 1.306, 1.291,
1.286, 1.277, 1.258, 1.425, 1.515, 1.546, 1.562, 1.575, 1.580, 1.586,
1.588, 1.589, 1.590, 1.593, 1.593, 1.594, 1.588, 1.538, 1.494, 1.465,
1.432, 1.399, 1.378, 1.361, 1.339, 1.314, 1.298, 1.292, 1.284, 1.272,
1.279, 1.457, 1.521, 1.550, 1.565, 1.575, 1.580, 1.584, 1.589, 1.590,
1.592, 1.594, 1.593, 1.593, 1.563, 1.506, 1.476, 1.438, 1.406, 1.374,
                          ... trimmed ...
1.354, 1.331, 1.308, 1.297, 1.287, 1.278, 1.260, 1.374, 1.495, 1.534,
1.556, 1.568, 1.578, 1.584, 1.587, 1.588, 1.589, 1.593, 1.593, 1.593,
1.592, 1.557, 1.515, 1.468, 1.438, 1.404, 1.380, 1.361, 1.342, 1.316,
1.300, 1.292, 1.284, 1.272, 1.251, 1.466, 1.519, 1.549, 1.564, 1.574,
1.580, 1.584, 1.588, 1.589, 1.590, 1.593, 1.593, 1.594, 1.586, 1.537,
1.498, 1.451, 1.418, 1.390, 1.369, 1.352, 1.323, 1.303, 1.295, 1.286,
1.277, 1.257, 1.408, 1.504, 1.542, 1.559, 1.570, 1.578, 1.585, 1.587,
1.589, 1.591, 1.593, 1.593, 1.593, 1.593, 1.541, 1.490, 1.455, 1.420,
1.388, 1.371, 1.353, 1.329, 1.303, 1.295, 1.286, 1.275, 1.255, 1.425,
1.510, 1.545, 1.564, 1.573, 1.580, 1.585, 1.588, 1.589, 1.590, 1.593,
1.594, 1.593, 1.589, 1.537, 1.498, 1.462, 1.429, 1.396, 1.375, 1.356,

stop
Program terminating.
```

1.3 Analyze History for Light Dips

Each second, the program must analyze the light samples that were captured by during the previous second (the history). It must count the number of dips in the light level that are revealed by the samples during that second. A dip in light level can be caused by blocking the light to the sensor for a brief moment (such as waving your hand across over top of the board). The number of light dips found in the previous second's data is reported on the terminal (section 1.4), the 14-seg display (section 1.6), and via UDP (section 1.2).

- A dip is when the light level drops below a threshold (a certain amount below the current average light level).
  - Another dip cannot be detected until the light level returns above the threshold.
  - A dip can be detected when the voltage is 0.1V or more away from the current average light level. The current average light level is computed using exponential smoothing (section 1.1).
  - Carefully consider if a lower light level leads to smaller voltages, or larger voltages.
  - Use hysteresis of 0.03V to prevent re-triggering a dip incorrectly if there is some noise in the readings.
    - i.e., The light level must drop by 0.1V to trigger a dip.
      It cannot retrigger another dip until the light level first rises to 0.07 below the average light level (or higher).
      Then if the light-level drops back below 0.1V it retriggers a dip.
- *Testing hint:*
  - *Wave your hand over the board once, interrupting the light, and it should detect 1 dip.*
  - *Wave back and forth; it should detect 2 dips, and then count down.*
  - *Wave fast and see!*
  - *Install a strobe light program on your phone! Section 1.5 adds a flashing LED to test it.*

1.4 Terminal Output

Each second, print the following to the terminal (via `printf()`). Use a fixed number of characters for each value so the alignment does not change.

- Line 1:
  - # light samples taken during the previous second
  - Raw value from the POT, and how many hertz (Hz) this is for the PWM (section *1.5)*
  - The averaged light level (from exponential smoothing), displayed as a voltage with 3 decimal places.
  - The number of light level dips that have been found in samples from the previous second
  - Timing jitter information (provided by `periodTimer.h/.c`) for samples collected during the previous second (section 1.7)
    - Minimum time between light samples.
    - Maximum time between light samples.
    - Average time between light samples.
    - Number of times sampled
    - Format:
      `Smpl ms[{min}, {max}] avg {avg}/{num-samples}`
- Line 2:
  - Display 20 sample from the previous second.
  - These values must be as evenly spaced across the collected samples as possible (for example, if you have ~100 samples collected, display every ~5th sample). If there are less than 20 samples from the previous second, display all the values.
  - Format: `{sample number}:{value}`

Hint: Use a thread to do the analysis and print these values and then sleeps for 1s; or combine these tasks with another module that runs every second.

Sample output

```
#Smpl/s = 547    POT @ 578 => 14Hz   avg = 1.411V    dips =  15    Smpl ms[ 1.361,  6.433] avg 1.848/547
  0:1.330   55:1.586  109:1.495  164:1.251  219:1.596  274:1.317  328:1.581  383:1.500  438:1.250  492:1.596
#Smpl/s = 552    POT @ 560 => 14Hz   avg = 1.432V    dips =  15    Smpl ms[ 1.361,  5.197] avg 1.829/552
  0:1.274   55:1.594  110:1.380  166:1.557  221:1.598  276:1.256  331:1.593  386:1.334  442:1.582  497:1.487
#Smpl/s = 554    POT @ 497 => 12Hz   avg = 1.435V    dips =  13    Smpl ms[ 1.367,  6.718] avg 1.825/554
  0:1.495   55:1.596  111:1.183  166:1.403  222:1.596  277:1.423  332:1.248  388:1.590  443:1.531  499:1.273
#Smpl/s = 549    POT @ 122 =>  3Hz   avg = 1.361V    dips =   4    Smpl ms[ 1.460,  6.141] avg 1.848/549
  0:1.458   55:1.189  110:1.181  165:1.358  220:1.181  275:1.301  329:1.175  384:1.598  439:1.177  494:1.165
#Smpl/s = 550    POT @   0 =>  0Hz   avg = 1.276V    dips =   1    Smpl ms[ 1.462,  7.070] avg 1.842/550
  0:1.161   55:1.160  110:1.159  165:1.163  220:1.171  275:1.167  330:1.156  385:1.153  440:1.150  495:1.150
#Smpl/s = 549    POT @ 454 => 11Hz   avg = 1.313V    dips =   7    Smpl ms[ 1.453,  4.708] avg 1.839/549
  0:1.186   55:1.179  110:1.300  165:1.366  220:1.174  275:1.333  329:1.342  384:1.338  439:1.330  494:1.176
#Smpl/s = 549    POT @ 440 => 11Hz   avg = 1.347V    dips =   9    Smpl ms[ 1.454,  6.447] avg 1.842/549
  0:1.239   55:1.224  110:1.566  165:1.589  220:1.595  275:1.568  329:1.190  384:1.575  439:1.195  494:1.375
#Smpl/s = 550    POT @ 442 => 11Hz   avg = 1.390V    dips =  12    Smpl ms[ 1.404,  5.246] avg 1.839/550
  0:1.268   55:1.240  110:1.372  165:1.580  220:1.594  275:1.597  330:1.598  385:1.462  440:1.338  495:1.273
```

1.5 POT controlling LED via PWM

Use the Zen cape's potentiometer (POT) to select the frequency at which to flash the LED.

- POT Reading
  - Ten times per second, read the Zen cape's potentiometer (via analog to digital: A2D). This will give you a number between 0 and 4095 (4k different values) inclusive.
- PWM LED Control
  - Using the provided guides, wire up a discrete LED (one of the lose components). Position the LED pointing at (or at least parallel to) the light sensor so that it sees the LED's light. Any of the 2-lead (two wires) LEDs should be fine.
  - Drive the LED using PWM (pulse width modulation),
  - Compute the desired PWM frequency by dividing the raw A2D reading of the Zen cape's POT by forty (40). Update the PWM frequency each time the POT is read.
  - If the PWM is already set to the desired frequency, do not re-set it because this will cause a brief interruption to the LED's flashing. i.e., if it's at 10Hz and you are setting it to 10Hz, do nothing.

1.6 14-Seg Display via I2C

Use the Zen cape's two digit 14-segment display to display the number of dips that the program has detected using the samples collected during the previous second.

- Display on the 2-digit 14 segment display the number of dips in the previous second.
  - Update this value once per second.
  - Display the number in base 10.
  - If the number to display is greater than 99, display 99 instead (hard to achieve!)
- For values < 10, you may prefix with a 0 or a blank.
  - For example, if the number is 5, you may display either "` 5`" or "`05`"
- You must figure out what bits to set for the 14-seg display to show the characters you want.
  - Consider setting one bit active at a time and recording which segment it lights up.
  - After you have mapped all the segments to bit patterns, you can then use this to construct the expected bit-patterns.
  - *Hint: Write a for-loop that turns on the bits one at a time and then waits for user input to move to the next so you can record what LED segment is turned on.*
- You may **not** assume the I2C pins are already configured for I2C use.
  - Your program should make a call to the `config-pin` program to control the pin setup for `P9_17` and `P9_18`, as described in the I2C guide.

1.7 Timing Jitter

The provided `periodTimer.h/.c` files are a ready-to-use module which supports recording the timestamps of certain events, and then calculating some statistics about those timestamps.

You may modify these files as needed for recording the timing (see below); **however, you may not use this file to store your light-level samples: it is only for timing.**

Expected Usage

1. In `periodTimer.h`, create your own category of event in the `Period_whichEvent enum`. Note that you can use the provided one, and/or create your own.
2. At startup, call `Period_init()`.
3. Each time the event of interest happens in your code, call the `Period_markEvent()` function. Pass in the `enum` for the event you are tracking. The module will record the timestamp for this event (stored internally).
4. Each time you want the current statistics, call the `Period_getStatisticssAndClear()` function for your event. Note that when called, this function will wipe out the timestamps that it has been storing for this event (hence not double-counting a time-stamp the next time you call this).
5. When shutting down, call `Period_cleanup()`.

2. Debug program "noworky"

The file `noworky.c` is provided on the course website. This program does not do what its comments say it will. You must debug it and fix it. The tool `gdb` will be discussed in class.

- Cross-compile the `noworky` for the target.
  - Compile using `-g` option (include debug symbols) in `gcc`. Recommended flags are:
    `-Wall -g -std=c99 -D _POSIX_C_SOURCE=200809L -Werror`
  - `noworky` generates the following output shown on the right.

```
[root@Boardcon bin]# ./noworky
noworky: by Brian Fraser
Initial values:
  0: 000.0 --> 000.0
  1: 002.0 --> 010.0
  2: 004.0 --> 020.0
  3: 006.0 --> 030.0
  4: 008.0 --> 040.0
  5: 010.0 --> 050.0
  6: 012.0 --> 060.0
  7: 014.0 --> 070.0
  8: 016.0 --> 080.0
  9: 018.0 --> 090.0
Segmentation fault
[root@Boardcon bin]#
```

- Use `gdbserver` and the `gdb` cross-debugger to debug `noworky`.
  - Do a full debugging session using the `gdb` text debugger. Using copy-and-paste, copy the full text of your debugging session into `as2-gdb.txt`. Your debugging session must show the bug, where it is, and how you (could reasonably have) found it.
  - Even if you used a graphical debugger initially to figure out the bug, you must still use `gdb` to re-investigate the problem and show a full debugging process (not just a listing on the program and say "There's the problem!")
- Setup a graphical cross-debugger (such as Eclipse or VS Code). Use it to re-debug `noworky` in a cross-debugging configuration (target device runs `noworky`, host runs the graphical debugger)
  - Create a screen shot named `as2-graphical.png` showing the graphical debugger debugging the program. If using Eclipse, this should show the debug perspective; if using VS Code show the “Run and Debug” view.
  - A single screenshot won't show your full debugging session, but it proves that there *was* a graphical debugging session, which is good enough for this assignment. Just make the screen-shot show some representative part of your debugging session.
- Correct the bug in `noworky.c` and comment your change. For example,
  `// Bug was here: It was doing.... but should be doing....`
  - *Hint: The fix is no more than a one word change!*
  - Submit the corrected `noworky.c` file to CourSys along with the screenshot and trace.

3. Deliverables

Submit the items listed below to the CourSys: https://coursys.sfu.ca

1. `as2.tgz`
   Compressed copy of your project. Delete the `build/` folder first so it's much smaller.

   Hint: Compress the as1/ directory with the command like:
   ```
   (host)$ tar cvzf as2.tgz as2
   ```
2. `as2-gdb.txt`
3. `as2-graphical.png`

Please remember that all submissions will be compared for unexplainable similarities. Please make sure you do your own original work; and not copied from GitHub. I have copies of all coed previously submitted, such as all copies on GitHub, and have a very efficient tool to find similarities. Please take this opportunity to do great work!

3.1 Informal Milestones

- **How to start**
  - If you want to work with a partner, start looking for one!
  - Follow the guides to get started:
    - I2C to get the 14-seg display working
    - A2D to read the Zen's POT
    - Wiring guide for the light-sensor (A2D)
    - PWM guide to learn how to control the light. Start with the Zen cape's buzzer to learn PWM, then follow wiring guide for the LED connected to the PWM
  - Read all sections of the assignment. Design HAL modules. Think about what modules your application will have.
  - Use the provided hardware test app to ensure light sensor and LED are wired correctly.
  - Try designing and coding the sections of this assignment one at a time.
    - When you get the light sampling and UDP working, use the provided Python program (run on the host) to show a graph of the samples you have collected.
  - Think about how you will shutdown the application correctly.
- **Half done**
  - Sampling light levels.
  - Sending history samples via UDP, checked with Python program.
  - Maybe also counting dips in the last second of data?
- **Final checks**
  - Review the learning objectives for this assignment (see webpage).
  - Complete the `noworky` debugging.
  - Double check your final ZIP file for correctness! No last minute refactoring bugs!

3.2 Revision History

- V1: initial version