Assignment 2: Embedded Linux Programming

- Submit deliverables to CourSys: https://courses.cs.sfu.ca/
- Late penalty is 10% per calendar day, maximum 2 days late.
- 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, or via email to cmpt-433-help@sfu.ca
- See the marking guide for details on how each part will be marked.

1. Write "`sorter`" Program

Write a C program (not C++) named `sorter` which runs on the target to sort arrays of integers and listen to a UDP socket. Plus, use the Zen cape's potentiometer to allow the user to select the size of array to sort. Use the Zen cape's 2-character (14-segment display) to display the number of arrays sorted per second. Your program must 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).

Think of this assignment as a number of mini-tasks put together. "`sorter`" uses different skills:
- C-programming with `malloc()` / `free()`
- pthread threads and thread synchronization
- UDP networking
- Analog to digital (A2D) reading and piece-wise-linear functions
- Output on a 14-segment display over I2C

I strongly encourage you to start early on the assignment and work on one part at a time. I recommend starting with the how-to guides. Try to design well encapsulated modules as it's easier to do some design up front than just refactor later. You may still need to refactor your code as you add functionality, but this will be easier than trying to complete the entire assignment at once.

1.1 Thread Sorting Arrays

In a separate thread (using p-threads), continually sort arrays of integers:
- Dynamically create an array of integers to be the array to sort.
  - The length of the array is controlled by the POT (see section 1.3); for now, just default to 100.
  - You must use `malloc()` to allocate each array.
- Initialize the array to be a random permutation of the numbers 1 through the size of the array. You must write your own permutation code, not copy code from online.
  - Hint: You can initialize the array to contain values 1 through the size of the array, and then for each element in the array, swap it with a random element in the array.
  - Hint: `rand()` returns a pseudo-random number (look it up). You don't need to worry about seeding your random number generation with `srand()` if you don't want to.
- Sort the array using bubble sort.
  - Yes, I did just say *bubble sort*. And yes, I do know it's O(n^2). Make sure you get the right algorithm!
- When finished sorting the array, free the memory.
- Count the total number of arrays which have been sorted.

Your module which sorts must support:

- Starting the above mentioned sorting thread (done once at startup).
- Stopping the sorting thread (done once at program end).
- Setting the size of the next array to sort.
- Provide access to the current state of the array being sorted (i.e., its current contents).
  - This is used by the UDP networking code.
  - This must give access to the current partially sorted array (not just after the sort has finished).
  - The data in the array the module returns must be a valid permutation (otherwise, your program has a threading bug).
- Make sure that all parts of your code are thread-safe. Consider what data will be used outside of this sorting thread (and how), and what needs to be done to synchronize access.
  - Hint: Consider who will be reading/changing the memory and variables.
  - How does this fit in with the modular design of your program? Are you able to localize all of thread synchronization code into a single module? A module may be called by multiple threads.
  - Hint: Have an array sorting module which internally spawns the background thread and then manages interacting with that thread.
  - *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.

```
// sorter.h
// Module to spawn a separate thread to sort random arrays
// (permutations) on a background thread. It provides access to the
// contents of the current (potentially partially sorted) array,
// and to the count of the total number of arrays sorted.
#ifndef _SORTER_H_
#define _SORTER_H_

// Begin/end the background thread which sorts random permutations.
void Sorter_startSorting(void);
void Sorter_stopSorting(void);

// Get the size of the array currently being sorted.
// Set the size the next array to sort (don't change current array)
void Sorter_setArraySize(int newSize);
int Sorter_getArrayLength(void);

// Get a copy of the current (potentially partially sorted) array.
// Returns a newly allocated array and sets 'length' to be the
// number of elements in the returned array (output-only parameter).
// The calling code must call free() on the returned pointer.
int* Sorter_getArrayData(int *length);

// Get the number of arrays which have finished being sorted.
long long Sorter_getNumberArraysSorted(void);

#endif
```

1.2 Thread Listening to UDP

In a new thread, listen to port 12345 for incoming UDP packets.

- Treat each packet as a command to respond to: your software will 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.
  - `count`
    - Return the number of arrays sorted so far (must support > 10 billion).
  - `get #`
    - Return the requested value from inside the current array being sorted. For example, "`get 10`" returns the tenth value in the array currently being sorted (1-indexed!). If the number is less than 1, or greater than the length of the current array being sorted, it is an error (see below).
  - `get length`
    - Returns the length of the array currently being sorted.
  - `get array`
    - Returns all the data in the current array being sorted.
    - Values must be comma separated, and display ten (10) numbers per line.
    - You may need to send multiple return packets if the array too big for one packet.
      - 1,500 bytes of data in a UDP packet is a reasonable amount and works across Ethernet over USB.
      - No value in the array may have its digits split across two packets.
  - `stop`
    - Stop sorting arrays and exit the program.
    - Should shutdown gracefully and close all open sockets, files, pipes, threads, and free all dynamically allocated memory.
  - All unknown commands print a message indicating it's unknown.
- **Error Handling**
  - Ensure that the requests don't request unknown elements of the array. For example, "get 101" when the current array length is 100 should display a meaningful error message which specifies the range of values that are acceptable.
  - 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!`"
  - You may make commands case sensitive or insensitive, but you must accept at least lower-case commands.
  - 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 to connect to `sorter`.
    ```
    $ netcat -u 192.168.7.2 12345
    ```
  - Now you can type commands on the host and `sorter`'s responses will be shown.
  - To exit `netcat` on the host, you'll have to press Control-C ("`stop`" only kills sorter, not the `netcat` client). Or, you can press enter a couple times when it is not connected to a server (the target).
- **Sample output on the host via `netcat`**
  - Commands sent from host are shown here in bold-underlined for ease of reading.
  - Your output need not exactly match the sample, but it must have the same elements.

```
brian@ubuntu:~$ netcat -u 192.168.7.2 12345
help
Accepted command examples:
count      -- display number arrays sorted.
get length -- display length of array currently being sorted.
get array  -- display the full array being sorted.
get 10     -- display the tenth element of array currently being sorted.
stop       -- cause the server program to end.
count
Number of arrays sorted = 2780.
get length
Current array length = 150.
get array
1, 2, 3, 4, 5, 6, 10, 11, 8, 12,
13, 14, 9, 15, 16, 17, 19, 20, 21, 22,
23, 24, 18, 25, 26, 27, 28, 29, 30, 7,
31, 32, 33, 34, 35, 36, 38, 39, 40, 41,
42, 43, 44, 37, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
101, 102, 103, 104, 105, 106, 107, 108, 109, 110,
111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
121, 122, 123, 124, 125, 126, 127, 128, 129, 130,
131, 132, 133, 134, 135, 136, 137, 138, 139, 140,
141, 142, 143, 144, 145, 146, 147, 148, 149, 150
get array
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
101, 102, 103, 104, 105, 106, 107, 108, 109, 110,
111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
121, 122, 123, 124, 125, 126, 127, 128, 129, 130,
131, 132, 133, 134, 135, 136, 137, 138, 139, 140,
141, 142, 143, 144, 145, 146, 147, 148, 149, 150
get 15
Value 15 = 17
get 15
Value 15 = 24
get 15
Value 15 = 69
count
Number of arrays sorted = 8658.
count
Number of arrays sorted = 8849.
get 0
Invalid argument. Must be between 1 and 150 (array length).
get 151
Invalid argument. Must be between 1 and 150 (array length).
stop
Program Terminating
```

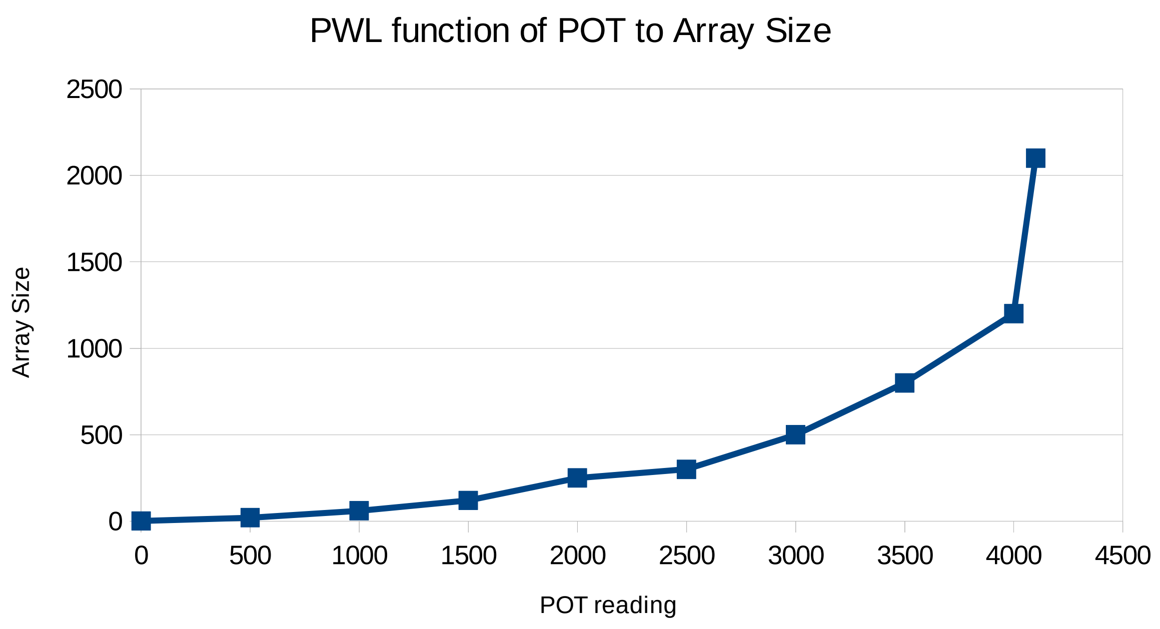
1.3 POT via Analog to Digital

Use the Zen cape's potentiometer (POT) to select the size of array to be sorted.

- Once 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 varues) inclusive.
- Convert this reading to the array size via the following piece wise linear (PWL) function:
  - Data Points:

A2D Reading	0	500	1000	1500	2000	2500	3000	3500	4000	4100
Array Size	1	20	60	120	250	300	500	800	1200	2100

  - Graph of the conversion function:



- Set the array size by calling a function on your sorting module.
  - Hint: Your sort module should allow the current array to be sorted as is (using the previous size), and then use the new size when allocating the next array to sort.
- Each time a new array size is calculated (i.e., once per second), display the new array size to the screen (via `printf()` to `stdout`). You may also display other values as well if you like, but output must be clear and easy to read during program execution.

1.4 14-Seg Display via I2C

Use the Zen cape's two digit 14-segment display to display the number of arrays that the program finished sorting in the previous second.

- Once per second, calculate the number of arrays sorted in the previous second.
  - Hint: May be reasonable to integrate this with the POT code's thread above.
  - Hint: You should have a function in your sorting module to give you the number of arrays sorted so far. Use this to compute the number sorted in the last second.
  - Hint: If it takes more than one second to sort a single array, then the second when an array finished sorting will show "1", otherwise show "0'.
- Display the number of arrays sorted in the last second on the 2-digit 14 segment display.
  - Display the number in base 10.
  - Hint: Follow the I2C guide first!
  - If the number to display is greater than 99, display 99 instead.
  - If the number is 5, you may display either " 5" or "05" (i.e., a 5 on the right, and either nothing or a '0' on the left).

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 Eclipse (or any other graphical debugger) as the graphical cross-debugger. Use it to re-debug `noworky` in a cross-debugging configuration (target running `noworky`, host running 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.
  - 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://courses.cs.sfu.ca/

1. `as2-sorter.tar.gz`
   Compressed copy of source code and build script (`Makefile`).

   Archive must expand into the following (without additional nested folders to find the `Makefile`)

   ```
   <as2 directory name>
    |-- Makefile
    |-- noworky.c    (debugged and corrected)
    \-- <dependencies such as .c, .h files>
   ```

   `Makefile` must support both the `` `make` `` and `` `make all` `` commands to build `noworky.c` and your sorter program to `$(HOME)/cmpt433/public/myApps/`
   (Do not use relative paths for getting to the `cmpt433/public/myApps/` directory because the TA may build from a different directory than you.)

   Hint: Compress the `as2/` directory with the command

   ```
   $ tar cvzf as2-sorter.tar.gz as2
   ```

   You may use a different build system than make (such as CMake). If you do, include a file named `README` which describes the commands the TA must execute to install the necessary build system under Ubuntu 16.04, and the commands needed to build and deploy your project to the `~/cmpt433/public/myApps/` directory. The process must be straightforward and not much more time consuming than running `make`.

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; I will be checking.