# Assignment 2: Debug & Light Sampling

- 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.

## *Version History*

1: Initial version

2: Corrected A2D pinout for Joystick

3: Corrected pins for LED Matrix; expanded hints for analysis; minor edits.

 

## 1. 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 -Wshadow`
  - `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 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.

 

# 2. Light Dip Analyzer

## 2.1 Overview

Your application will:

- Sample the current light level (using a photoresistor) ~500+ times a second.
- Analyze the collected samples about once a second to compute some useful statistics.
- Print some useful information to the screen (terminal) each time the statistics are computed.
- Display useful statistics on the 8x8 LED matrix. By pressing the joystick in different directions, the user can select which statistic is displayed.
- Exit when the user presses the USER/BOOT button on the BBG.

After the requirements sections (next), some detailed development suggestions are listed to get you coding!

## 2.2 Software Design

Your program must:

- Use good style
  - All modules, files, functions, and variables are named well
  - All `.h` files have a comment at the top
  - All code has excellent indentation

  You don't need to match the course's style guide, you just need to write beautiful code
- Have a clean modular design:
  - No externally linked global variables
  - Functions meant to be used by other modules are externally linked and listed in .h file
  - Functions not meant to be used by other modules are internally linked
  - Thread IDs and mutextes are encapsulated inside modules (not available from other modules)
  - Multiple modules, each with its own .h and .c file. Expected ~5 to ~20 modules (each with a .h/.c file).
- Be multi-threaded and threadsafe (no race-cases; use mutexes to protect critical sections)
- Use *no* busy waits (use sleep() or mutexes instead)
- Be written in C, or Rust (requires guidance to TAs for compiling and running)
- Use a `makefile` to build your application and copy it to `$(HOME)/cmpt433/public/myApps/light_sampler`

## 2.3 Hardware Wiring

The following is the required hardware configuration for completing this assignment.

- **Photoresistor** (see video)
  - P9.32 (`VDD_ADC`) to one side of photoresistor
  - P9.40 (`AIN1`) to other side of photoresistor, and 10k resistor
  - P9.34 (`GNDA_ADC`) to other side of 10k resistor
  - Basically, this is a voltage divider with the photoresistor and a 10k resistor in series. Connect 1.8V A2D ref voltage (`VDD_ADC`) as high voltage, connect A2D ground (`GNDA_ADC`) as low voltage, and read the middle point with the A2D (`AIN1`).
- **Joystick** (see A2D Guide)
  - P9.32 (`VDD_ADC`) to +V to the joystick
  - P9.34 (`GNDA_ADC`) to -V to the joystick
  - P9.37 (`AIN2`) to X output on the joystick

 

- P9.38 (`AIN3`) to Y output on the joysticks
- Place joystick with breakout board's text ("Analog Mini Thumbsick") to the left, and its "`+ Y X -`" pins at the top and bottom.

- **8x8 LED Matrix** (see I2C Guide)
  - P9.01 (`Ground`) to "-" pin on the LED Matrix
  - P9.03 (`3.3V`) to "+" pin on the LED Matrix
  - P9.17 (`I2C1-SCL`) to "C" pin on the LED Matrix
  - P9.18 (`I2C1-SDA`) to "D" pin on the LED Matrix
  - **Orient display with the pins at the bottom, and the display pointing out the top of the breadboard**. (This is 180 degree rotation, with text on the "backpack" board upside down).
- **User Button** (see GPIO guide)
  - Already wired on the BBG, see USER/BOOT button.

Use the provided hardware verification app to ensure that your hardware wiring matches the required setup. This wiring setup will be used by the TAs for marking, so your wiring should match it 100%.

## 2.4 Sampling

- Create a buffer to store photoresistor samples
- Read the photoresistor (A2D) in a thread with the following steps:
  - Read the A2D
  - Store the value in the buffer
  - Sleep 1ms
- The samples only need to be stored for ~1s: they will be read by the analysis module and then discarded. So, you can use a single array to store the values as you read them each second; you don't need a circular buffer or any more complicated data type.
- For each light sample, record:
  - the A2D voltage reading, and
  - the timestamp of when the reading happened (microsecond or nanosecond accuracy)
- When samples are extracted by the analysis module, clear the buffer and restart filling it with the next read. Make the buffer big enough so it won't be filled during a second.

## 2.5 Sample Analysis & Printing

Each second, compute the following using the samples that have been taken since the last second (the previous time analysis was performed):

- Number of samples analyzed during this second (likely between 500 and 1000 samples)
- Average sample voltage, using an exponential averaging function for all samples seen so far (not just this second's worth of samples)
  - For exponential averaging, weight the previous average at 99.9%
  - Use the first reading as the initial value
- Minimum and maximum sample value (in volts)
- Number of dips in the light level this second (see below for details)
- Minimum, maximum, and average interval time between samples
  - Computed based on the time between one sample and the next
  - Must be computed for each sample and the next, starting from the last sample of the previous second (if any), through to the most recent sample for this second.
  - *Hint: store the timestamp of the last sample of the previous second so you can use it when analyzing the next second's data.*

After computing the statistics above, display them to the screen (`printf()`) using the following format (all on one line)

- Interval ms (<min interval>, <max interval>) avg=<avg interval>
- Samples V (<min sample V>, <max sample V>) avg = <exponential averaged voltage>
- # Dips: <number dips this second>
- # Samples: <number samples this second>

See sample output on website.

### *Computing Dips*

The program must analyze the light samples for this second and count the number of dips in the light level that are revealed by these samples. 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), or by using a strobelight app. The number of light dips found is reported on the terminal and the 8x8 LED matrix.

- A dip is when the light level drops below a threshold (a certain amount away from 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.
  - Use hysteresis of 0.03V to prevent re-triggering a dip incorrectly if there is some noise in the readings.
- *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.*
  - *Wave fast and see!*
  - *Install a strobelight program on your phone; should work up to at least 20hz.*

## 2.6 Display & Joystick

- Display must be able to show:
  - integer between 0 and 99
  - floating point between 0.0 and 9.9
- Values bigger than maximum should show 99 or 9.9 (for integer or floating point)
- Based on the direction the user presses the joystick, display the following on the 8x8 LED matrix. Each value is the value calculated the last time the board analyzed light samples.
  - Joystick released (centred): number of dips (integer)
  - Joystick UP: maximum sample voltage (in V, floating point)
  - Joystick DOWN: minimum sample voltage (in V, floating point)
  - Joystick LEFT: minimum interval between samples (in ms, floating point)
  - Joystick RIGHT: maximum interval between samples (in ms, floating point)
- If multiple directions are pressed at once (up and left), then arbitrarily pick one.
- Use a reasonable threshold on the joystick position to make it reliable. Suggestion is if you map the joystick reading onto the interval [-1.0, 1.0], then make values less than -0.5, or greater than 0.5 register as being pressed.

## 2.7 USER Button to Exit

When the user presses the `USER` (also called `BOOT`) button on the BBG, exit the program.

- It's OK to poll the button every ~100ms to see if it is pressed.
  - So, you may assume the user holds down the button for at least 100ms+, if needed.
  - You don't have to **make** them press and hold the button; it's OK to assume they do.
- The program must exit smoothly by:
  1. Shutting down all running threads
  2. Freeing all dynamically allocated memory
  3. Exiting without any runtime errors (such as a segfault or divide by zero)
- Program may *not* call `exit()`.

## 2.8 Suggested Development Process

Here are some suggestions. You don't need to follow them, but they'll help you get started.

**Start small, test often, test with the state visible (printf or GDB debugger).**

### *Joystick Readings*

- Start by making a module which reads the joystick.
- Wire up the joystick as per the A2D guide.
- Use the hardware test app to ensure it works as expected.
- Give it a function like:
  `Joystick_readXY(double* x, double* y);`
  which uses pass-by-pointer to read the current x and y state of the joystick; or perhaps:
  `double Joystick_readX();`
  `double Joystick_readY();`
- To test, create a loop in `main()` which prints these readings to the screen.
- Note the joystick must be mounted sideways on the breadboard, so the joystick's physical X and Y must be changed to map to our app's needs (common with hardware designs!) Use the hardware test application to ensure yours is wired the expected way.

### *8x8 LED Matrix*

This can take some time; write a good module that makes your life easy now and in the future.

- Wire up the LED matrix and use the Linux commands to get it working (see I2C guide).
- Use the hardware test app to ensure it works.
- Create a module for the 8x8 LED matrix and design the `.h` file's functions.
  - You'll need to display integers between 0 and 99 (inclusive)
  - You'll need to display floating point values between 0.0 and 9.9 (inclusive)
  - I suggest having a function which accepts an `int` parameter and displays it, and another one which accepts a `double` value and displays it.
- Create the initialization function to write to the I2C file the necessary commands to initialize the display.
- Create some test code to prove that you can display patterns on the LED matrix (perhaps iterate through and turn on each LED one at a time. This ensures your low-level control routines work, and shows you how to access each LED.
- In an array in your module's `.c` file, define the bits for each needed digit (0-9, '.' and space).
  - Design each digit to be 3 columns wide, and 7 rows tall

- Use the bottom row only for the decimal point
- If helpful, define a `struct` data type to hold info about each digit, and store the digits in an array so your program can loop through all digits. This allows your code to handle any digit, and not be hard-coded with big `if` or `case` statements.
- Define just a couple digits ('0', '1', '2') to start; test with these. Define the other digits once everything else works in case you need to change your representation.
- Define each digit in a sane way that you can understand. For example, use 7 bytes in an array: one byte for each row of the digit. Make each bit correspond to a column in the digit. Use a bit of 1 to indicate on, 0 off.
- Consider using binary: 0b101 to easily "see" the digits in the code

- You'll need to compose your individual digits together to make up the LED matrix output. Create a logical "frame" of 8x8 bits (likely an array of 8 bytes, each byte for a row, each bit for a column). This logical frame is independent of LED matrix wiring; it just stores the bits in a reasonable format. Compose your digits, as needed, into this logical frame.
- Map the logical frame to a physical frame that corresponds to the rows and bits of the actual display.
  - This is necessary because the row/column order of the physical hardware does not neatly line up with something we can work with.
  - You'll likely need to rotate some bits from your logical frame's bytes to work for the physical frame.
- Write your physical frame to the LED matrix's registers by writing to the I2C file.
- Test your code by displaying values like 0, 1, 2; 11, 22, ..., 99; 0.0, 1.1, 2.2, ..., 9.9.

## *Light Sampling*

- Wire up the photoresistor as described. It should be read by `in_voltage1_raw`.
- Start the sampling thread (done once at startup).
- Stop the sampling thread (done once at program end).
- Make your sampling thread short and fast.
- Store samples into an array, such as a buffer named `buff`.
- The first sample goes into `buff[0]`, then `buff[1]`, .. etc.
- When the data is extracted, for analysis, clear the buffer and reset the write position back to the start of the buffer so the next sample is stored at the front.
- Encapsulate the 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).
  - You need to store the time of each sample. Consider changing the provided getTimeInMs() function from assignment 1 to instead return microseconds, or nanosecnods.
- Make sure that all parts of your code are thread-safe. Consider what data will be used outside of the sampling thread (and how). What needs to be done to synchronize access.
  - Consider which modules and threads will be reading/changing the memory and variables.
- For modules that need a thread, have the module create a thread and manages interactions with that thread 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.

```
// sampler.h
// Module to sample light levels in the background (thread).
// It provides access to the raw samples and then deletes them.
#ifndef _SAMPLER_H_
#define _SAMPLER_H_

typedef struct {
    double sampleInV;
    long long timestampInNanoS;
} samplerDatapoint_t;

// Begin/end the background thread which samples light levels.
void Sampler_startSampling(void);
void Sampler_stopSampling(void);

// Get a copy of the samples in the sample history, removing values
// from our history here.
// 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.
// Note: function provides both data and size to ensure consistency.
samplerDatapoint_t* Sampler_extractAllValues(int *length);

// Returns how many valid samples are currently in the history.
int Sampler_getNumSamplesInHistory();

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

#endif
```

## *Analysis*

- Create a thread that runs every second.
- Have the thread compute all the statistics of interest.
- Have the thread print out to the screen the statistics.
  - You could have a different thread that does the printing. However, you would need a mechanism to signal from the analysis thread to the printing code because you only want to print once per second, and after the analysis has been completed: you don't want to be half way through printing and have the analysis run and update the values.
- Have your module provide functions to access the desired statistics (no externally linked global variables).
- Make sure that your time-between-samples statistics includes the time between each second (the last sample of the previous second to the first sample of the current second).

## 3. Deliverables

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

1. `as2-gdb.txt`
2. `as2-graphical.png`
3. `as2.tgz`
   Compressed copy of source code and build script (`Makefile`).

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

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

   `Makefile` must support the `make` targets to build `noworky.c` and your 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.tgz 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 20.04, 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`.

Please remember that all submissions will be compared for unexplainable similarities. Please make sure you do your own original work; I will be checking.

 