Assignment 4: Linux Drivers

- Submit deliverables to CourSys: https://courses.cs.sfu.ca/
- This assignment is to be done **individually or in pairs**.
- Post questions to Piazza discussion forum.
- Do not give your work to another student/group, do not copy code found online, and do not post questions about the assignment to other online forums.
- See the marking guide for details on how each part will be marked.

1. Morse Code Driver

In this assignment you will create a Linux kernel driver which will flash Morse code on the Zen cape's LEDs.

Morse Code (see Wikipedia):

- Morse code is a way of transmitting text (such as a message "Hello World") as short and long beeps, or in our case, short and long flashes on an LED. The short beeps/flashes are called "dots", and the long ones called "dashes".
- A "dot" is the basic unit of time. For now, assume the "dot" time is 200 ms.
- A "dash" is three dot-times long.
- Each dot/dash is separated by one dot-time. During this time the LED is off.
- Two letters in a message are separated by three dot-times. During this time the LED is off.
- Spaces between words are equal to seven dot-times total (no additional 3 dot-time inter-character delay).
- See the Wikipedia link for more explanation of timing.
- Morse code does not differentiate between upper and lower case letters.

1.1 General Requirements

1. All `printk()`'s must use reasonable log-levels (`KERN_INFO, KERN_ERR,...`)
2. Your driver must correctly load and unload when using `insmod` and `rmmod`.
   - It should clean-up after itself so you can re-load it often.

1.2 Create a 'Misc' Driver to Flash Morse Code

1. Use the driver-creation guide to create a skeleton driver named `morsecode`.
   - Setup the driver to build into a file named `morsecode.ko`
   - In your makefile, when building the driver, automatically copy or build the driver to `~/cmpt433/public/drivers/`
   - Your makefile must work with any user ID (don't hard code your ID!). You can use ~ to refer to the home directory. It must build on a Linux PC that has been setup in accordance with the driver creation guide.
2. Make your driver's `init` and `exit` methods display a message using `printk()`.
3. Make your driver a misc-character driver which supports a read and write method.
   - Have it setup its device virtual file (node) as `/dev/morse-code`
4. Add a new LED trigger named "`morse-code`".
   - Register the trigger at the driver's initialization, and unregister at exit.
5. Implement the `write()` method as follows:
   - Make it a blocking call (do not return until all flashing has completed).

- Flash the message to the LEDs using the kernel's LED library and the `morse-code` trigger you created.
  - You'll need to loop through all characters in the input buffer, determining the flash code for each letter, and then flashing it out to the LED(s).
- The course website provides an encoding for each letter. **You must use these.**
- Any character not a space or a letter (a-z, or A-Z) should be skipped with no delay. (i.e., 'Hi There' should flash the same as '!H_I th_ER&e@#09.,-5%!'
- You may assume that only one call to your write method will be happening at once so you don't need to synchronize multiple calls. (If multiple writes do happen concurrently, it will likely inter-mix the flashes from both messages, but that's OK.)
- You must use `copy_from_user()` (or similar) to access buffers from user-space.

6. Tips:
   - Test using either of the following methods:
     ```
     # echo sos > /dev/morse-code
     ```
     or
     ```
     # cat > /dev/morse-code
     ```
     - In the second one, you can type and when you hit enter, it will send the data. Cancel with Ctrl+C.
   - Before any LEDs will flash, you need to set them to the `morse-code` trigger, much like was done in assignment 1.
     - If you `rmmod` your driver, the `morse-code` trigger is removed. When you `insmod` your driver again, you'll need to reset an LED to be `morse-code`.
   - Consider writing a trivial Linux script to unload the driver, reload the driver, and set the LED trigger to save time when testing new changes to your driver.
   - Use the `msleep()` kernel method (in header `linux/delay.h`) to suspend for a set number of milliseconds. It's a blocking write-call so this holds up the calling thread until the write is complete.
   - In order for the LEDs to function under your custom kernel, you will likely need to install all the runtime loadable kernel modules you compiled with your kernel. See the Driver Creation Guide, section "Coping Modules to RFS".

1.3 Driver Parameters

1. Add the following driver parameter:
   - `dottime`: Sets the timing of the "dot", in ms.
     - When you change the time for the "dot", all other flash-timings should change relative to this (as described in the Morse Code section above).
     - Default value should be 200ms.
     - Value must be limited to [1 – 2000] (inclusive). If out of range, display an error message and set the value to the default.
2. To configure the parameters at driver load time (from the console) use:
   ```
   # insmod morsecode.ko dottime=150
   ```
3. Tips:
   - Make sure all your timings are relative to the "dot" time, which is configurable. This includes the time for "dot", "dash", between dots and dashes, and spaces.
   - When using the Morse code letter-encoding that is provided, each bit represents one dot-time. Therefore, by setting this time correctly the "dot", the "dash" and between-dot/dash times are likely to be just one setting (i.e., the dot time).

1.4 Read Dashes and Dots via FIFO

Add a read method to your driver so that it returns the dots and dashes it has transmitted since the last read call.

1. This method is executed when a program reads from `/dev/morse-code`, such as:
   `# cat /dev/morse-code`
2. Create a FIFO character queue for the dots, dashes, and spaces being transmitted. This will be populated by your driver's write function and read in your driver's read function.
   - Each time a dot is flashed out, add a '.' (period) to the queue.
   - Each time a dash is flashed out, add a '−' (minus) to the queue.
   - Each break between letters adds one space to the queue.
   - Each break between words adds three spaces total to the queue.
   - At the end of transmitting a message, add a line-feed ('\n') to the queue.
   - For example, "`Hi me`" generates the following
     `.... ..  -- .\n`
     - Below is the same output, but with additional annotations for additional understanding (your program does not produce this annotated output):

```
.... ..   -- .\n      (dot's and dashes, and line-feed)
H    i    m  e       (letters in the message)
    s  sss  s        (s = space in dashes and dots message)
```

3. Make the read function non-blocking: When called, any data waiting in the queue is returned to the caller but if there is no data available then it immediately returns 0 bytes read.
   - Each character read from the queue by the read function removes that data from the queue. You must safely access the buffer from user space using something like `copy_to_user()`.
   - You must correctly implement the case where the read buffer is smaller than the amount of data waiting to be read. Hint: Use the `readfile` program on the course website to test. If using the kernel queue data structure, it may be trivial to correctly implement this.
4. Examples:
   - Simple SOS pattern (easy to recognize). Note that calling `cat` twice generates no output the second time because the queue is empty.

```
# echo 'SOS' > /dev/morse-code
# cat /dev/morse-code
... --- ...
# cat /dev/morse-code
#
```

- Adding spaces between the letters means the output has three spaces between the words instead of one space (as above) between the letters.

```
# echo 'S O S' > /dev/morse-code
# cat /dev/morse-code
...   ---   ...
```

- The letters E I S H are just dots (1 through 4); and T M O are dashes (1 through 3), which makes the following easy to debug:

```
# echo 'e i s h t m o  eishtmo' > /dev/morse-code
# cat /dev/morse-code
.   ..   ...   ....   -   --   ---   . .. ... .... - -- ---
```

- And, of course:

```
# echo 'Hello world!' > /dev/morse-code
# cat /dev/morse-code
.... . .-.. .-.. ---   .-- --- .-. .-.. -..
```

- A longer output:

```
# cat > /dev/morse-code
But soft! What light through yonder window breaks?
# cat /dev/morse-code
-... ..- -   ... --- ..-. -     .-- .... .- -   .-.. .. --.
.... -   - .... .-. --- ..- --. ....   -.-- --- -.
-.. . .-.   .-- .. -. -.. --- .--   -... .-. . .- -.- ...
```

5. **Test real-time output with two terminals open:**
   Terminal 1: Display, in real time, what is being output from `/dev/morse-code`:
   `# for((;;)) do cat /dev/morse-code; sleep 0.1; done;`
   This will show you the '.' and '-' characters as they are generated by your driver!

   Terminal 2: When you want to test, echo new text into `/dev/morse-code`:
   `# echo 'hello world!' > /dev/morse-code`
6. Tips:
   - Assume there is at most one write and one read happening at the same time.
   - If a read happens part-way through flashing out a transmission, it will return the dashes and dots that have been flashed so far; further reads will return the later dashes and dots. This is by design.
   - While your write routine is flashing out dots and dashes, you'll want to interpret the bits to detect dots and dashes to put '.', '-', and ' ' characters into the queue.
     - Count the number of dashes and dots by incrementing a counter whenever one is detected.
     - When a dash, a dot, an inter-character break, an inter-word break, or end of transmission is detected, push the correct character into the queue for the read function to retrieve.
     - You can detect a dash with the bit pattern: 1110 (four msb's).

    - You can detect a dot with the bit pattern: 10 (two msb's). However, if you are shifting the Morse-code bit pattern to the left each time step, a dash when shifted left two will look like a dot. Therefore you may need to add a cool-down feature to prevent detecting spurious dot's. (i.e., once you detect a dash, don't detect a dot for a certain number of bit-times).
- There is a function for kfifo which works with user-space buffers.

1.5 Flash on Zen LED & Capture Output

On the **target**, boot into your custom compiled Linux kernel (downloaded using UBoot) and capture the console interaction of executing the following commands. Save the capture into a file named `capture.txt`:

1. *First ensure you can load the Zen cape's LED support, as shown in the ZenLEDGuide.*
2. Display the kernel version:
   ```
   # uname -r
   ```
3. Display the booted kernel's command line:
   ```
   # cat /proc/cmdline
   ```
4. Display target's networking configuration (must show Ethernet because it's built into the kernel):
   ```
   # ifconfig
   ```
5. Load the Zen cape's LED virtual cape (see guide).
6. Run `modinfo` on `morsecode.ko`.
7. Load the Morse code driver (`insmod`), **selecting a dot time of 40 ms.**
8. List loaded modules (`lsmod`).
9. Set the Zen cape's red LED to be driven by the `morse-code` trigger.
10. Display the list of triggers for the Zen cape's red:
    ```
    # cat /sys/class/leds/zencape\:red/trigger
    ```
11. Flash out the message `Hello world!`:
    ```
    # echo 'Hello world.' > /dev/morse-code
    ```
12. Display the dash-dot flash codes:
    ```
    # cat /dev/morse-code
    ```
13. Remove the module (`rmmod`).
14. List loaded modules (`lsmod`).
15. Display all the kernel messages your driver output:
    ```
    # dmesg | tail -100
    ```
    - Delete all `dmesg` lines not from your driver just to keep the output smaller.

Include in `capture.txt` any notes to the marker about how to get your code to work if there are any unusual steps (you may assume the marker has completed the Zen Cape LEDs guide and the Driver Creation Guide. You need not include the output from UBoot.

Make sure your capture file is human readable! If there are any funny characters (like ^D or [[^C] please edit the file to make its contents clearer. (These can arise due to colours in the console). An easy way to capture this text is execute the commands via SSH, and then copy-and-paste the contents of the SSH session into a text file.

2. LED Wiring

Wire up one of the large discrete (loose component) LEDs included with your BeagleBone Green kit into the breadboard for being safely controlled by software. You may choose any colour LED. Then registered it as an LED with the kernel, which makes it accessible inside the `/sys/class/leds/` folder.

Hint: Carefully follow the Wiring An LED guide.

Warning: Incorrectly wiring the circuit could damage your BeagleBone Green. I strongly recommend you have another person (such as a partner on the assignment, or another student in the class) double check your circuit before applying power.

Record a video of you controlling the LED via the Linux command line:

1. Must register it as an LED with the kernel (you need not show this in the video).
2. Show the breadboard with the wired LED circuit.
3. Show the Linux terminal where you enter the command to turn on the LED.
4. Show the LED that has turned on.
5. Show the Linux terminal where you enter the command to turn off the LED.
6. Show the LED that has turned off.
7. Show the Linux terminal where you enter the command to set the LED to the heartbeat trigger.
8. Show the LED turning on and off for ~10 seconds.

Upload your video to some place it will be accessible to the TA, such as YouTube, your SFU Vault drive (share file with TA), Vimeo, ... Video must be less than 5 minutes long! Shorter is better!

3. Deliverables

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

1. `as4-morse.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`)

   ```
   <assignment directory name>
    |-- Makefile
    \-- <dependencies such as .c, .h files>
   ```

   `Makefile` must support the `make` command to build your driver to `$(HOME)/cmpt433/public/drivers/`
   (Do not use relative paths for getting to the `cmpt433/public/drivers/` directory because the TA may build from a different directory than you.)

   Hint: Compress the `as4/` directory with the command

   ```
   $ tar cvzf as4-morse.tar.gz as4
   ```

2. Driver capture: `capture.txt`

(including mention of any unusual steps to get your code to build/work).
3. URL of uploaded video showing working LED on bread-board via Linux LED commands.

Remember that all submissions will automatically be compared for unexplainable similarities.