**EE414 Embedded System**

**20180745 – Tung-Duong Mai**

**Lab 2. LED Driver**

1. Purpose:

Grab a firm concept on serial communication by programing a serial input for the metronome. This program will be used to send a command string from the PC to the embedded board via RS-232C serial cable (actually USB cable).

The application gets user commands – tempo, time-signature, start, and stop - from the PC keyboard via USB serial cable.

Target board: Beaglebone (containing 4 GPIO LEDs) with Linux.

Host computer: PC with Linux, cross compiler, NFS, and minicom.

1. Experiment sequence:
2. Setup cross development environment:
3. Connection

Connect Beaglebone with Ethernet cable and Power adapter.

Connect Beaglebone with Linux PC by USB cable.

By this, the PC can access the Beaglebone via the network or USB.

1. Start PC NFS server:

After this step, PC can act as NFS server and it waits for Beaglebone to mount as an NFS client.

1. Connect to Beagle bone:

Using minicom or ssh

Difficulty in this step:

I prefer ssh connection method since it is more reliable (I was worried that the stale lock file might happen again) and more intuitive (I can execute directly from the PC terminal without switching to minicom interface). The result I showed is from ssh connection.

However, for complete experiment and item 1 in the discussion, I also retest all experiments with minicom.

1. Start Beaglebone nfs client

I switch to root user to have enough permission to run sysfs command (I tried give gpio permission to john (my user) but there is some incompatibility, so I think it is the best to just run in root)

Now, the Beaglebone can access the directory on PC that we setup in the previous lab. We are ready for cross development.

NFS or scp both work correctly. However, I think NFS is more intuitive since it gives me the sentiment that the data are shared and I can work as if the data is locally stored (for scp, I have to explicitly transfer the data).

1. Test single key input

This step aim to experiment reading the key input without enter

We perform 2 modes: blocking and non-blocking

After this step, we learned how to read input without enter key in both blocking and non-blocking mode. This is important because we do not want our input to be ‘buffered’, we prefer an immediate response (our application is quite time-sensitive). It is also important to see the different between blocking and non-blocking mode, which will be discussed further in the discussion

Since we do not need Beaglebone response, we test on PC for easy debugging.

1. Test the algorithm of Metronome\_TUI.c on PC

From now on, I use blocking mode (to be consistent with TA’s comment on Piazza) and since it is necessary for some (but not all) parts from now on. Further detail is in discussion.

This steps aims to confirm the validity of the response to keyboard input.

We write algorithm to read input and perform appropriate actions to control time signature, tempo or on/off status.

This is the early debug for our code, since it might become very complicated later on.

Since we do not need Beaglebone response, we test on PC for easy debugging.

4. Test metronome\_tui\_thread.c on Beaglebone

4.1 Debug on PC

This is the early debug for our code, since we would like to make sure that everything in the input handling is correct before moving on the Beaglebone control

The program simply add the output of character corresponding to the LED ( ‘7’ means 3 LEDs, ‘3’ means 2 LEDs, and ‘1’ means 1 LED. After this step, we confirm that the algorithm pass the desired correct LEDs status to the Beaglebones and the Beaglebones now can realize the LED status.

We put a function to calculate the status and output it into a separate thread, since we want both jobs to run simultaneously.

Difficulty: Threading and shared states

Since I am not familiar with thread programming, it takes quite sometimes for me to grasp the working principle of it.

We need to share multiple information between gpio thread and main thread. Therefore, I bring many shared variables outside to be global variables. By this, we could share information between threads.

I also face some difficulty when making while loop for gpio, as we need to print out characters sequentially and continuously. We also need to make sure that we start at the first beat of the new state after the state (time signature/tempo/on-off) is changed. I do that by using a (wrap-around) counter and reset it to 0 (first beat) when state is changed.

I also made some silly errors, such as calculating the period, I forgot to cast to float and the period becomes 0.

4.2. Test on Beaglebones

Finally, we write the code to manipulate GPIO and realize the LED on/off status.

We migrate the thread function into the previous lab’s GPIO file. We slightly modify code to fit with our new setting (with a new while loop introduced).

Difficulty: Functionality separation (modularization)

It is hard since the lab requires we put divide the functionality into 4 files (actually 3, as 1 is header file) and keep sharing the global variables.

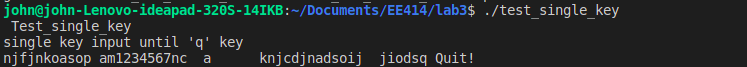
I use the headerfile (usermmapLED) as API between main (metronome\_tui\_thread) and gpio thread (gpio\_led\_fu.c). All the shared sunctiona nd variables are declared in the header file and initialize in either file (metronome\_tui\_thread or gpio\_led\_fu).

Before and after experiment, I reuse the shell file from last lab to turn off all LEDs and restore LEDs.

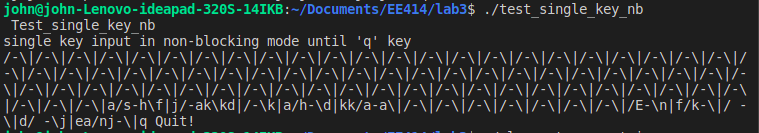
1. Experimental results:

1. Test single key input on PC

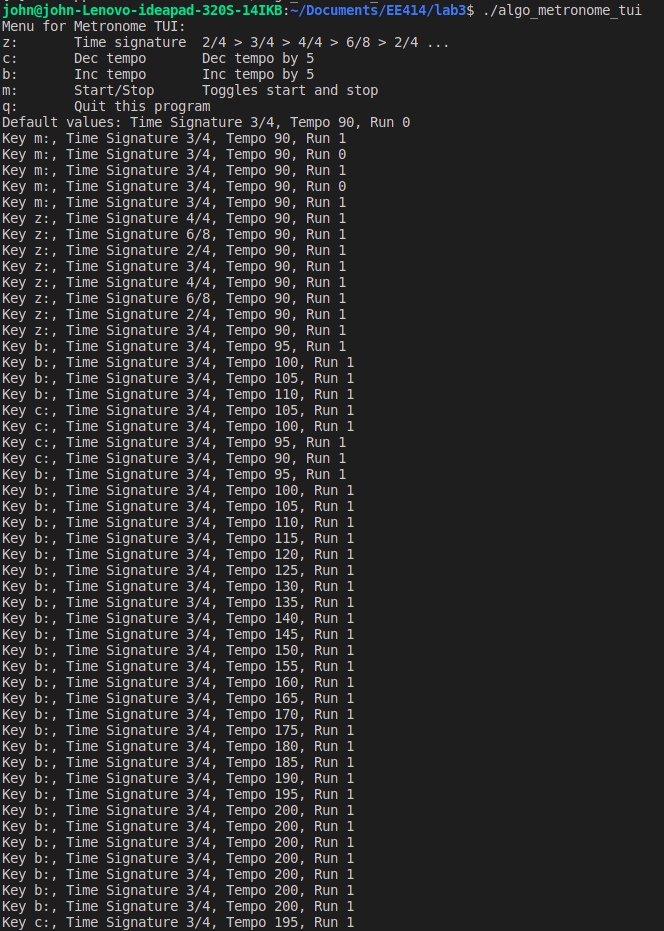
a. Blocking mode

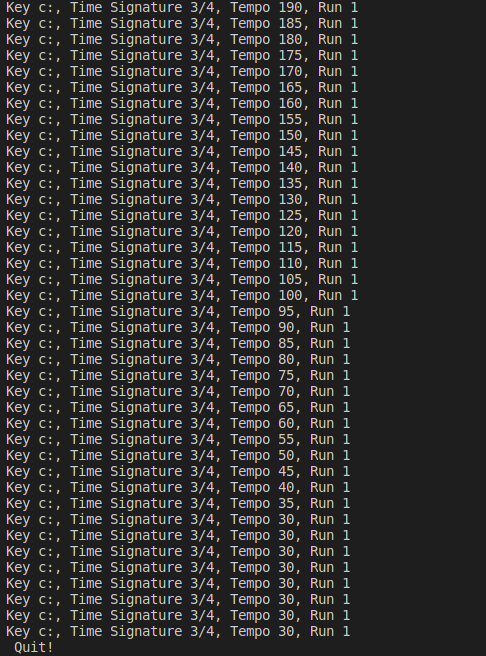


b. Non-blocking mode



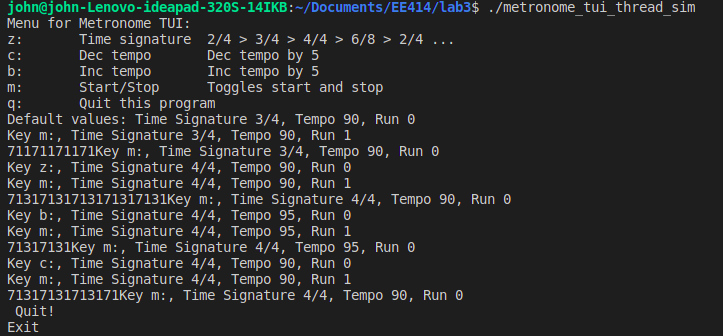
2. Test the algorithm of Metronome\_TUI.c on PC



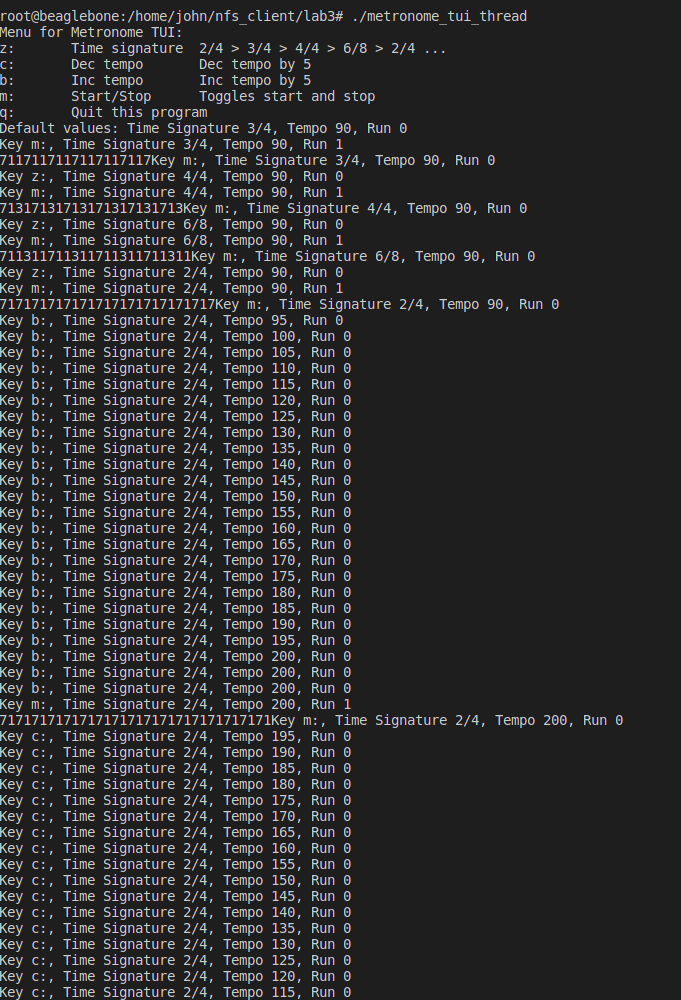
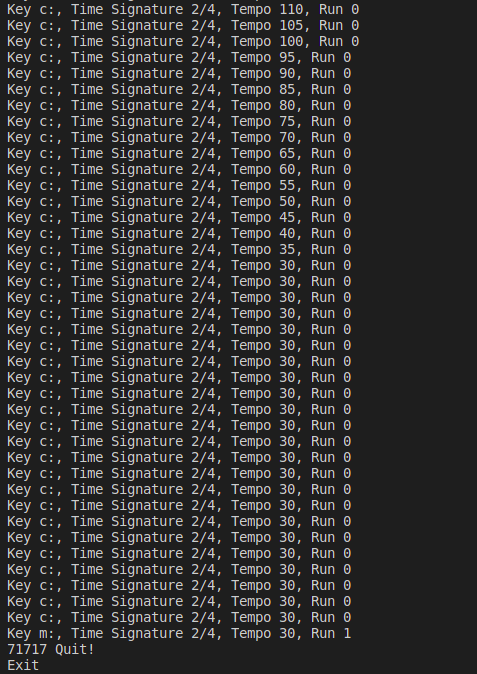


3. Test metronome\_tui\_thread.c on Beaglebone

3.1. Simulation on PC



3.2. Actual test on Beaglebones

Command: make met  

1. Discussion:
2. Explain how the console serial input/output is routed to USB in the Beaglebone.

Diagram:

User I/O (keyboard and display) <-> PC <-> USB <-> USB serial cable <-> Embedded processor -> GPIO -> LEDs.

When we provide (or observe) input/output via PC keyboard and screen display, the information can be access by USB port via Pentium

USB port and UART0 on processor is connected by serial connection with a USB cable. This assists serial debug. USB protocol allows peripherals and computers to interconnect in a standard way. (Both device will need the serial transmission converted back to parallel so that it can be used)

Then, via mmap mechanism, we can manipulate GPIO. The GPIO module implements the set-and-clear protocol register update for the data output and interrupt enable registers.

There are a total of 66 GPIO pins available on the Beaglebone, including pin for controlling LEDs. Therefore, we can manipulate LED by IO with PC.

1. About blocking mode and non-blocking mode:

In non-blocking mode, we do a busy loop while waiting for input (so it’s not ‘waiting for input’)

In blocking mode, we immediately go to next step without the loop (so it got stuck until we provide input)

1. The disabling of echo mode and canonical mode

We turn off echo so that the terminal does not automatically write (‘echo’) the read input into terminal. Instead, we want to check that we correctly read the input by writing the observed input to the terminal. The echoed output is rather redundant and confusing and does not serve out validation purpose/

We disable canonical mode, since we want input to be immediately available without pressing enter (in other words, we do not want line editing or buffered IO).

1. Explain how you apply pthread in your lab briefly. Find the summary of functions what you used.

I used data structure pthread\_t to stores the thread id (tid).

Then, I create the new thread from main:

int pthread\_create(pthread\_t &tid, const NULL, &gpio\_led, NULL)

(gpio\_led is starting function, first NULL denotes that we do not have attribute, second NULL denotes no argument)

I exit the calling thread using thread\_exit when I finish the gpio\_led or when Ctrl+C happens

int pthread\_exit(tid)

I use thread\_join in main to make sure main does not terminate until gpio\_led terminate

Int pthread\_join(tid, NULL)

(NULL denotes that we do not care about return value)

V- References:

[1] Getting Started with Beaglebone and Beaglebone Black, http://www.beagleboard.org/Getting%20Started

[2] Beaglebone Rev. A5. System Reference Manual,

http://circuitco.com/support/index.php?title=Beaglebone#Rev\_A5. NOTE. The version of Beaglebone boards in the Lab is Rev. A5. Be sure to download the correct version.

[3] “Embedded Linux Primer”, C. Hallinan, Prentice Hall.

[4] Lab material, EE414 Teaching Staffs, KLMS

[5] EBC Exercise series, elinux.org

[6] GPIO Programming series, ics.com

[7] “Loadable kernel module”, Wikipedia

[8] Leslie, Ben & Chubb, Peter & Fitzroy-Dale, Nicholas & Götz, Stefan & Gray, Charles & Macpherson, Luke & Potts, Daniel & Shen, Yue-Ting & Elphinstone, Kevin & Heiser, Gernot. (2005). User-Level Device Drivers: Achieved Performance. J. Comput. Sci. Technol.. 20. 654-664. 10.1007/s11390-005-0654-4.