[EE414] Embedded Systems
Lab3 Report

School of Computing, KAIST
20180480 Woojin, Lee

1. Purpose

In order to grab a firm concept on serial communication, a serial input for the metronome will be programmed. 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).
Also, we can get hands-on experience with threads. Thread is a precise concept that is compared to another concept, process. It's a powerful tool to parallelize stuff, and C provides <pthread.h> API to utilize threads efficiently.
For these purposes, we will write an application program "Metronome_tui" on the embedded board, which gets our commands(tempo, time-signature, start, and stop) from the PC keyboard via USB serial cable.

2. Experiment sequence

We first explore how to get an input character without an Enter key. (test_single_key.c)
To do so, you have to use the <termios.h> API, which is made for terminal control. termios provides its own structure, and we can control our terminal as we want.

```
struct termios
{
    tcflag_t c_iflag;    /* input flags */
    tcflag_t c_oflag;    /* output flags */
    tcflag_t c_cflag;    /* control flags */
    tcflag_t c_lflag;    /* local flags */
    cc_t     c_cc[NCCS]; /* control chars */
    speed_t  c_ispeed;   /* input speed */
    speed_t  c_ospeed;   /* output speed */
};
```

Basically, we will turn off canonical mode. In canonical mode, the terminal fires inputs buffer when the Enter key is pressed.

Next, we build algo_metronome_tui.c to utilize keyboard input. Its workflow will be like this:

1. Set termios
2. Print title and menu
3. Set default values to parameters (TimeSig ¾, Tempo 90, Stop), and print default vaules
4. Loop
    a. Get user input key without enter

b. If ‘q’, break
c. Interpret the key
    i. If ‘z’ (Time-Signature):
        1. inc TimeSig
        2. Make TimeSig rotating
    ii. If ‘c’ (Dec Tempo)
        1. Tempo = Tempo - 5
        2. Make TimeSig roating
    iii. If ‘b’ (Inc Tempo)
        1. Tempo = Tempo + 5
        2. Make TimeSig roating
    iv. If ‘m’ (Start/Stop)
        1. Run = 1 if run == 0
        2. Run = 0 if run == 1
d. Print single line message: Input & status

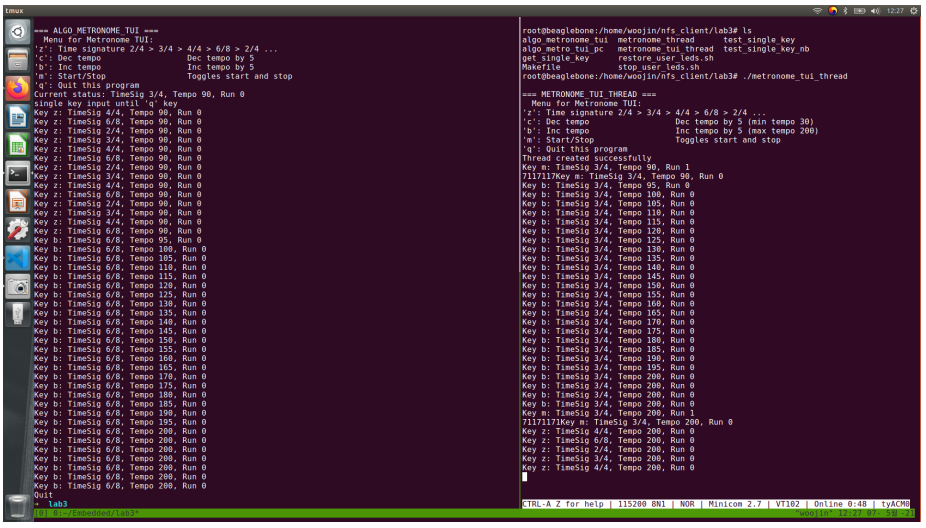
5. Print quit message
6. Reset termios

It’s not that difficult, since you did everything in test_single_key.c file. The main task here is just setting termios and controlling configurations through keys.

Finally, we combine algo_metronome_tui.c and lab2(LED output) so that we can control LED with keyboard input accordingly.
You are going to use pthread to work with LED and Key together. Also, it would be better to organize the code with split files. Note that you have to put -pthread compile option.

3. Experimental results

test_single_key and test_single_key_nb:
In test_single_key, we used termios so that we can get a single character from the terminal input. You set ~ICANON flag so that you can turn off canonical mode. By doing so, you don’t have to wait until ‘enter’ key.
In test_single_key_nb, it made the key_hit() function for non-blocking mode. By doing so, this code can manage other tasks(here, printing waitchar).



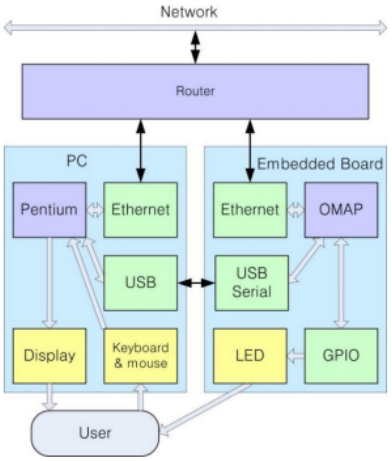
algo_metronome_tui and metronome_tui_thread works well

For metronome_tui_thread, my code has 4 different files. 1) gpio_led_fu.c, 2) key_input_fu.c, 3) metronome_tui_thread.c, 4) userLEDmmap.h. “metronome_tui_thread.c” is the main function. “gpio_led_fu.c” and “key_input_fu.c” are set of function utilities for controlling keyboard and LED, respectively. “userLEDmmap.h” contains functions and variables so that these files can communicate and operate smoothly.
I used blocking I/O, so I could feel some delay.
I attached my code and demo video.

4. Discussion

A. Explain how the console serial input/output is routed to USB in the Beaglebone.

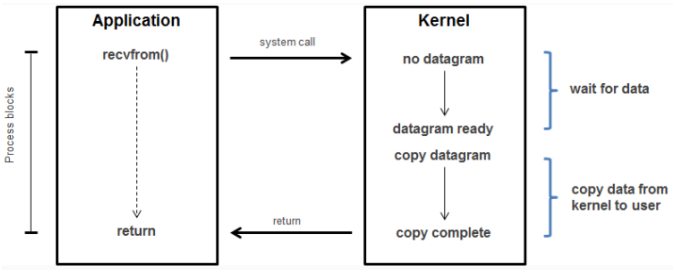
Most Beaglebone include a USB cable, providing a convenient way to provide both power to your Beagle and connectivity to your computer.



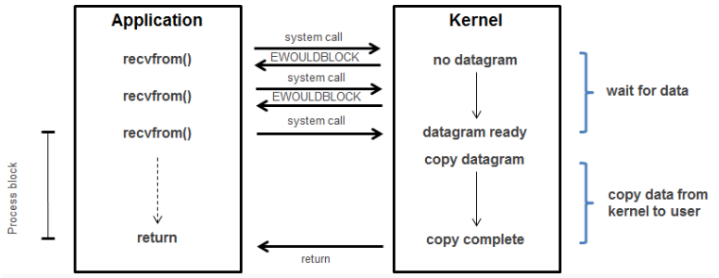
You will first make input via keyboard, and this information will be sent to PC through I/O bus. That information will then be sent to the embedded processor through USB/USB serial cable. After the embedded processor processes the information, it’ll send a command to the LED through the GPIO pin.
The Beaglebone Black board has a miniUSB connector that connects the USB0 port to the processor. This is the same connector as used on the original BeagleBone.

B. Compare single key input methods: blocking and non-blocking

I/O task is done in kernel space, not in user space. User process or thread asks the kernel for I/O task, and waits for the kernel's return value.
Blocking mode is default mode. User asks read() for the kernel, and waits until the kernel returns. After the kernel returns, the user program then resumes its task. It may waste resources.



On the other hand, non-blocking mode doesn't stall after asking the kernel for a task. When the user asks the kernel for the read() task, the result will be immediately returned. It doesn't have to be a finished task. It will repeat this process until there is input data. By doing so, processing time doesn't relate to I/O processing time. However, it also wastes some resources since you repeatedly use system calls.



In our lab manual, to use non-blocking mode, it implemented a function "key_int". You can implement it with select() function. This function checks for changes in fd.

C. Explain why you turn off canonical and echoing modes

Both are controlled through <termios.h>. By setting c_flag as ~ICANON and ~ECHO, you can turn off canonical and echoing modes.

[Canonical mode]

Canonical input is, the entire line is gathered and edited up until the end of the line character 'return' is pressed. So the whole line is made available to waiting programs.

On the other hand, non-canonical input is when you press a character, and it is immediately available to the program. It doesn't wait until 'return'. If you turn off canonical mode, then you can get keyboard input immediately. (You don't have to do pressing enter, stuffs like that)

[Echo mode]

When echo is on, the characters type at the keyboard are normally echoed to the screen. If echo is clear, input characters are not echoed. Since you are going to print status with input character, we don't have to turn on echo mode.

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

We have to make two loops, one for keyboard input from the user and the other for LED. I used thread for controlling LEDs.

By adopting thread concept, we can concurrently execute both keyboard input and LED output. Also, by adopting the MUTEX concept, we can exclusively protect their own regions and avoid deadlock.

Below, here comes a brief explanation of the functions that I used for lab code:

To compile with pthread, you have to give -lpthread option.

- functions related to thread

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

This function creates a thread. Thread id will be stored in pthread_t* thread, and the thread will conduct a given function(3rd argument), with parameters specified in 4th argument. the 2nd argument is for specifying thread property. If thread is successfully created, it will return 0.

```
int pthread_join(pthread_t th, void **thread_return);
```

This function is used for waiting thread to be ended. You can specify thread with pthread_t th(1st argument), and get its return value with void **thread_return(2nd argument). Thread will return every resource. If it succeeds, this function will return 0, and error codes otherwise.

```
void pthread_exit(void *retval);
```

As the name implies, this function terminates the thread.

- functions related to mutex

In thread, there is a concept called “critical section”. As multiple threads share data together, sometimes you may want to protect it so that specific thread can exclusively conquer that shared data. MUTEX is the mechanism that allows only one thread to shared data at once.

```
int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutex_attr
*attr);
```

This function is used to initialize mutex object. You specify mutex object with pthread_mutex_t* mutex(first argument), and can control properties with pthread_mutex_attr* attr(second argument). When NULL, it’ll be set to “fast”, which is default value.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The code fraction from pthread_mutex_lock to pthread_mutex_unlock will be “critical section”. So, only one thread can occupy shared data, and other threads will wait until that thread is done.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

As the name implies, this code will destroy a mutex object.

5. References

https://velog.io/@codemcd/Sync-VS-Async-Blocking-VS-Non-Blocking-sak6d01fhx
https://brainbackdoor.tistory.com/26
https://ju3un.github.io/network-basic-1/
https://beagleboard.org/getting-started