Prelab: 09
Demo: 34+25
Report: 37
Delay: 0

Total: 101

Real-Time Embedded Computing
Lab-6 Report
Musical Keyboard

Haoxiang Zhang

Due date: November 18, 2019

37/50

Post Lab

Objectives and Lab Description

Part-1: Speaker function

This part, we are going to trying two different approaches to create periodic tasks in kernel space, and think about which one is better. For the task, we have to make a speaker function with these two different approaches. Make sure use bit masking in order to avoid the situation that disrupt the work of other pins.

Part-2: The Keyboard

Since part 1 we implement that let speaker make sound, for this part, the requirements is that make five buttons that each button could change the frequency of the sound while pressing the button.

Part-3: Master/Slave Control

5/5

This part is about communication between kernel space and user space. Client could send message that represent one of the five notes to be played to all of the device that under the same network. The message is @A, @B, @C, @D, or @D. Every device that receive the message should change the frequency of the sound. If the board is a master, when it receives a note message, then it should broadcast the message to every slave devices.

Client could send message to specific board. just type "& #" where # is the last two digits of ip address. Then, type the message and send it to the device.

Extra Credit

For this part, the requirement is that if your board is the master, and a button is pressed, your board should send the note to all the slave boards.

✓

Implementation

Flow Chart

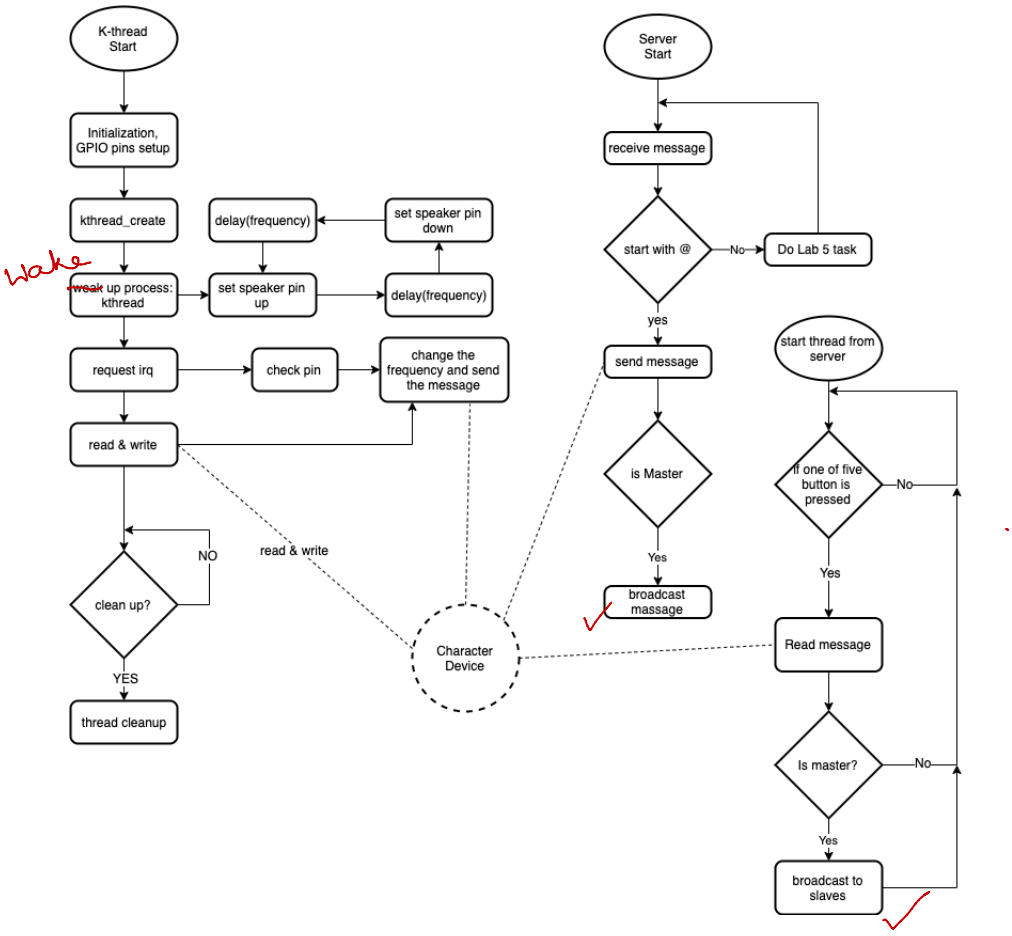

Figure 1: Lab-6 Flow Chart

kernel space

First of all, part 1, I create a kthread in the thread initial function and wake this process up. In the kthread_fn function, in the while loop, I set up the square wave for the speaker. Then, in the part two, in the thread_init function, I setup gpio pins and its function, which could operate the pins directly. Also, build an interrupt request which allowed pause the process and run the code first, and change the frequency of the sound depends on the pin that is detected by GPEDS0 in the function button_isr.

Now, in order to implement the function that the message from client or servers could change the frequency of sound, I register the character device in the initial function. In the button_isr function, I store the button message when the button is detected that is pressed. Because kernel space will receive the data from user space, in the device_write function, I check the message and change the frequency of sound via comparing the message and the note: @A, @B, @C, @D, or @E.

Where is this message stored? -1

9/10

user space

Just keep everything same in the lab 5. What I have changed is that add the message checking with first character is "@" and check if is master then to broadcast to slaves.

✓

extra credit

This part, I create a pthread that could receive the checking result that detect the button is pressed or not. If pressed then check the message from character device. If is master then broadcast to slaves. ✓

Should have added screenshots accompanied by a short explanation. -5

Experiments and Results

Music Keyboard

10/15

First, I do make file and create the .ko file. After install the ko file, the speaker is making sound. When I press the button, the tone is changed. Then, I give the character device write permission, and run the client program and server program. When I type note message (such as @A) in the client, server receive the message, also ,the sound is changed. When I pressed the button, only in the server (so far still is slave), display the note message. After VOTE the master, the master board receive the note message from client, it change the sound and broadcast the message to all the slave boards. When I press the button, master board display the note message, and slave board displayed as well.

Discussion and Post-lab questions

1. Based on your observations, which approach did you like better? Did either of them produce a better sound? Which one do you think is more reliable/accurate?

   - After I tried kthread and hrtimer, I think kthread is better. Actually, I did not find any difference of the sound between these two approaches. The disadvantage of the kthread is that every time should wait the jiffies time go after 10HZ, which means we have to wait about 10 second then will start the process, but hrtimer could start the process immediately. The most inefficient part of the hrtimer is that it not much stable. Every time I stop the kernel, the system always has some problem and I have to reboot the system. Also, it cannot allows me to run again. It could be that my program has some issues. Anyway, I think kthread is more reliable.

8/15

Should have added all the problems you have faced & the approach you made to fix the same. Also, should have included what you have learnt in this lab. -7

Comments: 5/5

Coding Section

34+25/40

Kernel Function - kthread

Listing 1: Kernel part: Control the frequency of sound and receive the data from user space

i) "register_char" explanation is wrong -3

ii) Interrupt config explanation is off. -3

```
#ifndef MODULE
#define MODULE
#endif
#ifndef __KERNEL__
#define __KERNEL__
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/io.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/interrupt.h>
#include <linux/delay.h>
#include <linux/kthread.h>  // for kthreads
#include <linux/sched.h>  // for task_struct
#include <linux/time.h>   // for using jiffies
#include <linux/timer.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

#define MSG_SIZE 50
#define CDEV_NAME "lab6_hx" // "YourDevName"

// Address
#define GPIOBASE     0x3F200000
#define SPKR_SEL     0x40000

// Button's frequency
#define BTN_1 1150
#define BTN_2 1050
#define BTN_3 950
#define BTN_4 850
#define BTN_5 750

// Give the permission to install module
MODULE_LICENSE("GPL");
```

```
38
39 // Masks for each part
40 unsigned long spkr_mask = 0x40;
41 unsigned long btn1_mask = 0x10000;
42 unsigned long btn2_mask = 0x20000;
43 unsigned long btn3_mask = 0x40000;
44 unsigned long btn4_mask = 0x80000;
45 unsigned long btn5_mask = 0x100000;
46 unsigned long btns_mask = 0x1F0000; // btn1 to btn5
47 unsigned long pins_mask = 0x1F0040; // bottons and speaker
48
49 // address
50 unsigned long* GPFSEL0;
51 unsigned long* GPSET0;
52 unsigned long* GPCLR0;
53 unsigned long* GPEDS0;
54 unsigned long* GPPUD;
55 unsigned long* GPPUDCLK0;
56 unsigned long* GPAREN0;
57
58
59 // structure for the kthread.
60 static struct task_struct *kthread1;
61 unsigned long *vaddr;
62 unsigned int sound = BTN_3;
63 // structure for isr
64 int mydev_id; // to identify the handler
65 // structure for device
66 static int major;
67 static char msg[MSG_SIZE];
68
69 bool pressed = false;
70
71 // function for get length of actual char number
72 size_t getlen(const char* str){
73   const char *s;
74   for (s = str; *s; ++s){
75     if(strncmp(s, "/0", 2) == 0){
76       break;
77     }
78   }
79   return (s - str);
80 }
81
82 // Function called when the user space program reads the character
       device.
83 // Some arguments not used here.
84 // buffer: data will be placed there, so it can go to user space
85 // The global variable msg is used. Whatever is in that string will
       be sent to userspace.
86 // Notice that the variable may be changed anywhere in the module
       ...
87 static ssize_t device_read(struct file *filp, char __user *buffer,
       size_t length, loff_t *offset){
88   // Whatever is in msg will be placed into buffer, which will be
       copied into user space
89   ssize_t dummy = copy_to_user(buffer, msg, length);  // dummy will
```

```
      be 0 if successful
90   // msg should be protected (e.g. semaphore). Not implemented here
       , but you can add it.
91   msg[0] = '\0';  // "Clear" the message, in case the device is
       read again.
92           // This way, the same message will not be read twice.
93           // Also convenient for checking if there is nothing new,
       in user space.
94   return length;
95 }
96
97 // Function called when the user space program writes to the
       Character Device.
98 // Some arguments not used here.
99 // buff: data that was written to the Character Device will be
       there, so it can be used
100 //       in Kernel space.
101 // In this example, the data is placed in the same global variable
       msg used above.
102 // That is not needed. You could place the data coming from user
       space in a different
103 // string, and use it as needed...
104 static ssize_t device_write(struct file *filp, const char __user *
       buff, size_t len, loff_t *off){
105   ssize_t dummy;
106
107   if(len > MSG_SIZE)
108     return -EINVAL;
109
110   // unsigned long copy_from_user(void *to, const void __user *from
       , unsigned long n);
111   dummy = copy_from_user(msg, buff, len); // Transfers the data
       from user space to kernel space
112   if(len == MSG_SIZE)
113     msg[len-1] = '\0';  // will ignore the last character received.
114   else
115     msg[len] = '\0';
116
117   // You may want to remove the following printk in your final
       version.
118   printk("Message from user space: %s\n", msg);
119   if (strncmp(msg, "@A", 2) == 0){
120     sound = BTN_1;
121   } else if (strncmp(msg, "@B", 2) == 0){
122     sound = BTN_2;
123   } else if (strncmp(msg, "@C", 2) == 0){
124     sound = BTN_3;
125   } else if (strncmp(msg, "@D", 2) == 0){
126     sound = BTN_4;
127   } else if (strncmp(msg, "@E", 2) == 0){
128     sound = BTN_5;
129   }
130   return len;   // the number of bytes that were written to the
       Character Device.
131 }
132
133 // structure needed when registering the Character Device. Members
```

```
    are the callback
// functions when the device is read from or written to.
static struct file_operations fops = {
  .read = device_read,
  .write = device_write,
};

// ================================ K-Thread ======================================

// Function to be associated with the kthread; what the kthread executes.
int speaker_fn(void *ptr){
  unsigned long j0, j1;
  printk("In kthread1\n");
  j0 = jiffies;   // number of clock ticks since system started;
            // current "time" in jiffies
  j1 = j0 + 10*HZ;  // HZ is the number of ticks per second, that is
            // 1 HZ is 1 second in jiffies
  while(time_before(jiffies, j1)) // true when current "time" is less than j1
      schedule();   // voluntarily tell the scheduler that it can schedule
            // some other process
  printk("Before loop\n");

  // The ktrhead does not need to run forever. It can execute something
  // and then leave.
  while(1)
  {
    // In an infinite loop, you should check if the kthread_stop
    // function has been called (e.g. in clean up module). If so,
    // the kthread should exit. If this is not done, the thread
    // will persist even after removing the module.
    // comment out if your loop is going "fast". You don't want to
    // printk too often. Sporadically or every second or so, it's okay.
    if(kthread_should_stop()) {
      iowrite32(0x40, vaddr + 10);  //GPCLR0
      do_exit(0);
    }
    iowrite32((uint32_t)GPSET0 | spkr_mask, GPSET0);  //GPSET0
    udelay(sound);
    iowrite32((uint32_t)GPCLR0 | spkr_mask, GPCLR0);  //GPCLR0
    udelay(sound);
  }

  return 0;
}

static irqreturn_t button_isr(int irq, void *dev_id) {
  // In general, you want to disable the interrupt while handling it.
  disable_irq_nosync(79);

```

```
182   // This same handler will be called regardless of what button was
        pushed,
183   // assuming that they were properly configured.
184   // How can you determine which button was the one actually pushed
        ?
185
186   // DO STUFF (whatever you need to do, based on the button that
        was pushed)
187   if (ioread32(GPEDS0) == btn1_mask){
188     sound = BTN_1;
189     sprintf(msg, "@A");
190   }
191   if (ioread32(GPEDS0) == btn2_mask){
192     sound = BTN_2;
193     sprintf(msg, "@B");
194   }
195   if (ioread32(GPEDS0) == btn3_mask){
196     sound = BTN_3;
197     sprintf(msg, "@C");
198   }
199   if (ioread32(GPEDS0) == btn4_mask){
200     sound = BTN_4;
201     sprintf(msg, "@D");
202   }
203   if (ioread32(GPEDS0) == btn5_mask){
204     sound = BTN_5;
205     sprintf(msg, "@E");
206   }
207
208   // IMPORTANT: Clear the Event Detect status register before
        leaving.
209   iowrite32(btns_mask, GPEDS0);
210
211   printk("Interrupt handled\n");
212   enable_irq(79);   // re-enable interrupt
213
214   return IRQ_HANDLED;
215 }
216
217 int thread_init(void){
218   int dummy = 0;
219
220   char kthread_name[11]="my_kthread";
221   // try running  ps -ef | grep my_kthread
222   // when the thread is active.
223   printk("In init module\n");
224
225   // register the Characted Device and obtain the major (assigned
        by the system)
226   major = register_chrdev(0, CDEV_NAME, &fops);
227   if (major < 0) {
228         printk("Registering the character device failed with %d\n",
        major);
229         return major;
230   }
231   printk("Lab6_cdev_kmod example, assigned major: %d\n", major);
232   printk("Create Char Device (node) with: sudo mknod /dev/%s c %d
```

```
    0\n", CDEV_NAME, major);
233
234   // gpio set
235   vaddr = (unsigned long*)ioremap(GPIOBASE, 4096);
236   GPFSEL0   = vaddr + 0;
237   GPSET0    = vaddr + 7;
238   GPCLR0    = vaddr + 10;
239   GPEDS0    = vaddr + 16;
240   GPAREN0   = vaddr + 31;
241   GPPUD     = vaddr + 37;
242   GPPUDCLK0 = vaddr + 38;
243   iowrite32(SPKR_SEL, GPFSEL0);  //GPFSEL0
244   iowrite32((uint32_t)GPCLR0 | spkr_mask, GPCLR0);
245   iowrite32(0x01, GPPUD);
246   udelay(100);
247   iowrite32(btns_mask, GPPUDCLK0);
248   iowrite32(btns_mask, GPAREN0);
249
250   kthread1 = kthread_create(speaker_fn, NULL, kthread_name);
251
252   if((kthread1)){ // true if kthread creation is successful
253     printk("Inside if\n");
254     // kthread is dormant after creation. Needs to be woken up
255     wake_up_process(kthread1);
256   }
257   dummy = request_irq(79, button_isr, IRQF_SHARED, "Button_handler"
      , &mydev_id);
258
259   printk("Button Detection enabled.\n");
260       return 0;
261 }
262
263 void thread_cleanup(void) {
264   int ret;
265
266   // disable reading from device
267   unregister_chrdev(major, CDEV_NAME);
268   printk("Char Device /dev/%s unregistered.\n", CDEV_NAME);
269
270   // Good idea to clear the Event Detect status register here, just
       in case.
271   iowrite32(0, GPEDS0);
272   // Disable (Async) Rising Edge detection for all 5 GPIO ports.
273   iowrite32(0, GPAREN0);
274   // Remove the interrupt handler; you need to provide the same
       identifier
275       free_irq(79, &mydev_id);
276
277   printk("Button Detection disabled.\n");
278
279   // the following doesn't actually stop the thread, but signals
       that
280   // the thread should stop itself (with do_exit above).
281   // kthread should not be called if the thread has already stopped
       .
282   ret = kthread_stop(kthread1);
283
```

```
284   if(!ret)
285     printk("Kthread stopped\n");
286 }
287
288
289
290
291
292
293
294
295 /*
296 // ================================  hrtimer
        ================================
297 unsigned long timer_interval_ns = 1e6;  // timer interval length (
        nano sec part)
298 static struct hrtimer hr_timer;     // timer structure
299 static int dummy = 0;
300
301 // Timer callback function: this executes when the timer expires
302 enum hrtimer_restart timer_callback(struct hrtimer *
        timer_for_restart)
303 {
304     ktime_t currtime, interval; // time type, in nanoseconds
305   unsigned long overruns = 0;
306
307   short spkr_mask = 0x40;
308   vaddr = (unsigned long*)ioremap(GPIOBASE, 4096);
309   iowrite32(SPKR_SEL, vaddr);  //GPFSEL0
310
311
312   // Re-configure the timer parameters (if needed/desired)
313     currtime  = ktime_get();
314     interval = ktime_set(0, timer_interval_ns); // (long sec, long
        nano_sec)
315
316   // Advance the expiration time to the next interval. This returns
         how many
317   // intervals have passed. More than 1 may happen if the system
        load is too high.
318     overruns = hrtimer_forward(timer_for_restart, currtime,
        interval);
319
320
321   // The following printk only executes once every 1000 cycles.
322   if(dummy == 0){
323     printk("mark here\n");
324     //int s;
325     // comment out if your loop is going "fast". You don't want to
326     // printk too often. Sporadically or every second or so, it's
        okay.
327     //for (s = 1; s <900; s++){
328     if(kthread_should_stop()) {
329       iowrite32(0x40, vaddr + 10);  //GPCLR0
330       do_exit(0);
331     }
332     //  if (s >= 895){
```

```
333     //     s = 1;
334     //   }
335     iowrite32((uint32_t)(vaddr+7) | spkr_mask, vaddr + 7);  //
        GPSET0
336     udelay(s);
337     iowrite32((uint32_t)(vaddr+7) | spkr_mask, vaddr + 10); //
        GPCLR0
338     udelay(s);
339     //}
340   }
341   dummy = (dummy + 1)%1;  // kind of timer here
342
343
344   return HRTIMER_RESTART; // Return this value to restart the timer
        .
345                 // If you don't want/need a recurring timer, return
346                 // HRTIMER_NORESTART (and don't forward the timer).
347 }
348
349 int timer_init(void)
350 {
351   // Configure and initialize timer
352   ktime_t ktime = ktime_set(0, timer_interval_ns); // (long sec,
        long nano_sec)
353
354   // CLOCK_MONOTONIC: always move forward in time, even if system
        time changes
355   // HRTIMER_MODE_REL: time relative to current time.
356   hrtimer_init(&hr_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
357
358   // Attach callback function to the timer
359   hr_timer.function = &timer_callback;
360
361   // Start the timer
362   hrtimer_start(&hr_timer, ktime, HRTIMER_MODE_REL);
363
364   return 0;
365 }
366
367 void timer_exit(void)
368 {
369   int ret;
370     ret = hrtimer_cancel(&hr_timer);  // cancels the timer.
371     if(ret)
372     printk("The timer was still in use...\n");
373   else
374     printk("The timer was already canceled...\n");  // if not
        restarted or
375                               // canceled before
376
377     printk("HR Timer module uninstalling\n");
378
379 }
380 */
381
382 // Notice this alternative way to define your init_module()
383 // and cleanup_module(). "thread_init" will execute when you
```

```
    install your
// module. "thread_cleanup" will execute when you remove your
    module.
// You can give different names to those functions.

module_init(thread_init);
module_exit(thread_cleanup);

//module_init(timer_init);
//module_exit(timer_exit);
```

Server Function

Listing 2: Socket part: communication with devices under same network and communicate with kernel space

```
/*
    Project: RT_Embedded_computing Lab-6
     Author: Haoxiang Zhang
  Description: UDP Kernel music control
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <inttypes.h>
#include <stdbool.h>
#include <time.h>
#include <fcntl.h>
#include <wiringPi.h>

#define MSG_SIZE 40
#define CHAR_DEV "/dev/lab6_hx" // "/dev/YourDevName"


bool fromMaster = true;
bool isMaster = false;
int master_num = -1;
char master_ip[14] = "";
int people = 0;
int remain = -1;
bool comparing = false;

char btn[2];
char buf[MSG_SIZE]; // buffer                ===
int cdev_id, dummy, summy; // device id and dummy    ===

  // ================    Declare Area    ======================
```

```
41   //                                ===
42     int server_fd;     // socket              ===
43     int sa_in_len;     // length of struct sockaddr_in    ===
44     socklen_t fromlen;  // length of sockaddr_in (socklen_t) ===
45     int msg_check;     // check the receive and send    ===
46     int option = 1;    // socket option          ===
47     struct ifreq ifr;      //              ===
48     struct sockaddr_in server;  //             ===
49     struct sockaddr_in from;  //              ===
50
51     char* ip_addr;     // ip address for this server    ===
52
53   pthread_t thread_r;
54     //                                ===
55     // ==================   Declare Area   ====================
56
57
58 void error(const char *msg){
59   perror(msg);
60   exit(0);
61 }
62
63 void* devRead(){
64   // setup for wiringPi
65   wiringPiSetup();
66   // setup pins
67   pinMode(27,INPUT);  // btn1
68   pinMode(0,INPUT); // btn2
69   pinMode(1,INPUT); // btn3
70   pinMode(24,INPUT);  // btn4
71   pinMode(28,INPUT);  // btn5
72   while(1){
73       if(digitalRead(27)||digitalRead(0)||digitalRead(1)||
     digitalRead(24)||digitalRead(28)){
74         summy = read(cdev_id, btn, sizeof(btn));
75         if(summy != sizeof(btn)) {
76           printf("Write failed, leaving...\n");
77           break;
78         }
79         if (strlen(btn) > 0) printf("%s\n",btn);
80         if (isMaster){
81           from.sin_addr.s_addr = inet_addr("128.206.19.255");
82           if (sendto(server_fd, btn, sizeof(btn), 0, (struct
     sockaddr*)&from, fromlen) < 0) {
83             error("sendto");
84           }
85         }
86         // btn store the info from device
87         // btn should send to slave
88       }
89
90       delay(150);
91   }
92   pthread_exit(0);
93 }
94
95 int main(int argc, char *argv[]){
```

```
96  // check the argument, have to have port number
97  if (argc < 2) {
98    printf("ERROR: no port find\n");
99    exit(0);
100 }
101
102 srand(time(NULL));
103
104
105
106
107
108 // Open the Character Device for writing
109 if((cdev_id = open(CHAR_DEV, O_RDWR)) == -1) {
110   printf("Cannot open device %s\n", CHAR_DEV);
111   exit(1);
112 }
113
114
115   // setup socket
116   if ((server_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
117     error("Socket Failed");
118   }
119
120   // erase the data from memory
121   sa_in_len =  sizeof(server);  // size: 16
122   bzero(&server, sa_in_len);    // erase the size of memory
123
124   // setup server socket
125   // INADDR_ANY tells socket to listen on all available
      interfaces
126   server.sin_family = AF_INET;
127   server.sin_addr.s_addr = INADDR_ANY;
128   server.sin_port = htons(atoi(argv[1]));
129
130   // bind socket
131   if (bind(server_fd, (struct sockaddr*)&server, sizeof(server))
      < 0) {
132     error("Bind Failed");
133   }
134
135   // set broadcast option
136   if (setsockopt(server_fd, SOL_SOCKET, SO_BROADCAST, &option,
      sizeof(option)) < 0) {
137     error("Socket option sets Failed");
138   }
139
140   // define ifreq
141   ifr.ifr_addr.sa_family = AF_INET;
142   strcpy(ifr.ifr_name, "wlan0"); // this is the local network
      name
143 // on desktop: enp0s25
144 // on       Pi: wlan0
145
146   // get the address from sock
147   ioctl(server_fd, SIOCGIFADDR, &ifr);
148   memcpy(&server, &ifr.ifr_addr, sizeof(server)); // copy the
```

```
    memory to address
149     ip_addr = inet_ntoa(server.sin_addr);
150     printf("Your IP address: %s\n", ip_addr);
151     //strcpy(master_ip, ip_addr);
152
153   // start thread
154   pthread_create(&thread_r, NULL, devRead, NULL);
155
156
157
158
159
160
161     // receiving and sending
162     fromlen = sizeof(struct sockaddr_in);
163     while(1){
164       // erase buffer
165       bzero(buf, MSG_SIZE);
166
167
168
169       // receiving
170       if (recvfrom(server_fd, buf, MSG_SIZE, 0, (struct sockaddr*)&
      from, &fromlen) < 0){
171         error("recvfrom");
172       }
173
174
175
176
177       //printf("master ip: %s, value: %d\n", master_ip, master_num)
      ;
178
179       // print receive
180       if (strncmp(buf, "WHOIS", 5) == 0) {
181         // ============================ WHOIS
     ==========================
182           printf("Someone: Asking who is master.....\n");
183       if (strncmp(ip_addr, master_ip, 14) == 0){
184         isMaster = true;
185       } else {
186         isMaster = false;
187       }
188         if (isMaster){
189           printf("System : You are the Master!!!!\n");
190           char msg[MSG_SIZE];
191           sprintf(msg, "Haoxiang on %s is Master!!\n", ip_addr);
192           from.sin_addr.s_addr = inet_addr("128.206.19.255");
193           if (sendto(server_fd, msg, sizeof(msg), 0, (struct
      sockaddr*)&from, fromlen) < 0) {
194           error("sendto");
195         }
196         } else {
197           printf("System : You are not master...\n");
198         }
199       } else if (strncmp(buf, "VOTE", 4) == 0){
200       // ============================ VOTE ============================
```

```
201       isMaster = false;
202       remain = people;
203       master_num = -1;
204       bzero(master_ip, 14);
205         printf("Start Voting...\n");
206         int vote_num = -1;
207         char msg[MSG_SIZE];
208         vote_num = rand() % 10;
209         sprintf(msg, "# %s %d\n", ip_addr, vote_num);
210         from.sin_addr.s_addr = inet_addr("128.206.19.255");
211         if (sendto(server_fd, msg, 40, 0, (struct sockaddr*)&from,
    fromlen) < 0) {
212           error("sendto");
213         }
214
215       } else if(strncmp(buf, "# ", 2) == 0){
216         remain = remain -1;
217       comparing = true;
218       char delim[] = " ";
219       char* ptr = strtok(buf, delim);
220       char* arr[3];
221       int i = 0;
222       while (ptr != NULL){
223         arr[i++] = ptr;
224         ptr = strtok (NULL, delim);
225       }
226       if(atoi(arr[2]) > master_num) {
227         master_num = atoi(arr[2]);
228         strcpy(master_ip, arr[1]);
229       } else if (atoi(arr[2]) == master_num) {
230         if (strncmp(arr[1], master_ip, 14) > 0){
231           master_num = atoi(arr[2]);
232           strcpy(master_ip, arr[1]);
233         }
234       }
235       } else if(strncmp(buf, "@", 1) == 0){
236       dummy = write(cdev_id, buf, sizeof(buf));
237       if(dummy != sizeof(buf)) {
238         printf("Write failed, leaving...\n");
239         break;
240       }
241       if (isMaster && fromMaster){
242         fromMaster = false;
243         continue;
244       }
245       if(isMaster){
246         from.sin_addr.s_addr = inet_addr("128.206.19.255");
247         if (sendto(server_fd, buf, sizeof(buf), 0, (struct sockaddr
    *)&from, fromlen) < 0) {
248           error("sendto");
249         }
250         fromMaster = true;
251       }
252     }else {
253         printf("Received a data: %s\n", buf);
254     }
255
```

```
256     }
257   close(cdev_id); // close the device.
258     return 0;
259 }
260 // The IP address: 128.206.19.machine_num
261 // The destination address: 204.159.253.118
262
263
264
265
266
267
268
269 /*
270 ================================================================================
271                                     Task note
272 ================================================================================

273 One master computer and several slaves comupter
274 implement server on Raspberri Pi
275 a client will ask all the student( include master ) "WHOIS"
276 if no master, the clients can ask "VOTE" to vote a new master
277
278 at the begining no one is master
279
280 "WHOIS" - ask who is master
281 "VOTE"  - ask everyone to vote a new master
282
283 to vote:
284   each client sent a broadcast "# ip_addresss vote_number"
285   highest number win
286   if number are same then check highest ip win
287
288 dynamically get ip
289 message size: 41
290 vote range [1,10]
291 the port should be an argument of the program
292
293 ================================================================================
294                           Understanding about the project
295 ================================================================================

296 1. Server and clients should be run under same network.
297 2. In order to communicate with other device, should enter in a
       same port.
298 3. To set as TCP (Connection based), shoud set socket as
       SOCK_STREAM type
299 4. To set as UDP (Connectionless), should set socket as SOCK_DGRAM
       tyep
300
301 Socket:
302   Scoket Structures:
303     sockaddr_in
304       struct sockaddr_in{
305         short sin_family;
```

```
306         unsigned short sin_port;
307         struct in_addr sin_addr;
308         char sin_zero[8];
309       }
310     in_addr
311       struct in_addr{
312         unsigned long s_addr; // address
313       }
314     sockaddr
315       struct sockaddr{
316         unsigned short sa_family;
317         char sa_data[14];
318       }
319
320 ------------------------------------------------------------------------

321   Useful Functions:
322     in_addr inet_addr(char addr)  // IPV4 format
323       same type  with struct in_addr, convert string to long
324     char* inet_ntoa(struct in_addr in)
325       convert long to string
326
327 ------------------------------------------------------------------------

328   Setting up a Socket
329     int socket(int domain, int type, int protocol)
330           AF_INET     TCP/UDP     default:0
331       set up a socket's attributes
332     int bind(int desc, struct sockaddr* addr, int addrlen)
333
334 ------------------------------------------------------------------------

335   sendto() and recvfrom() is same as write() and read()
336
337
338 It is better to add a error() function to print error if there is a
      error
339
340 void error(char* msg){
341   perror(msg);
342   exit(1);
343 }
344
345 ------------------------------------------------------------------------

346 #include <strings.h>
347 bzero(&memory_addr, mem_size);
348 bzero() erase the mem_size data from memory start from memory_addr
349
350
351 ========================================================================

352                           Ways to copy ip address
353 ========================================================================

354   struct ifreq ifr;
355   struct sockaddr_in sin;
```

```
1.
  memcpy(&sin, &ifr.ifr_addr, sizeof(sin));
  *inet_ntoa(sin.sin_addr);
2.
  struct sockaddr_in* ipaddr = (struct sockaddr_in*)&ifr.ifr_addr;
  *inet_ntoa(sin.sin_addr);
*/
```