|  |  |
| --- | --- |
| \\cetus.ece.missouri.edu\users\fischerjd\Desktop\Current Semester\common\logo-clear.png | University of Missouri – Columbia  Department of Electrical Engineering & Computer Science  ECE 4220 Real Time Embedded Computing |

**Airborne Virus Sanitation System**

Jacob Alongi | Sam Bishop | Michael Boeding

AVSS

December 2, 2020

# Introduction & Motivation

In this experiment, students will be exposed to:

* Different approaches for creating periodic tasks in Kernel space.
* Learn about communication between *Kernel space* and *User space*.
* Use the **MQTT** protocol to communicate between Server and Client to pass message over the network.

# Objective

In this experiment, students will be exposed to:

* Different approaches for creating periodic tasks in Kernel space.
* Learn about communication between *Kernel space* and *User space*.
* Use the **MQTT** protocol to communicate between Server and Client to pass message over the network.

# System Overview

Initially I took Lab 1 kernel module and added a few changes to allow it to work:

* Adding Kthreads function from the example file given and modifying it to work with my code

The only changes I made here since it was working code for creating a kthread, was adding the gpio mapped with ioremap, and inside the while loop adding iowrite32(0X40, gpset0); which wrote to pin 6 on the Pi which is where the speaker is located. After that line of code I added a udelay(period); which is set in another section, but the udelay dictated the frequency that was output. Then I cleared the corresponding bit in gpclr0 to clear the register.

* Removing the code inside the switch statement that decided which pin to use for the buttons and replacing it with arbitrary values that were within the confines of human hearable decibels

Since the buttons were already correctly mapped, I just used a global variable called period that was set based on the button pressed. This value is later used in the Kthread to dictate how long the udelay is done for

* Adding device\_write from another given file, Lab6\_cdev\_kmod.c

The logic of “device write” is backwards to what you would assume. It actually reads the input in copy\_from\_user and writes it into the device driver. I initially assumed it was device as in “device write to user” instead its “device read from user and write to device” from the code the message sent from userspace is put into the msg variable which takes the first character in the array, ignoring everything else since we are expecting single character to dictate a note of A B C D and E, and then sets the period from that input

* Added Static struct file operations from given code

This is also from given code in Lab6\_cdev\_kmod.c which is needed to have device write work. I ignored device read since it was not needed for this part and leaving it caused errors during compile time

* Added to init\_button the steps to instantiate the kthread

which was given to us as well, but required you to actually start the thread with the wake up process, which was different then pthread that did not require you to explicitly start and stop the thread.

* Added registering the Major id for the process

Major = register\_chardev(0, PS\_NAME, &FOPS); is required as part since we need to know the thread id number to create the node located at /dev/Wugga. Once we knew the ID then we had to manually in the terminal make the node with sudo mknod /dev/Wugga c MajorNumber 0

# Project Description

After the Kernel module was making sound via the corresponding buttons, then I moved on the user space. I also took the given code from the class and made some subtle changes to make it work.

* Replaced gets(buffer); with scanf(‘%s”, &buffer);

this is due to the fact that the gets method is deprecated and didn’t work when I used it. Simply changing it to scanf fixed the problem

# Experimentation and Results:

The only trouble I had was recognizing the difference in the meaning of Device write and not read as I explained above about how from a user standpoint its actually reading from the user and writing to the device, which I initially thought as the device writing to user space. The other issue I had was when testing the sound, initially I didn’t hear anything and tried several other test to figure out what was wrong. Turns out it was because the range of values I specified couldn’t be heard by human ears, so I tampered with it for a little until I had a range of values I could hear. No idea if they actually correspond to notes.

# Outcome

# Code

**Kernel Module**

/\* Lab6\_kernel.c

\* ECE4220/7220

\* Author: Jacob Alongi

\*/

#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>

//new modules

#include <linux/fs.h>

#include <asm/uaccess.h>

#include <linux/module.h>

#include <linux/hrtimer.h>

#include <linux/ktime.h>

#include <linux/kthread.h> // for kthreads

#include <linux/sched.h> // for task\_struct

#include <linux/time.h> // for using jiffies

#include <linux/fs.h>

#include <linux/uaccess.h>

#include <asm/uaccess.h>

MODULE\_LICENSE("GPL");

/\* Declare your pointers for mapping the necessary GPIO registers.

You need to map:

- Pin Event detect status register(s)

- Rising Edge detect register(s) (either synchronous or asynchronous should work)

- Function selection register(s)

- Pin Pull-up/pull-down configuration registers

\*/

int irq\_ptr; // variable needed to identify the handler

// Helper functions

int init\_button(void);

void exit\_button(void);

//(PG 90) of manual list addresses

//Important: remember that the GPIO base register address GPSETis 0x3F200000,

//not the one shown in the BCM2835 ARM Peripherals manual.

#define GPIO\_BASE 0x3F200000 // GPSEL0 is at base address

#define BLOCK\_SIZE 4096

//0x 7E20 0004 -> 04/4 = x1

#define GPSEL1\_OFFSET 0x01

//0x 7E20 0008 -> 08/4 = x2

#define GPSEL2\_OFFSET 0x02

//0x 7E20 001C -> 1C -> 28/4 = x7

#define GPSET0\_OFFSET 0x07

//0x 7E20 0028 -> 28 -> 40/4 = 10 -> xA

#define GPCLR0\_OFFSET 0x0A

//0x 7E20 0034 -> 34 -> 52/4 = 13 -> xD

#define GPLEV0\_OFFSET 0x0D

//0x 7E20 0040 -> 64 -> 64/4 = 16 -> x10

#define GPEDS0\_OFFSET 0x10

//0x 7E20 007C -> 124 -> 124/4 = 31 -> x1F

#define GPAREN0\_OFFSET 0x1F

//0x 7E20 0094 -> 148 -> 148/4 = 46 -> x25

#define GPPUD\_OFFSET 0x25

//0x 7E20 0098 -> 152 -> 152/4 = 38 -> x26

#define GPPUDCLK0\_OFFSET 0x26 //

#define LED 3

#define CLR\_LED 6

//NEW DATA

unsigned long period = 0;

static struct task\_struct \*kthread1;

#define PS\_NAME "Wugga"

#define MSG\_SIZE 50

static int major;

static char msg[MSG\_SIZE];

static char send\_msg[MSG\_SIZE];

// Uses content stored in the buffer by USP to determine the frequency of the note

static ssize\_t device\_write(struct file \*filp, const char \_\_user \*buff, size\_t len, loff\_t \*off)

{

ssize\_t dummy;

if(len > MSG\_SIZE)

return -EINVAL;

unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long n);

dummy = copy\_from\_user(msg, buff, len); // Transfers the data from user space to kernel space

if(len == MSG\_SIZE)

msg[len-1] = '\0'; // will ignore the last character received.

else

msg[len] = '\0';

printk("Message from user space: %s\n", msg);

if(msg[1] == 'A'){

period = 500;

printk("Period is set to: %lu\n", period);

}else if(msg[0] == 'B'){

period = 600;

printk("Period is set to: %lu\n", period);

}else if(msg[0] == 'C'){

period = 700;

printk("Period is set to: %lu\n", period);

}else if(msg[0] == 'D'){

period = 800;

printk("Period is set to: %lu\n", period);

}else if(msg[0] == 'E'){

period = 1000;

printk("Period is set to: %lu\n", period);

}

return len; // the number of bytes that were written to the Character Device.

}

static struct file\_operations fops = {

.write = device\_write,

};

static irqreturn\_t button\_isr(int irq, void \*dev\_id){

disable\_irq\_nosync(79); // Disable 79

// Instructions/Configuration for Red LED and Button reads

unsigned long \*gpsel0 = (unsigned long\*)ioremap(GPIO\_BASE, BLOCK\_SIZE);

unsigned long \*gpset0 = gpsel0 + GPSET0\_OFFSET;

unsigned long \*gpclr0 = gpsel0 + GPCLR0\_OFFSET;

unsigned long \*gpeds0 = gpsel0 + GPEDS0\_OFFSET;

unsigned long \*gplev0 = gpsel0 + GPLEV0\_OFFSET;

unsigned long event\_detection = ioread32(gpeds0);

unsigned long mask\_BCM\_pins\_16\_to\_20 = 0x000F8000;// 0b 0000 0000 0000 1111 1000 0000 0000 0000

unsigned long pin\_used = event\_detection & mask\_BCM\_pins\_16\_to\_20;//16-20 mapped to buttons

//Button 1: 16

//2: 17

//3: 18

//4: 19

//5: 20

printk("%lx", pin\_used);

//select all LEDs for output mode

// need to shift 2 bits as LEDs are bit2-5

// iowrite32(0x2490 << 2, gpsel0); //1001 0010 0100 0000

// iowrite32(0x9240, gpsel0);

iowrite32(0X40000, gpsel0);//0100 0000 0000 0000 0000

//2490

// 0010 0100 1001 0000

//<< 2

// 1001 0010 0100 0000

// 9240

switch (pin\_used)

{

/\*

pin allocation bit2 - bit5 for LEDs:

b2,3,4,5: red,yellow,green,blue

\*/

case 0X10000://red BCM 16

// ((ioread32(gplev0) & 1 << 2) == 0) ? iowrite32(1 << 2, gpset0) : iowrite32(1 << 2, gpclr0);

period = 500;

break;

case 0X20000://yellow 17

// ((ioread32(gplev0) & 1 << 3) == 0) ? iowrite32(1 << 3, gpset0) : iowrite32(1 << 3, gpclr0);

period = 600;

break;

case 0X40000://green 18

// ((ioread32(gplev0) & 1 << 4) == 0) ? iowrite32(1 << 4, gpset0) : iowrite32(1 << 4, gpclr0);

period = 700;

break;

case 0X80000://blue 19

// ((ioread32(gplev0) & 1 << 5) == 0) ? iowrite32(1 << 5, gpset0) : iowrite32(1 << 5, gpclr0);

period = 800;

break;

case 0://all 20

// ((ioread32(gplev0) & 1 << 5) == 0) ? iowrite32(0xF << 2, gpset0) : iowrite32(0xF << 2, gpclr0);

period = 1000;

default:

break;

}

iowrite32(0x001F0000, gpeds0);

enable\_irq(79); // Re-enable 79iowrite32(1 << 3, gpclr0); // GPCLR pin 3 to LOW

printk("Interrupt handled\n");

//for event detection status registers:

//The bit is cleared by writing a “1” to the relevant bit. (pg 96 GPIO)

iowrite32(0xFFFFFFFF, gpeds0);

return IRQ\_HANDLED;

}

int kthread\_fn(void \*ptr){

unsigned long \*gpsel0 = (unsigned long\*)ioremap(GPIO\_BASE, BLOCK\_SIZE);

unsigned long \*gpset0 = gpsel0 + GPSET0\_OFFSET;

unsigned long \*gpclr0 = gpsel0 + GPCLR0\_OFFSET;

unsigned long j0, j1;

j0 = jiffies;

j1 = j0 + 10\*HZ;

while(time\_before(jiffies, j1)){

schedule();

}

while(1){

// msleep(1000);

if(kthread\_should\_stop()){

do\_exit(0);

}

//0010 0000

iowrite32(0X40,gpset0);// 0100 0000 -> pin 6

udelay(period);

iowrite32(0X40, gpclr0);

udelay(period);

}

return 0;

}

int init\_button()

{

int dummy = 0;

unsigned long \*gpsel0 = (unsigned long\*)ioremap(GPIO\_BASE, BLOCK\_SIZE);

// Then you can offset to the addresses of the other registers

unsigned long \*gpsel1 = gpsel0 + GPSEL1\_OFFSET;

unsigned long \*gpsel2 = gpsel0 + GPSEL2\_OFFSET;

unsigned long \*gppud = gpsel0 + GPPUD\_OFFSET;

unsigned long \*gppudclk0 = gpsel0 + GPPUDCLK0\_OFFSET;

unsigned long \*gparen0 = gpsel0 + GPAREN0\_OFFSET;

printk("Begin INIT Instructions.\n");

// configure all ports connected to the push buttons as inputs. (Pg 101)

// 4 for gpsel1 (buttons 1-4 BCM 16-19)

iowrite32(0x0, gpsel1); // GPSEL pin 16 to input mode

// 1 for gpsel 2 (button 5 BCM 20)

iowrite32(0x0, gpsel2); // GPSEL pin 20 to input mode

/\*

follow steps from page 101:

The GPIO Pull-up/down Clock Registers control the actuation of internal pull-downs on

the respective GPIO pins. These registers must be used in conjunction with the GPPUD

register to effect GPIO Pull-up/down changes. The following sequence of events is

required:

\*/

// 1. Write to GPPUD to set the required control signal (i.e. Pull-up or Pull-Down or neither

// to remove the current Pull-up/down)

iowrite32(0b01, gppud);

// 2. Wait 150 cycles – this provides the required set-up time for the control signal

udelay(100);

// 3. Write to GPPUDCLK0/1 to clock the control signal into the GPIO pads you wish to

// modify – NOTE only the pads which receive a clock will be modified, all others will

// retain their previous state.

iowrite32(0x001F0000, gppudclk0);

// 4. Wait 150 cycles – this provides the required hold time for the control signal

udelay(100);

//5. Write to GPPUD to remove the control signal

iowrite32(0x0, gppud);

//6. Write to GPPUDCLK0/1 to remove the clock

iowrite32(0x0, gppudclk0);

// Enable (Async) Rising Edge detection for all 5 GPIO ports.

iowrite32(0x001F0000, gparen0);

// Request the interrupt / attach handler

dummy = request\_irq(79, button\_isr, IRQF\_SHARED, "Button\_handler", &irq\_ptr);

//NEW KTHREAD

char kthreadid[5] = "Wugga";

kthread1 = kthread\_create(kthread\_fn, NULL, kthreadid);

if((kthread1)){

wake\_up\_process(kthread1);

}

major = register\_chrdev(0, PS\_NAME, &fops);

if (major < 0) {

printk("Registering the character device failed with %d\n", major);

return major;

}

printk("major: %d\n", major);

printk("/dev/%s %d\n", PS\_NAME, major);

printk("Init Complete.\n");

return 0;

}

void exit\_button()

{

int ret;

printk("Init Clean.\n");

// Good idea to clear the Event Detect status register here, just in case.

//The bit is cleared by writing a “1” to the relevant bit. (pg 96 GPIO)

// iowrite32(0xFFFFFFFF, gpeds0);

// Disable (Async) Rising Edge detection for all 5 GPIO ports.

// iowrite32(0x00000000, gparen0);

// Remove the interrupt handler; you need to provide the same identifier

free\_irq(79, &irq\_ptr);

ret = kthread\_stop(kthread1);

if(!ret)

printk("Kthread stopped\n");

unregister\_chrdev(major, PS\_NAME);

}

module\_init(init\_button);

module\_exit(exit\_button);

**User Space**

/\* Lab6\_users.c

\* ECE4220/7220

\* Author: Jacob Alongi

This program allows you to enter a message on the terminal, and then it writes the

message to a Character Device. The Device should be created beforehand, as described

in the Lab6\_cdev\_kmod.c file.

Try this example together with the Lab6\_cdev\_kmod module. Once you enter a message, run

dmesg | tail ( | tail is not needed. With that, you'll only see the last few entries).

on a different terminal. You should see the message in the system log (printk'd in the

module). That would mean that the message is getting to kernel space.

Use elements from this example and the Lab6\_cdev\_user module in your Lab 6 programs. You may

need to modify a bit the callback functions in the module, according to your needs.

For the extra credit part of lab 6, you'll need to think about how to read messages coming

from kernel space, and how to create those messages in the module...

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#define CHAR\_DEV "/dev/Wugga" // "/dev/YourDevName"

#define MSG\_SIZE 50

int main(void)

{

int cdev\_id, dummy;

char buffer[MSG\_SIZE];

// Open the Character Device for writing

if((cdev\_id = open(CHAR\_DEV, O\_WRONLY)) == -1) {

printf("Cannot open device %s\n", CHAR\_DEV);

exit(1);

}

while(1)

{

printf("Enter message (! to exit): ");

scanf("%s", &buffer);

// fflush(stdout);

// gets(buffer); // deprecated function, but it servers the purpose here...

// One must be careful about message sizes on both sides.

if(buffer[0] == '!') // If the first character is '!', get out

break;

dummy = write(cdev\_id, buffer, sizeof(buffer));

if(dummy != sizeof(buffer)) {

printf("Write failed, leaving...\n");

break;

}

}

close(cdev\_id); // close the device.

return 0; // dummy used to prevent warning messages...

}