CHARACTER DEVICE

Op de Raspberry PI type 1A

SUZANNE PEERDEMAN & TIM VISSER

1 INTRODUCTION

For this assignment we decided to create a loadable kernel module. A loadable kernel module means that the module in question can be compiled without recompiling the entire kernel, so rather than changing the existing kernel the module will extend the kernel. This will save both time when compiling, and effort when coding. This paper will highlight two kernel modules that we have created: the first will deliver a message at boot and shutdown, and the second will control one of the GPIO (general purpose input/output) pins on the raspberry pi in order to blink a LED. The code will be written in the programming language C.

2 PREPARATIONS

Before getting started on writing the code for the driver, a few preparations are in order. Because the driver will be cross compiled on a windows machine, the following prerequisites are required:

- A virtualbox image with a Linux-OS. For this paper Ubuntu 16.04 was used
- Clion IDE
- RPi toolchain for cross-compilation (https://github.com/ raspberrypi/tools)

When developing software on a different platform than where the software will eventually be running, one is faced with several options. One could for example write the code on one platform, then transfer it to the target platform for compiling. This approach ensures that the software will run properly in the target environment. The downside, however, is that this method takes needless amounts of work and is often times (depending on the target environment) very slow.

This is why the RPi toolchain is very useful. The toolchain contains everything that is needed to develop software for the RPi externally, so that one can rely on the far superior CPU of a pc, rather than that of the RPi, when compiling. Information on how to set up the toolchain for Clion can be found here: https://stackoverflow.com/questions/19162072/installing-raspberry-pi-cross-compiler

3 KERNEL MODULE

As mentioned earlier, we will avoid having to compile the entire kernel by writing a kernel module instead. The result will be similar to writing a custom kernel, but instead allows us to only code the part that we want, and adding this to the existing kernel.

4 APPROACH

First and foremost we must make sure that the OS is up to date.

```
#Standard Linux-OS update routine
apt-get update -y
apt-get upgrade -y

#Update kernel
rpi-update
```

Once this is done the system will need to reboot. When the system has finished rebooting we can begin writing our driver. The first step to this is to download the RPi source code. The Wget tool will help with this.

```
sudo wget
    https://raw.githubusercontent.com/notro/rpi-source/master/rpi-
source -O /usr/bin/rpi-source
```

Now to make sure we can execute:

```
sudo chmod +x /usr/bin/rpi-source
```

Now to indicate that this is the final version of the script, and it will not be changed:

```
/usr/bin/rpi-source -q --tag-update
```

Then the code is ready to run.

```
rpi-source
```

And we are ready to start working on our driver.

5 CODING THE DRIVER

Now that the RPi is prepped and ready for our software, it is time to begin coding said software. We will begin by creating a folder, in our case naming it KernelModuleRPI. In this folder we create several files. To compile everything we will need a Makefile.

```
obj -m := suzanne_tim.o
```

Next is a header file which helps with calling our driver.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

void LED_aan(void) {
  signed int fd, ret;
  fd = open("/dev/suzanne_tim_driver", O_RDWR);
  if (fd < 0) {
    perror("Failed to open character device...");
  }
  ret = write(fd, "1", 1);
  if (ret < 0) {
    perror("Failed to write 1 to the character device...");
  }
}

void LED_uit(void) {
  signed int fd, ret;
  fd = open("/dev/suzanne_tim_driver", O_RDWR);
  if (fd < 0) {
    perror("Failed to open character device...");
  }
  ret = write(fd, "0", 1);
  if (ret < 0) {
    perror("Failed to write 0 to the character device...");
  }
}
```

The driver:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/gpio.h>
#include <asm/gpio.h>


#define DEVICE_NAME "suzanne_tim_driver"
#define DEVICE_MAJOR  69
#define GPIO_LED_PIN 21


MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Character Device");
MODULE_AUTHOR("Suzanne Peerdeman & Tim Visser");
```

```
static unsigned long procfs_buffer_size = 0;
static char buffer_data[3];
static char led_status = 0;

static int device_open(struct inode *inode, struct file *file);
static int device_release(struct inode *inode, struct file
    *file);
static ssize_t device_write(struct file *file, const char
    *buffer, size_t len, loff_t *offset);

static struct file_operations fops =
{
  .owner    = THIS_MODULE,
  .open     = device_open,
  .release  = device_release,
  .write    = device_write,
};


int suzanne_tim_init(void) {
        pr_alert("Character Device van Suzanne Peerdeman & Tim
            Visser\n");
        int ret;
        ret = register_chrdev(DEVICE_MAJOR, DEVICE_NAME, &fops);

        return 0;
}

void suzanne_tim_exit(void) {
        gpio_free(GPIO_LED_PIN);
        unregister_chrdev(DEVICE_MAJOR, DEVICE_NAME);
        pr_alert("Character Device ontkoppeld...\n");
}

static ssize_t device_write(struct file *file, const char
    *buffer, size_t len, loff_t *offset) {
        procfs_buffer_size = len;
        if (copy_from_user(buffer_data, buffer,
            procfs_buffer_size)) {
          return -EFAULT;
        }
        *offset += len;
        if (buffer_data[0] == '1') {
          led_status = 1;
          gpio_set_value(GPIO_LED_PIN, led_status);
          printk(KERN_ALERT "GPIO %d is set HIGH\n",
              GPIO_LED_PIN);
        } else if (buffer_data[0] == '0') {
          led_status = 0;
          gpio_set_value(GPIO_LED_PIN, led_status);
          printk(KERN_ALERT "GPIO %d is set LOW\n",
              GPIO_LED_PIN);
        }
```

```
        pr_info("Received data...");
        pr_info("Data received: %s\n", buffer_data);
        return procfs_buffer_size;
}

static int device_open(struct inode *inode, struct file *file) {
        gpio_set_value(GPIO_LED_PIN, led_status);
        printk(KERN_ALERT "GPIO %d is set HIGH\n", GPIO_LED_PIN);
        return 0;
}

static int device_release(struct inode *inode, struct file
    *file) {
        gpio_set_value(GPIO_LED_PIN, led_status);
        printk(KERN_ALERT "GPIO %d is set LOW\n", GPIO_LED_PIN);
        return 0;
}

module_init(suzanne_tim_init);
module_exit(suzanne_tim_exit);
```

And finally to test the driver:

```
#include "suzanne_tim.h"
#include <unistd.h>

int main (int argc, char **argv) {

    while (1) {
        LED_aan();
        usleep(1000000);
        LED_uit();
        usleep(1000000);
    }

    return 0;
}
```

When all this is done the driver needs to be compiled as 'kernel module'.

```
make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
```

This yields a .ko file. This is a 'kernel object', which can be loaded into the kernel by the command line actions:

```
sudo insmod suzanne_tim.ko
sudo mknod -m /dev/KernelModuleRPI c 666 0
```

And we are done!