Lab 4: Motor Control in Linux

Success is not measured by what you accomplish, but by the opposition you have encountered, and the courage with which you have maintained the struggle against overwhelming odds.

14-642, 18-349 Introduction to Embedded Systems

Submission Due: Thursday April 26, 2018, 3pm EST

Demo Due: Tuesday May 1, 2018 – even if you are using grace or late days

Contents

1 Introduction and Overview

This write up is long and contains a lot of information. Be sure to read it in its entirety before starting the lab.

1.1 Goal

The objective of this lab is to install Linux on your Pi and learn how to develop loadable kernel modules (LKMs) that allow user-space applications to interact with hardware devices and communicate over the network. The LKMs you build in this lab will allow you to interface with the encoders and DC motor on your I/O board. Using these kernel modules, you will build a user space application to control the position of the wheel using a PID controller and communicate between two Pis over ethernet.

1.2 Task List

1. Get the starter code from GitHub classroom (see Section 2)
2. Read the handout in its entirety, carefully.
3. Set up Raspbian on both Pi's. Make sure you can login via serial. (see Section 5.1)
4. Follow the instructions to set up static ip addresses in both pi's. Make sure you can ssh via your ethernet connection. (see Section 5.3).
5. Run the echo client and server to make sure your pi's can talk to each other. These will run in userspace. (see Section 5.4)
6. Make sure you can run the sample lkm.
7. Write, test, and debug your kernel space LKM('s).
8. Write your position PID, using the rotary encoder to control the position. (see Section 9.2)
9. Design your network protocol for PID over the network. This is a good time to do a draft version of your architecture. (see Section 9.4)
10. Implement and fine-tune the PID over the wire. (see Section 9.3)
11. Update your architecture.
12. Demo to TAs
13. Submit to GitHub (see Section 12). Make sure your code has met style standards!
14. Submit your GitHub link to canvas.

1.3 Grading

Kernel Space	70 Points
Motor Driver	15
PWM Driver	30
Encoder Driver	25
User Space	65 Points
Position control with PID	25
PID over the network	30
Organization and Architecture	10
Other	15 Points
Startup configuration	5
Style (including documentation following submission protocols)	10
TOTAL	150 pts

2 Starter Code

Use the magic link to create a repository for lab 4. This will contain starter code and the handout. Only commit source files. You should **not** commit and push your object files, executable files, and downloaded Linux kernel sources. These clog your commits with useless data, and they can cause conflicts when you run ”make” because you may have different versions of compiled files. You may run ”make clean” before ”make” to be sure you have the newest compilation.

3 Hardware Setup

Most boards should already be set up from last semester. Your board may be partially assembled. Use this section useful to double-check your board.

As a first step, add the platform and motor to your lab I/O kit. Directions for this can be found in the supplemental assembly guide. To interface with the motors and encoder simultaneously, we will want to configure the various dip switches on the I/O board. Set your switches to match the configuration described below.

SW301			
1	2	3	4
ON	OFF	ON	ON

SW501			
1	2	3	4
ON	ON	ON	OFF

SW601			
1	2	3	4
ON	ON	ON	ON

SW602			
1	2	3	4
ON	OFF	ON	ON

SW604			
1	2	3	4
ON	ON	ON	ON

SW606			
1	2	3	4
OFF	OFF	ON	OFF

SW702			
1	2	3	4
OFF	OFF	OFF	OFF

Note: SW501 is located underneath the debugger board

Please refer to `motor-kit-assembly-guide.pdf` for more information on how to setup your board. Make sure you connect the DC motor to the screw terminals for motor 1 (`P601`).

When connecting the AC adapter, make sure the black wire is connected to the screw terminal labeled (`GND`) and the white or red wire is connected to (`VS`).

WARNING: WIRING THE AC ADAPTER INCORRECTLY WILL CAUSE YOUR H-BRIDGE TO BLOW UP. PLEASE DOUBLE CHECK YOUR WIRING BEFORE POWERING THE PI. IF YOU MANAGE TO BLOW YOUR BOARD BY NOT FOLLOWING THE STEPS ON HOW TO CONNECT THE WIRES, YOU WILL LOSE 20% OF YOUR GRADE.

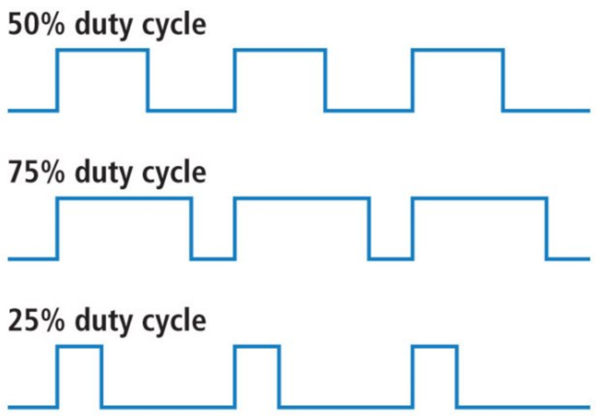
4 PWM and Encoders

In this section we will briefly describe how Pulse Width Modulation (PWM) and encoders work and how we use them in the context of this lab.

4.1 Pulse Width Modulation

Analog signals are continuous in both time and magnitude. There are multiple ways analog signals can be represented in the digital domain. Pulse Width Modulation (PWM) is a technique for producing analog signals from a digital output. In the context of this lab, we use PWM to control the speed of the motor.

The motor connected to your I/O board can be driven by a GPIO pin on the Pi through an H-Bridge for amplification and depending on whether you supply power to this pin, you can either power the motor at full power (100 % duty cycle) or no power (0 % duty cycle). To get intermediate speeds, we use PWM. In essence PWM modulates a digital signal by altering the proportion of time the signal is high over a consistent time interval. The consistent time interval over which PWM operates is governed by the PWM clock frequency and the proportion of time the signal is high in one clock period is called the duty cycle. The following image shows examples of various duty cycles:



By varying the duty cycle of the PWM, we vary the effective output voltage from the GPIO pin. For example, a square wave that has an amplitude of 5V operating with a 50 % duty cycle will effectively output 2.5V. Thus, we are able to regulate the input voltage to the motor to control its speed.

4.2 Encoders

Encoders are angular measurement devices. They are often used to measure the speed and direction of an object's rotation. Encoders do this through 2 channels (channel A and channel B) producing a signal that registers movements. The 2 channels of the encoder output two square waves that are out of phase by 90° as described below:

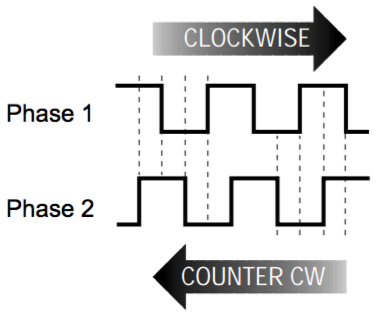
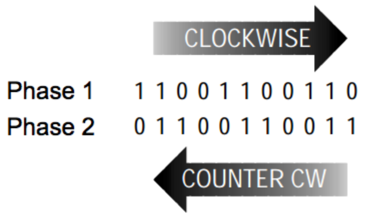


Figure 1. Quadrature waveforms from a rotary encoder contain directional information.



The output of the encoder is a Gray code, which we can use to determine its position. For example, if channel A is high and channel B has a rising edge, the encoder is moving clockwise. If channel A is low and channel B has a rising edge, the encoder is moving counter clockwise. By capturing all transitions of both channels, we can accurately determine the position of the encoder as it rotates.

5 Environment Setup

5.1 Installing Raspbian OS

In order to run Linux, we will need to reformat the SD card and load a new Linux image. The following instructions are applicable for most Linux machines.

First, we will need to format the SD card to be FAT32. Open GParted with your SD card inserted into your computer and use the drop down in the top right corner to select your SD card (the size should be approximately 7.4 GiB). If you do not have GParted installed then install it from the command line with the following:

```
$ sudo apt-get install gparted
```

In the GParted GUI, delete all partitions currently on the SD card by clicking on a partition and selecting ”Delete” from the right-click menu (be sure to commit the changes by clicking on the check mark in the top menu bar). Once they are all deleted, then select ”partition” and then ”new” from the top menu bar. From here, select the partition type to be FAT32 (be sure to commit this change as well). If you are using a Mac, you can use Disk Utility to format your SD card to be MS-DOS (FAT) format. On Windows `https://www.minitool.com/partition-manager/partition-wizard-home.html` works well.

Next, we will flash our Linux image to the newly formatted SD card. Download the Raspbian Jessie image from September 2016 at `http://downloads.raspberrypi.org/raspbian/images/raspbian-2016-09-28/` and extract the image file with `unzip`. To make sure the Linux image you have is not corrupted, run the following command in the directory which contains your zipped Linux image:

```
$ sha1sum 2016-09-23-raspbian-jessie.zip
```

```
e0eeb96e2fa10b3bd4b57454317b06f5d3d09d46  2016-09-23-raspbian-jessie.zip
```

or if you are on a Mac

```
$ openssl sha1 2016-09-23-raspbian-jessie.zip
SHA1(2016-09-23-raspbian-jessie.zip)= e0eeb96e2fa10b3bd4b57454317b06f5d3d09d46
```

and verify that the SHA1 sum matches the one listed on the Raspberry Pi website where you downloaded the image

Next, follow the instructions at `https://www.raspberrypi.org/documentation/installation/installing-images/README.md` to get the downloaded image onto your SD card based on your OS. After copying the file over to your SD card, run the following command to ensure your writes are flushed out to disk.

```
$ sudo sync
```

Now you can safely unmount the SD card. Your SD card is now set up to run Linux!

5.2 Connecting to your Pi

Now that our kernel and SD card are configured, we can boot into the Raspberry Pi kernel. There are two ways to do this: (1) over serial or (2) with Ethernet. We highly recommend that you use ethernet to connect to your Pi since it is much faster and you will need to use ethernet for the last part of the lab.

NOTE: The login is `pi` and the password is `raspberry`.

Option 1: UART

The kernel has its own UART driver that you can interface with. This option requires no setup on your part. Just use FTDIterm as you have in the past and connect to the Pi once it is plugged in with the USB cable. The kernel will output logs during boot, but it won't repeat prompts (like login and shell) until you poke it. If you don't see anything for a few seconds, press enter to repeat the last prompt. Use the ftditerm `--lf` flag to only send `\n` (versus `\r\n`) when you press enter. The kernel listens for 115200 baud UART by default. Eventually, you should see this:

```
$ sudo ftditerm.py -b 115200 --lf
opening first available ftdi device...
--- Miniterm on None: 115200,8,N,1 ---
--- Quit: Ctrl+]  |  Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---

Raspbian GNU/Linux 7 raspberrypi ttyAMA0

raspberrypi login: pi
Password:
```

If you don't see it after waiting for a few seconds, try pressing enter to poke it (as mentioned earlier).

You can copy files between your computer and the Pi using a USB drive. When a USB drive is plugged in to the Pi's USB port, it should be accessible at `/media/pi/`.

Option 2: Ethernet

You can connect to your Pi to via ethernet and then use SSH to communicate with it. It is difficult to connect to your Pi to CMU's network, so we only recommend using an ethernet adapter or direct connection to your computer. In general, SSH will be faster and it allows you to more easily move files with `scp` etc. This method requires you to know the IP address of your Pi once it is on the network. You should use the UART method to connect the first time to the Pi and configure the network. See the next section figure out how to configure your network.

5.3 Setting up your Pi to use Ethernet

We will configure the Pi to use a static IP address and connect an ethernet cable directly between the Pi and your computer. **Make sure that you assign different static IP addresses to your group's Pis, so that they will not conflict when they are connected.**

Check the settings for your VM. The networking should be NAT. These instructions expect that you will connect an ethernet cable directly from your computer to the Pi, or through a switch that is not connected to the internet. They use the "autoconfigure" IP address range (169.254). Other network solutions are possible – if your switch does DHCP (e.g. a home router), you will need to use the subnet and netmask that your router uses. (This will make it more challenging to demo in the lab, so we recommend not using this configuration unless you are comfortable reconfiguring the networking).

First, connect to your Pi over serial as described in the previous session. **Make sure that the computer is connected to the Pi with an ethernet cable as well.** This could be directly or through a hub/switch. Once you login you should have access to a shell. **Make sure you are doing this on the Raspberry Pi and not your VM or computer! Your prompt should look like** `pi@raspberrypi:~$`. Run the command `ifconfig`. The output should be something like as follows (your numbers will likely be different):

```
eth0      Link encap:Ethernet  HWaddr b8:27:eb:1f:2c:ea
inet addr:169.254.53.35  Bcast:169.254.255.255  Mask:255.255.0.0
inet6 addr: fe80::29e7:9986:5678:1651/64 Scope:Link
UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
RX packets:139 errors:0 dropped:0 overruns:0 frame:0
TX packets:74 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:23452 (22.9 KiB)  TX bytes:13373 (13.0 KiB)

lo         Link encap:Local Loopback
inet addr:127.0.0.1  Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING  MTU:65536  Metric:1
RX packets:256 errors:0 dropped:0 overruns:0 frame:0
TX packets:256 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:20736 (20.2 KiB)  TX bytes:20736 (20.2 KiB)
```

Write down the inet addr, bcast, and mask values for eth0. Next, run `netstat -nr`. The output should look like the following.

```
Kernel IP routing table
Destination      Gateway        Genmask         Flags   MSS     Window  irtt     Iface
169.254.0.0     0.0.0.0           255.255.0.0      U               0     0               0     eth0
```

Write down the destination and gateway values as well.

Finally, open the file `/etc/network/interfaces` in your favorite text editor. Note that the pi does not have Vim installed, but you can use vi (which is Vim's predecessor), instead (e.g. `sudo vi /etc/network/interfaces`). When you open the file, you will see something like this:

```
# interfaces(5) file used by ifup(8) and ifdown(8)

# Please note that this file is written to be used with dhcpcd
# For static IP, consult /etc/dhcpcd.conf and 'man dhcpcd.conf'
# Include files from /etc/network/interfaces.d:
source-directory /etc/network/interfaces.d

auto lo
iface lo inet loopback

iface eth0 inet manual

allow-hotplug wlan0
iface wlan0 inet manual
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf

allow-hotplug wlan1
iface wlan1 inet manual
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
```

You need to edit the file so that the `eth0` has a static ip. To do this, change `manual` to `static`, add `auto eth0` above it, and add the following fields immediately under it:

- address inet addr
- netmask mask
- network destination
- broadcast Bcast
- gateway gateway

where the second item on each line is the corresponding value you wrote down earlier. Here is what it should look like when it is done. Note that the values here correspond to the values used above.

```
# interfaces(5) file used by ifup(8) and ifdown(8)

# Please note that this file is written to be used with dhcpcd
# For static IP, consult /etc/dhcpcd.conf and 'man dhcpcd.conf'

# Include files from /etc/network/interfaces.d:
source-directory /etc/network/interfaces.d

auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
address 169.254.53.35
netmask 255.255.0.0
network 169.254.0.0
broadcast 169.254.255.255
gateway 0.0.0.0

allow-hotplug wlan0
iface wlan0 inet manual
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
```

```
allow-hotplug wlan1
iface wlan1 inet manual
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
```

Now reboot the pi using `sudo reboot`.

Ensure that your changes persisted by checking the `inet addr` by running `ifconfig` again and by checking the content of the `/etc/network/interfaces` file.

You should now be able to ssh into your pi from the VM by using the address as the hostname.

```
$ ssh pi@169.254.53.35
```

Once you have picked your communication method and logged in for the first time, run `uname -a` and you should see something like the following:

```
$ uname -a
Linux raspberrypi 4.4.21-v7+ #911 SMP Thu Sep 15 14:22:38 BST 2016 armv7l GNU/Linux
```

Now that you can ssh, you can also use `scp` to copy files between your pi and your computer, instead of the sd card.

If you're using an ethernet adapter and VMWare isn't detecting it, you may need to go into VMWare settings:virtual machine:network adapter:network adapter settings and select your adapter under bridged mode. You'll also need to go to the IPv4 settings for the ethernet adapter on the VM and change the drop-down setting to "shared with other computers" (see `https://askubuntu.com/questions/169473/sharing-connection-to-other-pcs-via-wired-ethernet`).

5.4 Testing a simple Network

The second user space application you write will require that your Pi's are able to communicate with each other using sockets. To help get started and test your network, we provided a simple echo client and server in the handout. First, connect the Pi's with an ethernet cable. Run server on one Pi and the client on the other. Instructions for using the echo client and server are in `echo_client.py` and `echo_server.py`. Make sure that this works before trying to do PID over the network.

6 Writing a Loadable Kernel Module

6.1 Loadable Kernel Modules

The most basic way to add code to the Linux kernel is to add source files to the kernel source tree and recompile the kernel. As you can imagine, this would be extremely slow and if you have a bug, the kernel could crash in ways that are difficult to debug. Your code also becomes tied to a specific kernel version and must be ported to each new Linux kernel (and there are A LOT of Linux kernel versions). The preferred method is to use an LKM so that you can add code to the Linux kernel while it is running. Your kernel code becomes more portable and users can load your module regardless of the Linux kernel version they may be running. You also get better debugging abilities because a bug that would have required a full reboot and kernel recompile to fix can now be a quick fix and short compile away!

These modules can accomplish many tasks, but they typically are one of three things: device drivers; file system drivers; system calls. The Linux kernel isolates certain functions, including these, especially well so they don't have to be intricately wired into the rest of the kernel and can be extended at runtime by LKMs. The important thing is that a LKM is **not a user program**. They run in kernel land and can easily crash the Linux kernel if you are not careful.

6.2 Preparing Kernel Sources

In this lab you'll have to use the VM as your host machine since the toolchain is only provided by the Raspberry Pi Foundation for Linux.

A kernel module interacts directly with the data structures and functions in the kernel. After compilation, these are known as symbols. Between versions of the kernel, some of these data structures and function prototypes change. To get our kernel module to match, we need to get an exact copy of the sources (Raspbian Jessie is using kernel 4.4.21), the configuration it was compiled with, and a table of symbols the compiler generated.

We can extract the configuration from a running kernel by reading `/proc/config.gz`. Most linux distributions support this, including Debian. The /proc/ filesystem is only available while the system is running (it's not actually on the SD card), so we'll copy the configuration to our home directory like this:

```
pi@raspberrypi:~$ sudo modprobe configs
pi@raspberrypi:~$ sudo zcat /proc/config.gz > raspbian.config
```

The best way to get this file into your VM is to do (from the VM), `scp pi@169.254.53.35:raspian.config .` (use your IP address and note the "." at the end of the command)

If you haven't configured ethernet, you'll need to copy this to your VM by first moving the SD card to your computer, copying the file to your computer and then the VM. You can move the `raspbian.config` to the `/boot` directory if you want, so that it's easier to find on the SD card. Move the SD card to your host machine and copy the `/home/pi/raspbian.config` file from your SD card to your working directory.

We can't get the symbol table from the running image, but the Raspberry Pi does provide it to us. For our 4.4.21 kernel on ARMv7 we want `https://github.com/raspberrypi/firmware/blob/8979042/extra/Module7.symvers`. Note the git commit description attached to this file. Another file in this repository, `https://github.com/raspberrypi/firmware/blob/8979042/extra/git_hash`, tells us which version of the kernel sources generated this file. We will use this information, summarized in the table below, to download and configure the sources.

Repo	Contents	Version
raspberrypi/linux	Kernel source (extended from kernel.org)	2d31cd5
raspberrypi/tools	Compilation tools (gcc, ld, as)	master
raspberrypi/firmware	Kernel and initial loader binaries for each version number	8979042

We've provided a script to download, decompress, and configure these sources automatically by running **in the VM**:

```
$ make sources
```

NOTE: Do NOT commit and push these kernel source files to GitHub. They are too large, and you do not need to track them.

The script downloads the repositories from the table and configure the sources, copies the configuration and symbol table and loads the configuration. More details can be found in the `Makefile`

If you want more information about compiling kernel modules, the Arch Linux wiki and Kernel.org docs are always informative:

1. `https://wiki.archlinux.org/index.php/Compile_kernel_module`
2. `https://www.kernel.org/doc/Documentation/kbuild/modules.txt`

6.3 Compiling a LKM

Run `make` to compile your module. This will compile the simple stub file that contains the skeleton implementation of a simple LKM. To compile the LKM, we need the Linux kernel source code header files, which we downloaded and built previously. These allow us to link code against the kernel internals so that once the LKM is loaded, the linker symbols will be filled out and actually call the right functions inside the kernel. A compiled LKM has a `.ko` file extension.

In order to give you some freedom with respect to how you decide to architect your system, you can modify the provided Makefile and/or add additional Makefiles with appropriate make targets. Keep in mind that when we run your code, we will simply run make at the top level directory and copy the relevant files onto your SD card, so modify your Makefile appropriately.

6.4 Getting your LKM onto the Pi

Depending on if you are using UART or Ethernet to communicate with your pi, you can transfer your files over in 2 different ways (but really, we **strongly** recommend you get ethernet working so you don't have to move the SD card back and forth a lot):

Option 1: UART

Cross compile your LKM on your host machine and then using the USB SD card adapter you were given with the course materials, insert the SD card into your host machine. The SD card should appear as a disk named ”boot”. Copy the compiled kernel module from your host machine onto the SD card. Then, remove the SD card, and insert it into your Pi and reboot. After logging in after reboot, you should see the corresponding LKM appear in the `/boot` directory.

Option 2: Ethernet

Cross compile your LKM on your host machine and SSH into the pi as described before. Then use the `scp` command to copy files over onto your pi using the following command

```
    $ scp driver.ko pi@169.254.53.35:~/
```

Now you should see `driver.ko` in your home directory.

6.5 Running a LKM

The following commands are used from the command line to interact with a LKM:

To load your LKM:

```
    pi@raspberrypi:\~\$ sudo insmod <your LKM>.ko
```

To unload your LKM:

```
    pi@raspberrypi:~$ sudo rmmod <your LKM>.ko
```

To list all LKMs your kernel has loaded:

```
    pi@raspberrypi:~$ sudo lsmod
```

To list the info about a LKM:

```
    pi@raspberrypi:~$ sudo modinfo <your LKM>.ko
```

To see the output from your LKM, you can run `dmesg | tail`

6.6 Debugging a LKM

The Linux kernel does not have JTAG enabled so we cannot use GDB to debug our LKMs (however, a great exercise would be to write a LKM that enables the JTAG interface in the Linux kernel so that GDB debugging could be possible...). Since we cannot use GDB, we will rely on `printk` (sound familiar?). `printk` is the kernel's version of userland `printf`. However, unlike `printf`, the output isn't shown to the user once it is called (otherwise the UART shell would be covered in kernel logging). Instead, we use the command `dmesg` to view the kernel `printk` log. Run `dmesg` and see the output of all the `printk`s across the system. To view just the most recent `printk` outputs, you can run `dmesg | tail`. The syntax for `printk` is simple:

```
    printk(KERN_INFO "Hello world\n");
```

The `KERN_INFO` is just a kernel macro to indicate the log level. (Note that there is no comma between the `KERN_INFO` and the format string). More info about other levels can be found at `http://www.makelinux.net/ldd3/chp-4-sect-2`. On that note, a lot of information about LKMs can be found online. Programmers have been writing LKMs for a long time and there are quite a few examples that are just a google search away. **But be warned**! Copying and pasting kernel code from some random website is a great way to invite bugs into your code. Always look up any kernel function you see used in some example code before just adding it to your own kernel.

6.7 Tips for Testing your LKM

1. Running the command `cat /dev/kernel_driver` will essentially preform an `open()`, `read()`, and then `close()` on your LKM and report the value from `read()` to the console output.
2. Running the command `echo 0 > /dev/kernel_driver` will essentially preform an `open()`, `write()`, and then `close()` on your LKM with the value "0".
3. If your kernel crashes when you load your kernel module, look at the output. It will likely tell in which function the exception was triggered. You will need to reboot your Pi on a kernel crash.
4. If using UART, be careful to not damage the adapter when removing and inserting the SD card over and over.
5. Note, if removing a kernel module crashes your kernel, the issue probably is in how you're cleaning up kernel resources (including GPIO pins and timers).

7 Tutorial

You may find parts 1 and 2 of this tutorial useful:

`http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/`

7.1 Making Character Device Driver

Once we create and load a LKM, how do we interact with it? One standard way is to use the character device driver interface. This allows for your LKM to appear as a device in `/dev/` inside the kernel filesystem. Your LKM then implements handlers for various file system operations (`write()`, `read()`, `open()`, `close()`, etc). A userland application can then get a file descriptor for your LKM and communicate to it using basic `write()` and `read()`. The kernel handles mapping the user file system requests to the file system handlers in your LKM.

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the /dev directory. Special files for char drivers are identified by a "c" in the first column of the output of ls -l. Block devices appear in /dev as well, but they are identified by a "b."

If you issue the ls -l command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. Traditionally, the major number identifies the driver associated with the device. Modern Linux kernels allow multiple drivers to share major numbers, but most devices that you will see are still organized on the one-major-one-driver principle.

The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how your driver is written, you can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices. Either way, the kernel itself knows almost nothing about minor numbers beyond the fact that they refer to devices implemented by your driver.

Module Init & Exit

The first 2 important functions which you need for any LKM are your init and exit functions. These dictates what happens when you call `insmod <yourlkm.ko>` and `rmmod <yourlmk>`.

The declaration for these special functions is as follows:

```
static int __init my_init(void){}
static void __exit my_exit(void){}
.
.
.
module_init(my_init);
module_exit(my_exit);
```

File Operations Structure

Once you are able to load an LKM successfully, we can now interact with it. One way is to use a character device driver. Character devices are represented as file structures in the kernel. The `file_operations` structure lists the callback functions that you can associate with the file operations. These include(`write()`,`read()`, `open()`, `close()`, etc) as mentioned previously.

```
static struct file_operations fops =
{
    .open = mydriver_open,
    .read = mydriver_read,
    .write = mydriver_write,
    .release = mydriver_release,
};
```

In this case, we are able to write our own logic to correspond to what should occur on a read or write to the file by editing those functions.

Prototypes are:

```
static int mydriver_open(struct inode *inodep, struct file *filep);
static int mydriver_release(struct inode *inodep, struct file *filep);
static ssize_t mydriver_read(struct file *filep, char *buffer, size_t len,loff_t *offset);
static ssize_t mydriver_write(struct file *filep, const char *buffer,size_t len, loff_t *offset);
```

Register Device

However, before we are able to use the nifty file operations, we have to first create our device class and device driver to use it.

Register a major number for character devices:

```
int register_chrdev (unsigned int major, // major device number or 0 for dynamic allocation
const char*  name,                  // name of this range of devices
const struct file_operations* fops); // file operations associated with this devices
```

For more information, `https://www.fsl.cs.sunysb.edu/kernel-api/re941.html`.

Register the device class:

```
struct class * class_create (
struct module* owner, // pointer to the module that is to own this struct class; usually 'THIS_MODULE'
const char*    name); // pointer to a string for the name of this class
```

For more information, `https://www.fsl.cs.sunysb.edu/kernel-api/re814.html`.

Register the device driver:

```
struct device * device_create (
struct class* class,   // pointer to the struct class that this device should be registered to
struct device* parent, // pointer to the parent struct device of this new device, if any; NULL works
dev_t devt,            // the dev_t for the char device to be added, MKDEV(major_number,0) works
void * drvdata,        // data to be added to device callbacks, if any; NULL works
const char* fmt,       // string for the device's name
...);
```

For more information, `https://www.fsl.cs.sunysb.edu/kernel-api/re812.html`.

Deregister Device

Similarly, once we are done with the device driver, we should destroy it.
Remove the device:

```
void device_destroy(
struct class* class, // pointer to the struct class that this device was registered with
dev_t devt);         // the dev_t of the device that was previously registered
```

For more information, `https://manned.org/device_destroy/ecf2591b`.

Unregister the device class:

```
class_unregister(struct class* class); // pointer to the struct class that this device was registered w
```

Remove the device class:

```
class_destroy(struct class* class); // pointer to the struct class that this device was registered with
```

Unregister the major number:

```
unregister_chrdev(int majorNumber,    // this the number returned from register_chrdev()
char[] DEVICE_NAME);// your string for the device name
```

7.2 Communicating to GPIO in a LKM

Your kernel in labs 1-3 ran in physical memory with no virtual memory. This is not the case with the Raspberry Pi Linux kernel. In order to use the MMIO GPIO on the Pi, we must translate the physical MMIO addresses to virtual ones. Luckily, the kernel does this for us. Inside a LKM, we use the following functions to request that the kernel virtually map a given physical address region into the virtual address mapping of a process.

```
void *ioremap(unsigned long phys_addr, unsigned long size);
```

This function returns a base *virtual address* that is mapped to the physical address `phys_addr` for `size` bytes. To unmap a given virtual memory region, we use the following kernel function:

```
void iounmap(void * virtual_addr);
```

This function takes in the virtual address returned by `ioremap` and removes the virtual to physical mapping requested previously by `ioremap`.

However, Linux offers a library to interface with GPIO. We can use the following functions:

```
#include <linux/gpio.h>      // Required for the GPIO functions
#include <linux/interrupt.h> // Required for the IRQ code

static unsigned int gpioMotorA = 1;
static unsigned int gpioMiddle = 0;

// how to set a gpio pin
gpio_request(gpioMotorA, "gpioMotorA");  // gpio for Motor A requested
gpio_direction_output(gpioMotorA, true); // Set the gpio to be in output mode and on
gpio_set_value(gpioMotorA, false);       // Set gpio to off
gpio_export(gpioMotorA, false);          // Causes gpio20 to appear in /sys/class/gpio
// the bool argument prevents the direction from being changed

// if the gpio is a button (middle)
gpio_request(gpioMiddle, "gpioMiddle"); // Set up the gpioMiddle
gpio_direction_input(gpioMiddle);       // Set the button GPIO to be an input
gpio_set_debounce(gpioMiddle, 200);     // Debounce the button with a delay of 200ms

// if we want IRQ for every gpio pin change
irqNumberMiddle = gpio_to_irq(gpioMiddle);
// GPIO numbers and IRQ numbers are not the same! This function performs the mapping for us

// This next call requests an interrupt line
resultMiddle = request_irq(irqNumberMiddle,            // The interrupt number requested
(irq_handler_t) my_irq_handler,// The pointer to the handler function below
IRQF_TRIGGER_RISING,          // Interrupt on rising edge (button press, not release)
"middleButton_gpio_handler",  // Used in /proc/interrupts to identify the owner
NULL);                        // The *dev_id for shared interrupt lines, NULL is okay

// prototype for IRQ Handler
static irq_handler_t my_irq_handler(unsigned int irq, void *dev_id, struct pt_regs *regs){}
```

8 Kernel Space

In the kernel space, you will be required to write one or more drivers to interact with the motor and encoders. Below, we have broken up the kernel space requirements roughly by functionality. You may choose to include all the functionality into one driver or into multiple separate drivers as you see fit. When deciding how many drivers you want and what functionality each should have, you should think about how you would use them, especially in the context of how your PID will use them.

You should make sure that the full functionality described below is implemented and functional when all kernel modules are loaded.

8.1 Motor Direction driver

Now that you have learned about LKMs and how they interact with the Linux kernel though the character device driver interface, you will write a LKM that can turn on and off the power to the motor as well as change direction. Use the stubs provided in the tutorial above as a starting point.

You will need to create and register your class in the init function and destroy it in the exit function. The write function will need to be overwritten to handle setting GPIO pins to change direction. For example, `echo 1 > /dev/motor` would allow the wheel to start moving clockwise, `echo 2 > /dev/motor` would allow the wheel to start moving counterclockwise, and `echo 0 > /dev/motor` would stop any motor movement. It would be a phenomenally good idea to overwrite the read function to give debugging information to the user.

You should refer to the datasheets to figure out the GPIO pin numbers needed for this. You will also need to understand how the direction is set based on the motor and H-Bridge datasheet. You might need to change the configurations of 2 switches as well so please refer to the schematic in the lab I/O specification sheet.

Furthermore, you need to configure the down button on the keypad to change direction. Note that other buttons may overlap with various other peripherals and buses.

This isn't graded, but it's a good way to get you on your way to writing a PID position controller and should help you understand some of the motivation for PID.

8.2 PWM driver

Once you have the basic motor wheel turning, you need to control the speed of the wheel using PWM.

You need to be able to echo a duty cycle between 0 to 100 and watch the speed of the wheel change accordingly.

For this task, it involves using a timer to keep track of each cycle and for what period of that cycle will you be sending 1's and 0's. One way to do this is to use the kernel's high resolution timers. The hr_timer has an interface which allows you to set the number of clock ticks (by passing in a ktime struct, which is also another way the kernel keeps track of time) it will count down to as well as passing in a function at the end of the clock period which is able to restart the timer on top of another computation needed.

Some useful functions:

```
#include <linux/hrtimer.h>
#include <linux/ktime.h>

static struct hrtimer hr_timer;
ktime_t ktime;

ktime = ktime_set( <secs>, <nsecs> );
hrtimer_init( <hr_timer struct address>, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
hr_timer.function = &my_hrtimer_callback;
hrtimer_start( <hr_timer struct address>, ktime, HRTIMER_MODE_REL );
```

```
    hrtimer_cancel( <hr_timer struct address> );


    enum hrtimer_restart my_hrtimer_callback( struct hrtimer *timer ) {

        hrtimer_forward_now(&hr_timer, ktime); // ensure the timer is done
        return HRTIMER_NORESTART;               // doesnt restarts the timer
    }
```

If you do not wish to use the hr_timer and are able to figure out an easier solution, we would love to see it!

8.3 Encoder driver

The last piece of functionality you will need is to do be able to read the motor as well as the rotary switch encoders. This involves figuring out which GPIO pins need to be read as well as how to maintain variables to remember the encoder values to help a user setting the target position using the encoders.

For every change in the encoder, your variables should be updated and be accessible through a device read since the user space applications will be using these variables constantly.

9 User Space Application

Now that you have implemented all the kernel modules, we will now interact with these kernel modules through two user space application. In particular, you will implement a PID controller to control the position of the wheel. Then, we will extend the application to do PID over the wire. For information regarding PID control and how to tune the parameters, refer to the lecture slides.

Since we are now running Linux on the Pi, we now have access to a variety of tools that we didn't before. Therefore you have a lot of freedom in deciding how to implement your user space application. The Linux distribution on your Pi comes with a Python interpreter, as well as the standard C compiler. So you can implement your use space application as a Python script, as a C file that gets compiled on the Pi into an executable or even as Bash script! **We highly recommend using Python since it tends to be easier for development of applications that do not require low-level hardware access or high performance.**

The following sections describe the requirements of the different components of the user space application

9.1 Setting target position using rotary encoder

Turning the rotary encoder should allow you to set a target position for the wheel. One full rotation of the rotary encoder should correspond to one full rotation of the wheel. Setting a target position should cause the motor to move to that target position.

9.2 Position Control using PID

The naive implementation of position control isn't robust to disturbances. To improve upon this, you will implement a PID controller that will restore the wheel back to its target position even if it is displaced, as demonstrated in lecture. Tune the constants of your PID controller to restore the position of the wheel as accurately as possible. To test if your position control is working correctly, set a target position for the motor and then displace the wheel. If your PID controller is working correctly you should feel greater resistance the more farther away you displace the wheel and letting it go should result in the wheel coming back to its target position. You will be evaluated on your position control based on the four metrics below:

- Rise time / Response
- Peak time
- Overshoot
- Settling Time

You are not required to use all three terms, but may do so if you wish. Be sure to spend adequate time tuning your PID as this will also make the second user space application much easier.

9.3 PID over the network

The final user space application will extend your previous position controller to control two motors over ethernet. The goal of this controller is that the two motors should maintain the same position in the face of disturbances. Some behaviors that we will expect to see are as follow:

- If you move one motor while not disturbing the other, the other motor should follow the position of the motor you moved, and vice-versa.
- If you hold the two motors at different positions, when you let them go, they should converge to the same position (this position will probably not be the original position of either motor).
- If your network PID controller is working correctly, if you hold one motor while turning the other away, you should feel increased resistance in both motors.

You are responsible for designing the network protocol that the Pi's will communicate over as well as any additional tuning of the PID that may be necessary.

You can look at the video in the slides a reference for the behavior.

While you are free to architect the PID over the network as you wish, we recommend doing the following. To set up communication between your Pi's, use a client server model. Designate one Pi as the client and the other as the server. On each Pi, run two threads. One thread should be the PID controller and the other thread should be the network thread (either the client thread or the server thread). Since you are using threads, you can use global variables to communicate between the network thread and the PID thread. What you choose to send over the network is up to you!

9.4 Organization and Architecture

For the last part of this assignment, you need to include a PDF in the top level of your repository called `lab4-architecture.pdf`. This pdf should have a diagram that describes the architecture and design of your network protocol. You should also include a few sentences that describe the protocol. If you considered multiple alternatives, state the trade offs between them and why you chose the option you did. Protocol transaction diagrams like the one shown in class are great ways to visually explain how network protocols function.

You can receive credit for this portion without actually implementing the PID over the network for a thoughtful explanation of how you would design such a protocol and trade-offs between multiple designs as well.

10 Startup Configuration

Most real-world embedded systems need to be able to cleanly restart on their own. For this reason, we want you to explore the startup configuration options in Linux. You should create an **init script** that executes all commands needed to setup and start running the networked PID controller. You should configure this init script to run at boot time. Make sure you run your python program in the background by using `&`.

A correctly set up startup configuration will allow the Pis to begin running your networked PID implementation immediately upon boot, without requiring you to log in via UART or SSH and manually start your program. It is acceptable if you require that the Pi running the server is booted before the Pi running the client.

11 Additional Resources

Information regarding which pins are connected to what can be found in the I/O board schematic. Consult the data sheet for the encoder regarding information about what the outputs of the 2 channels from the encoder looks like. More information about the motor can be found at `https://www.pololu.com/product/2822`

12 Submission

Unlike previous labs, we will have checkoffs near the due date for this lab for you to demo your system. You will still however need to submit your code on GitHub via the normal mechanism. Don't forget to upload the link to your GitHub repository on canvas. Note that you can upload this link at any time, even now! Make sure to tag your submission with the tag `lab4-submit`!

For this lab's style, we ask that you conform to PEP8 if you're using Python. We've provided a new makefile goal, `make sources`, to check the parts of PEP8 that we care about (this is what we'll use to grade your submission). If you're not using Python, we'll look through your code ourselves, but it's much easier for us (and you, since you'll know what we're grading against) if you use Python.

12.1 Demo

For the demo, we'll ask you to demo the following:

Basic PID – one board: Load your kernel module(s), run your user program, and test your PID implementation. Once you run your user program, the wheel's position should be set by the rotary encoder – turning the rotary encoder should cause the wheel to turn. Your PID controller, as listed above, should be responsive, avoid oscillation, and avoid large overshoots.

Networked PID – both boards: We'll ask you to demonstrate the networked PID controller as demonstrated in the videos in Section 14.

Headless Mode – both boards: You should be able to boot both boards and have them start running the Networked PID demo without typing any commands. It's fine if you need to boot boards in a specific order.

13 Tips and Tricks

1. Use `dmesg | tail` in order to only see the last few lines of the kernel log instead of the whole log every time you run `dmesg`.

2. Running the command `cat /dev/_driver` will essentially preform an `open()`, `read()`, and then `close()` on your LKM and report the value from `read()` to the console output.

3. Running the command `echo "0" > /dev/_driver` will essentially preform an `open()`, `write()`, and then `close()` on your LKM with the value "0".

4. If your kernel crashes when you load your kernel module, look at the output. It will likely tell in which function the exception was triggered. You will need to reboot your Pi on a kernel crash.

5. If using UART, be careful to not damage the adapter when removing and inserting the SD card over and over.
6. VMWare's HGFS (shared folders) are slow relative to the virtual disk. You will probably see a good performance improvement by working on the VM's virtual disk instead.
7. Your character device does not have to be thread safe (meaning you don't need to use mutexes) or support multiple instances in the same system. We will only be testing it by having a single process read or write characters to it.
8. Always check your error codes which are returned from most functions to ensure it does not return negative numbers (error occurred in that case)
9. If running `make` starts building the entire kernel, `root` is probably the owner of the `rpi/` directory. If `root` is the owner, run `sudo make veryclean`, followed by `make sources`. The owner of the `rpi/` directory should now be `ece349` (if you're using the VM). Running `make` should finish in under 2 minutes.

14 References

1. `http://derekmolloy.ie`
2. `http://www.makelinux.net/ldd3/chp-3-sect-2`
3. Single PID controller: `https://www.youtube.com/watch?v=tB3B-XFTsEc&feature=youtu.be`
4. Networked PID controller: `https://www.youtube.com/watch?v=3Y4-tWKp8yQ&feature=youtu.be`