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 Fundamentals of Embedded Systems

Due: December 6, 2016

Contents

1 Introduction

1.1 Overview

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. The LKM's you will build in this lab will allow you to interface with the encoders and motor on your I/O board. Using these kernel modules, you will build a user space application to control the position and speed of the wheel using a PID controller and visualize it using the screen.

1.2 Grading

This lab is broadly divided into 2 sections:

- Kernel Space **(50 points)**
  - Motor Driver **(10 points)**
  - PWM Driver **(25 points)**
  - Encoder Driver **(15 points)**
- User Space **(40 points)**
  - Setting the target position using the rotary encoder **(10 points)**
  - Position Control using PID **(15 points)**
  - Speed Control using PID **(10 points)**
  - Interfacing with the screen **(5 points)**
- Bonus: Startup Configuration **(5 points)**
- Code Style and Organization **(10 points)**

Updating Your Repository

To get the starter code and necessary software packages for this lab, download the `lab4` folder by running

```
$ git pull upstream master
```

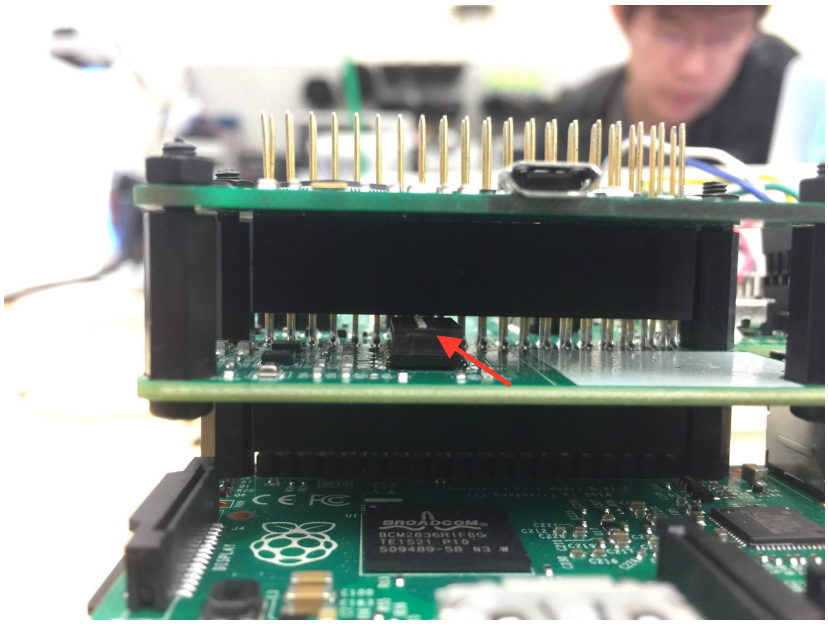
This folder contains various libraries needed by Linux to interface with the hardware on the I/O board.

2 Hardware Setup

As a first step, please add the platform and motor to your lab I/O kit. Directions for this can be found an a 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 showed in the images below.

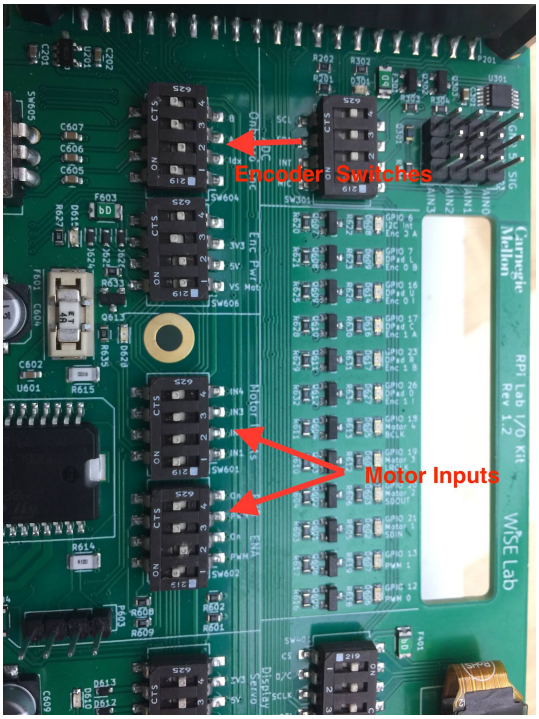
2.1 Switch Configuration

Switches Below the Debugger



SW301 Configurations

Switches on the IO Board



SW604, SW606, SW601, SW602, Configurations

2.2 Board Setup

If both members of your group missed the lab kit assembly lecture on 11/17 Thursday, please refer to the pdf posted on how to setup your board. You can obtain parts from a TA during office hours.

WARNING: WIRING THE AC ADAPTED 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. THIS APPLIES TO SETUP IN CLASS AS WELL.

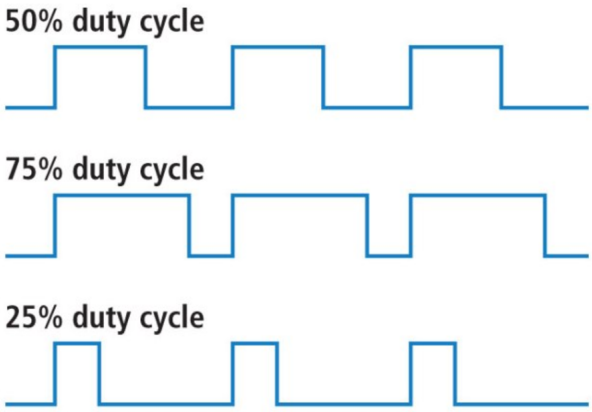
3 PWM and Encoders

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.

3.1 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 compared to when it is low 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 what various duty cycles look like



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 and so, we are able to control the speed of the motor.

3.2 Encoders

An encoder is an angular measurement device. They are often used to measure the speed and direction of rotation of an object. It does 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

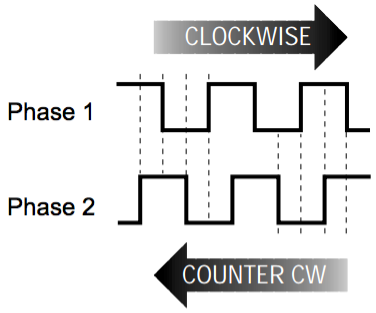
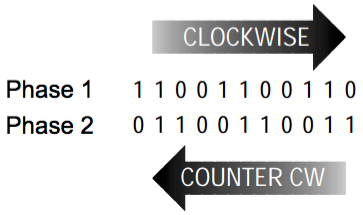


Figure 1. Quadrature waveforms from a rotary encoder contain directional information.



The output of the encoder can be interpreted to a gray code and we can determine changes to the position of the encoder based on this gray code. For example, if Channel A is high and Channel B has a rising edge, the encoder is moving clockwise and similarly 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

4 Environment Setup

4.1 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

Next, we will flash our Linux image to the newly formatted SD card. Download the latest Raspbian Jessie Lite image from `https://www.raspberrypi.org/downloads/raspbian/` 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-lite.zip
3a34e7b05e1e6e9042294b29065144748625bea8  2016-09-23-raspbian-jessie-lite.zip
```

or if you are on a Mac

```
$ openssl sha1 2016-09-23-raspbian-jessie-lite.zip
SHA1(2016-09-23-raspbian-jessie-lite.zip)= 3a34e7b05e1e6e9042294b29065144748625bea8
```

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!

4.2 Connecting to your Pi

Now that our kernel and SD card are configured, we can boot into the RPi kernel. There are two ways to do this: (1) over serial or (2) with Ethernet.

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:

```
$ ftditerm -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:
```

Option 2: Ethernet

If you can connect an Ethernet cable to your Pi, then you can use SSH and communicate to it. 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. This can be done by using the UART method to connect the first time to the Pi and then run `ifconfig` after logging in with the Ethernet cable attached. You should see something like the following:

```
pi@raspberrypi ~ $ ifconfig
eth0      Link encap:Ethernet  HWaddr b8:27:eb:16:21:af
          inet addr:192.168.0.123  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:249 errors:0 dropped:0 overruns:0 frame:0
          TX packets:190 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:23595 (23.0 KiB)  TX bytes:50249 (49.0 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:232 errors:0 dropped:0 overruns:0 frame:0
          TX packets:232 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:19248 (18.7 KiB)  TX bytes:19248 (18.7 KiB)
```

The key piece of information is the `inet addr:` field. Using this IP address, we can use SSH to connect with our Pi (no more UART)! From another terminal window, run the following using the IP address field from the `ifconfig` command:

```
$ ssh pi@192.168.0.123
```

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 3.18.11-v7+ #781 SMP PREEMPT Tue Apr 21 18:07:59 BST 2015 armv7l GNU/Linux
```

For many operations we want to do, you'll need to run with superuser permissions. You can easily drop into a root shell with:

```
pi@raspberrypi:~$ sudo su
root@raspberrypi:/home/pi#
```

We are now set up to build our own Linux kernel modules.

5 Writing a Loadable Kernel Module

5.1 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.

5.2 Preparing Kernel Sources

In this lab you'll have to use the VM or a native linux machine 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 configuartion 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:

```
root@raspberrypi:/home/pi# zcat /proc/config.gz > raspbian.config
root@raspberrypi:/home/pi# sudo sync
```

In case you don't have a `config.gz` file, run the following command

```
root@raspberrypi:/home/pi# sudo modprobe configs
```

and then extract the configuration.

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:

```
$ make sources
```

The script downloads the repos 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`

5.3 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.

5.4 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:

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 kernel\_driver.ko pi@192.168.0.123:~/
```

Now you should see an `kernel_driver.ko` in your home directory.

5.5 Running a LKM

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

To load your LKM:

```
$ sudo insmod <your LKM>.ko
```

To unload your LKM:

```
$ sudo rmmod <your LKM>.ko
```

To list all LKMs your kernel has loaded:

```
$ sudo lsmod
```

To list the info about a LKM:

```
$ sudo modinfo <your LKM>.ko
```

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

5.6 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.

5.7 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.

6 Tutorial

6.1 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
const char* fmt,      // string for the device's name
...);
```

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

Deregister Device

Simlarly, 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 with
```

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
```

6.2 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 RPi 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){}
```

7 Kernel Space

7.1 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 read file operation will need to be overwritten to handle the setting of the GPIO pins to change direction. For example, `echo 1 > /dev/motor` might allow the wheel to start moving clockwise and the corresponding write function would be setting the appropriate pins based on the number written.

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.

7.2 PWM driver

Once you have the basic motor wheel turning, you should write another driver which is able 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!

7.3 Encoder driver

For the last device driver, you will need to 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.

8 User Space Application

Now that you have implemented all the kernel modules, we will now interact with these kernel modules through a user space application. In particular, you will implement a PID controller to control the speed and position of the wheel. 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!

In order to interface with the screen, we need to setup Linux to use SPI and install some libraries. In order to set up the supporting libraries, we first downloaded the necessary packages from the Debian packages website. These packages are supporting libraries used by several applications (e.g. compression utilities like `zip`, a font engine like `freetype`). Next we set up the Python development environment and then compiled the necessary Python libraries like `PIL` and `spidev` from source and added them to our `PYTHONPATH`. For more information about compiling libraries like `PIL` from source, refer to `http://www.pythonware.com/products/pil/`

In order to configure your Pi with the appropriate libraries, first edit the `config.txt` file under `/boot` to uncomment the line that says `dtparam=spi=on`. Next, navigate to the `Imaging-1.1.7` and `spidev-3.2` directories and run the following command in each directory.

```
$ python setup.py install
```

That's it, you should now have all the required libraries to interface with the screen. To test, you can run

```
$python image.py
```

or

```
$python shapes.py
```

If you don't see an image on the screen, make sure your switches are configured correctly and that you followed the steps described above.

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

8.1 Setting target position using rotary encoder

Turning the rotary encoder should allow you to set a target position or target speed for the wheel. You can empirically measure what the maximum speed of the wheel is by supplying it 100% duty cycle and scale the output of the rotary encoder appropriately. Setting a target position/speed should cause the motor to move to that target position/rotate at that speed

8.2 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

8.3 Speed Control using PID

The naive implementation of speed control will not maintain its speed if some resistance is applied on the wheel. To improve upon this, you will implement a PID controller that will appropriately increase the power supplied to the motor when it encounters resistance in order to try and maintain the target speed. To test your system, you can apply some resistance to the motor. Increasing the resistance applied should increase the power output by your controller to ensure the motor maintains the same speed, so it should be harder and harder to apply resistance to the motor.

8.4 Interfacing with the screen

Lastly, in order to visualize the parameters being set and changed, you will interface with the screen to display the target position and the current position of the motor(for position control) or the target speed and current speed of the motor (for speed control). Since you have freedom with respect to how you implement your user space application, we provide a standard interface to interact with the screen driver. To run the screen driver, simply run

```
$ python screen.py
```

The screen driver for you that simply reads lines from 2 files called `tar_file` and `cur_file` that contain the target parameter and current parameter respectively. The screen driver will update the values being displayed on the screen based on what it reads from the `tar_file` and `cur_file`. If no new inputs are provided to either file, the screen will continue to hold the last displayed value.

So for example `tar_file` could look like

```
10
20
```

and `cur_file` could look something like

```
5
12
9
11
10
15
24
18
21
20
```

The screen will continuously update as it reads lines from either file and will hold its output when it's done parsing the last line of the `cur_file`. Note that the two files do not need to have the same number of entries. The screen driver will update if either file contains a new entry.

You are welcome to change the screen driver and how you interface with it, either by modifying the existing driver or creating your own. Just document it in your README.

9 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 can either use the `init.d` file or check out how to set cron jobs to be able to execute all commands needed to setup the PID controller; i.e: you should not need to do anything (echo-ing or running a Python file).

This will allow it to model a real world system where only thing required is to power up a device and it should automatically be able to perform what it was configured to do.

10 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`

11 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 gitlab. Add the relevant files into your working directory and call it `lab4`. Include a `README.txt` that tells us how to compile and run your system and include in there anything that we should keep in mind when running your system. Tag your submission with the tag `lab4-submit`

12 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)

13 References

1. `http://derekmolloy.ie`
2. `http://www.makelinux.net/ldd3/chp-3-sect-2`