Lab 5: 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.

Submission Due: Friday, May 1, 2020, 11:59pm
Demo deadline: Demos sessions to be announced on Piazza

Contents

1 Introduction and Overview

1.1 Goal

The objective of this lab is to install Linux on your RPi and learn how to use SPI to communicate between the RPi and STM32. The RPi will act as a master that is responsible for performing PID control and sending commands to the STM32. The STM32 will act as a slave and control the motor based on commands issued by the RPi.

1.2 Task List

1. Get the starter code from GitHub Classroom (see Section 2)
2. Set up Raspbian on your RPi (see Section 3.1)
3. Connect to the RPi via SSH (see Section 3.2).
4. Implement the motor encoder.
5. Fill in the SPI interrupt.
6. Tune and modify your PID controller.
7. Submit to GitHub (see Section 6). Make sure your code has met style standards!
8. Submit your GitHub link to canvas.
9. Demo at TA office hours.

1.3 Grading

Encoder Driver	40
SPI	25
Position control with PID	25
Style (including documentation following submission protocols)	10
TOTAL	100 pts

2 Starter Code

Use the link on Canvas to create a GitHub repository for lab 3. Create a new team if your partner hasn't made one or join your partner's team upon accepting the assignment.

Then copy over the previously implemented files. You will not need any of your syscall or real-time kernel code:

```
kernel/src/i2c.c
kernel/src/led_driver.c
kernel/src/svc_handler.c
kernel/src/uart.c (or kernel/src/uart_polling.c, you may use either)

kernel/asm/boot.S
```

You will be working on the following files (and may wish to create your own files as well):

```
kernel/src/spi.c
kernel/src/encoder.c
kernel/src/kernel.c

kernel/asm/boot.S

pid.py
```

3 Environment Setup

3.1 Installing Raspbian OS

Begin by downloading the latest version of `Raspbian Buster with desktop` from `https://www.raspberrypi.org/downloads/raspbian/`.

Once you have successfully downloaded the `.zip` file, you will want to ensure it has not been corrupted. To do so, run the following command in the directory which contains the the file:

```
$ sha256sum 2019-09-26-raspbian-buster.zip
```

or if you are on Windows:

```
>CertUtil -hashfile 2019-09-26-raspbian-buster.zip SHA256
```

or if you are on a Mac:

```
$ shasum -a 256  2019-09-26-raspbian-buster.zip
```

Ensure that the resulting hash matches the SHA-256 hash listed on the download page.

In order to run Linux, we will need to flash your SD card with the downloaded Raspbian image. Follow the instructions at `https://www.raspberrypi.org/documentation/installation/installing-images/README.md` to flash your SD card using Etcher. Your SD card is now set up to run Linux!

3.2 Connecting to your Pi

Unfortunately, on recent versions of Raspbian, the SSH server is not enabled by default. You will need to enable it manually. To do so, follow the instructions at: `https://www.raspberrypi.org/documentation/remote-access/ssh/`. If you do not have access to a monitor and an HDMI cable, you will need to perform the headless setup laid out in Step 3.

Next reboot your RPi (with the SD card and Micro USB power cable inserted of course), and plug in the ethernet cable between the RPi and your computer (you may need an ethernet-to-USB adapter if you do not have an ethernet port). Alternatively, if you have access to your router, you can connect your Pi to your router.

Now you will be able to SSH into RPi using the address: `raspberrypi.local` with the username `pi` and the password `raspberry`. It is suggested that you do so from your host rather than the VM, as it can be difficult to configure your VM network settings correctly to allow SSH connection to the RPi on some systems.

If you are on Windows, you may not be able to SSH from WSL. Instead, you should either SSH from CMD/Powershell or use a SSH client like PuTTy (`https://www.chiark.greenend.org.uk/~sgtatham/putty/`).

If you are not able to connect the Pi to your laptop via an Ethernet cable, feel free to use the lab computers as they have PuTTY and WinSCP.

3.3 Transferring files to your Pi

After you have verified that you can SSH, you should now be able to transfer files from your computer to your RPi. To do so, you can use the `scp` command as follows:

```
$ scp <source path> pi@raspberrypi.local
```

Alternatively, if you have a favorite SFTP client (e.g. WinSCP, Filezilla, Cyberduck, etc.) you may use that.

3.4 Enabling SPI on your Pi

In order to use SPI on your PI, you must first enable the SPI interface. To do so, use the following command:

```
sudo raspi-config
```

This will launch the RPi Configuration Tool. Navigate to `Interfacing Options > SPI`, and enable SPI.

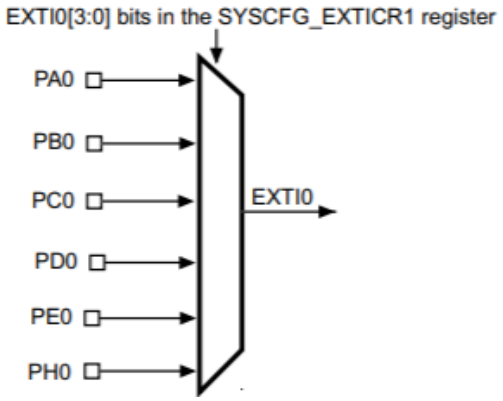
4 Writing The Encoder Driver

You will implement a driver which will track the position of the motor using the built-in encoder. You will only have to support the use of one encoder for this lab.

4.1 EXTI Overview

The encoder we are using is a standard quadrature encoder as discussed in class. There are two encoder lines connected to GPIO pins on your STM32 for each encoder. These will be set up as external interrupts. An external interrupt is an IRQ caused by a configurable edge condition from an external signal, in our case an enocder via GPIO pins.

Take a look at the external interrupt(EXTI) setup for the STM32 on page 205 of the reference manual. You will see that each EXTI is connected to a single channel, and the port is selected through a mux. Meaning, you may not have an external interrupt on both PA_1 and PB_1. You must choose one port for each channel.



The EXTI library has already been implemented for you. Look at exti.h for documentation on how to use the library. You will have to fill out the boot.S table with your encoder IRQ handler based on the IRQ numbers of the EXTIs your system uses. Look at the table on pages 203-205 to find the IRQ numbers. Don't forget to enable the IRQs in the NVIC.

4.2 Encoder Overview

You can track the position of a quadrature encoder by traversing through a state-machine. The current state is based on the current GPIO input values coming from both channels of the encoder. Turning the wheel will cause one of the channels to transition from high to low or low to high at a time. This state transition can be detected via EXTIs. Moving clockwise around the state-machine represents moving in one direction, and counterclockwise the other direction. The state diagram is shown in Figure 1 below.

This allows you to track the position in a discrete number of ticks. Review Lecture 23: Embedded Control for more information.

4.3 Encoder Driver

You will be responsible for writing the following functions in `kernel/src/encoder.c`:

- `encoder_init` initializes the encoder. The `exti` and `nvic` libraries will be useful here.
- `encoder_stop` should disable EXTIs for the two encoder pins.
- `encoder_read` should return the position of the motor. You will need to empirically determine how many ticks of the rotary encoder and the wheel correspond to a full circle. Some tape along one spokes of the wheel can make this task slightly easier.

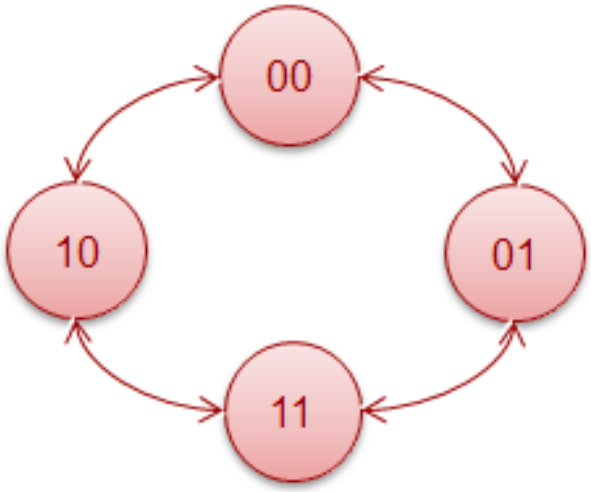


Figure 1: SPI data layout

- `encoder_irq_handler` will be responsible for updating the position. You should then place the updated position into the SPI data register so that it is transferred to the RPi.

In addition, your STM32 must display the current position (which you will send over SPI) on the LED Display. Take care to not hurt timing of your interrupts while doing this.

5 Linux Python Application

Now that the STM32 can track the position of the wheel, we will interact with the STM32 over SPI. In particular, you will tune a PID controller to control the position of the wheel. This PID controller has been mostly implemented for you in `pid.py`. You will be responsible for tuning the constants and adding any additional logic you wish in order to control the wheel with little overshooting or oscillation.

Transfer `pid.py` from your repo to the RPi (See Section 3.3). Then, you can run `pid.py` using the following command:

```
$ python3 pid.py
```

5.1 Communicating between the RPi and the STM32 using SPI

The RPi and the STM32 will communicate over SPI with the RPi acting as the master. Since there are only two devices, slave select is unnecessary. SPI will be set up with Polarity and Phase set to high. SPI is full duplex, so we can send and receive at the same time. With every SPI transfer, the Pi will need to send a speed and direction to the STM32, and the STM32 will send the current motor position back to the RPi.

On the RPi's end, to initiate a SPI transfer to the STM32, you will need to use the `spi.xfer2` function from within the main loop in `pid.py`. The input into this function is an array of data to be transferred, and the return value is an array of data received as a reply. See `http://tightdev.net/SpiDev_Doc.pdf` for documentation of the spidev library.

On the STM32's end, you will need to intialize SPI and set up SPI interrupts. See the provided SPI functions in `spi.c`. You are responsible for filling out the `spi_slave_irq_handler` to receive and process the data from the RPi. Note that current code will interrupt you on both RXNE and TXE. In your final implementation, you need not handle TXE interrupts, as you can write the updated position into the SPI data register in `encoder_irq_handler`. This works because the SPI peripheral does not clear the TX data register on transmit. Rather, if you do not manually overwrite the data, the peripheral will repeatedly transmit whatever was previously in the register whenever the master drives the clock.

To test that SPI works between the RPi and the STM32, you can begin by passing messages back and forth between

the RPi and STM32 and printing the received messages to their respective terminals.

5.2 Position Control using PID

Once you are able to send data between the RPi and STM32 using SPI, you can now implement the PID controller. You will need to fill in the main loop in `pid.py` to receive the current position, then determine the error from the target position. Using this error, you can then calculate the required speed and direction to spin the wheel toward the target. You then need to send this speed and direction to the STM32 to drive the motor.

On the STM32, when you receive this data in the `spi_slave_spi_handler`, you should set the motor to spin in the given direction at the given speed. We have provided a motor driver library in `motor_driver.h` to help you do so. **Note that if your timer channel has an N at the end of it (check the Arduino pin layout!), you will need to make a slight modification to your motor_driver.c. You will need to add the following line of code between at line 38:**

```
IS_COMP = 1;
```

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 by typing a number between 0-255 in the terminal from which you are running `pid.py` 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

Feel free to modify the given PID algorithm in any way you want to achieve the best behavior for your specific motor.

As a final note. Many of the motors will brown out at max speed due to insufficient current from the power supply. You may wish to set a maximum speed to prevent this issue.

6 Submission

Since the due date is the last day of class (May 1st), we will be arranging demo sessions during finals week as needed, based on the number of students who decide to do Lab 5. More details will be announced on Piazza. Stay tuned!

As with previous labs, we ask that you conform to doxygen for your C files.

6.1 Github

As always, if you have any comments or suggestions, please feel free to let us know in README.txt.

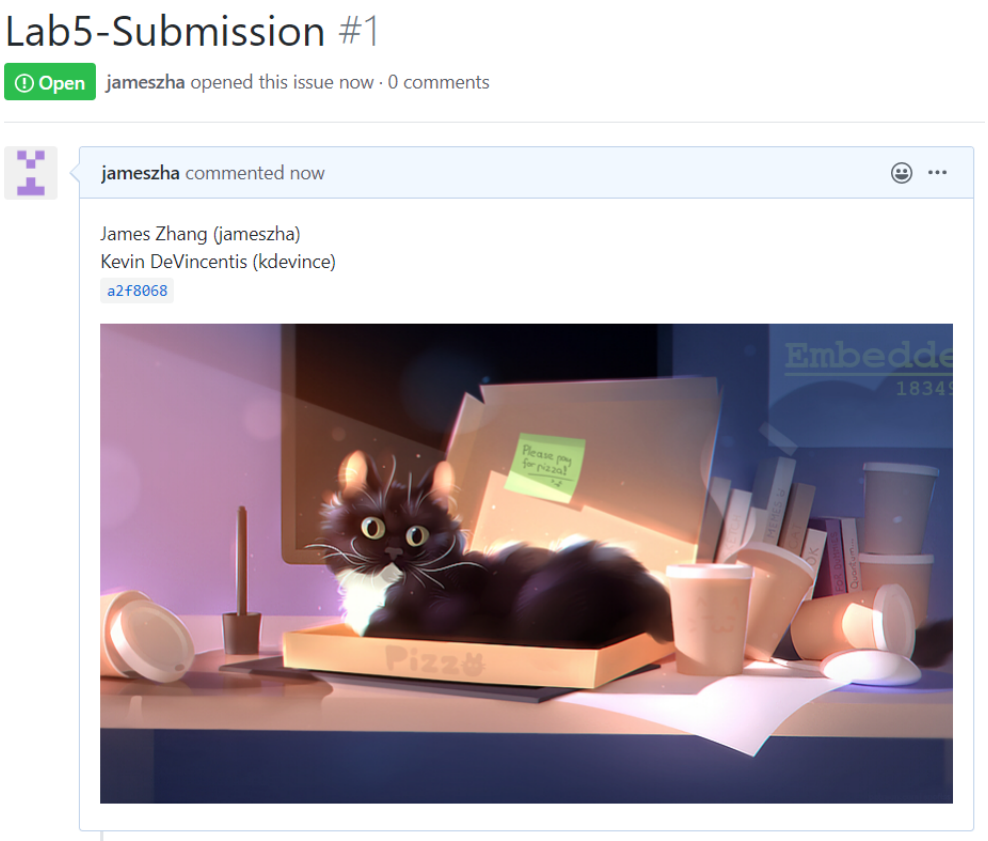
1. To submit the completed lab, create an issue on GitHub titled `Lab5-Submission`. In the comments section include your and your partner's name, AndrewID, and the commit-hash you want to submit.

When you are done, it should look like the screenshot in Figure 6.1. Pictures of cute animals are not needed but are recommended.

6.2 Demo

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

PID – one board: Show us your PID control. We will set the target to several different position, and look at how your wheel reacts. The LED display should also display the current position of the motor.



7 Tips and Tricks

1. Look at the lecture slides for a guide on how to tune the PID controller
2. The PID may need to change for each motor. We recommend tuning the PID together on a single board. You only need to demo with one board.
3. Don't over complicate the communication between the Pi and STM32. Keep it simple.

8 Style

8.1 Doxygen

Doxygen is a framework that allows you to automatically generate documentation from comments and markup tags inserted directly into the source. This style of embedding documentation in source is often used in industry. We will be using `doxygen` for code documentation for this course moving forward. A tutorial is available here if unfamiliar with it:`http://www.doxygen.nl/manual/docblocks.html`. Steps for setup and use of Doxygen are below:

```
Installation in the VM:
$ sudo apt-get install doxygen

Generating documentation:
$ make doc
```

The handout code contains a default configuration file called `doxygen.conf`. While most of the tags provided in the file already have the correct value, you are responsible for making sure that the `INPUT` tag specifies the correct source files (for this lab and future labs). **Specifically, you will need to have zero doxygen warnings for `kernel`.**

If the above command runs successfully, then you should have a `/doxygen_docs` directory with an `index.html` file. View it locally in a browser to see the documentation created from the code you wrote. When running this command, a file called `doxygen.warn` should have been created in the directory you ran `make doc`. Open this file to inspect any warnings. If there are any documentation warnings in the file about code you have written, then fix them. The TAs will check this file and take off style points if there are *any* warnings in this file.

8.2 Good Documentation

Good code should be mostly self-documenting: your variable names and function calls should generally make it clear what you are doing. Comments should not describe what the code does, but why; what the code does should be self-evident. (Assume the reader knows C better than you do when you consider what is self-evident.)

There are several parts of your code that do generally deserve comments:

- File header: Each file should contain a comment describing the purpose of the file and how it fits in to the larger project. This is also a good place to put your name and email address.
- Function header: Each function should be prefaced with a comment describing the purpose of the function (in a sentence or two), the function's arguments and return value, any error cases that are relevant to the caller, any pertinent side effects, and any assumptions that the function makes.
- Large blocks of code: If a block of code is particularly long, a comment at the top can help the reader know what to expect as they're reading it, and let them skip it if it's not relevant.
- Tricky bits of code: If there's no way to make a bit of code self-evident, then it is acceptable to describe what it does with a comment. In particular, pointer arithmetic is something that often deserves a clarifying comment.
- Assembly: Assembly code is especially challenging to read – it should be thoroughly commented to show its purpose, however commenting every instruction with what the instruction does is excessive.

8.3 Good Use of Whitespace

Proper use of whitespace can greatly increase the readability of code. Every time you open a block of code (a function, "if" statement, "for" or "while" loop, etc.), you should indent one additional level.

You are free to use your own indent style, but you must be consistent: if you use four spaces as an indent in some places, you should not use a tab elsewhere. (If you would like help configuring your editor to indent consistently, please feel free to ask the course staff.)

8.4 Good Variable Names

Variable names should be descriptive of the value stored in them. Local variables whose purpose is self-evident (e.g. loop counters or array indices) can be single letters. Parameters can be one (well-chosen) word. Global variables should probably be two or more words.

Multiple-word variables should be formatted consistently, both within and across variables. For example, "hashtable_array_size" or "hashtableArraySize" are both okay, but "hashtable_arraySize" is not. And if you were to use "hashtable_array_size" in one place, using "hashtableArray" somewhere else would not be okay.

8.5 Magic Numbers

Magic numbers are numbers in your code that have more meaning than simply their own values. For example, if you are reading data into a buffer by doing "fgets(stdin, buf, 256)", 256 is a "magic number" because it represents the length of your buffer. On the other hand, if you were counting by even numbers by doing "for (int i = 0; i ¡ MAX; i += 2)", 2 is not a magic number, because it simply means that you are counting by 2s.

You should use #define to clarify the meaning of magic numbers. In the above example, doing "#define BUFLEN 256" and then using the "BUFLEN" constant in both the declaration of "buf" and the call to "fgets".

This is especially important when putting constants in memory-mapped registers.

8.6 No "Dead Code"

"Dead code" is code that is not run when your program runs, either under normal or exceptional circumstances. These include "printf" statements you used for debugging purposes but since commented. Your submission should have no "dead code" in it.

8.7 Modularity of Code

You should strive to make your code modular. On a low level, this means that you should not needlessly repeat blocks of code if they can be extracted out into a function, and that long functions that perform several tasks should be split into sub-functions when practical. On a high level, this means that code that performs different functions should be separated into different modules; for example, if your code requires a hashtable, the code to manipulate the hashtable should be separate from the code that uses the hashtable, and should be accessed only through a few well-chosen functions.

8.8 Consistency

This style guide purposefully leaves many choices up to you (for example, where the curly braces go, whether one-line "if" statements need braces, how far to indent each level). It is important that, whatever choices you make, you remain consistent about them. Nothing is more distracting to someone reading your code than random style changes.