Chandler Janzen
Rev 3

CME331 – Lab 4

Microcontrollers
Pulse Width Modulation and Stepper Motors

Safety: In this lab, voltages used are less than 15 volts and this is not normally dangerous to humans. However, you should assemble or modify a circuit when power is disconnected and do not touch a live circuit if you have a cut or break in the skin.
Preparing for emergencies protects our lives and property. A classroom emergency posting is located in each classroom and lab near the main door of the room. Students are advised to review and be familiar with this classroom emergency posting and the College of Engineering Emergency Response Plan (ERP) The building must be evacuated when an alarm sounds for more than 10 seconds.

Objectives: By the end of this lab, you should have learned the following:

1. Know what a header file is and how to use one.
2. Understand what PWM is and how to use it.
3. Know what duty cycle is and how to calculate it.
4. Know how to configure the TM4C123GH6PM to output a desired PWM frequency and duty cycle.
5. Know what a stepper motor is and how to program an MCU to use one.
6. Become familiar with common uses of PWM and Stepper Motors.

Expectation: Getting the lab done is good. Getting expected results is better. Learning how to work with MCUs properly so that your code and work is modular, expandable, reusable and robust is best. Although these labs are to be done individually, you should envision yourself working on a small portion of a larger project and design team.

Code Submission: Always create your CCS projects with your NSID(s) in the project name (your NSID should be in the 'tree' on the left of CCS). Generic naming of projects (such as 'Lab 1' may result in a grade of 0. Always submit the **entire** CCS project folder (which should already include your NSID in the folder name) with relevant outputs electronically in a zip file (named <NSID>-<Lab#-Part#>) for each part (if applicable) through Canvas/BlackBoard no later than a week after your lab. Enhance the readability of your code by adding explanatory comments wherever appropriate. Always include your name, NSID, and date at the top of each code file. If you are unsure about what is expected, ask your instructor.

Working with a Partner: If you've worked with a partner, please indicate your partner's Name and NSID at the top of your PDF Document, as well as at the top of any source files (as a comment). Only one submission per pair is required.

Late Policy: Each lab is due one week after the scheduled lab time. Every student is granted 2 days late submission without penalty and 10% per day after; however, all submissions must be in prior to the start of the next lab.

1 Introduction

Labs 2 and 3 discussed the system timer and general purpose timers. It also mentioned that there are more timer based modules available in the MCU. This lab will explore one of them - the Pulse Width Modulation (PWM) module. We will also introduce a practical application of using a microcontroller: driving a stepper motor.

2 Background

For your convenience, here is a link to the MCU Datasheet: TM4C123GH6PM Microcontroller Datasheet

2.1 Header Files

One focus of these labs is get you into writing good, modular code. Modular, in this case, refers to the ability to reuse your code in another project without breaking existing functionality. An easy way to add existing items into your project is through header files. Header files (in C) are used to include **declarations** of functions or **symbol definitions** (also called macro definitions).

To use a header file, we include it into another source file using the syntax:

```
#include "my_header_file.h"
```

This include statement should always appear at the **top** of your source file. Why? That's where declarations go! When you hit the compile button, the source file gets process from top to bottom. We always start with declarations at the top of the file because it tells the compiler that the definition for that item exists (somewhere). If we started with function definitions, the first time the compiler gets to function call that has not been declared yet, it will throw an error and complain that it doesn't know what it is. *We must declare a function before we use it.*

When we include a header file at the top of a source file, it is (essentially) the same thing as copy-pasting the contents of the header file at the top of the source file. **You never want to include another source file (.c)**. If we want to use the same function in multiple source files, we can include the header file in each file; this will give each source file access to the functions that are declared in the header file. The definition for the function that is declared in the header file should only be defined **once** in a source file in the entire project (this is why we don't include source files in other source files - it's making duplicates!). Technically, the declaration can only be included once as well. This will cause problems when we include the same header file in multiple source files. To prevent multiple declaration errors, we need to add a **Macro Guard.** See Example 2.1:

Example 2.1 (*Header Files*)

```
/******* main.c ********/
#include "foo.h" //You can replace the contents of this line with the
               //contents of the header file.
int main(void)
while(1)
{
  GPIO_PORTF_DATA |= 0x02; //This is defined in the header file.
  foo_add(1,2); //This is declared in the header file.
}

--------------------------------------------------------------------------------
/******* foo.c ********/
int foo_add(int a, int b)
{
       return a + b;
}
```

```
--------------------------------------------------------------------------------
/******* foo.h ********/
#ifndef FOO_H //this is called a Macro Guard. It skips this file if FOO_H is
              //already defined.
#define FOO_H //first time the compiler runs this it creates a symbol
              //called FOO_H

// Only put declarations and symbol definitions in here...

// Symbol Definitions:
#define GPIO_PORTF_DATA (*((volatile unsigned long *)0x400253FC))

// Declarations:
int foo_add(int, int);

#endif //close the Macro Guard
```

A macro guard is a simple check to see if the contents of the file have already been discovered by the compiler (or not). The first two lines in the header file could be read as: "If FOO_H is not defined, then define FOO_H". The first time the file is processed, the symbol in the macro guard will not be defined yet. The very next thing it does is create the symbol (FOO_H). *The name of the symbol does not matter, but it must be unique!* One way we can ensure that the symbol is unique is to make the macro guard symbol the same as the file name - we can't have two files in our project with the same name! If the header file is included multiple times throughout the project, only the first time the compiler reaches the include statement does it actually include the contents. Remember - the compiler eventually combines all files in a project into a single table of objects; declarations are only needed to occur in a project once. We are simply ensuring that the order of operations of the compiler are correct.

Using header files will eventually save you time. It will also reduce how many lines of code that you have in one file; thus, making it easier to find lines of interest. They will also help prevent the infamous copy-paste errors that occur when grabbing code from other projects.

By now, you've probably mastered the art of creating symbol definitions for all of the registers. Normally when we choose a microcontroller from a vendor, we would expect some support in the form of debugging tools, header files, code examples, and datasheets. The header files, in particular, are used to create all of the necessary symbol definitions for the user to use - as you found out, it can be tedious! The header file "tm4c123gh6pm.h" provided includes over 10000 symbol definitions. You can, *and should*, use it in your future code.

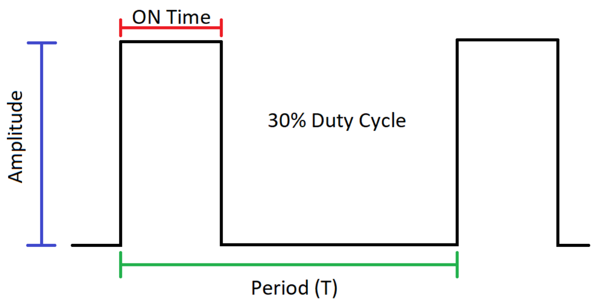
2.2 PWM

Pulse width modulation is a timer-based module built into the MCU. PWM allows the user to specify a *frequency* and a *duty cycle*. Upon configuring the PWM, a wave with the specified parameters will be automatically generated. PWM modules are commonly used for several things including but not limited to:

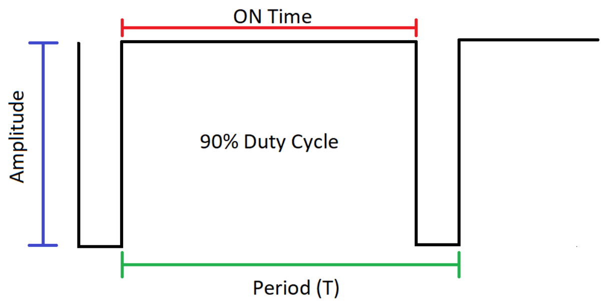
1. Controlling servo and stepper motors.
2. Controlling the power to a circuit through adjustment of the duty cycle (for example, speed control of a DC motor).
3. Generating frequencies for signals used as clocks.
4. Telecommunications for encoding purposes.
5. Audio/Video amplifiers.

2.2.1 Frequency and Duty Cycle

PWM modules can (in real-time) adjust their frequency and duty cycle. The frequency is related to overall time that the pulse is on and off for, where as the duty cycle is the ratio of on-time to the period. See Figure 1a and 1b. The



(a) Wave with 30% Duty Cycle



(b) Wave with 90% Duty Cycle

Figure 1: Pulse Width Modulation Examples

amplitude of the wave is typically equivalent to the logic levels used in the microcontroller. In the TM4C123GH6PM, it will output a 3.3V logic high and 0V (GND) logic low. As with anything microcontroller related, the PWM module should not be used to drive any large loads directlyf (including LEDs, motors, etc.) but should rather be used to control these devices through a switching component such as a transistor or mosfet.

2.2.2 Configuration and Options

This particular microcontroller contains two modules that each contain four different PWM generators, each capable of outputting two PWM signals (A and B for each generator for a total of 8 PWM pin outputs per module). Each generator can run at a different frequencies and duty cycles, but the outputs for each generator (A and B) are related.

A block diagram on Page 1232 gives a nice overview of the PWM modules - you will want to reference that when configuring the PWM because the naming convention can be confusing! Page 1233 outlines a list of external PWM pin options. You'll need to reference that table in order to determine which PWM module you are needing to use.

The Datasheet outlines a step-by-step procedure to initialize and use the PWM modules. See Page 1239: Section 20.4 Initialization and Configuration. **Note:** The first step is to enable the clock for the PWM module. The method (using RCC) is legacy and does not include the PWM1 module. To enable the clock to the PWM0 and PWM1 modules, you should use the updated register on Page 354: RCGCPWM. Also, if you are not using M0PWM1, you will need to pay attention to the values for each register to adjust for using different generators and whether you are using the A signal or B signal of the generator.

To assist you with configuring the PWM module and understand some of the configuration options, here are some explanations:

- **16 Bits**: Since the PWM timer only supports 16 bits, you can assume that all configurations related to time are 16 bits wide. This includes LOAD, CMP, COUNT, etc.
- **Run-Mode Clock Configuration (RCC)**: Page 254 - Adjusts the clock source to the PWM module. If your base clock frequency is 16Mhz, you may need to run a slower clock to your PWM module to achieve the desired frequencies. Included in this register is the option to *divide* the system clock that feeds the peripherals. Specific to PWM, there is an option in this register to only slow down the PWM module without effecting other modules (such as general purpose timers). *NOTE: changing the wrong bits in this register can have negative effects on the launchpad.*
- **Count Modes** Each PWM generator can be used to count down to zero from a specified value (PWM LOAD register), or use count-up/down mode that will count down to zero and then back up to the LOAD value. The impact of this is the difference between a left/right aligned pulse or a center aligned pulse. See page 1235 for more information. *One application of this is to implement a phase shift in the signal.*
- **PWM CMP (Duty Select)**: The value in this register establishes what the desired duty cycle of the wave is to be. The PWM generator will always count to the LOAD register, but will toggle the PWM output when it reaches the CMP value. The relationship between the LOAD register and CMP register is the duty cycle as a ratio (For

example, for generator 0A):

$$DutyCycle = \frac{PWM0CMPA}{PWM0LOAD} * 100 \tag{1}$$

or, if using an inverted PWM signal:

$$DutyCycle = \frac{PWM0LOAD - PWM0CMPA}{PWM0LOAD} * 100 \tag{2}$$

From these equations you can see that there is no option to use a different LOAD value for the A and B outputs for generator 0 - this implies that the A and B outputs can have different duty cycles but have a common frequency.

- **Synchronization between modules**: This MCU offers a huge complement of additional options and uses for the PWM module. One of which is the ability to synchronize several PWM modules such that their count values line up (if you have several generators with the same LOAD value (frequency) then how do make sure they count at the same time?) This can be extremely useful when signal inputs and outputs are time sensitive (an important factor when applying DSP techniques..)

There are many other configuration options that have been overlooked in this lab including dead-band control, synchronization with other peripherals, fault detection, and interrupts.

2.3 Stepper Motors

This is an excellent source for information on Stepper Motors, and much of the information presented here is taken from there.
A stepper motor is a type of motor that requires a sequence of pulses in order to drive the motor. They are useful in applications where accuracy of rotations are important. There are two types of stepper motors: unipolar and bipolar stepper motors. Furthermore, the drive configuration of the bipolar motors can be configured in several ways: bipolar-series, bipolar-parallel, or bipolar half-coil. Additionally, stepper motors can come in a variety of configurations including 4-wire, 5-wire, 6-wire, and 8-wire setups.

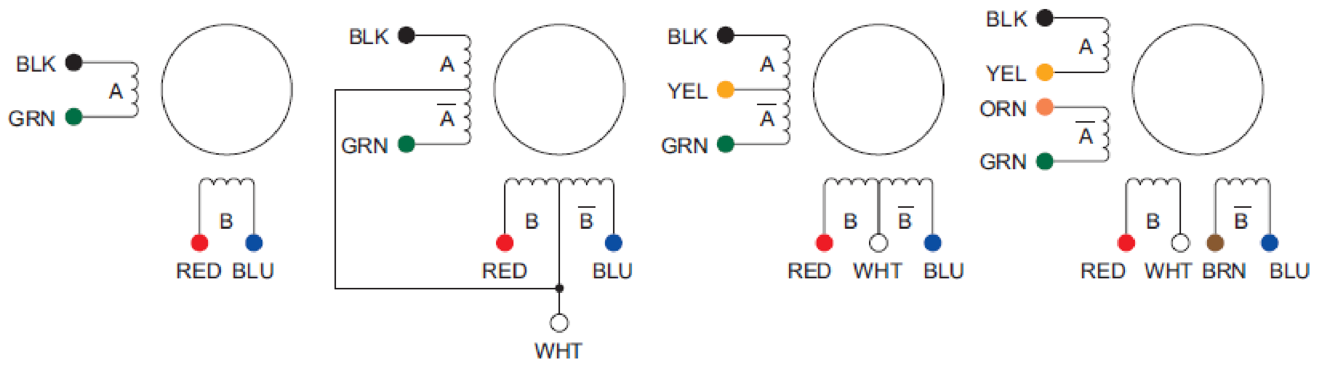


Figure 2: Common Stepper Motor Configurations

2.3.1 Unipolar vs Bipolar

The main difference between unipolar and bipolar stepper motors is the center tap wire, which splits the full coils of the winding in half. This could be done with one wire or two wires. If you remove the center tap, then it becomes a bipolar-series connection. The motor itself is not unipolar or bipolar per se, but manufacturers can classify stepper motors

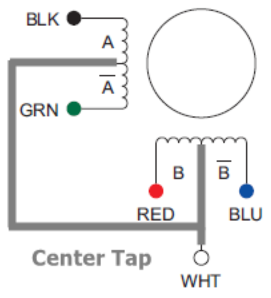


Figure 3: Unipolar Configuration

as "unipolar" or "bipolar" according to the number of lead wires. Therefore, a six wire stepper motor can be classified as a "unipolar" motor, and a four wire stepper motor can be classified as a "bipolar" motor. However, remember that the "unipolar" motor can always be converted to "bipolar".

While "unipolar" and "bipolar" are terms referring to the type of driver used, "unipolar", "bipolar-series", and "bipolar-parallel" are used to describer the wiring between the motor and driver. By changing from unipolar to bipolar, or vice versa, we are actually changing the electrical characteristics of the winding inside the motor, such as voltage, resistance, and inductance, as well as the torque characteristics.

2.3.2 Which wiring method is the best?

This is a trick question. The answer is it really depends on your application. The type of wiring configuration is usually included during the motor sizing portion of the machine design phase. These wiring tricks also allow more repurposing of the same motor for different applications.

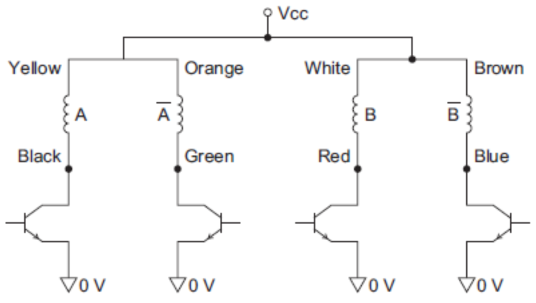
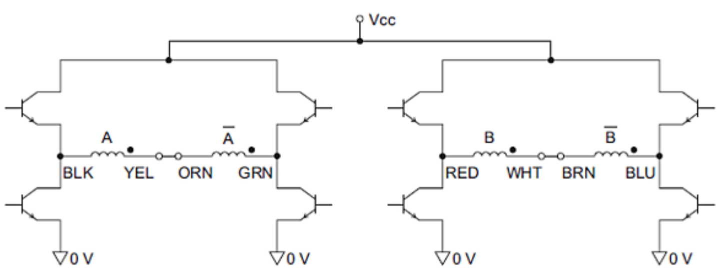
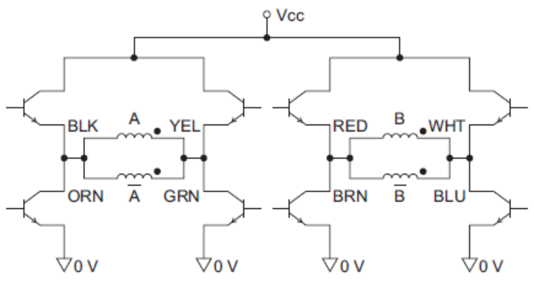


Figure 4: Unipolar or Bipolar Half-Coil Driver Circuit



(a) Bipolar Series



(b) Bipolar Parallel

Figure 5: Bipolar Stepper Motor Driver Circuits

For example, if you are using a unipolar stepper motor setup and would like to increase its low speed torque for a different application, it's worth investigating the bipolar-series wiring configuration in order to keep the same motor size. For the best combination of speed and torque, try bipolar-parallel. However, it requires more current from the driver. Remember that it also depends on what type of driver you have, and how much current it can output.

Here are the differences in motor performance shown with each individual speed-torque curve overlapped. It's easy to see how bipolar-parallel (or parallel bipolar) performs the best.

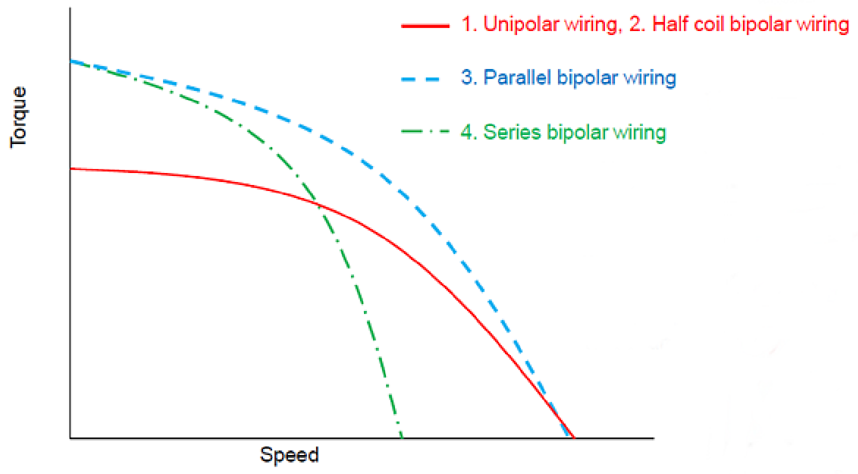


Figure 6: Configuration Performance

2.3.3 Driving a Stepper Motor

Along with the different ways of wiring a stepper motor, we can also drive the motors in different ways. The trade-off between each of the different waves (wave-drive, full-step and half-step) includes power/speed/torque versus power consumption. The more windings that are enabled at the same time, the more power the stepper will draw. This applies to both unipolar and bipolar wiring configurations; however the implementation of driving a bipolar motor is much more difficult than the unipolar motor.

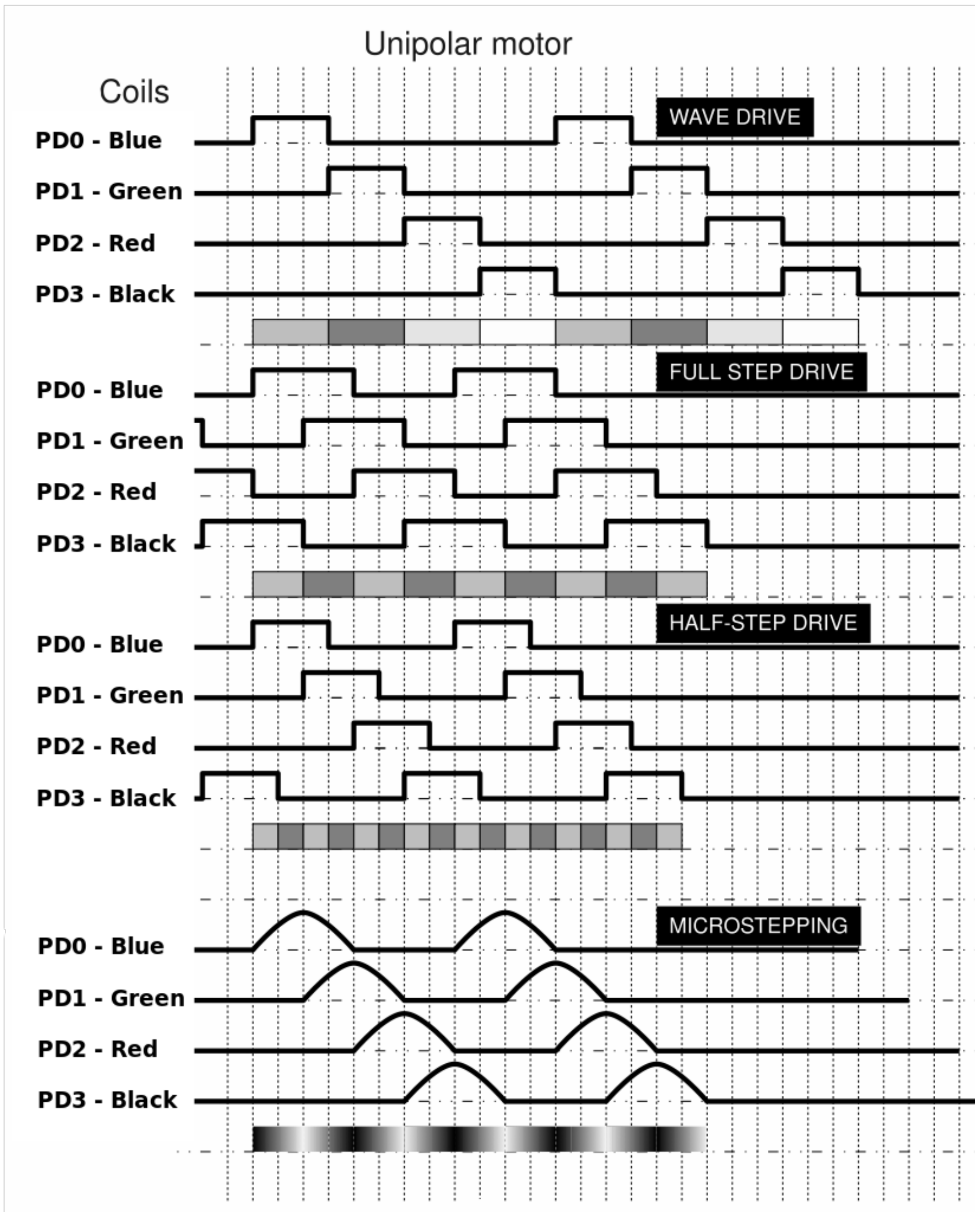


Figure 7: Driving a Stepper Motor

2.3.4 Driving a Unipolar Stepper Motor with the Launchpad

As with all applications involving a microcontroller, it's bad practice to have the microcontroller do any of the work. Something like driving a motor would require way too much power to be sourced or sank from the MCU; instead you should control the motor through a driver circuit as shown in Figure 8.

There is a caveat to be aware of: two pins on PORTB are tied to two pins on PORTD as shown in Figure 9. This information can be found the circuit schematics from the launchpad's user manual (the same one you would

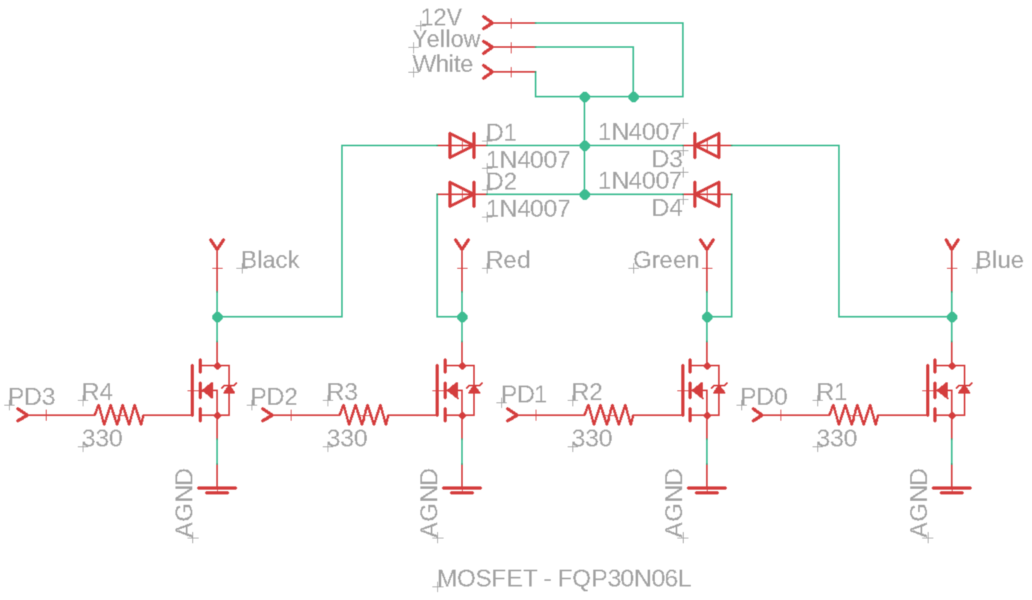


Figure 8: Stepper Motor Driver Schematic

have used in Lab 1). Thus, when you are driving the stepper motor from PORTD, you need to make sure PORTB is set to be an input or make sure you are driving PORTB's pins with the same logic level as PORTD.

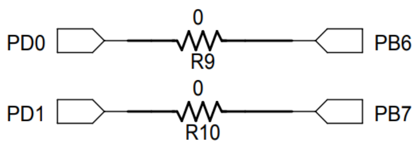


Figure 9: Launchpad Pins Tied Together

The last bit of information that you need to know is for how long to pulse the motor. That information can be found in the stepper motor's (SY42STH38-0406A) datasheet. *Note: PPS stands for Pulses per Second.* ***Also note that the motor is rated for 12V, but the operating chart assumes 28V, and so you can expect the maximum PPS to be around 400 instead of 1000***. The pulse length will vary with the chosen wave drive that you use.

When connecting the motor to the launchpad, connect the wires to the corresponding pins. The push-connectors are for tool-less interfacing; the springs can be tough to push, but be careful to to push too hard and break the plastic. You will also need to connect VCC and GND to the benchtop powersupply. Since our microcontroller is driving a MOSFET, and a MOSFET's gate-source voltage (Vgs) determines how 'on' the FET is, we should make sure the ground references are (relatively) the same. We can do that by connecting the GND to the negative terminal on the power supply.

Initially, you should configure the power supply to output 12V, and set a current limit of 2A. If you notice you are hitting the current limiter when running the motor, increase it slightly. You'll know you are hitting the current limiter because the 'CC' light will come on, particularly on startup. If the motor is not spinning but it is still drawing current, **or if the CC light is on**, you probably have done something wrong and should turn off the power supply. You should investigate what is wrong before proceding.

If you are unsure how to use the power supply, check out the User Manual.

3 Procedure

1. Create a new project called NSID_Lab_4. Refer to Lab 1 if you need help.
2. Configure PWM to control the power of the RED LED. Make a 1KHz signal, 10% duty cycle. Verify this output is working correctly with your ADM/ADALM (or visually verify it's working as the LED will be dimmer than normal. If the LED is on and bright, try setting your duty cycle to 90
3. Starting from a duty cycle of zero, increase the duty cycle by ten percent per second. Verify that your LED output gets brighter each second.
4. Turn the power on the launchpad board off (switch in top left). It's always a good idea to turn power off before connecting anything to a MCU.
5. Connect the Stepper motor to the launchpad. You'll also need to use the power supply on the table.. Make sure it is turned off, set it up for 12V, 2A current limit (to start with). Do not turn on the power supply until your launchpad is programmed!
6. *It is recommended to program the board demo **(lm4flash /usr/local/bin/Board_Demo.bin)** to make sure the motor driver circuit is connected and working correctly. After initialization, the motor should start spinning.*
7. Program the launchpad to pulse the motor at 400PPS to make it spin clockwise. *Note: The instructions/example in the datasheet are for M0PWM0, but you need to use M1PWM5. Make sure you make the necessary adjustments...* You will find the timers/interrupts from the previous labs to be useful to generate the waveforms...
8. Turn on the power supply and verify your design works. **NOTE:** If your motor is getting hot but is not turning, something is probably wrong. TURN OFF THE POWER SUPPLY.
9. Optional: enable interrupts for PWM.

Remember to clean up your station before you leave!

Exercise 3.1 (*Driving a Stepper Motor*) ∗

1. Write a program that does the following (one program that does all of the following):
   (a) Set the brightness of the LED based on a value that represents the speed of the stepper motor. Use PWM to do this (1KHz wave).
   (b) Each time SW1 is pressed, spin the motor **exactly** 360 degrees (one rotation), then stop.
   (c) Each time SW2 is pressed, increase the speed of the motor. Start at 0 (stopped), and increase the speed by 100 PPS each time the button is pressed up to a maximum of 500 PPS. Note: You will need to press SW1 to spin the motor.
2. Make a PDF that has any information that will benefit the marker in regards to the coding question, and answers to the following questions:
   (a) Which wire configuration did you use for this lab? Indicate what the pulse width should be to reach 1000 PPS for each of the following drive waveforms:
      i. Wave Drive
      ii. Half-Step
      iii. Full-Step
   (b) PWM is often used to drive 3 different types of motors: Servo motors, Stepper motors, and DC motors. Explain (briefly) how each of these can use PWM and how they differ from each other.
3. Go back to the start of this manual and verify that you have met the objectives of the lab.

4 Hand-in Instructions

Make a folder called 'Documents' inside of the folder that is your solution to Exercise 3.1 (that should be named NSID_Lab_4). Place your PDF with the answers to your questions in the Documents Folder. Zip the entire contents of the folder named NSID_Lab_4, and hand-in the zip file to canvas/blackboard before the due date. The due date is one week from the day of the lab (at midnight), unless otherwise posted.

A Appendix: Tips on Enabling PWM

Here are some help with commonly asked questions:

1. **How do I read/use the table on page 1351?** First, you must tell the microcontroller to use an alternate function for a pin, then you must choose the peripheral you want to use by setting the value from the table in the PCTL register.

   The AFSEL register simply tells the microcontroller that the pin is not to be used as a GPIO pin (neither input or output) but rather it should be controlled by hardware via some peripheral. The options for peripherals are chosen from the table on page 1351: so for PF1, we can choose to set the pin be controlled by M1PWM5. Choosing *which* peripheral the pin is to use is done by setting the value in the table to the correct spot in the PCTL register. Keep in mind that each pin can be assigned a value from 1-15, so each pin uses 4 bits in the PCTL register.

2. **The Datasheet uses M1PWM5 naming scheme, but the register names follow a different naming scheme.. what is the equivalent name in the registers for M1PWM5?** Have the block diagram open from page 1232 to visually see this explanation.

   First, note that there are 2 modules, so the entire block diagram is for a single module. Module 0 is the first module (which has 8 PWM outputs) and Module 1 is the second module, which also has 8 PWM outputs. From the naming convention M1PWM5, we need to use Module 1. PWM5 comes from the second half of Generator 2. Thus, we could refer to the output PWM5 as Gen2B.

   The naming scheme of the registers is PWM(module_number)_(generator_number)_(A or B). Putting this all together, M1PWM5 is equivalently named or PWM1_2_B