**PWM Driver Testing**

Lorenzo Javier (005979623)

*Computer Engineering Department, College of Engineering*

*San Jose State University, San Jose, CA 95192*

*E-mail:* loj90@sbcglobal.net

**Abstract**

*PWM (Pulse Width Modulation) is a technique of converting analog signals to digital signals with specific frequency, duty cycle, and timing. General Purpose Input/Output (GPIO) pins/registers are a critical part of embedded systems for controlling peripheral devices. This paper will discuss the implementation of a kernel module to drive a stepper motor using GPIO and PWM pins directly through a microcontroller.*

**1. Introduction**

The embedded system used for this lab was a Raspberry Pi 1 Model B+ [1], which is based off an ARM architecture. It has integrated GPIO pins which are configurable for various functionalities, including PWM. It’s an open system where custom kernel drivers, and certain user applications that can be installed and executed in order to set the frequency and duty cycle of a stepper motor, through the various GPIO registers.

The PWM function is dependent on setting its frequency and duty cycle. The frequency allows for the dynamic speed of the motor, while the duty cycle sets the power supplied between each full clock cycle, or period. For example, if the duty cycle is 50%, and the frequency is 1000 Hz, this means the period is 1 ms, so the power supplied will be HIGH for 0.5 ms, and then LOW for 0.5 ms.

The EasyDriver Stepper Motor Driver [2] is capable of handling micro stepping, for full step, half step, ¼ step, and 1/8 step. This allows most stepper motors, which are usually 200 steps/rev, to be more precise in its rotations, from 200 steps/rev, up to 1600 steps/rev. It’s also capable of driving motors up to 30V, 2A.

**2. Code Design**

First, we have to design our kernel driver that would initialize the selected GPIO pins for our circuit, including setting one of the GPIO pins as a PWM functionality. The kernel module is our low-level application that will control the physical address space of our selected GPIO pins, which will control any peripheral devices. It’s a safer way of allowing users to interact with these pins, without messing up the physical aspect of the system.

Second, we have to design our user application that will give any user, at a higher-level, the ability to control thus mentioned GPIO pins, and peripheral devices. This method allows the user to call an API, where the physical registers have already been set to enable the peripheral devices. The user does not need to know, or worry about any physical address space.

**2.1. Kernel Module**

Our low-level module has to configure the physical address registers in order to control the various GPIO pins. Starting with the initialization, all GPIO pins needed for this lab, must be enabled, then set the direction of the pins as either input or output. Also, a module class (registered device) must be created for a user to be able to access the kernel drivers.

For the PWM stepper motor, the microcontroller needs (2) GPIO pins to control both the rotation direction, and the speed/power. When initializing and setting the pins, the frequency was set as 0 Hz, duty cycle as 50%, and rotation direction as clockwise. If needed for micro stepping, an additional (2) GPIO pins should be initialized and enabled as output for the 2-bit micro stepping pins of the EasyDriver.

**2.2. User Application**

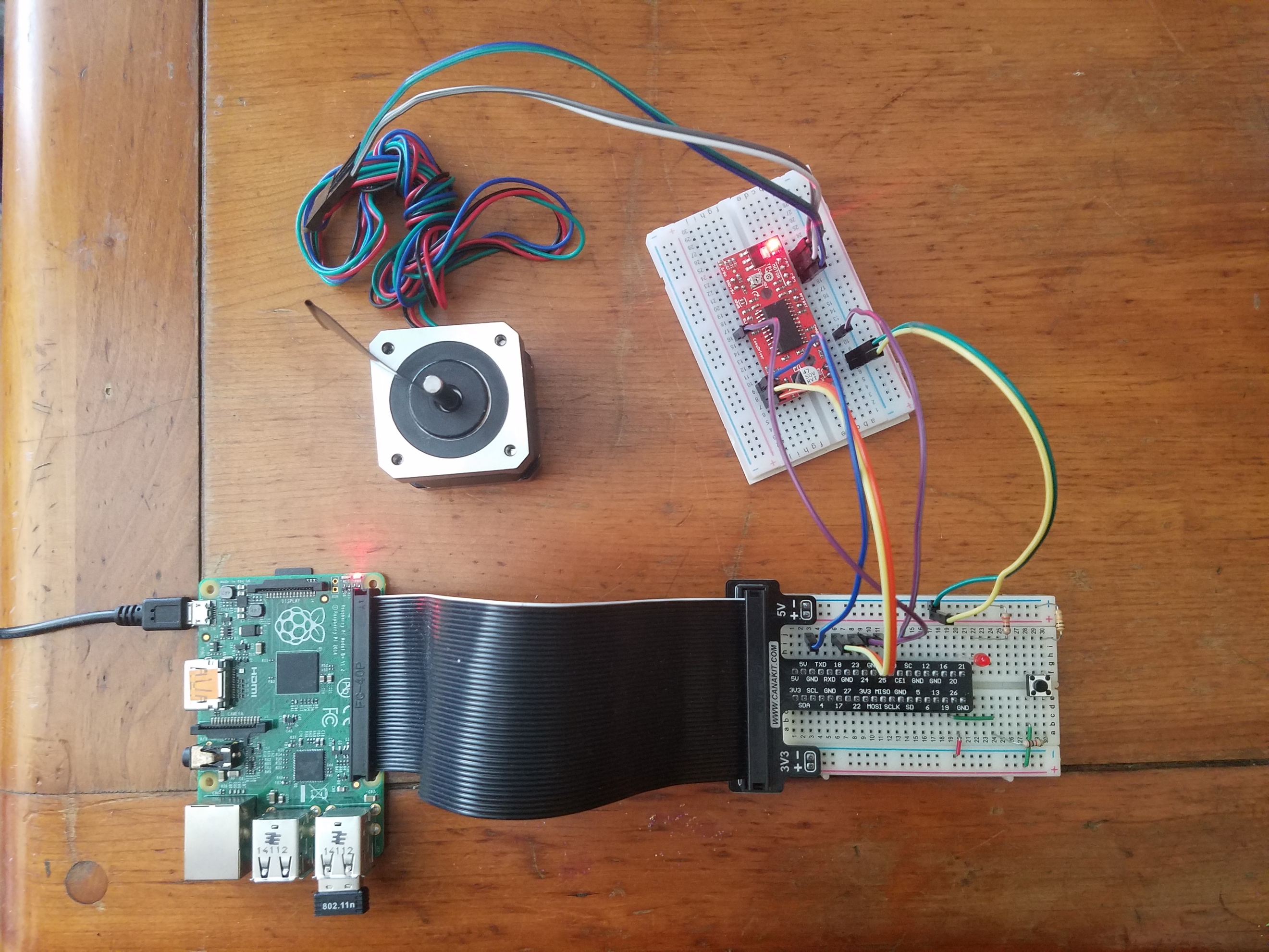
Designing the user application must have the kernel driver in mind. What would be the easiest way for the user to control external peripheral devices through the GPIO pins? In this case, when the kernel driver creates a device class (i.e. /dev/pwm0), the user can simply open this path to gain control of the device associated with it.

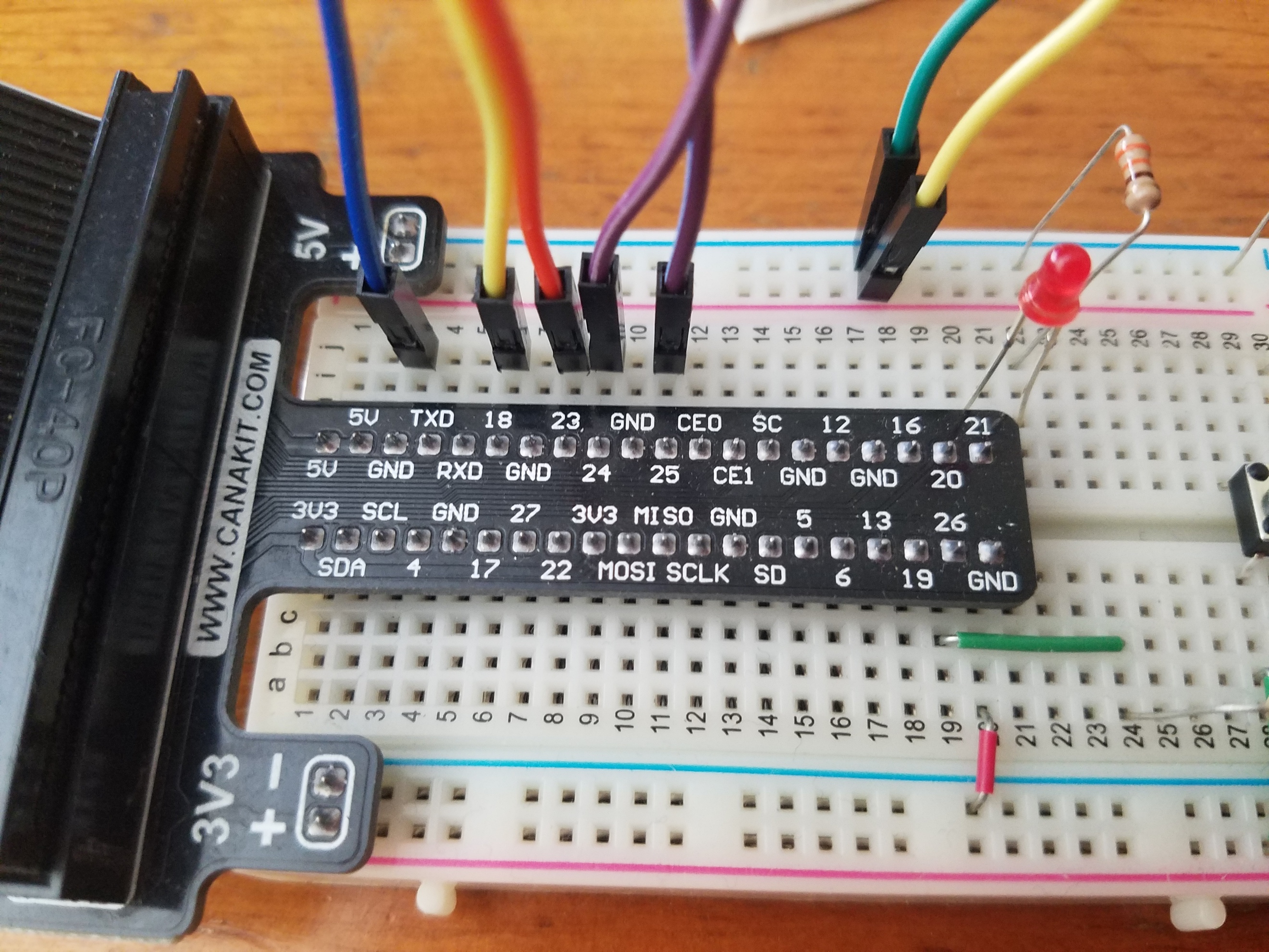
Using the “ioctl” library function, we can send values through the device path to the kernel driver, which then interprets the data and sets the values given to it. Using a data structure, multiple values can be sent at one time, making it easier than sending multiple user to kernel requests.

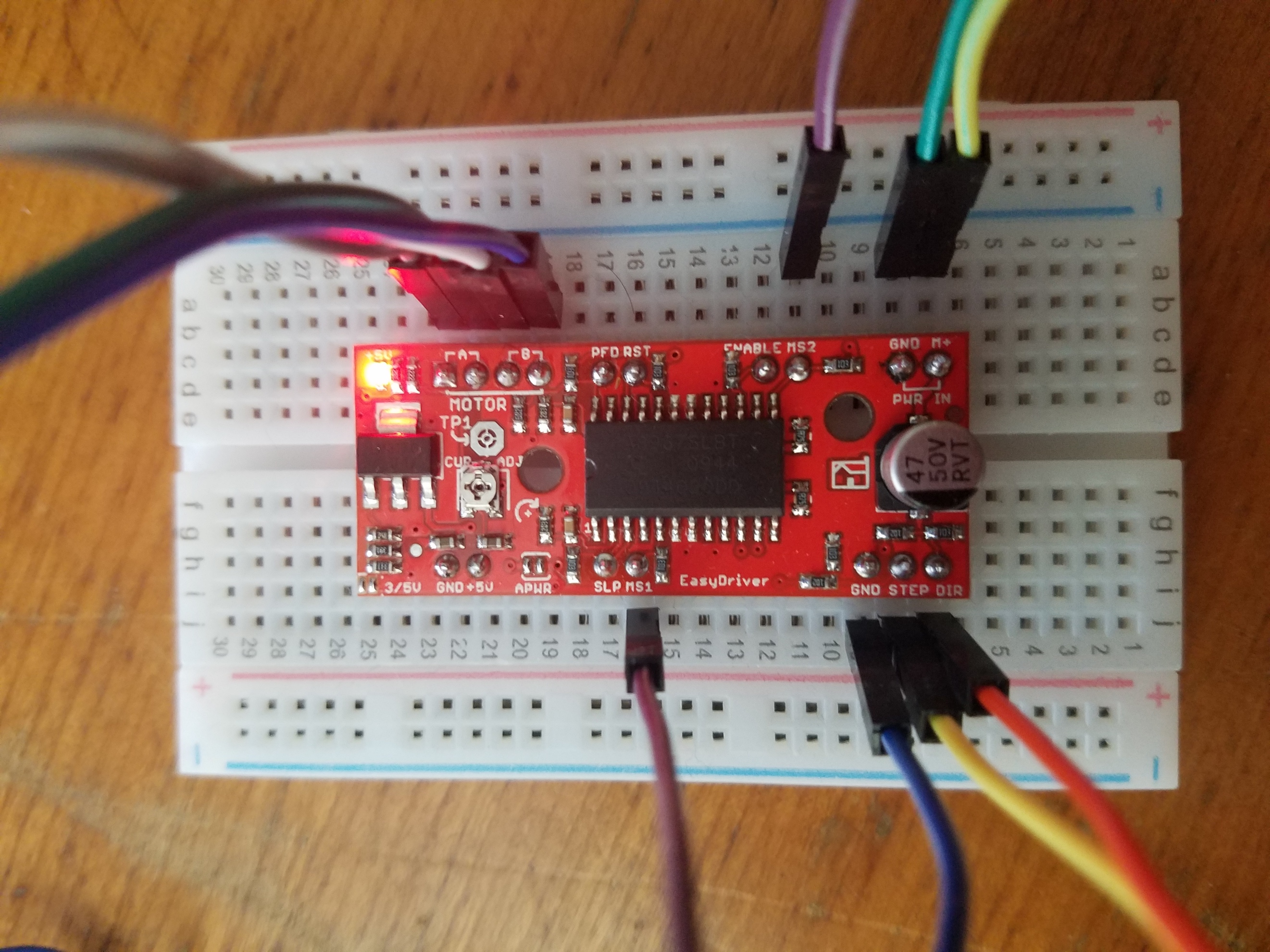
**3. Implementation**

Once the kernel driver and user application has been developed, it must be cross-compiled using an ARM toolchain. This will allow our code to only run on ARM platform devices, rather not x86 systems. After compilation, the kernel driver must be installed as a kernel module, and the user application can be executed from command line.

Second, the hardware setup must also be configured appropriately. From the images below, we can see how the Raspberry Pi to EasyDriver to Stepper Motor [3] devices are physically attached per pin.



****

****

****

**3.1. Kernel Module**

From the main directory of the kernel module, the following commands should be executed:

1. sudo make
2. sudo insmod pwm.ko
3. sudo rmmod pwm (\*Removing the pwm module)

For debugging, run either command while running the kernel module to see any printk messages:

1. dmesg
2. tail –f /var/log/kern.log &

**3.2. User Application**

From the main directory of the user application, the following commands should be executed:

1. sudo make
2. sudo ./pwm\_test

This particular PWM driver test runs in a for loop, increasing the frequency every 5-10 seconds.

**4. Testing and Verification**

The main goal of this assignment is to be able to control a PWM driver device by dynamically increasing or decreasing the frequency. It was successful by following the below commands to test both the kernel module and user application.

The results were positive, as for the frequency changing. There was a noticeable difference in speed from the stepper motor.

**4.1. Kernel Module**

From the main directory of the kernel module, the following commands should be executed:

1. sudo make
2. sudo insmod pwm.ko
3. sudo rmmod pwm (\*Removing the pwm module)

For debugging, run either command while running the kernel module to see any printk messages:

1. dmesg
2. tail –f /var/log/kern.log &

**4.2. User Application**

From the main directory of the user application, the following commands should be executed:

1. sudo make
2. sudo ./pwm\_test –m 1 –r 0 –f 1000 –d 50

This particular PWM user application test runs in a for loop so the user application can control the frequency, duty cycle, and rotation direction values to be sent to the kernel module.

**5. Conclusion**

By understanding the CPU’s datasheet, where all the physical addresses are listed out, any number of external peripheral devices can be attached to the physical GPIO pins and controlled through software, by referencing them in virtual addresses. By using kernel modules, any user doesn’t need to know the low-level specifications to design software for any physical peripheral.

Using a development kit allows any user to work on a project with specific hardware and software before designing a unique integrated embedded system device.

**A. Resources**

[1] https://www.raspberrypi.org/products/raspberry-pi-1-model-b-plus/

[2] http://www.schmalzhaus.com/EasyDriver/

[3] https://www.omc-stepperonline.com/hybrid-stepper-motor/nema-17-bipolar-45ncm-64ozin-2a-42x40mm-4wires-w-1m-cable-and-connector-17hs16-2004s1.html