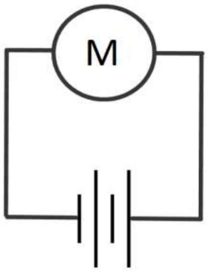
CMPT 433 Course Project How-To Guide

Using TB6612FNG DC Motor Controllers with a BeagleBone Green

David, Joy, Jordan

Summer 2018

Motivation



Why would you want to use a motor controller to control your DC motors? Consider the trivial circuit in figure 1 depicting a motor being powered by a battery. This is the simplest circuit you can build that will make DC motors move. All you need to do for this circuit to work is match the voltage ratings on the motor and battery and make sure the battery can supply sufficient current to move the motors.

The issue with the following circuit is that once you build it, there is no way to change the behavior of the motor without changing the circuit. As soon as you complete this circuit the motor will start running at full power and the only way you can stop it is by breaking the circuit. If you want to be able to control what the motor does using software running on a microcontroller, you need to use a motor controller. A DC motor controller such as the TB6612FNG sits between a DC motor, a power supply or battery, and a microcontroller. The battery powers the DC motors (but not the microcontroller) and the microcontroller tells the motor controller how power from the battery should be used to control the behavior of the motors.

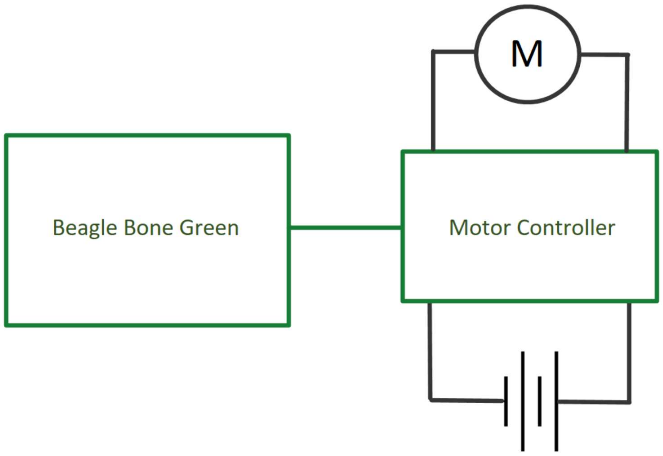


Figure 2 - DC motor controller circuit block diagram

Using a motor controller such as the TB6612FNG allows you to control how fast the motors spin and which direction the motors should spin in using GPIO and PWM signals. At a high level, motor controllers allow you to isolate the task of powering motors and controlling motors which is essential when you wish to control motors with a device like a BeagleBone, which cannot supply the current necessary to power any kind of motor.

Introduction

This guide shows you how to use a TB6612FNG motor controller breakout board to control one or more DC motors. The TB6612FNG is a fairly mainstream motor controller that is available (at great prices) by vendors like sparkfun.com on a variety of boards that you can easily interface with your microcontroller of choice.

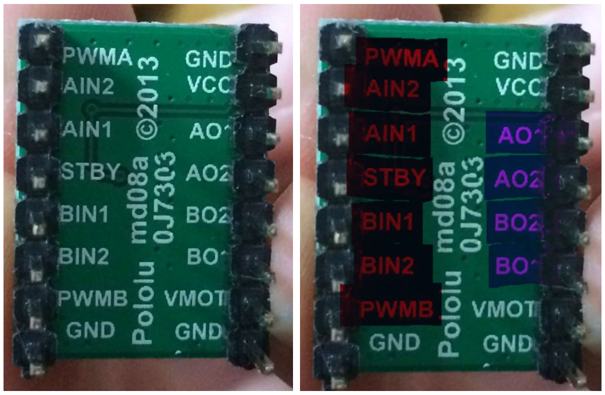
There is a lot of excellent documentation on the TB6612FNG online. Sparkfun.com sells TB6612FNG breakout boards in a variety of configurations and has hookup guides that are very easy to use. The following URL links to a hookup guide for the TB6612FNG at sparkfun.com that you may find easier to use and more extensive than this guide.

https://learn.sparkfun.com/tutorials/tb6612fng-hookup-guide?_ga=2.106715840.2111960893.1529422690-809850030.1529422690

If you want to interface the TB6612FNG with an Arduino, the above guide details how to use a pre-existing software library that lets you access the motor controllers in an Arduino sketch with simple function calls. Because the above guide is so extensive, this How-To Guide will focus on how to interface the motor controllers with a BeagleBone green and create your own software library or module using the bone's GPIO and PWM pins from within a C program.

TB6612FNG Motor Controller Board Overview

The topology of the motor controller circuit in figure 2 is apparent when looking at the pins on a TB6612FNG breakout board. It is import to fully familiarize yourself with DC motor controller circuits further before you try and construct the circuit suggested in this guide because you can easily damage your motor controller or your microcontroller with improper wiring.



Purple: Outputs to DC motors

Red: Inputs from Beagle Bone Green

Figure 3 - TB6612FNG breakout pins

To connect one DC motor to the motor controller, connect the two terminals of the motor to the AO1 and AO2 pins (or BOX pins, just be consistent when you hook up the input pins to your microcontroller). For now it doesn't matter which lead of the motor connects to which AOX pin because DC motors are bi-directional. Connect the positive terminal of the battery you wish to power your motors with to the VMOT pin and connect the negative terminal to one of the ground (GND) pins. BE CAREFUL when you do this because it is easy to destroy the board when you incorrectly connect the power pins. Connect one of the 3.3V pins on your BeagleBone Green to the VCC pin and connect one of the ground pins to a GND pin. You will connect the BeagleBone to the PWMA, AIN2, AIN2, and STBY pins as inputs to control the TB6612FNG. This will require 3 GPIO pins and 1 PWM pin.

Operation

If you have wired the motor controller circuit correctly, you will be able to control what the DC motor does by driving signals to pins PWM0, AI1, AI2, and STBY on the motor controller using the BeagleBone green. The three GPIO signals can be from any GPIO pins on the BeagleBone and you can follow Brian Fraser's GPIO guide to configure them as outputs and drive them high or low. The PWM pin needs to be one of a few designated pulse width modulation pins on the BeagleBone and you need to take a few extra steps to make the PWM pins usable.

Pulse Width Modulation Pins (PWM)

You need to use a device tree overlay if you want to use PWM pins. If you wish to control 1 or 2 DC motors independently, I recommend you use the BB-PWM0 device tree overlay to enable pins 22 and 21 on the P9 header of the BeagleBone green as PWM pins to control the board. Using more than 2 motors is outside the scope of this guide and will require exporting a compatible universal device tree overlay. Use the following command to export the BB-PWM0 device tree overlay.

```
# export SLOTS=/sys/devices/platforms/bone_capemgr/slots

# echo BB-PWM0 > $SLOTS
```

Check that this command succeeded by checking the exported device tree overlays with the following command.

```
# cat $SLOTS
```

You should see a slot in your slots file containing the BB-PWM0 device tree overlay. If you have exported the overlay properly, the directory "/sys/class/pwm/pwmchip0" will exist.

Once the pmwchip0 directory exists, you can control the PWM signal on pins 22 and 21 similarly to how you would control the GPIO pins on the BeagleBone. From the /sys/class/pwm/pwmchip0 directory, you can run the following commands

```
# echo 0 > export
# echo 1 > export
```

to create two directories called pwm0 and pwm1 inside the pwmchip0 directory. The files in /sys/class/pwm/pwmchip0/pwm0 control the PWM signal on pin 22 and the files in /sys/class/pwm/pwmchip0/pwm1 control the PWM signal on pin 21.

The basic idea behind pulse width modulation is you are controlling the duty cycle of a square wave signal. By modulating the duty cycle of the signal, you modulate the pseudo analog voltage of the signal.

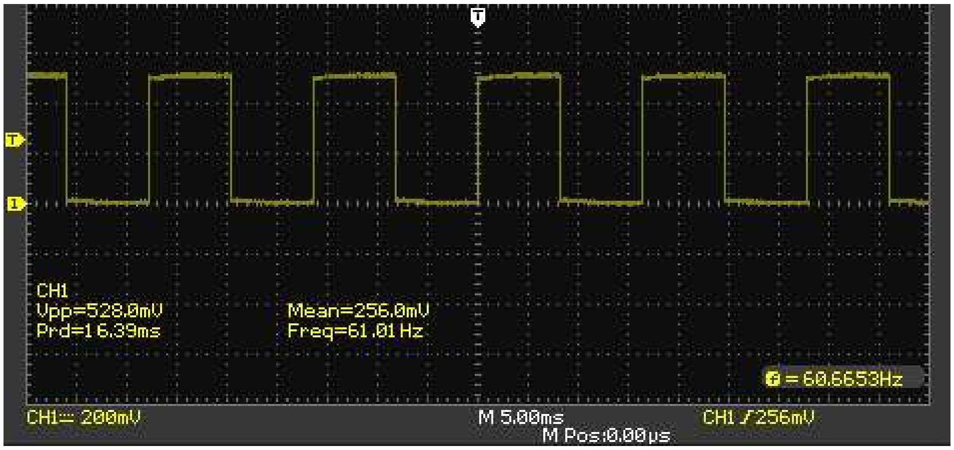
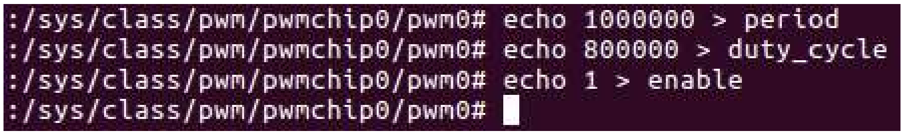


Figure 4 - A PWM signal with around a %50 duty cycle on an osciliscope

If you wired PWMA on the microcontroller to pin 22 on the BeagleBone, change to the /sys/class/pwm/pwmchip0/pwm0 directory. You should see three files called enable, period, and duty_cycle. These 3 files affect the PWM signal on pin 22. The enable file contains a “1” or “0” and toggles the PWM pin on or off. The period file contains the period of the PWM square wave in nanoseconds. The duty_cycle file contains the duration of the high portion of the square wave in the PWM signal also in nanoseconds. A sample PWM pin write is shown in figure 4.



```
:/sys/class/pwm/pwmchip0/pwm0# echo 1000000 > period
:/sys/class/pwm/pwmchip0/pwm0# echo 800000 > duty_cycle
:/sys/class/pwm/pwmchip0/pwm0# echo 1 > enable
:/sys/class/pwm/pwmchip0/pwm0# █
```

Figure 5 - Creating a 1kHz square wave with an 80% duty cycle on pin 22

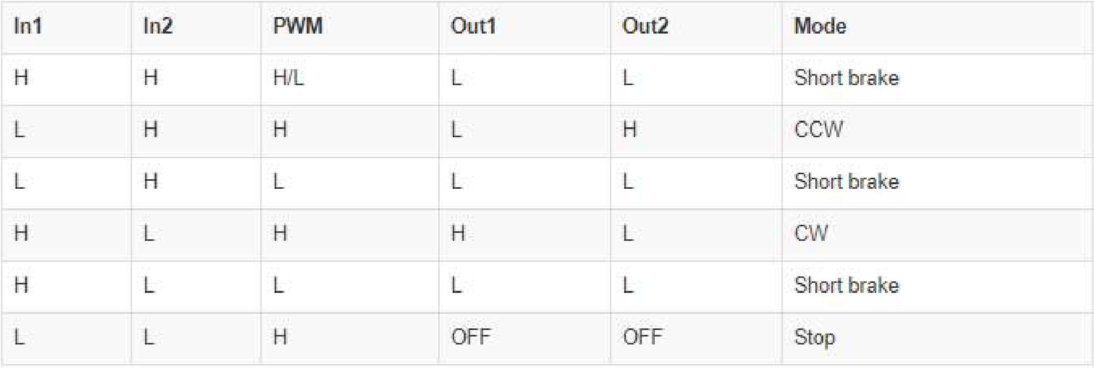
Combining Signals to Control Motors

The Input pins on the motor controller determine how the motors will move. You must send the proper signals to the motor controller for your motors to work the way you want them to.

The STBY pin is a universal toggle that allows or disallows the motors to operate regardless of what kind of voltage or signal is driven to any of the other input pins on the motor controller. You must drive the GPIO pin you connected to the STBY pin high before the motors will work. If you are lazy you could short this pin to VCC and your motor controller will always be enabled.

The AI1 and AI2 pins control the direction the motors will spin in. The GPIO high and low signals you send to these pins determine which direction the motor controller will send current through your DC motors. When you set AI1 low and A2 low no power will be sent to the motors. When AI1 is high and AI2 is low the motors will move. When AI1 is low and AI2 is high the motors will run in the opposite direction. See figure 4 for details.

The PWM pin allows you to control how much power is sent to the motors. Figure 4 summarizes how you should drive your GPIO and PWM pins to get the behavior you desire out of your motors.



In1	In2	PWM	Out1	Out2	Mode
H	H	H/L	L	L	Short brake
L	H	H	L	H	CCW
L	H	L	L	L	Short brake
H	L	H	H	L	CW
H	L	L	L	L	Short brake
L	L	H	OFF	OFF	Stop

Figure 6 - Motor controller input/output behavior (Sparkfun.com)

Making a Motor Controller Module in C

Whatever project you may be working on that requires motors, it is usually a good idea to create a software module for them. This module will allow you to isolate the logic of your main program from the file writing operations that change the behavior of your GPIO and PWM pins. You will find it very convenient to have simple functions like MotorsWrite(int power) at your disposal when you are writing other modules or your mainline routine.

The bulk of your C module will be file writing operations to your PWM and GPIO files. You will have to find a convenient way to manipulate C strings containing full paths to files and check for errors every time you make a write.

If you are controlling your motors from a C program, there are a number of good practice guideline you should try to follow:

-Give your module init() and deinit() functions that export GPIO and PWM pins, set pin modes, and drive the standby pin.

-Make the pin numbers you use on you BeagleBone constants at the top of your program. This will make swapping pins easy.

-Make some path constants at the beginning of your program to make your string operations neater and more portable.

-Create your own file writing function with error checking that has a nice name. This will make your code much more readable.

-Use lots of comments because your sea of file writes and string operations are likely going to look messy and hard to read no matter what.

-Extra tip: write a testing program for you motors module. Having a simple command line style application that calls the functions in your motors module goes a long way when debugging!