Adding External Motors to the Crazyflie 2.0 with PWM

Bradley Pospeck

Tutorial Overview

This tutorial discusses how to setup an external motor with PWM on the Crazyflie 2.0 platform. You will need to go through "Building the Project in Eclipse" first. This is because it contains necessary precursor information to the later section.

For this tutorial, you will need a Crazyflie development environment setup on a computer. You will also need an assembled, working Crazyflie 2.0 with the Crazyradio PA. Instructions for both can be found here. For physical setup, you can use the Crazyflie Prototyping deck found here, a printed circuit board (PCB), or a basic breadboard. I will be using a breadboard for demonstrations in this tutorial.

Sections

Building the Project in Eclipse

I. Overview

This part of the tutorial is a step that is required at the end of both “Adding an LED” and “Adding a Motor” sections. It will be introduced beforehand so you can make sure your project is setup correctly initially. You can then refer to it again after finishing either of the other 2 later sections. It also provides a small overview of some necessary tabs.

II. Building

1) Figure 1 below shows the main Eclipse window and highlights the necessary parts for building.
    a) The large blue boxed section on the left is the “Project Explorer” tab. Use this to navigate through the directory to find files.
    b) The orange section on the bottom is the console window. Outputs are displayed here.
    c) Once your files are ready to be built, start first by cleaning the project. This can be found on the right side. In the “Make Target” tab (boxed in red) you will see “clean” boxed in green. The console will show progress and say when cleaning is finished.
    d) Next, hit the build icon,  , boxed in purple towards the upper left hand corner. Again, the console will show the build progress and state when it is finished.
    e) Now to upload the code to the Crazyflie 2.0.
        - Start the Crazyflie 2.0 in bootloader mode by holding the power button until 2 blue lights start blinking. It should take around 2-3 seconds.
        - Ensure the Crazyradio PA is ready to go. That’s what uploads the code from the computer.
        - Hit “Flash using radio”, highlighted in the “Make Target” tab. When flashing is finished, the Crazyflie 2.0 will restart itself in its normal mode and you are done.

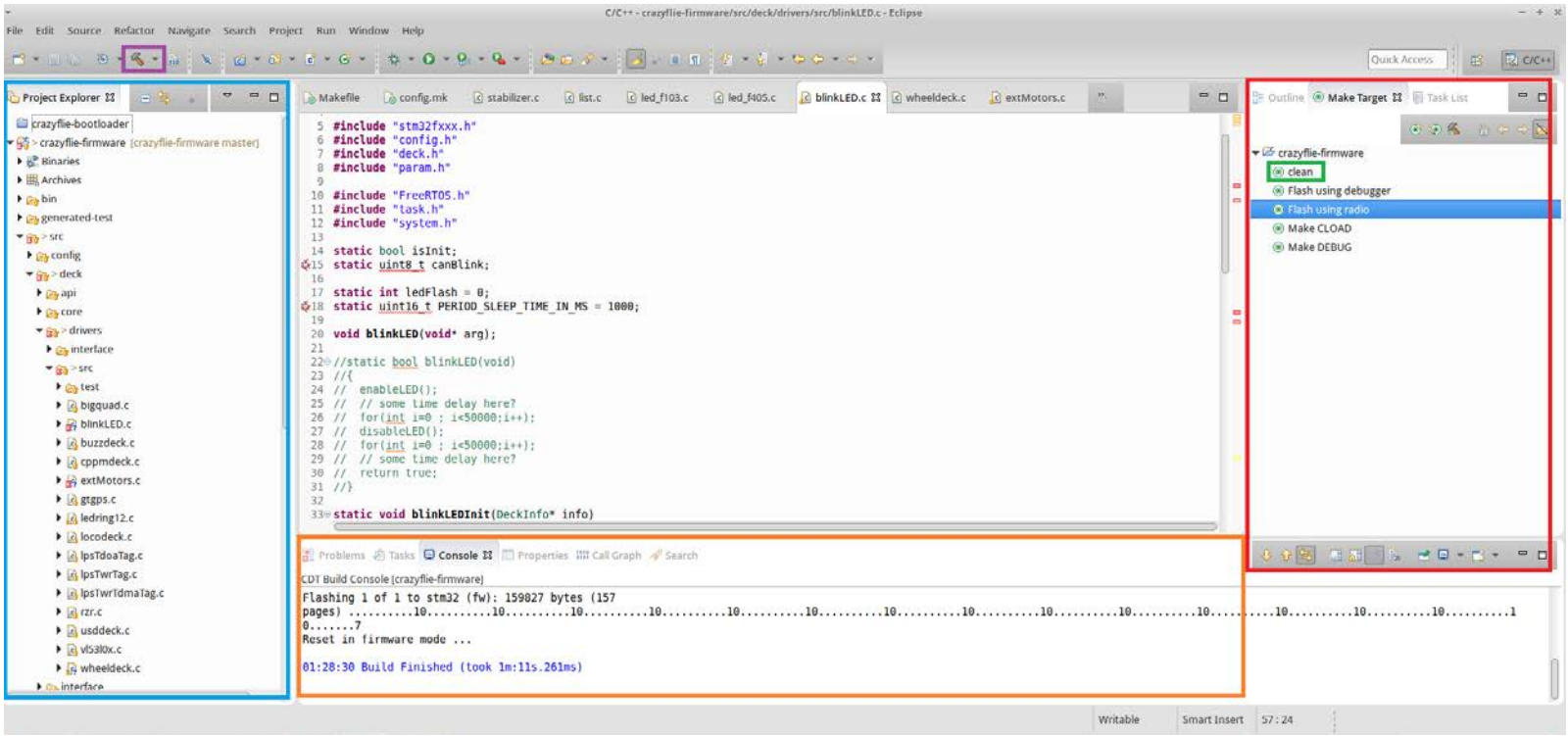


Figure 1: Eclipse overview

Adding a Motor with PWM

I. Overview

For this section, a small dc is motor is needed as well as a battery to power it. The battery I have is a 3.7 volt, 2000mAh lithium ion battery, similar to this one. The motor will need a motor driver in order to be used as desired. The one used in this tutorial will be Sparkfun's Dual TB6612FNG found here. The required pins on this motor driver will be VM, VCC,GND, A01 and A02, AIN1 and AIN2, PWMA, and STBY. The required Crazyflie 2.0 pins will be PA2, PB5, PB8, VCC, and GND. In order to activate PWM on PA2, some extra setup is required with the Microcontroller on the Crazyflie 2.0. One of the libraries used in this platform is the STM32Fxxx HAL Libraries. This library helps abstract the registers you need to access in the Microcontroller to make coding a little bit easier. If you want to explore the microcontroller registers in more detail beyond what the library provides, here is the datasheet.

II. Circuit Setup

1) Start by inserting the motor driver into the breadboard and bridging both power and ground rails.
2) Next, setup all of the ground wires. Refer to Figure 2 below as necessary.
    a) Connect GND of the Crazyflie to one of the ground rails on the breadboard.
    b) Connect GND of the motor driver to the nearest ground rail.
    c) Connect the battery's ground to the ground rail nearest VM of the motor driver.

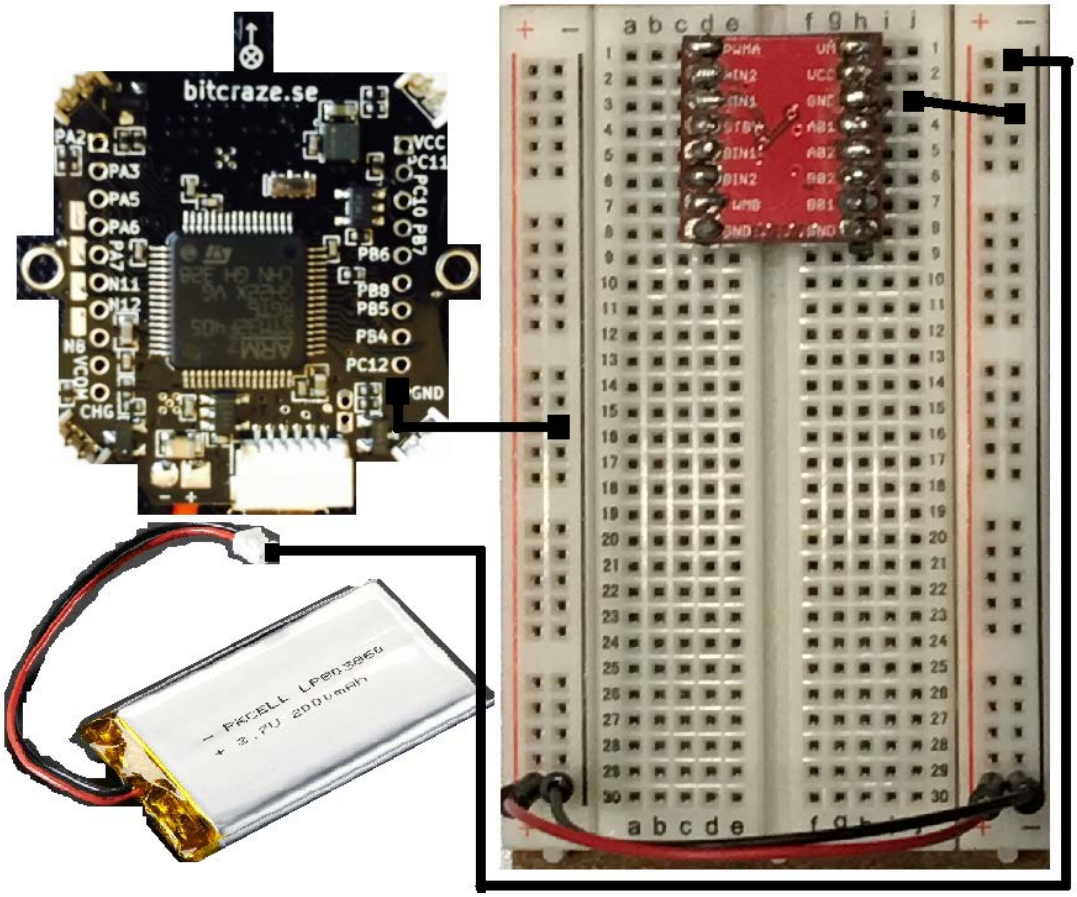


Figure 2: All grounds connected

3) Now, it is time to set up all power connections. Refer to Figure 3 below as necessary.
    a) Attach Crazyflie VCC to one of the breadboard's power rails.
    b) Attach VCC of the motor driver to the nearest power rail.
    c) Attach the power of the battery to VM of the motor driver.

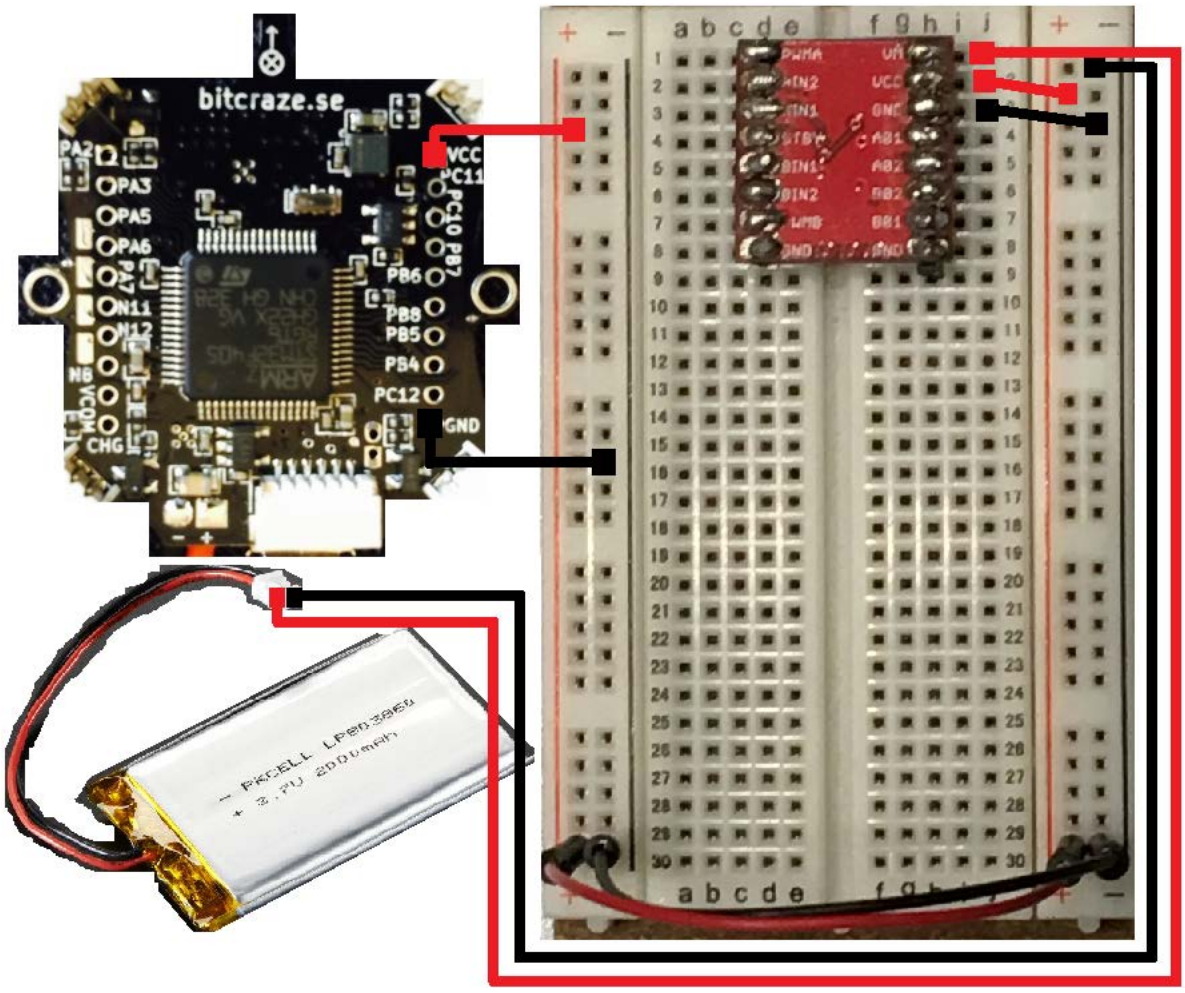
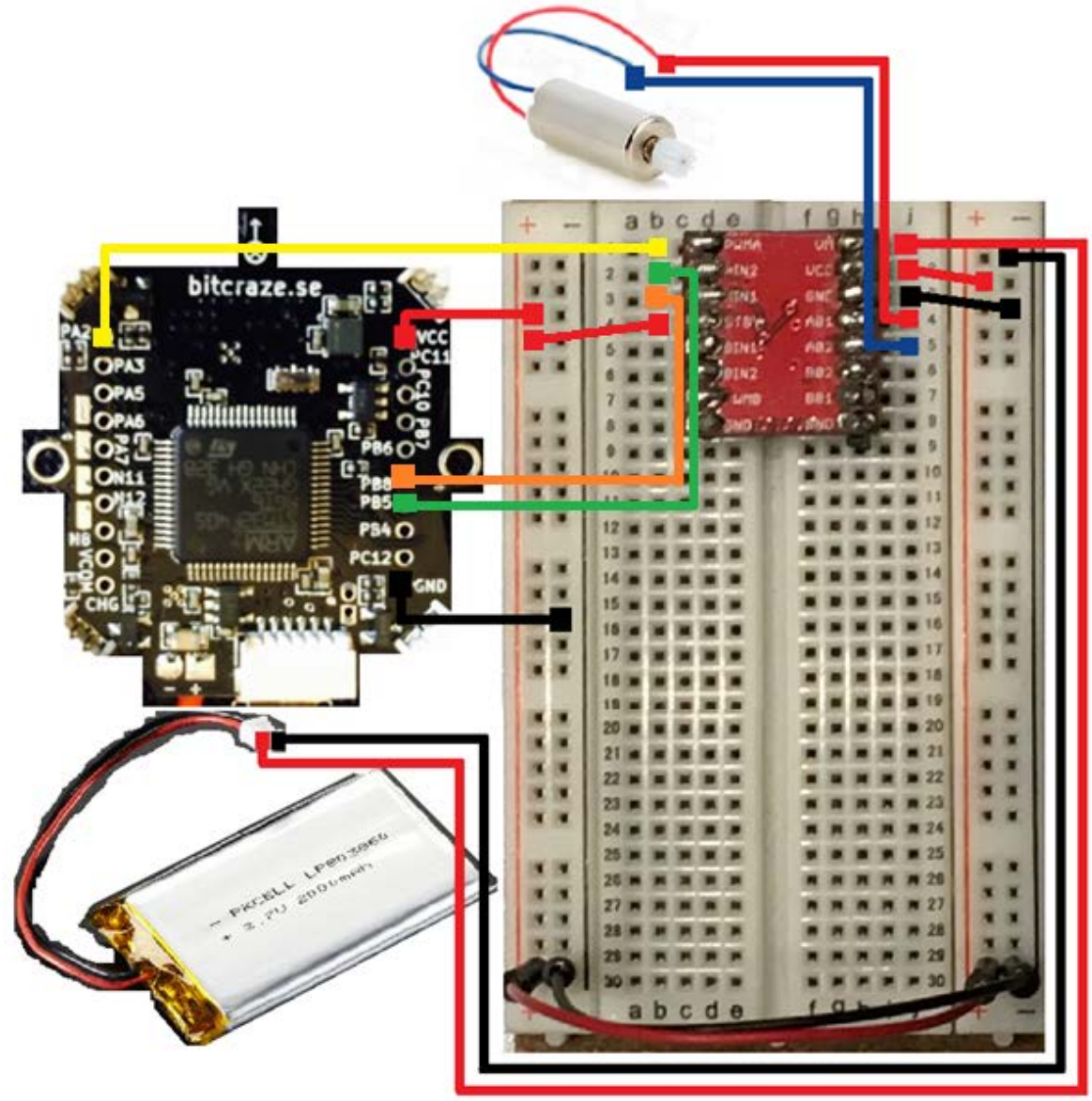


Figure 3: All power and ground connections completed

4) With this particular motor driver, STBY being low will deactivate the motors. This is not desired. Therefore, connect STBY to the nearest power rail so that VCC can hold it high.
5) Now to attach inputs of the Crazyflie to PWMA, AIN1, and AIN2 of the motor driver to be able to control the motor. It can go clockwise, counterclockwise, or be stopped.
    a) PA2 to PWMA: This will control the motor's speed via pulse width modulation (PWM).
    b) PB8 to AIN1
    c) PB5 to AIN2
6) The final piece is to connect the two motor leads to the motor driver.
    a) Connect the red lead to A01
    b) Connect the blue (it may also be black) lead to A02

7) That is all for the circuitry. The final diagram with all wiring is shown in Figure 4 below.



r

Figure 4: Diagram of the entire circuit setup

III. About Parameters

Using the Crazyflie client, certain variables can be accessed within the Crazyflie quadcopter by using what are called parameters. In the Crazyflie client, under the "Parameters" tab, you will find a list of all parameters in the Crazyflie 2.0. There will be groups that can all contain one or more parameters. Some are read only, while others are read and write. Any with write permissions can be changed right there in the client. This will send a command (via radio) in real time to the quadcopter in order to change the variable associated with the parameter. These will be used to control whether the motor is on or off. If the motor is on, the parameters also control which direction the motor will spin (clockwise or counterclockwise) as well as the motor's speed.

IV. Driver Coding

1) First, create your '.c' and '.h' files for the motor driver. I named mine "motorsDriver.c" and "motorsDriver.h".
    a) Navigate to this directory: "crazyflie-firmware/src/drivers". From here, the '.h' file will go in the "interface" folder and the '.c' file will go into the "src" folder.
    b) Right click the directories and navigate to "New->Source File" for the '.c' and "New->Header File" for the '.h'.
    c) Enter the name and do not forget the extension ('.h' or '.c')
2) Setup the '.h' file will all appropriate headers found in Figure 5 below.

```
#ifndef SRC_DRIVERS_INTERFACE_MOTORSDRIVER_H_
#define SRC_DRIVERS_INTERFACE_MOTORSDRIVER_H_


#include <stdint.h>
#include <stdbool.h>
#include "config.h"
```

Figure 5: motorsDriver.h header information

3) Setup the function definitions that you see in the figure below. These will be implemented in the '.c' file. That's the entirety of the '.h' file.

```
// Setup and configuration functions. Called by motorDriverInit()
void GPIOInit();
void timerInit();
void pwmInit();

// Motor initialization
void motorDriverInit();

// Set motor ratio
void motorSetRatio(uint8_t ratio);

// Set motor freq in Hz
void motorSetFreq(uint16_t freq);

#endif /* SRC_DRIVERS_INTERFACE_MOTORSDRIVER_H_ */
```

Declear all the funs

Figure 6: motorsDriver.h function definitions

4) Let's start setting up the '.c' file with necessary headers and variables. Reference Figure 7.
    a) *BASE_FREQ* is setup so that the frequency of the PWM can be changed via a function call later.
    b) *isInit* is just a Boolean used to determine if the PWM is initialized on the MCU or not.

```c
#include <stdbool.h>

/* ST includes */
#include "stm32fxxx.h"

#include "motorsDriver.h"

//FreeRTOS includes
#include "FreeRTOS.h"
#include "task.h"

#define BASE_FREQ (328125)

static bool isInit = false;
```

Figure 7: motorsDriver.c header files and variable initialization

5) We will be implementing “GPIOInit” function next as in Figure 8. This will setup GPIO Port A.
   a) *GPIO_Struct* is used in order to gather all necessary register configurations into one place so they can all be initialized at once by the “Init” function later.
   b) The long line function call of “RCC_AHB1PeriphCLockCmd” is just enabling the clock peripheral on GPIO Port A.
   c) “GPIO_PinAFConfig” is used to configure GPIO Port A, pin 2 to timer 5.
   d) Configuring the GPIO_Struct has a few options for each field. If you want to learn more about the ones used here and what other options are available for them, I encourage you to consult the library and datasheet resources linked above.

```c
// Initialize output
void GPIOInit(){
    // Struct to initialize/configure GPIO
    GPIO_InitTypeDef GPIO_Struct;
    // setup a clock on GPIO Port A
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
    // configure alternate function with pinsource2 and TIM5 timer
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource2,GPIO_AF_TIM5);
    // Configure GPIO_Struct
    GPIO_Struct.GPIO_Mode = GPIO_Mode_AF;
    GPIO_Struct.GPIO_OType = GPIO_OType_PP;
    GPIO_Struct.GPIO_Pin = GPIO_Pin_2;
    GPIO_Struct.GPIO_PuPd = GPIO_PuPd_DOWN;
    GPIO_Struct.GPIO_Speed = GPIO_Speed_2MHz;
    //Initialize GPIOA with setup config
    GPIO_Init(GPIOA, &GPIO_Struct);
}
```

Figure 8: GPIO Port A configuration setup

6) Initialization of timer 5 is up next. Use Figure 9 below as reference.
   a) The *TIM_TimeBaseStruct* serves the same purpose as the GPIO struct above.
   b) Setting up the clock is as simple as calling the “RCC_APB1PeriphClockCmd” function with the desired timer, timer 5 in our case, and enabling it.
   c) Let’s keep it simple: leaving the prescaler and clock division at 0 is fine as our frequency is well within the 84 MHz frequency of the clock. The period is set at 255 because we don’t need a ton of resolution and it maxes out the range of an 8-bit integer.
   d) Again, use the datasheet and library references if you want to learn more about any of the fields. Some comments are provided in the code as a general guideline.

```
// Initialize Timer, TIM5
void timerInit(){
    // Struct to initialize/configure TIM5
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStruct;
    // Setup clock
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE);
    // Configure TIM_TimeBaseStruct
    TIM_TimeBaseStruct.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseStruct.TIM_Prescaler = 0;
    TIM_TimeBaseStruct.TIM_ClockDivision = 0;
    TIM_TimeBaseStruct.TIM_Period = 255; // 328,125Hz PWM with 84MHz clock
    TIM_TimeBaseStruct.TIM_RepetitionCounter = 0;
    // Initialize the timer
    TIM_TimeBaseInit(TIM5, &TIM_TimeBaseStruct);

    // Enable TIM5
    TIM_CtrlPWMOutputs(TIM5, ENABLE);
    // Set compare register values
    TIM_SetCompare3(TIM5, 0x00);
    //TIM_SetCompare4(TIM5, 0x00);

    // Start timer
    TIM_Cmd(TIM5, ENABLE);
}
```

Figure 9: Timer 5 initialization

7) The final portion of setting up the MCU comes with configuration of which channel we will be using on Timer 5. Channel 3 is the corresponding channel, and the output compare of it is the final piece of setting up PWM. Use Figure 10 below.
    a) The “OCMode” is set to PWM1 as this is the simplest.
    b) The “OCPolarity” is simple. Setting it to high ensures that the higher the number provided to “TIM_Pulse”, the faster the motors will go. Setting it low reverses this.
    c) Since our period is set to 255 above, setting the pulse to 255 means an initial duty cycle of 100%.

```
void pwmInit(){
    // Struct to initialize/configure channels for pwm
    TIM_OCInitTypeDef TIM_OCStruct;
    // Configure TIM_OCStruct
    TIM_OCStruct.TIM_OCMode = TIM_OCMode_PWM1;
    TIM_OCStruct.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCStruct.TIM_OCPolarity = TIM_OCPolarity_High;
    TIM_OCStruct.TIM_Pulse = 255;
    // Initialize on Channel 3
    TIM_OC3Init(TIM5, &TIM_OCStruct);
    // Enable preload
    TIM_OC3PreloadConfig(TIM5, TIM_OCPreload_Enable);
}
```

Figure 10: Output compare setup on channel 3 of Timer 5

8) Almost done with the driver. All that's left is to implement the last 3 function definitions from "motorsDriver.h". Use Figure 11 below.
    a) "motorDriverInit" simply gives our deck code (that we'll be writing in the next section) one easy function to call for initialization.
    b) "motorSetRatio" is a function for our future deck code to call when changing the motor's speed. The new resulting duty cycle for the PWM will be the *ratio* divided by the period, which is 255 as you recall.
    c) "motorSetFreq" is less likely to be used, but allows users to change PWM frequency. The function updates the prescaler with division of *BASE_FREQ* by the given frequency, *freq*.

```
void motorDriverInit(){
    if(isInit)
        return;
    GPIOInit();
    timerInit();
    pwmInit();

    isInit = true;
}

void motorSetRatio(uint8_t ratio){
    TIM_SetCompare3(TIM5,ratio);
}

void motorSetFreq(uint16_t freq){
    TIM_PrescalerConfig(TIM5, BASE_FREQ/freq, TIM_PSCReloadMode_Update);
}
```

Figure 11: Final function implementations for "motorsDriver.c"

V. Deck Coding

1) First, create your file for the motor control deck. I named mine "pwmMotors.c".
    a) Navigate to this directory: "crazyflie-firmware/src/deck/drivers/src"
    b) Right click this directory and navigate to "New ->Source File". Click it.
    c) Enter the name for the file and do not forget the '.c' extension.
2) Setup the file with all appropriate headers found in Figure 12 below.
    a) "FreeRTOS" is a real time operating system that can be implemented in microcontrollers. I won't go over any details of it, but some of its functions will be used in this code. You can read about it here if you want.
    b) Note the inclusion of our driver file, "motorsDriver.h". This is necessary in order for us to use the functions we have setup there.

```
#include <stdint.h>
#include <string.h>
#include <unistd.h>

#include "stm32fxxx.h"
#include "config.h"
#include "deck.h"
#include "param.h"

#include "FreeRTOS.h"
#include "task.h"
#include "system.h"

#include "motorsDriver.h"
```

Figure 12: All necessary header files to include in "pwmMotors.c"

3) Now to setup all necessary variables and function definitions. Refer to Figure 13.
    a) The bool *isInit* is used in initialization.
    b) *prevDir* keeps track of the previously set mode. *prevSpeed* does the same with speed.
    c) *mode* is an enumeration used to specify 3 states for the motor: stopped, clockwise, or counterclockwise. It is accessed by a parameter later in the code.
    d) The function definition, "pwmMotors", is where our motor control will happen.
    e) The "pwmWrite" function calls our "motorSetRatio" from the driver with a provided number from 0-255. "stopMotor" does the same, but with a value of zero to stop the motor.

```
static bool isInit;
static int prevDir = -1;
static uint8_t prevSpeed = 255;
static uint8_t speed;
static uint16_t PERIOD_SLEEP_TIME_IN_MS = 100;

typedef enum{
    stop = 0,
    cw = 1,
    ccw = 2
} Mode;
static Mode mode;

void pwmMotors(void* arg);

void pwmWrite(uint8_t ratio){
    motorSetRatio(ratio);
}

void stopMotor(){
    motorSetRatio(0);
}
```

Figure 13: Variable and function definitions

4) The actual functions are next. Refer to Figure 14 for the code used.
    a) "pwmMotorsInit" takes care of initializing and setting up this file and the "pwmMotors" task. It sets PB5 and PB8 as output and sets up the motor to initially run clockwise at full speed.
    b) "pwmMotors" is setup such that the task sleeps for .1 second before checking the mode. If the mode hasn't changed, it just goes back to sleep, letting the motor continue what it is doing. Otherwise it adjusts the motors to match the new mode provided.

```
static void pwmMotorsInit(DeckInfo* info)
{
    if (isInit)
        return;

    motorDriverInit();

    pinMode(DECK_GPIO_IO1, OUTPUT); // used as output1
    pinMode(DECK_GPIO_IO2, OUTPUT); // used as output2
    mode = 1;
    speed = 255;
    motorSetFreq(10000);

    xTaskCreate(pwmMotors, "pwmMotors", configMINIMAL_STACK_SIZE, NULL, /*priority*/ 1, NULL);

    isInit = true;
}
void pwmMotors(void* arg)
{
  systemWaitStart();
  TickType_t xLastWakeTime;

  while (1) {
    xLastWakeTime = xTaskGetTickCount();
    if(prevDir == mode && prevSpeed == speed){
        vTaskDelayUntil(&xLastWakeTime, M2T(PERIOD_SLEEP_TIME_IN_MS));
        continue;
    }
    switch(mode){
        case stop:
            digitalWrite(DECK_GPIO_IO1, 0);
            digitalWrite(DECK_GPIO_IO2, 0);
            stopMotor();
            break;
        case cw:
            stopMotor();
            digitalWrite(DECK_GPIO_IO1, 1);
            digitalWrite(DECK_GPIO_IO2, 0);
            pwmWrite(speed);
            break;
        case ccw:
            stopMotor();
            digitalWrite(DECK_GPIO_IO1, 0);
            digitalWrite(DECK_GPIO_IO2, 1);
            pwmWrite(speed);
            break;
        default:
            //do nothing here
            break;
    }
    prevDir = mode;
    prevSpeed = speed;

    vTaskDelayUntil(&xLastWakeTime, M2T(PERIOD_SLEEP_TIME_IN_MS));
  }
}
```

Figure 14: Function implementations and initialization

5) Setting up the parameters to access *mode* and *speed* is the next task. Use Figure 15 below.
    a) *xMotorPWM* is naming the parameter group.
    b) The “PARAM_ADD” portion adds a parameter to the created group of the chosen type.
    c) I named the parameters the same as the variables they access for simplicity.

```
PARAM_GROUP_START(xMotorPWM)
PARAM_ADD(PARAM_UINT8, mode, &mode)
PARAM_ADD(PARAM_UINT8, speed, &speed)
PARAM_GROUP_STOP(xMotorPWM)
```

Figure 15: Parameter setup for the LED

6) The last piece of code sets up the file to be considered a deck. This is standard for external peripherals used with the Crazyflie 2.0. Refer to Figure 16 for details.
    a) Pay attention to what you name it, “pwmMotors” in my case. The name is needed later.
    b) The ports being used (besides VCC and GND) must be specified with the “DECK_USING” format. PB5, PB8, and RX2 are PB5, PB8, and PA2 respectively on the quadcopter.

```
static const DeckDriver pwmMotors_deck = {
        .vid =0,
        .pid =0,
        .name = "pwmMotors",
        .usedPeriph = DECK_USING_TIMER5,
        .usedGpio = DECK_USING_PB5 | DECK_USING_PB8 | DECK_USING_RX2,
        .init = pwmMotorsInit
};

DECK_DRIVER(pwmMotors_deck);
```

Figure 16: Setting up the file as a deck

7) After saving the file, the next step is to add the appropriate flags in “Makefile”.
    a) The makefile is found right in the “crazyflie-firmware” directory.
    b) Open up the makefile and scroll down to where you see a comment: “# Drivers”. It should be around line 140.
    c) In this drivers section, type in the flag shown at the bottom of Figure 17. The name before the ‘.o’ extension will be whatever you named your driver .c and .h files.

```
135 # Init
136 PROJ_OBJ += main.o
137 PROJ_OBJ_CF1 += platform_cf1.o
138 PROJ_OBJ_CF2 += platform_cf2.o
139
140 # Drivers
141 PROJ_OBJ_CF2 += motorsDriver.o
```

Figure 17: Adding driver Makefile flags

    d) Next, scroll down to where you see a comment: “# Decks”. It will be somewhere shortly before line 190.
    e) In this decks section, type in the flag shown at the bottom of Figure 18. The name before the ‘.o’ extension will be whatever you named your deck file.

```
182 # Deck API
183 #PROJ_OBJ_CF2 += wheeldeck.o
184 PROJ_OBJ_CF2 += deck_constants.o
185 PROJ_OBJ_CF2 += deck_digital.o
186 PROJ_OBJ_CF2 += deck_analog.o
187 PROJ_OBJ_CF2 += deck_spi.o
188
189 # Decks
190 PROJ_OBJ_CF2 += pwmMotors.o
```

Figure 18: Adding deck Makefile flags

8) Finally, setup the config.mk file.
    a) It is located in “crazyflie-firmware/tools/make”. Open it.
    b) Type the following: `CFLAGS += -DDECK_FORCE=pwmMotors`
    c) Remember the deck name from earlier? That’s what “pwmMotors” is here. Substitute accordingly if you used a different name.

DDECK_FORCE

9) That is it. Now all that is left is to build the project and your Crazyflie 2.0 will be able to run a motor clockwise or counterclockwise at a desired speed.