ABSTRACT

Four Channel Bidirectional Motor Control on the MSP430

William J. Richards

Director: Mr. John Miller

This project explores the initial development of a four-channel bidirectional motor control for the Texas Instruments MSP430F5529 Launchpad. Motor control is one of the most common tasks for an embedded microcontroller to perform. Chip manufacturers are keen to capture the hobbyist and do-it-yourself (DIY) microcontroller market, to encourage development of commercial products in their ecosystem. As a result, development boards have readily available modular expansion boards for common tasks. A survey of popular hobbyist retailers revealed that most available motor control expansion boards for hobbyist microcontroller development boards are limited to control of four motors with single direction control, or two motors with bidirectional control, and offer no position feedback. The result of this project is a circuit and driver package that allows control of four motorized linear potentiometers, including resistive position feedback, storage, and recall, as well as an interrupt. The resulting package is intended to be released and maintained by the open source community.

APPROVED BY DIRECTOR OF HONORS THESIS:

_____

Mr. John Miller, Department of Electrical Engineering

APPROVED BY THE HONORS PROGRAM:

_____

Dr. Andrew Wisely, Director

DATE: _____

FOUR CHANNEL BIDIRECTIONAL MOTOR CONTROL ON THE MSP430

A Thesis Submitted to the Faculty of

Baylor University

In Partial Fulfillment of the Requirements for the

Honors Program

By

William J. Richards

Waco, Texas

May 2015

TABLE OF CONTENTS

CHAPTER ONE

History of Audio Equipment

The advent of electrical sound amplification and reproduction fundamentally changed public performance. Since ancient time, musicians and orators relied on the natural acoustics of the performance space to propagate sound in such a way that their audience could hear it. In 1877, Thomas Edison was awarded the patent for the carbon button microphone, which was the first practical electric microphone, but it was not until 1911 that Edwin Pridham and Peter Jensen designed a speaker which could practically amplify sound [1]. With the advent of the electronic amplification and reproduction of sound, it was now possible to address crowds that were beyond the constraints of human vocal ability. This technological development resulted in the social developments of radio broadcast and public address systems, which have had revolutionary impact on entertainment, news, and political campaigning.

Origin of Mixing Systems

One technologically revolutionary result of the transition to electrical amplification and recording is the practice of audio mixing. Audio mixing is the adjustment of several audio signal inputs and summing them of to several audio signal outputs. This stems from the ability to run several amplifying circuits in parallel, at different gain levels. These signals can then be summed and sent as a single output. Audio mixing was done entirely with analog circuits until the release of the first digital console, the Neve Capricorn, in 1993 [2].

The Rise of Digital Mixers

In studio settings, digital mixing consoles were adopted as the fidelity of digital systems improved, but in live settings the analog mixer remained dominant. Even in the present day, most live production is done on an analog mixing system. This is due, in part, to early digital systems having poor reliability and audio quality. These flaws have long since been resolved, but the image remains. In fact, in terms of sound quality, the opposite has become a problem. Digital systems are often criticized for being too transparent, giving an audio characteristic dubbed "harsh" or "sterile." Another reason for the difference in adoption rate is that digital mixers have, to date, retained the form factor of a conventional mixing console. In a studio context, there is still a benefit to digital mixers. Studio consoles typically have a large number of inputs and outputs, due to multi-track production methods. As a result of the studio console format, analog studio consoles get increasingly complex to construct and repair as they increase in channel count. Live mixing consoles, on the other hand, take a large number of inputs but reduce them to a small number of outputs. A small console, meant for small venues or traveling bands that provide their own equipemtn, may have as few as three output channels. This format served the needs of the production community well until the recent advent of in-ear monitoring systems.

Monitoring Systems: An Overview

A monitoring system provides audio feedback to a performer (or group of performers) so that they can hear, and adjust, their performance. This is especially useful in groups where one performer or instrument is significantly quieter than another or if an instrument is highly directional in sound. Prior to recent advances in wireless technology, it was common practice to have only one or two monitor channels, directed to speakers on stage (see Figure 1). Some large bands would have more complex monitoring arrangements, but this would require either a second sound technician to do the mixing for the monitors, or the technician to perform both monitoring and live mixing from the main mixer.

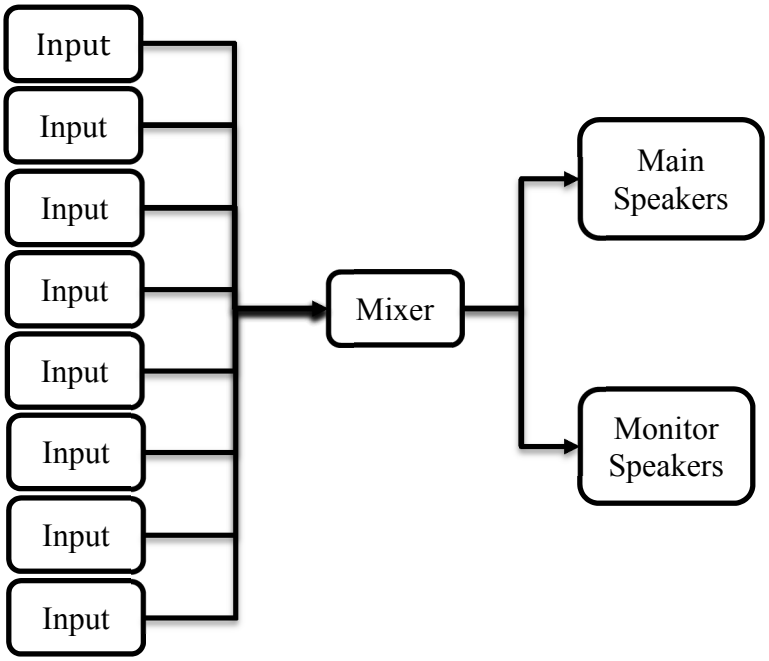

Figure 1: A Traditional Mixing and Monitoring Setup

One of the challenges the traditional monitoring method faces is that the ideal monitor mix is different for each performer or band member. The decreasing

expense of digital mixing technology has led to the current practice of personalized monitoring, which takes a duplicate of the incoming audio signals as they enter the mixer, digitizes them, and delivers them to a personal mixing station. The pinnacle of personalized mixing delivers the mix to earphones worn by the performer, which is called "in-ear monitoring." However, this solution requires an auxiliary system to the main mixing console, a number of cables running to the personal mixing stations, and often an equal number of cables to return the mixed audio signal to a wireless transmitter for the in-ear monitoring system (see Figure 2).

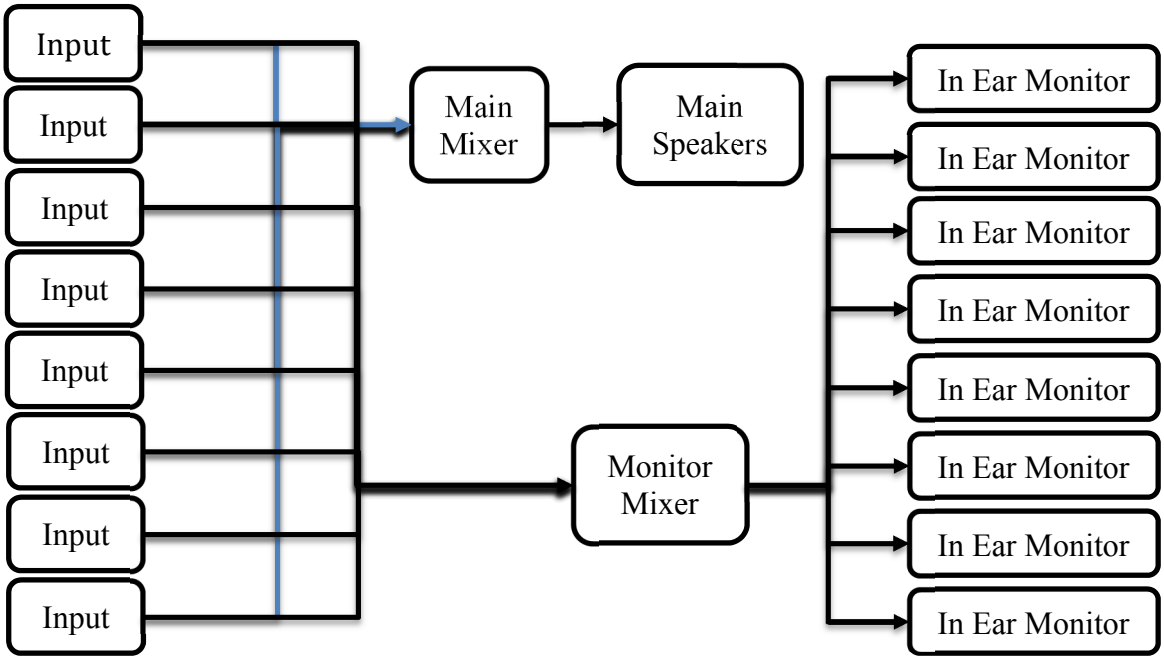

Figure 2: Conventional Mixing and Personal Monitoring System

CHAPTER TWO

Project Context

The Digital Monitoring Mixing System (DMMS) is a project started by Prof. John Miller. As a volunteer sound technician at his church, Prof. Miller found commercially available live mixing and monitoring solutions to be lacking in space efficiency and scalability. In commercially available mixers, changing the number of available channels required purchasing a new mixer that has the proper number of channels. In-ear monitoring required the purchase of an auxiliary system, which adds substantial amounts of cabling, as well as personal mixers for each of the performers, which are expensive.

The DMMS sought to replace the traditional mixing console and wireless monitoring systems with a system that consists of a mixing controller unit and breakout boxes for audio input and output (see Figure 3), all of which are rack-mountable for a tidy, compact system. The conventional mixing console surface is replaced by an electronic interface based on touchscreen devices like smartphones and tablets. This is excellent for in-ear monitor mixing, however, these devices provide minimal haptic feedback, and are limited in channel accessibility and data availability by their small screen size. To address this issue, a control surface was under development that replicates the experience of operating a standard mixing console. This console was intended to be modular, allowing addition or removal of channels as the need and budget of the customer changes.

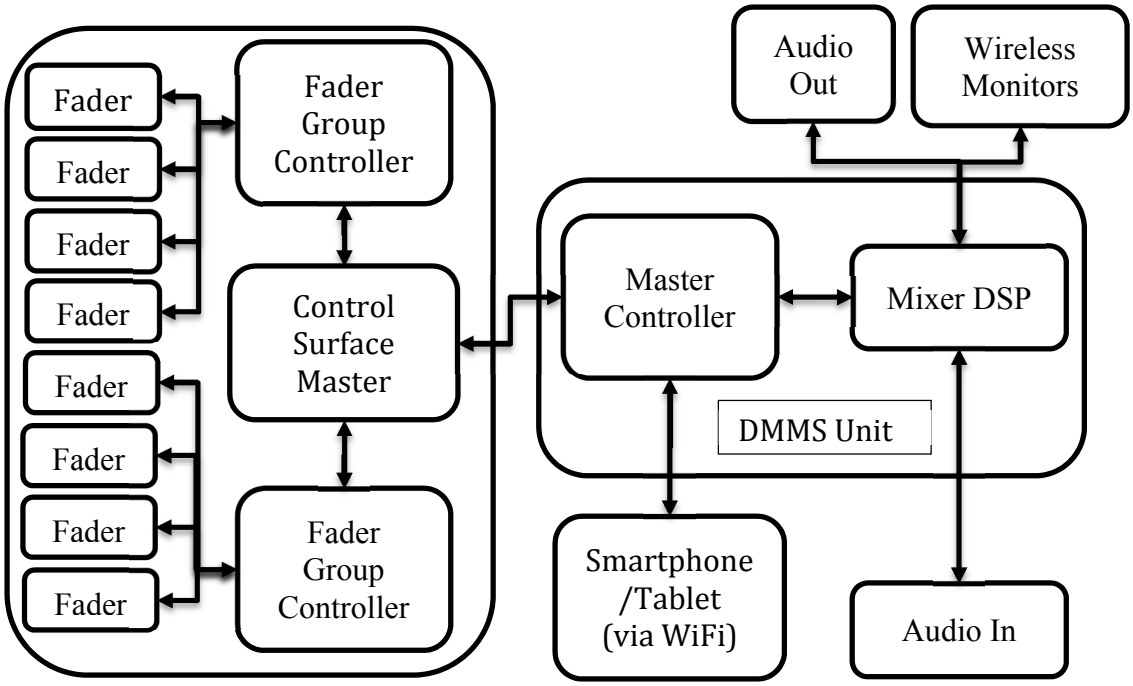

Figure 3: Block Diagram of DMMS

This project originated as designing a module to control a group of faders, a subsystem of this control surface. In commercial mixing designs, a fader controller often controls only one fader, which is economically feasible when the unit is mass-produced. The DMMS project, however, was under development with minimal funding, so a group controller made sense for the console.

A New Direction: The Open Source Community

Over the course of this project, products have come to market that integrate several features of the DMMS. The Mackie 1608 and Line6 StageScape M20d introduced iPad based mixing control, though targeted at small bands who have to

do their own mixing, and the Mackie DL32R added the ability to control monitoring from additional tablets or smartphones. Because the innovations of the DMMS system were now present in commercial products, development of the DMMS turned to developing technology that could be used by the do-it-yourself (DIY), hobbyist or "maker" community. The DIY community develops technology on the small scale to meet personalized needs, to develop better tools and equipment for others in the community to use, or to prototype commercial products. Supporting this community is important to the academic engineering community because many engineers find the start of their career in such tinkering.

The DMMS Control Surface seemed to be the most promising subsystem to focus on. Control surfaces are useful to many hobbyist areas, including audio recording and editing, video editing, robotics, automation, and model railroads. It is also promising that people with these hobbies tend to also belong to the DIY, open source, or maker communities. However, the generalized concept of a fader controller, a multichannel motor driver with feedback, is more useful to the DIY community than a fully developed control surface. There are two-channel motor drivers available, and feedback can be implemented, but no four-channel drivers exist, even though the hardware capacity exists. This project addresses that need, and develops a rudimentary multichannel DC motor control unit to be improved upon by the open source community.

CHAPTER THREE

Implementation

This project is intended to provide a basis for a four-channel motor control expansion board for common microcontroller development boards. This chapter will summarize the concepts behind implementing this project, as well as describe the process to duplicate the design produced in this project.

Important Concepts

Microprocessors and Microcontrollers

A microprocessor is a common example of an embedded computer. Embedded computers are computers that are designed to perform a simple set of tasks with high efficiency and reliability and to be inexpensive. A microcontroller is a microprocessor bundled with common peripherals like timers, analog-to-digital converters (ADCs), flash memory, communication, or pulse-width-modulation.

The most prevalent applications of microcontrollers are data collection and process control. In general, microcontrollers do not have complex user interfaces, like keyboards and monitors on a personal computer. Instead, microcontrollers primarily use sensors and actuators. It is unlikely that a person using a commercial implementation of a microcontroller will realize that they are using one.

Motor Control Overview

Motors are one of the three common actuators used by a microcontroller. (The other two are linear actuators and solenoids.) Motor control is the minimization of steady-state error in motors. Frequently this is done with proportional-integral-derivative (PID) control and PWM of the voltage. This implementation uses simplified PWM and does not use PID control, though PID algorithm are intended to be developed after the release of this project to open source.

PID Control Theory

PID control theory is a mathematical model that uses three terms in determining the error correction to be used. These terms are a proportional term, an integral term, and a derivative term, from which the theory gets its name.

The proportional term of a PID model asserts that correction should be proportional to the error, so a large error will result in a larger corrective reaction. In the context of motorized linear faders, this means that the farther the fader is from its desired position, the faster the motor will turn, and as it approaches the desired position, the motor will slow down. This term is the most prominent term in a PID model, and in many cases, is the only term required to get satisfactory results.

The integral term of a PID model acts as an accumulator, prioritizing small errors over time and forcing the error to zero. The integral term introduces

overshoot into the system, introducing a tradeoff. While integral terms increase system response time, the characteristic of overshoot is often undesirable, so the integral term should be kept as small as effectively eliminates steady-state error.

The derivative term of a PID model adds prediction to the PID model, counteracting the overshoot introduced by the integral term and promoting a shorter settling time. For situations where precision and stability are important, like fader automation, the derivative term will be strong.

Pulse-Width-Modulation

Pulse-width-modulation (PWM) is a common method for digital devices to emulate analog voltage outputs. PWM uses a pulse of high voltage, followed by a pulse of low voltage, to output an average value equivalent to the desired analog voltage. Because of the frequency that digital devices operate at and the inherent capacitance and inductance of high power components like motors, this averaging method works well for high power applications. If the pulsing does impact device performance, steps can be taken to mitigate this (e.g. a capacitor can be added to smooth the voltage transition.)

Development and Expansion Boards

Microprocessor manufacturers originally produced development boards (sometimes called evaluation boards) for potential customers to develop proof of concepts or prototype designs. In 2005, the Interaction Design Institute Ivera released a low-cost development board called Arduino for their students. This board proved extremely popular with the DIY community, and as a result, Texas

Instruments and Atmel (whose processor the Arduino is based on) designed low cost development boards, in an effort to encourage familiarity of the newly discovered community of innovators with their products. The development board chosen for this project is one of these, the Texas Instruments MSP430F5529 Launchpad.

Implementation of Motor Control on MSP430

The MSP430F5529 Launchpad was chosen because of its affordability and low power consumption. A similar concept could be implemented on an Arduino development board. All components in this project were mounted to a solderless breadboard except for the faders, which were mounted to a surface, and wires were run to the breadboard.

This motor controller used a pair of TI SN754410 ICs to drive the fader motors. The SN754410 is a standard Quadruple Half-H Driver chip, a common device for enabling a microcontroller to control a DC motor. A Half-H Driver takes an input of logic-level voltage (typically 3.3V or 5V) and connects or disconnects a higher voltage to provide power to loads that are too large for a microcontroller to directly power. A typical application circuit is displayed in Figure 4. For this project, he inversion gates on the inputs to the chip were implemented in software and the motor coils $\phi 1$ and $\phi 2$ were on two separate motors, instead of on one motor, as illustrated.

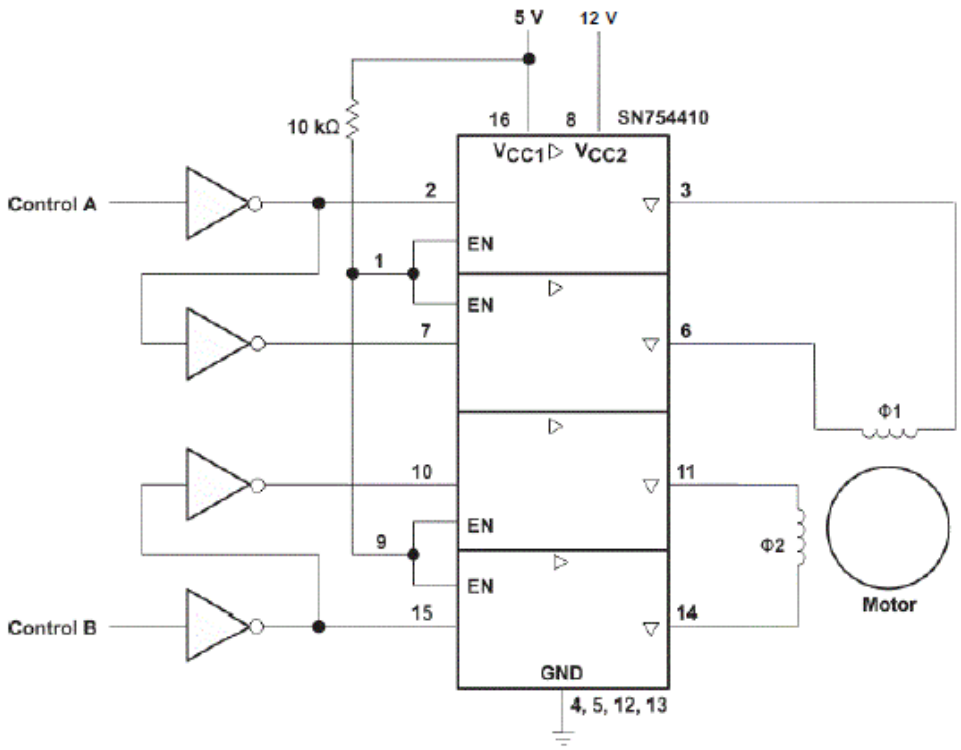

Figure 4: Typical Application Circuit for SN754410 [3]

The Tables in Appendix A give pin connections for all components. Beside the connections with the SN754410 and the faders, the MSP430 only needs to be connected to a common ground with the other ICs and faders.

Two example programs (available in appendices B and C) were written that demonstrate the functions of the circuit. These programs were compiled using Code Composer Studio 6, and were successfully implemented on the system. The first program demonstrates the ability to respond to input. The second program demonstrates an ability to restore saved positions. For this project simple PWM is used to control motor speed, and the electrical resistance of the fader is used to provide position feedback.

CHAPTER FOUR

Conclusions and Future Actions

This project demonstrated that a four channel motor controller is viable using a low-cost microcontroller and provides documentation for others to implement the design for themselves.

Challenges

One of the primary challenges of implementation is the PWM for controlling motor drive. The delay functions provided with the microcontroller required a constant integer input, forcing the pulse width to be of a fixed ratio. While this was adequate for a simple proof of concept, for better motor control a variable PWM is needed. This issue can be addressed using the MSP430's Timer_A module, but was not demonstrated due to time constraints.

Future of the Four Channel Motor Control Circuit

The future of this project is a release to the open source community for further development. This release will include refined schematics of the circuit used in this project, PCB design files that conform to the MSP430 Booster Pack and the Arduino Shield format. The release will also include a polished version of the example programs included in this project, adding variable PWM for motor speed control, PID, as well as drivers offering simple fader control. These drivers and examples will be available in C and in Energia/Arduino sketch format. This enables the DIY community easy access to both the hardware design and easy software implementation.

APPENDICES

APPENDIX A

Pin Connection Tables

Table A.1: Fader Pin Connections

	Fader 1	Fader 2	Fader 3	Fader 4
Pin 1	Vcc (3.3V)	Vcc (3.3V)	Vcc (3.3V)	Vcc (3.3V)
Pin 1’	Not Connected	Not Connected	Not Connected	Not Connected
Pin 2	ADC0	ADC1	ADC2	ADC3
Pin 2’	Not Connected	Not Connected	Not Connected	Not Connected
Pin 3	Ground	Ground	Ground	Ground
Pin 3’	Not Connected	Not Connected	Not Connected	Not Connected
Pin T	Not Connected	Not Connected	Not Connected	Not Connected
Motor Pos	Driver 1.1Y	Driver 1.3Y	Driver 2.1Y	Driver 2.3Y
Motor Ned	Driver 1.2Y	Driver 1.4Y	Driver 2.2Y	Driver 2.4Y

Table A.2: SN754410 Pin Connections

Pin	Driver 1	Driver 2
Vcc1	5V	5V
Vcc2	12V	12V
EN12	Pin 3.7	Pin 2.3
EN34	Pin 8.2	Pin 8.1
1A	Pin 2.5	Pin 1.3
1Y	Motor 1 Positive	Motor 3 Positive
2A	Pin 2.4	Pin 1.2
2Y	Motor 1 Negative	Motor 3 Negative
3A	Pin 1.5	Pin 4.3
3Y	Motor 2 Positive	Motor 4 Positive
4A	Pin 1.4	Pin 4.0
4Y	Motor 2 Negative	Motor 4 Negative

APPENDIX B

Example Code, Fader Chase

```
#include "driverlib.h"
/*
 * Recall Demonstration
 * By William Richards
 * 24 April 2015
 *
 * Description: This code demonstrates the ability of the Motor Control Circuit
 * to recall saved positions of faders, for initialization of a fader control
 * unit.
 *
 * This code is based on a MSP430F5529 Launchpad with four faders attached:
 * Fader *.1 - Vcc (3.3V)
 * Fader *.3 - Ground
 * Fader 1.2 - ADC0
 * Fader 2.2 - ADC1
 * Fader 3.2 - ADC2
 * Fader 4.2 - ADC3
 *
 * Driver 1.EN12 - Pin 3.7
 * Driver 1.EN34 - Pin 8.2
 * Driver 1.1A - Pin 2.5
 * Driver 1.2A - Pin 2.4
 * Driver 1.3A - Pin 1.5
 * Driver 1.4A - Pin 1.4
 * Driver *.Vcc1 - (5V)
 * Driver *.Vcc2 - (12V)
 *
 * Driver 2.EN12 - Pin 2.3
 * Driver 2.EN34 - Pin 8.1
 * Driver 2.1A - Pin 1.3
 * Driver 2.2A - Pin 1.2
 * Driver 2.3A - Pin 4.3
 * Driver 2.4A - Pin 4.0
 *
 */

// results stores the values retrieved from ADC12_A
uint16_t results [5] = {0,0,0,0,0};
// faderRes provides discrete segments of fader travel. Resolution finer than
// 128 positions introduces motor jitter due to lack of PID control
int16_t faderRes = 0xfff/128;
// Indicates status of driver chip enable, to prevent excessive GPIO writing
bool F1IsPaused = false;
bool F2IsPaused = false;
bool F3IsPaused = false;
```

```
bool F4IsPaused = false;
bool D2IsPaused = false;
// Approximate ADC12_A value for a fader at the top of its track
int16_t maxFaderVal = 0xEE0;
// These values are the stores positions on a scale of 128.
int16_t storedPos1 = 0;
int16_t storedPos2 = 30;
int16_t storedPos3 = 60;
int16_t storedPos4 = 90;

int main(void) {

    WDT_A_hold(WDT_A_BASE);

    // Setup Driver 1: Enable outputs, and set all low
    GPIO_setAsOutputPin(GPIO_PORT_P3,GPIO_PIN7);        // D1.EN12
    GPIO_setOutputLowOnPin(GPIO_PORT_P3,GPIO_PIN7);
    GPIO_setAsOutputPin(GPIO_PORT_P8,GPIO_PIN2);        // D1.EN34
    GPIO_setOutputLowOnPin(GPIO_PORT_P8,GPIO_PIN2);
    GPIO_setAsOutputPin(GPIO_PORT_P2,GPIO_PIN5);        // D1.1A
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN5);
    GPIO_setAsOutputPin(GPIO_PORT_P2,GPIO_PIN4);        // D1.2A
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN4);
    GPIO_setAsOutputPin(GPIO_PORT_P1,GPIO_PIN5);        // D1.3A
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN5);
    GPIO_setAsOutputPin(GPIO_PORT_P1,GPIO_PIN4);        // D1.4A
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN4);

    // Setup Driver 2: Enable outputs, and set all low
    GPIO_setAsOutputPin(GPIO_PORT_P2,GPIO_PIN3);        // D2.EN12
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN3);
    GPIO_setAsOutputPin(GPIO_PORT_P8,GPIO_PIN1);        // D2.EN34
    GPIO_setOutputLowOnPin(GPIO_PORT_P8,GPIO_PIN1);
    GPIO_setAsOutputPin(GPIO_PORT_P1,GPIO_PIN3);        // D2.1A
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN3);
    GPIO_setAsOutputPin(GPIO_PORT_P1,GPIO_PIN2);        // D2.2A
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN2);
    GPIO_setAsOutputPin(GPIO_PORT_P4,GPIO_PIN0);        // D2.3A
    GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN0);
    GPIO_setAsOutputPin(GPIO_PORT_P4,GPIO_PIN3);        // D2.4A
    GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN3);

    // Setup ADC Input Pins
    GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P6,
                                               GPIO_PIN0 + GPIO_PIN1 + GPIO_PIN2 +
                                               GPIO_PIN3
                                               );

    //Initialize the ADC12_A Module
    ADC12_A_init(ADC12_A_BASE,
                 ADC12_A_SAMPLEHOLDSOURCE_SC,
                 ADC12_A_CLOCKSOURCE_ADC12OSC,
                 ADC12_A_CLOCKDIVIDER_1);

    ADC12_A_enable(ADC12_A_BASE);
```

```
ADC12_A_setupSamplingTimer(ADC12_A_BASE,
                           ADC12_A_CYCLEHOLD_16_CYCLES,
                           ADC12_A_CYCLEHOLD_4_CYCLES,
                           ADC12_A_MULTIPLESAMPLESENABLE);

//Configure Memory Buffers
ADC12_A_configureMemoryParam param0 = {0};
param0.memoryBufferControlIndex = ADC12_A_MEMORY_0;
param0.inputSourceSelect = ADC12_A_INPUT_A0;
param0.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_AVCC;
param0.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
param0.endOfSequence = ADC12_A_NOTENDOFSEQUENCE;
ADC12_A_configureMemory(ADC12_A_BASE,&param0);

ADC12_A_configureMemoryParam param1 = {0};
param1.memoryBufferControlIndex = ADC12_A_MEMORY_1;
param1.inputSourceSelect = ADC12_A_INPUT_A1;
param1.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_AVCC;
param1.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
param1.endOfSequence = ADC12_A_NOTENDOFSEQUENCE;
ADC12_A_configureMemory(ADC12_A_BASE,&param1);

ADC12_A_configureMemoryParam param2 = {0};
param2.memoryBufferControlIndex = ADC12_A_MEMORY_2;
param2.inputSourceSelect = ADC12_A_INPUT_A2;
param2.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_AVCC;
param2.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
param2.endOfSequence = ADC12_A_NOTENDOFSEQUENCE;
ADC12_A_configureMemory(ADC12_A_BASE,&param2);

ADC12_A_configureMemoryParam param3 = {0};
param3.memoryBufferControlIndex = ADC12_A_MEMORY_3;
param3.inputSourceSelect = ADC12_A_INPUT_A3;
param3.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_AVCC;
param3.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
param3.endOfSequence = ADC12_A_ENDOFSEQUENCE;
ADC12_A_configureMemory(ADC12_A_BASE,&param3);

//Enable memory buffer 3 interrupt
ADC12_A_clearInterrupt(ADC12_A_BASE,
                       ADC12IE3);
ADC12_A_enableInterrupt(ADC12_A_BASE,
                        ADC12IE3);
// Program loop
while(1)
{
    //Start first sampling and conversion cycle
    ADC12_A_startConversion(ADC12_A_BASE,
                            ADC12_A_MEMORY_0,
                            ADC12_A_SEQOFCHANNELS);

    // Wait for conversion to complete
    //Enter LPM4, Enable interrupts
    __bis_SR_register(LPM4_bits + GIE);
```

```
//For debugger
__no_operation();

// If Fader 1 is too high, move it down
if(((int16_t)((maxFaderVal/128*storedPos1)-results[0])>faderRes)){
    F1IsPaused = false;
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN4); // D1.4A
    GPIO_setOutputHighOnPin(GPIO_PORT_P3,GPIO_PIN7); // D1.EN34
    GPIO_setOutputHighOnPin(GPIO_PORT_P2,GPIO_PIN5); // D1.3A

} else if(((int16_t)(results[0]-maxFaderVal/128*storedPos1)>faderRes)) {
    // Fader 1 too low, move up.
    F1IsPaused = false;
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN5); // D1.3A
    GPIO_setOutputHighOnPin(GPIO_PORT_P3,GPIO_PIN7); // D1.EN34
    GPIO_setOutputHighOnPin(GPIO_PORT_P2,GPIO_PIN4); // D1.4A

} else if (F1IsPaused == false){
    // If Fader 1 is where it belongs, stop motor
    GPIO_setOutputLowOnPin(GPIO_PORT_P3,GPIO_PIN7);
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN4);
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN5);
    F1IsPaused = true;
}

// If Fader 2 is too high, move it down
if(((int16_t)((maxFaderVal/128*storedPos2)-results[1])>faderRes)){
    F2IsPaused = false;
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN5); // D1.3A
    GPIO_setOutputHighOnPin(GPIO_PORT_P8,GPIO_PIN2); // D1.EN34
    GPIO_setOutputHighOnPin(GPIO_PORT_P1,GPIO_PIN4); // D1.4A

} else if(((int16_t)(results[1]-maxFaderVal/128*storedPos2)>faderRes)) {
    // Fader 2 too high, move up.
    F2IsPaused = false;
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN4); // D1.4A
    GPIO_setOutputHighOnPin(GPIO_PORT_P8,GPIO_PIN2); // D1.EN34
    GPIO_setOutputHighOnPin(GPIO_PORT_P1,GPIO_PIN5); // D1.3A

} else if (F2IsPaused == false){
    // If Fader 1 is where it belongs, stop motor
    GPIO_setOutputLowOnPin(GPIO_PORT_P8,GPIO_PIN2);
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN4);
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN5);

    F2IsPaused = true;
}

// If Fader 3 is too high, move it down
if(((int16_t)((maxFaderVal/128*storedPos3)-results[2])>faderRes)){
    F3IsPaused = false;
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN3); // D1.3A
    GPIO_setOutputHighOnPin(GPIO_PORT_P2,GPIO_PIN3); // D1.EN34
    GPIO_setOutputHighOnPin(GPIO_PORT_P1,GPIO_PIN2); // D1.4A
```

```
    } else if(((int16_t)(results[2]-maxFaderVal/128*storedPos3)>faderRes)) {
        // Fader 3 too low, move up.
        F3IsPaused = false;
        GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN2); // D1.4A
        GPIO_setOutputHighOnPin(GPIO_PORT_P2,GPIO_PIN3); // D1.EN34
        GPIO_setOutputHighOnPin(GPIO_PORT_P1,GPIO_PIN3); // D1.3A

    } else if (F3IsPaused == false){
        GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN3);
        GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN2);
        GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN3);

        F3IsPaused = true;
    }

    // If Fader 4 is too high, move it down
    if(((int16_t)((maxFaderVal/128*storedPos4)-results[3])>faderRes)){
        F3IsPaused = false;
        GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN3); // D2.3A
        GPIO_setOutputHighOnPin(GPIO_PORT_P8,GPIO_PIN1); // D2.EN34
        GPIO_setOutputHighOnPin(GPIO_PORT_P4,GPIO_PIN0); // D2.4A

    } else if(((int16_t)(results[3]-maxFaderVal/128*storedPos4)>faderRes)) {
        // Fader 4 too low, move up.
        F3IsPaused = false;
        GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN0); // D2.4A
        GPIO_setOutputHighOnPin(GPIO_PORT_P8,GPIO_PIN1); // D2.EN34
        GPIO_setOutputHighOnPin(GPIO_PORT_P4,GPIO_PIN3); // D2.3A

    } else if (F3IsPaused == false){
        GPIO_setOutputLowOnPin(GPIO_PORT_P8,GPIO_PIN1);
        GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN0);
        GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN3);

        F3IsPaused = true;
    }

    // Wait for motion to happen, then pause to check current status
    _delay_cycles(1500);
    // Fader 1
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN4);
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN5);
    // Fader 2
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN4);
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN5);
    // Fader 3
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN2);
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN3);
    // Fader 4
    GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN0);
    GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN3);
    _delay_cycles(4000);

}
```

```
    return (0);
}
// ADC interrupt handler - reads ADC results, stores them in array "results"
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=ADC12_VECTOR
__interrupt
#elif defined(__GNUC__)
__attribute__((interrupt(ADC12_VECTOR)))
#endif
void ADC12ISR(void)
{
    switch(__even_in_range(ADC12IV,34))
    {
        case  0: break;       //Vector  0:  No interrupt
        case  2: break;       //Vector  2:  ADC overflow
        case  4: break;       //Vector  4:  ADC timing overflow
        case  6: break;       //Vector  6:  ADC12IFG0
        case  8: break;       //Vector  8:  ADC12IFG1
        case 10: break;       //Vector 10:  ADC12IFG2
        case 12:              //Vector 12:  ADC12IFG3
            //Move results, IFG is cleared
            results[0] =
            ADC12_A_getResults(ADC12_A_BASE,
                               ADC12_A_MEMORY_0);
            //Move results, IFG is cleared
            results[1] =
            ADC12_A_getResults(ADC12_A_BASE,
                               ADC12_A_MEMORY_1);
            //Move results, IFG is cleared
            results[2] =
            ADC12_A_getResults(ADC12_A_BASE,
                               ADC12_A_MEMORY_2);
            //Move results, IFG is cleared
            results[3] =
            ADC12_A_getResults(ADC12_A_BASE,
                               ADC12_A_MEMORY_3);
            //Exit active CPU, SET BREAKPOINT HERE
            __bic_SR_register_on_exit(LPM4_bits);

        case 14: break;       //Vector 14:  ADC12IFG4
        case 16: break;       //Vector 16:  ADC12IFG5
        case 18: break;       //Vector 18:  ADC12IFG6
        case 20: break;       //Vector 20:  ADC12IFG7
        case 22: break;       //Vector 22:  ADC12IFG8
        case 24: break;       //Vector 24:  ADC12IFG9
        case 26: break;       //Vector 26:  ADC12IFG10
        case 28: break;       //Vector 28:  ADC12IFG11
        case 30: break;       //Vector 30:  ADC12IFG12
        case 32: break;       //Vector 32:  ADC12IFG13
        case 34: break;       //Vector 34:  ADC12IFG14
        default: break;
    }
}
```

APPENDIX C

Example Code, Recall

```
#include "driverlib.h"
/*
 * Double Channel Chase Demonstration
 * By William Richards
 * 24 April 2015
 *
 * Description: This code demonstrates the ability of the Motor Control Circuit
 * to read and write to multiple channels at once. When loaded into a MSP430
 * and the accompanying circuit, adjusting the first and third channels will
 * cause the microcontroller to move the second and fourth faders to mimic the
 * position of the respective odd numbered fader.
 *
 * This code is based on a MSP430F5529 Launchpad with four faders attached:
 * Fader *.1 - Vcc (3.3V)
 * Fader *.3 - Ground
 * Fader 1.2 - ADC0
 * Fader 2.2 - ADC1
 * Fader 3.2 - ADC2
 * Fader 4.2 - ADC3
 *
 * Driver 1.EN12 - Pin 3.7
 * Driver 1.EN34 - Pin 8.2
 * Driver 1.1A - Pin 2.5
 * Driver 1.2A - Pin 2.4
 * Driver 1.3A - Pin 1.5
 * Driver 1.4A - Pin 1.4
 * Driver *.Vcc1 - (5V)
 * Driver *.Vcc2 - (12V)
 *
 * Driver 2.EN12 - Pin 2.3
 * Driver 2.EN34 - Pin 8.1
 * Driver 2.1A - Pin 1.3
 * Driver 2.2A - Pin 1.2
 * Driver 2.3A - Pin 4.3
 * Driver 2.4A - Pin 4.0
 *
 */

// results stores the values retrieved from ADC12_A
uint16_t results [5] = {0,0,0,0,0};
// faderRes provides discrete segments of fader travel. Resolution finer than
// 128 positions introduces motor jitter due to lack of PID control
int16_t faderRes = 0xfff/128;
// Indicates status of driver chip enable, to prevent excessive GPIO writing
bool D1IsPaused = false;
```

```
bool D2IsPaused = false;

int main(void) {

    WDT_A_hold(WDT_A_BASE);

    // Setup Driver 1: Enable outputs, and set all low
    GPIO_setAsOutputPin(GPIO_PORT_P3,GPIO_PIN7);        // D1.EN12
    GPIO_setOutputLowOnPin(GPIO_PORT_P3,GPIO_PIN7);
    GPIO_setAsOutputPin(GPIO_PORT_P8,GPIO_PIN2);        // D1.EN34
    GPIO_setOutputLowOnPin(GPIO_PORT_P8,GPIO_PIN2);
    GPIO_setAsOutputPin(GPIO_PORT_P2,GPIO_PIN5);        // D1.1A
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN5);
    GPIO_setAsOutputPin(GPIO_PORT_P2,GPIO_PIN4);        // D1.2A
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN4);
    GPIO_setAsOutputPin(GPIO_PORT_P1,GPIO_PIN5);        // D1.3A
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN5);
    GPIO_setAsOutputPin(GPIO_PORT_P1,GPIO_PIN4);        // D1.4A
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN4);

    // Setup Driver 2: Enable outputs, and set all low
    GPIO_setAsOutputPin(GPIO_PORT_P2,GPIO_PIN3);        // D2.EN12
    GPIO_setOutputLowOnPin(GPIO_PORT_P2,GPIO_PIN3);
    GPIO_setAsOutputPin(GPIO_PORT_P8,GPIO_PIN1);        // D2.EN34
    GPIO_setOutputLowOnPin(GPIO_PORT_P8,GPIO_PIN1);
    GPIO_setAsOutputPin(GPIO_PORT_P1,GPIO_PIN3);        // D2.1A
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN3);
    GPIO_setAsOutputPin(GPIO_PORT_P1,GPIO_PIN2);        // D2.2A
    GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN2);
    GPIO_setAsOutputPin(GPIO_PORT_P4,GPIO_PIN0);        // D2.3A
    GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN0);
    GPIO_setAsOutputPin(GPIO_PORT_P4,GPIO_PIN3);        // D2.4A
    GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN3);

    // Setup ADC Input Pins
    GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P6,
                                               GPIO_PIN0 + GPIO_PIN1 + GPIO_PIN2 +
       GPIO_PIN3);

    //Initialize the ADC12_A Module
    ADC12_A_init(ADC12_A_BASE,
                 ADC12_A_SAMPLEHOLDSOURCE_SC,
                 ADC12_A_CLOCKSOURCE_ADC12OSC,
                 ADC12_A_CLOCKDIVIDER_1);

    ADC12_A_enable(ADC12_A_BASE);

    ADC12_A_setupSamplingTimer(ADC12_A_BASE,
                               ADC12_A_CYCLEHOLD_16_CYCLES,
                               ADC12_A_CYCLEHOLD_4_CYCLES,
                               ADC12_A_MULTIPLESAMPLESENABLE);

    //Configure Memory Buffers
    ADC12_A_configureMemoryParam param0 = {0};
    param0.memoryBufferControlIndex = ADC12_A_MEMORY_0;
```

```
param0.inputSourceSelect = ADC12_A_INPUT_A0;
param0.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_AVCC;
param0.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
param0.endOfSequence = ADC12_A_NOTENDOFSEQUENCE;
ADC12_A_configureMemory(ADC12_A_BASE,&param0);

ADC12_A_configureMemoryParam param1 = {0};
param1.memoryBufferControlIndex = ADC12_A_MEMORY_1;
param1.inputSourceSelect = ADC12_A_INPUT_A1;
param1.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_AVCC;
param1.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
param1.endOfSequence = ADC12_A_NOTENDOFSEQUENCE;
ADC12_A_configureMemory(ADC12_A_BASE,&param1);

ADC12_A_configureMemoryParam param2 = {0};
param2.memoryBufferControlIndex = ADC12_A_MEMORY_2;
param2.inputSourceSelect = ADC12_A_INPUT_A2;
param2.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_AVCC;
param2.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
param2.endOfSequence = ADC12_A_NOTENDOFSEQUENCE;
ADC12_A_configureMemory(ADC12_A_BASE,&param2);

ADC12_A_configureMemoryParam param3 = {0};
param3.memoryBufferControlIndex = ADC12_A_MEMORY_3;
param3.inputSourceSelect = ADC12_A_INPUT_A3;
param3.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_AVCC;
param3.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
param3.endOfSequence = ADC12_A_ENDOFSEQUENCE;
ADC12_A_configureMemory(ADC12_A_BASE,&param3);

//Enable memory buffer 3 interrupt
ADC12_A_clearInterrupt(ADC12_A_BASE,
                       ADC12IE3);
ADC12_A_enableInterrupt(ADC12_A_BASE,
                        ADC12IE3);

// Program loop
while(1)
{
    //Start first sampling and conversion cycle
    ADC12_A_startConversion(ADC12_A_BASE,
                            ADC12_A_MEMORY_0,
                            ADC12_A_SEQOFCHANNELS);

    // Wait for conversion to complete
    //Enter LPM4, Enable interrupts
    __bis_SR_register(LPM4_bits + GIE);

    //For debugger
    __no_operation();

    // Test fader levels, and initiate motor correction if needed
    if(((int16_t)(results[0]-results[1])>faderRes)){
        // Fader 2 > Fader 1, move down
        D1IsPaused = false;
```

```
            GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN5);  // D1.3A
            GPIO_setOutputHighOnPin(GPIO_PORT_P8,GPIO_PIN2); // D1.EN34
            GPIO_setOutputHighOnPin(GPIO_PORT_P1,GPIO_PIN4); // D1.4A

        } else if(((int16_t)(results[1]-results[0])>faderRes)) {
            // Fader 2 < Fader 1, move up.
            D1IsPaused = false;
            GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN4);  // D1.4A
            GPIO_setOutputHighOnPin(GPIO_PORT_P8,GPIO_PIN2); // D1.EN34
            GPIO_setOutputHighOnPin(GPIO_PORT_P1,GPIO_PIN5); // D1.3A

        } else if (D1IsPaused == false){
            // Fader 2 = Fader 1, stop motor
            GPIO_setOutputLowOnPin(GPIO_PORT_P8,GPIO_PIN2);  // D1.EN34
            GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN4);  // D1.4A
            GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN5);  // D1.3A
            // Set pause, so this does not execute constantly
            D1IsPaused = true;
        }

        if(((int16_t)(results[2]-results[3])>faderRes)){
            // Fader 4 > Fader 3, move down
            D2IsPaused = false;
            GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN3);  // D2.3A
            GPIO_setOutputHighOnPin(GPIO_PORT_P8,GPIO_PIN1); // D2.EN34
            GPIO_setOutputHighOnPin(GPIO_PORT_P4,GPIO_PIN0); // D2.4A

        } else if(((int16_t)(results[3]-results[2])>faderRes)) {
            // Fader 4 < Fader 3, move up.
            D2IsPaused = false;
            GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN0);  // D2.4A
            GPIO_setOutputHighOnPin(GPIO_PORT_P8,GPIO_PIN1); // D2.EN34
            GPIO_setOutputHighOnPin(GPIO_PORT_P4,GPIO_PIN3); // D2.3A

        } else if (D2IsPaused == false){
            // Fader 4 = Fader 3 stop motor
            GPIO_setOutputLowOnPin(GPIO_PORT_P8,GPIO_PIN1);  // D2.EN34
            GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN0);  // D2.4A
            GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN3);  // D2.3A
            // Set pause, so this does not execute constantly
            D2IsPaused = true;
        }

        // Simple PWM: 3:8 ratio.
        _delay_cycles(1500);
        GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN4); // D1.4A
        GPIO_setOutputLowOnPin(GPIO_PORT_P1,GPIO_PIN5); // D1.3A
        GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN0); // D2.4A
        GPIO_setOutputLowOnPin(GPIO_PORT_P4,GPIO_PIN3); // D2.3A
        _delay_cycles(4000);

    }

    return (0);
}
```

```
// ADC interrupt handler - reads ADC results, stores them in array "results"
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=ADC12_VECTOR
__interrupt
#elif defined(__GNUC__)
__attribute__((interrupt(ADC12_VECTOR)))
#endif
void ADC12ISR(void)
{
    switch(__even_in_range(ADC12IV,34))
    {
        case  0: break;       //Vector  0:  No interrupt
        case  2: break;       //Vector  2:  ADC overflow
        case  4: break;       //Vector  4:  ADC timing overflow
        case  6: break;       //Vector  6:  ADC12IFG0
        case  8: break;       //Vector  8:  ADC12IFG1
        case 10: break;       //Vector 10:  ADC12IFG2
        case 12:              //Vector 12:  ADC12IFG3
            //Move results, IFG is cleared
            results[0] =
            ADC12_A_getResults(ADC12_A_BASE,
                               ADC12_A_MEMORY_0);
            //Move results, IFG is cleared
            results[1] =
            ADC12_A_getResults(ADC12_A_BASE,
                               ADC12_A_MEMORY_1);
            //Move results, IFG is cleared
            results[2] =
            ADC12_A_getResults(ADC12_A_BASE,
                               ADC12_A_MEMORY_2);
            //Move results, IFG is cleared
            results[3] =
            ADC12_A_getResults(ADC12_A_BASE,
                               ADC12_A_MEMORY_3);
            //Exit active CPU, SET BREAKPOINT HERE
            __bic_SR_register_on_exit(LPM4_bits);

        case 14: break;       //Vector 14:  ADC12IFG4
        case 16: break;       //Vector 16:  ADC12IFG5
        case 18: break;       //Vector 18:  ADC12IFG6
        case 20: break;       //Vector 20:  ADC12IFG7
        case 22: break;       //Vector 22:  ADC12IFG8
        case 24: break;       //Vector 24:  ADC12IFG9
        case 26: break;       //Vector 26:  ADC12IFG10
        case 28: break;       //Vector 28:  ADC12IFG11
        case 30: break;       //Vector 30:  ADC12IFG12
        case 32: break;       //Vector 32:  ADC12IFG13
        case 34: break;       //Vector 34:  ADC12IFG14
        default: break;
    }
}
```

BIBLIOGRAPHY

[1] R. J. Burgess, *The History of Music Production*, 1st ed. New York, NY, USA: OUP, 2014, p. 30

[2] AMS Neve. Capricorn. Available: http://ams-neve.com/products/legacy-products/capricorn. Accessed April, 29, 2015.

[3] Texas Instruments, "SN754410 Quadruple Half-H Driver," Rep. no. SN754410, Datasheet., 2015. Accessed on Apr. 29 2015.