CSUN® | COLLEGE OF ENGINEERING AND COMPUTER SCIENCE

ECE 425

Microprocessor Systems

Final Project
TivaWAV

Adam Espinoza
Daniel Celnik
Instructor: Aaron Nanas
Spring 2025

INTRODUCTION

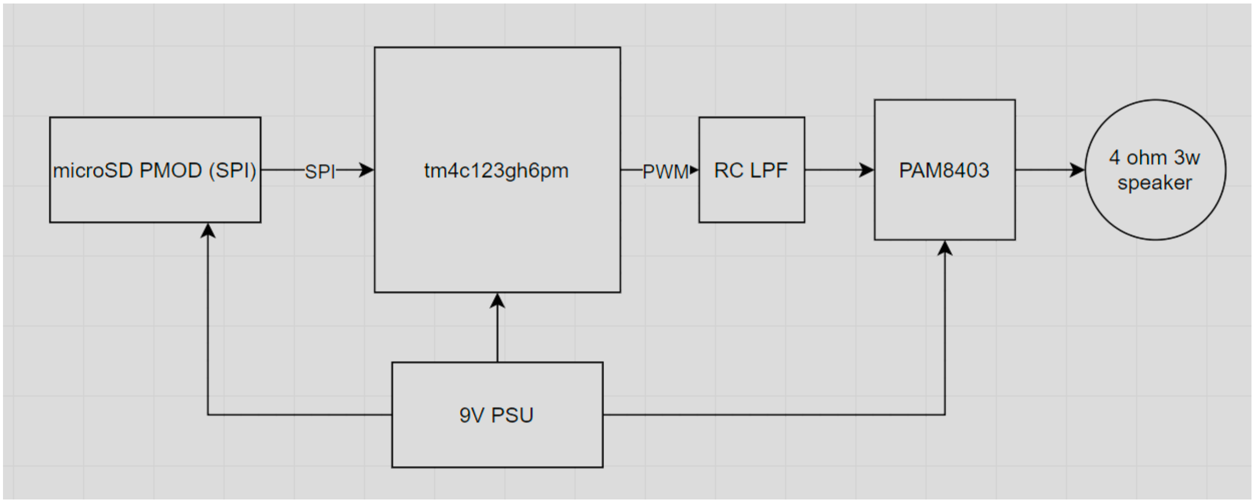
This project explores the design and implementation of a digital-to-analog audio output system using pulse-width modulation (PWM) on the Tiva C Series TM4C123 microcontroller. The goal was to read audio data from an SD card reader via serial peripheral interface (SPI) communication protocol and reconstruct an analog waveform capable of driving a speaker through the use of PWM, a low-pass filter, and an amplifier. By leveraging embedded C programming, hardware timer modules, and signal conditioning techniques, this system simulates a basic digital-to-analog converter (DAC) without relying on dedicated audio hardware. The project serves as a hands-on application of digital signal processing concepts in embedded systems, soldering, integrating software, electronics, and real-time control.

BACKGROUND AND METHODOLOGY

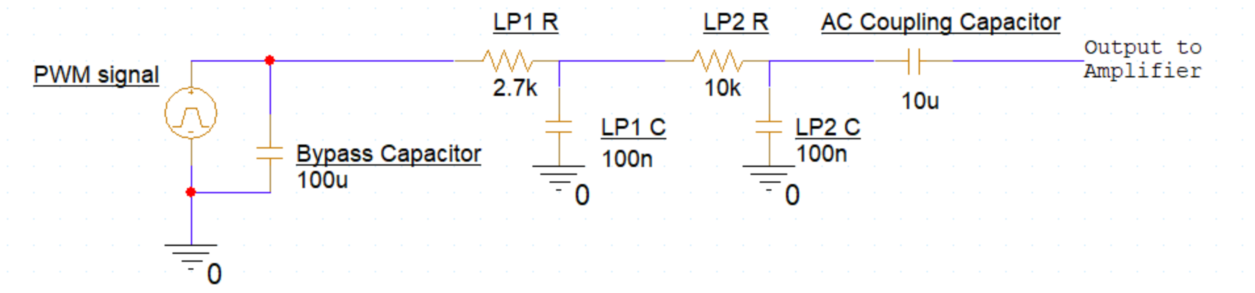
The realization of this project requires background knowledge of embedded systems concepts such as pulse width modulation (PWM) implementation, serial peripheral interface (SPI) and serial synchronous interface (SSI) communication, digital-to-analog conversion (DAC), timer peripheral initialization, low-pass filtering, fixed-point arithmetic and scaling techniques, and general purpose input/output (GPIO) configuration. By implementing all these required techniques, we aim to deepen our understanding of signal processing and strengthen our circuit design and soldering skills for embedded systems applications.

To begin, we load 8-bit PCM data onto an SD card using Linux. We then initialize the SSI module on the Tiva MCU to begin the data transfer from the PMOD SD card reader to the MCU on a sector-by-sector basis. From there, the MCU will load 512 bytes of data from the SD card into a local buffer. This is because the natural sector size of an SD card is 512 bytes, all 512 bytes must be read or the SD card could enter an error state. Since there is only 1 byte of PCM data, we do not need to worry about proper framing and syncing, as long as the sector counter is incrementing properly the audio will be in sync. The data is sized properly for the PWM 2 byte register, and then a Systick delay is utilized to allow the PWM module to drive out the new value for the appropriate sample rate time.

Once the PCM data has been successfully imported, we initialize the PWM module 1 generator 0 on the MCU and adjust the duty cycle of the PWM wave according to the raw value of the PCM data. We also configured the MCU's UART0 module to serve as a serial command-line interface, allowing the user to start and stop audio playback via USB communication from a connected PC. Once the PCM data has been encoded, the PWM signal propagates to the low-pass filter. The digital PWM signal is converted into an analog waveform by continuously varying the duty cycle over time; this changing duty cycle, when passed through the low-pass filter, results in a smoothed analog signal. The signal then passes through the class D amplifier, boosting the analog signal's strength and enabling it to drive the speaker. This all occurs at a sample rate of 44.1 kHz, which is fast enough to accurately reconstruct audio signals up to 22.05 kHz, satisfying the Nyquist criterion for full-range human hearing.



Regarding the low-pass filter's design, we first started designing with a specific cutoff frequency in mind, however in practice then output had too much noise in the form of clicking. We then opted to add a second stage for a steeper cutoff, which worked perfectly for our application. We used a 100nF capacitor for both stages and used trial-and-error to select the proper resistor values, with 2.7k and 10k respectively offering the best quality. A PSPICE schematic of the circuit can be found below:



BLOCK DIAGRAM

COMPONENTS USED

Component Name	Subcomponent/Datasheet Link if Applicable	Quantity
Low-Pass Filter	2.7 kΩ Resistor 10 kΩ Resistor 100 nf Capacitor	x 1 x 1 x 2
Coupling/Bypass Capacitors	10 uF Capacitor 100 uF Capacitor	x 1 x 1

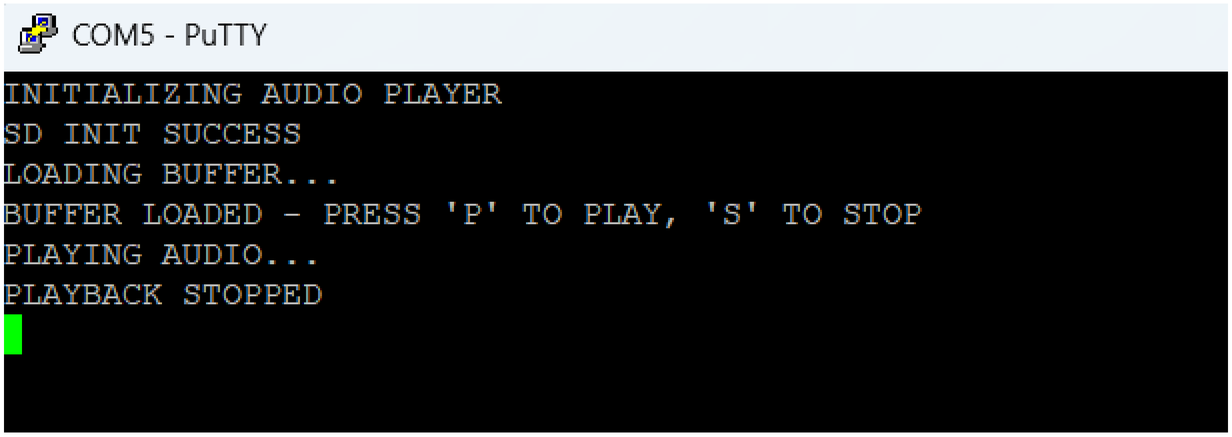
PAM8403 Class D Amp.	https://www.mouser.com/datasheet/2/115/PAM8403-247318.pdf?srsltid=AfmBOopRJu9x9uj2fy9vZLFt8JrCqvba7m6HYr39HYpEDCSr1NX7WDlU	x 1
4 Ohm 3W Speaker	https://www.aliexpress.us/item/3256808081586003.html?spm=a2g0o.productlist.main.25.19d728f8d0Ig2t&algo_pvid=f0a517ad-4b48-4095-ac77-03fdf754640f&algo_exp_id=f0a517ad-4b48-4095-ac77-03fdf754640f-12&pdp_ext_f=%7B%22order%22%3A%2260%22%2C%22eval%22%3A%221%22%7D&pdp_npi=4%40dis%21USD%216.43%216.23%21%21%2146.71%2145.26%21%402103146f17454296823325412ecfb4%2112000044426988632%21sea%21US%210%21ABX&curPageLogUid=xX0TN8CBGAXY&utparam-url=scene%3Asearch%7Cquery_from%3A#nav-specification	x 1
Tiva 4C123GH6PM		x 1
PMOD MicroSD Card Slot		x 1
PSU		x 1

PINOUT USED

SSI2Clk	PB4 (2)	SSI module 2 clock
SSI2Fss	PB5 (2)	SSI module 2 frame signal
SSI2Rx	PB6 (2)	SSI module 2 receive
SSI2Tx	PB7 (2)	SSI module 2 transmit
M1PWM0	PD0 (5)	Module 1 PWM Generator 0

ANALYSIS AND RESULTS

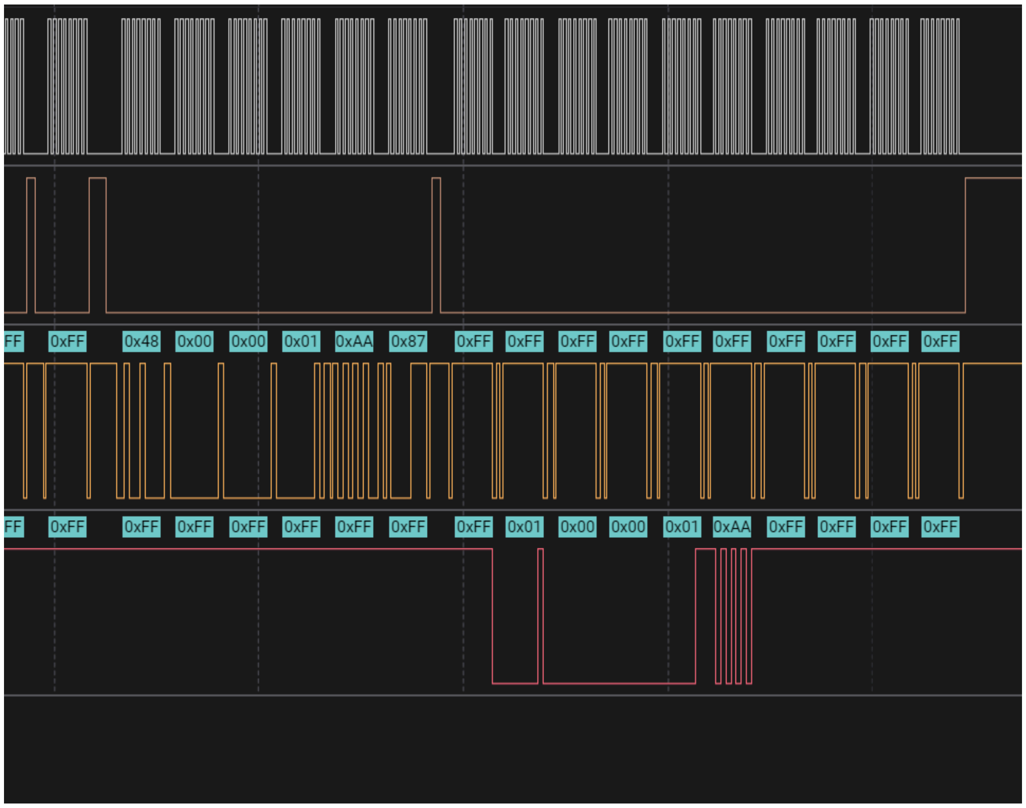
Regarding challenges faced, the low-pass filter and circuit design was the most time consuming aspect of this project due to the amount of trial-and-error involved with selecting resistor and capacitor values. We opted for a 2-stage low-pass filter to improve audio quality and two DC coupling capacitors to help isolate and bias the PAM8403.
On successful compiling of our C code, the following can be observed on the UART0 terminal:



COM5 - PuTTY

```
INITIALIZING AUDIO PLAYER
SD INIT SUCCESS
LOADING BUFFER...
BUFFER LOADED - PRESS 'P' TO PLAY, 'S' TO STOP
PLAYING AUDIO...
PLAYBACK STOPPED
```

Pressing 'P' causes the PLAYING AUDIO… response with the current values on the SD card being read and driven into the PWM system. The SD card system can be observed using a logic analyzer, and below is a screenshot of the initialization phase



Below is a link to a demonstration video of our completed project:
https://youtu.be/y95SkHDTvMQ

CONCLUSION
This project successfully demonstrated a low-cost, software-driven approach to audio playback using PWM on an embedded microcontroller. By leveraging the TM4C123's onboard peripherals, we implemented a basic digital-to-analog conversion system without the need for a dedicated DAC. 8-bit PCM data for the Game of Thrones main theme was sent to the microcontroller via SPI, encoded into PWM, smoothed using a passive low-pass filter, and finally sent through a coupling capacitor to an amplifier for output through a speaker. In addition, a UART-based command-line interface allowed real-time control of audio playback over USB, enhancing the system's usability during development and testing. Through this project, we gained valuable experience in embedded signal processing, hardware interfacing, low-level timing, and analog circuit behavior — all essential skills for real-world embedded systems design.

APPENDIX
Appendix A - main.c

```
// Main Code for tivaWAV
// Written by Daniel Celnik and Adam Espinoza

#include <stdint.h>
#include <stdbool.h>
#include "TM4C123GH6PM.h"
#include "SSI0.h"
#include "SysTick_Delay.h"
#include "UART0.h"
#include "PWM1_0.h"

#define BUFFER_SIZE    16384          // 32 sectors of 512 bytes
#define PWM_PERIOD     1133           // ~44.1 kHz PWM at 50 MHz
#define START_SECTOR   0
#define MARKER_BYTE    0x53

uint8_t buffer[BUFFER_SIZE];
volatile bool playing = false;

// Non-blocking UART character check
char UART0_CheckForCharacter(void) {
   // Check if there's data in the receive FIFO
   if((UART0->FR & UART0_RECEIVE_FIFO_EMPTY_BIT_MASK) == 0) {
       // Return the received character
       return (char)(UART0->DR & 0xFF);
   }
   return 0; // Return 0 if no character available
}

int main(void) {
   // Init hardware
   SSI0_Init();              // SPI for SD
   SysTick_Delay_Init();     // Microsecond delay
   UART0_Init();             // Debug UART
   UART0_Clear_Terminal();
   PWM1_0_Init(PWM_PERIOD, PWM_PERIOD / 2);  // 50% duty to start

   UART0_Output_String("INITIALIZING AUDIO PLAYER\r\n");


   UART0_Output_String("SD INIT SUCCESS\r\n");
   SD_SetHighSpeed();

   uint32_t current_sector = START_SECTOR;

   UART0_Output_String("LOADING BUFFER...\r\n");

   // Fill buffer initially
   for (int i = 0; i < BUFFER_SIZE / 512; i++) {
       SD_ReadSector(current_sector++, &buffer[i * 512]);
   }

   UART0_Output_String("BUFFER LOADED - PRESS 'P' TO PLAY, 'S' TO STOP\r\n");

   // Main playback loop
   while (1) {
       // Check for key presses
       char input = UART0_CheckForCharacter();

       if (input == 'P' || input == 'p') {
           if (!playing) {
               playing = true;
               UART0_Output_String("PLAYING AUDIO...\r\n");
           }
       }
       else if (input == 'S' || input == 's') {
```

```
        if (playing) {
            playing = false;
            // Silence output when stopped
            PWM1_0_Update_Duty_Cycle(PWM_PERIOD / 2);
            UART0_Output_String("PLAYBACK STOPPED\r\n");
        }
    }

    // Only process audio if in playing state
    if (playing) {
        for (int sector = 0; sector < BUFFER_SIZE / 512; sector++) {
            uint8_t* sector_ptr = &buffer[sector * 512];

            // Process all samples in this sector
            for (int i = 0; i < 512 - 1; i++) {
                // Check for key presses during playback
                char key = UART0_CheckForCharacter();
                if (key == 'S' || key == 's') {
                    playing = false;
                    PWM1_0_Update_Duty_Cycle(PWM_PERIOD / 2);
                    UART0_Output_String("PLAYBACK STOPPED\r\n");
                    break;
                }

                if (sector_ptr[i] == MARKER_BYTE) {
                    uint8_t sample = sector_ptr[i + 1];
                    // Convert unsigned 8-bit (0–255) to 16-bit biased range (0–65535)
                    uint16_t biased = ((uint16_t)sample) << 8;  // scale to full range
                    uint16_t duty = (biased * PWM_PERIOD) >> 18;
                    PWM1_0_Update_Duty_Cycle(duty);
                    SysTick_Delay1us(21);
                    i++;  // skip sample byte
                }
            }

            // If playback was stopped, break out of sector loop
            if (!playing) {
                break;
            }

            // Load next sector
            SD_ReadSector(current_sector++, sector_ptr);
        }
        }
        else {
            SysTick_Delay1ms(10);
        }
    }
}
```

SSI0.C

```
/**
 * @file SSI0.c
 *
 * @brief Source code for the SSI0 driver (for SD card SPI).
 *
 * @note Uses 400 kHz for initialization and switches to 10 MHz afterward.
 */

#include "SSI0.h"
#include <stdint.h>
#include <stdbool.h>
#include "TM4C123GH6PM.h"
#include "SysTick_Delay.h"

// --- Manual CS Control ---
#define CS_PIN  (1 << 3)
void SPI_CS_Low(void)  { GPIOA->DATA &= ~CS_PIN; }
void SPI_CS_High(void) { GPIOA->DATA |= CS_PIN; }

// --- SSI Status Register bits ---
#define SSI_SR_TFE  0x01
#define SSI_SR_TNF  0x02
```

```
#define SSI_SR_RNE  0x04

#define SSI_SR_RFF  0x08

#define SSI_SR_BSY  0x10


// --- 400 kHz at startup ---

void SSI0_Init(void)

{

    SYSCTL->RCGCSSI |= 0x01;

    SYSCTL->RCGCGPIO |= 0x01;

    while ((SYSCTL->PRGPIO & 0x01) == 0) {}


    GPIOA->AFSEL |= (1 << 2) | (1 << 4) | (1 << 5);

    GPIOA->PCTL &= ~((0xF << 8) | (0xF << 16) | (0xF << 20));

    GPIOA->PCTL |= (0x2 << 8) | (0x2 << 16) | (0x2 << 20);

    GPIOA->DEN |= 0x3C;


    GPIOA->AFSEL &= ~(1 << 3);

    GPIOA->DIR |= (1 << 3);

    GPIOA->DATA |= (1 << 3);


    SSI0->CR1 &= ~0x02;

    SSI0->CR1 &= ~0x01;

    SSI0->CR1 &= ~0x04;

    SSI0->CC = 0x0;


    // Set to 400 kHz: 50 MHz / (125 * 1)

    SSI0->CPSR = 125;

    SSI0->CR0 = 0x07; // 8-bit, mode 0
    SSI0->CR1 |= 0x02;
}

// --- Switch to 10 MHz after init ---
void SD_SetHighSpeed(void) {
    SSI0->CR1 &= ~0x02;     // Disable SSI during configuration
    SSI0->CPSR = 2;         // 50 MHz / 4 = 12.5 MHz
    SSI0->CR0 = 0x07;       // 8-bit data, SPI mode 0
    SSI0->CR1 |= 0x02;      // Enable SSI
}

// --- SPI byte transfer ---
uint8_t SPI_Transfer(uint8_t byte) {
    while ((SSI0->SR & SSI_SR_TNF) == 0);
    SSI0->DR = byte;
    while ((SSI0->SR & SSI_SR_RNE) == 0);
    return SSI0->DR & 0xFF;
}

// --- SPI burst transfer ---
void SPI_TransferBurst(const uint8_t *tx, uint8_t *rx, int len) {
                            SPI_CS_High();
                            SPI_CS_Low();
    for (int i = 0; i < len; ++i) {
        uint8_t out = tx ? tx[i] : 0xFF;
        rx[i] = SPI_Transfer(out);
    }
                            SPI_CS_High();
```

```
}

// --- Wait for R1 response ---
uint8_t SD_WaitForR1(void) {
   for (int i = 0; i < 10; i++) {
      uint8_t r1 = SPI_Transfer(0xFF);
      if ((r1 == 0x01)) return r1;
   }
   return 0xFF;
}

uint8_t SD_WaitForResponse(void) {
   SPI_CS_High();
           //          SysTick_Delay1us(1);
                       SPI_CS_Low();
           //          SysTick_Delay1us(1);
             uint8_t response = SPI_Transfer(0xFF);
   SPI_CS_High();
           //          SysTick_Delay1us(1);
   return response;
}

// --- Send command to SD card ---
void SD_SendCommand(uint8_t cmd, uint32_t arg, uint8_t crc) {

   uint8_t packet[6];
   packet[0] = 0x40 | cmd;
   packet[1] = (arg >> 24) & 0xFF;
   packet[2] = (arg >> 16) & 0xFF;
```

```
    packet[3] = (arg >> 8) & 0xFF;
    packet[4] = arg & 0xFF;
    packet[5] = crc;

    SPI_TransferBurst(packet, (uint8_t *)0, 6);

}

uint8_t SD_SendCommand0(uint8_t cmd, uint32_t arg, uint8_t crc) {
    uint8_t packet[6];
    packet[0] = 0x40 | cmd;
    packet[1] = (arg >> 24) & 0xFF;
    packet[2] = (arg >> 16) & 0xFF;
    packet[3] = (arg >> 8) & 0xFF;
    packet[4] = arg & 0xFF;
    packet[5] = crc;
                        uint8_t r1  = 0x00;

                        while (r1 != 0x01) {
                        SPI_CS_Low();
    SPI_TransferBurst(packet, (uint8_t *)0, 6);
            SPI_CS_High();
//                      SysTick_Delay1us(1);
                        SPI_CS_Low();
    r1 = SD_WaitForR1();
            SPI_CS_High();
          //            SysTick_Delay1us(1);
                        }
    for (int i = 0; i < 8; i++) {
```

```
                SPI_CS_Low();
            //          SysTick_Delay1us(1);
                        SPI_Transfer(0xFF);
                        SPI_CS_High();
            //          SysTick_Delay1us(1);

                        }

    return r1;
}




// --- SD card initialization ---
bool SD_Initialize(void) {
    // Send 80+ clocks (10 bytes) with CS high
    SPI_CS_High();
                         bool done = false;
                        uint8_t reply = 0;
    for (int i = 0; i < 500; i++) SPI_Transfer(0xFF);

                        SD_SendCommand(0,0x00000000,0x95);
                        for (int i = 0; i < 12 ; i++) {
                                    reply = SD_WaitForResponse();
                                    if (reply == 0xC1) {
                                    done = false;
                                    return done; }
                        }
                        SD_SendCommand(8,0x000001AA,0x87);
```

```
                    for (int i = 0; i < 12 ; i++) {
                              reply = SD_WaitForResponse();
                              if (reply == 0xC1) {
                                        done = false;
                                        return done; }
                    }
                              for (int i = 0; i < 1000 ; i++) {
                              SD_SendCommand(55,0x00000000,65);

                                        for (int j = 0 ; j < 8; j++) {
                                        reply = SD_WaitForResponse();
                                        if (reply == 0x00) done = true;
                                                  }

                                SD_SendCommand(41,0x40000000,0x77);

                                        for (int j = 0 ; j < 8; j++) {
                                        reply = SD_WaitForResponse();
                                        if (reply == 0x00) done = true;
                                                  }
                                        if (done == true) break;

                              }

          return done;

}

bool SD_ReadSector(uint32_t lba, uint8_t *buffer)
```

```
{
    // CMD17: Read single block (0x11)
    SD_SendCommand(17, lba, 0xFF);

    // Wait for R1 response (should be 0x00 for success)
    for (int i = 0; i < 10; i++) {
        uint8_t r1 = SD_WaitForResponse();
        if (r1 == 0x00) break;
        if (i == 9) return false; // timeout
    }

    // Wait for data token (0xFE)
    for (int i = 0; i < 1000; i++) {
        uint8_t token = SD_WaitForResponse();
        if (token == 0xFE) break;
        if (i == 999) return false; // timeout
    }

    // Read 512 bytes
    for (int i = 0; i < 512; i++) {
        buffer[i] = SD_WaitForResponse();
    }

    // Allow CRC to clear
    for (int i = 0; i < 10; i++) {
        SD_WaitForResponse();
    }
    return true;
}
```

UART0.C

```
/**
 * @file UART0.c
 *
 * @brief Source code for the UART0 driver.
 *
 * This file contains the function definitions for the UART0 driver.
 *
 * @note For more information regarding the UART module, refer to the
 * Universal Asynchronous Receivers / Transmitters (UARTs) section
 * of the TM4C123GH6PM Microcontroller Datasheet.
 * Link: https://www.ti.com/lit/gpn/TM4C123GH6PM
 *
 * @note Assumes that the system clock (50 MHz) is used.
 *
 * @author Aaron Nanas
 */

#include "UART0.h"

void UART0_Init(void)
{
	// Enable the clock to the UART0 module by setting the
	// R0 bit (Bit 0) in the RCGCUART register
	SYSCTL->RCGCUART |= 0x01;

	// Enable the clock to Port A by setting the
	// R0 bit (Bit 0) in the RCGCGPIO register
```

```
    SYSCTL->RCGCGPIO |= 0x01;

    // Disable the UART0 module before configuration by clearing
    // the UARTEN bit (Bit 0) in the CTL register
    UART0->CTL &= ~0x0001;

    // Configure the UART0 module to use the system clock (50 MHz)
    // divided by 16 by clearing the HSE bit (Bit 5) in the CTL register
    UART0->CTL &= ~0x0020;

    // Set the baud rate by writing to the DIVINT field (Bits 15 to 0)
    // and the DIVFRAC field (Bits 5 to 0) in the IBRD and FBRD registers, respectively.
    // The integer part of the calculated constant will be written to the IBRD register,
    // while the fractional part will be written to the FBRD register.
    // BRD = (System Clock Frequency) / (16 * Baud Rate)
    // BRDI = (50,000,000) / (16 * 115200) = 27.12673611 (IBRD = 27)
    // BRDF = ((0.12673611 * 64) + 0.5) = 8.611 (FBRD = 8)
    UART0->IBRD = 27;
    UART0->FBRD = 8;

    // Configure the data word length of the UART packet to be 8 bits by
    // writing a value of 0x3 to the WLEN field (Bits 6 to 5) in the LCRH register
    UART0->LCRH |= 0x60;

    // Enable the Transmit FIFO and the Receive FIFO by setting the FEN bit (Bit 4) in the LCRH
register
    UART0->LCRH |= 0x10;

    // Select one stop bit to be transmitted at the end of a UART frame by
    // clearing the STP2 bit (Bit 3) in the LCRH register
    UART0->LCRH &= ~0x08;

    // Disable the parity bit by clearing the PEN bit (Bit 1) in the LCRH register
    UART0->LCRH &= ~0x02;

    // Enable the UART0 module after configuration by setting
    // the UARTEN bit (Bit 0) in the CTL register
    UART0->CTL |= 0x01;

    // Configure the PA1 (U0TX) and PA0 (U0RX) pins to use the alternate function
    // by setting Bits 1 to 0 in the AFSEL register
    GPIOA->AFSEL |= 0x03;

    // Clear the PMC1 (Bits 7 to 4) and PMC0 (Bits 3 to 0) fields in the PCTL register before
configuration
    GPIOA->PCTL &= ~0x000000FF;

    // Configure the PA1 pin to operate as a U0TX pin by writing 0x1 to the
    // PMC1 field (Bits 7 to 4) in the PCTL register
    // The 0x1 value is derived from Table 23-5 in the TM4C123G Microcontroller Datasheet
    GPIOA->PCTL |= 0x00000010;

    // Configure the PA0 pin to operate as a U0RX pin by writing 0x1 to the
    // PMC0 field (Bits 3 to 0) in the PCTL register
    // The 0x1 value is derived from Table 23-5 in the TM4C123G Microcontroller Datasheet
    GPIOA->PCTL |= 0x00000001;

    // Enable the digital functionality for the PA1 and PA0 pins
```

```
	// by setting Bits 1 to 0 in the DEN register
	GPIOA->DEN |= 0x03;
}

char UART0_Input_Character(void)
{
	while((UART0->FR & UART0_RECEIVE_FIFO_EMPTY_BIT_MASK) != 0);

	return (char)(UART0->DR & 0xFF);
}

void UART0_Output_Character(char data)
{
	while((UART0->FR & UART0_TRANSMIT_FIFO_FULL_BIT_MASK) != 0);

	UART0->DR = data;
}

void UART0_Input_String(char *buffer_pointer, uint16_t buffer_size)
{
	int length = 0;
	char character = UART0_Input_Character();

	while(character != UART0_CR)
	{
		if (character == UART0_BS)
		{
			if (length)
			{
```

```
                    buffer_pointer--;
                    length--;
                    UART0_Output_Character(UART0_BS);
                }
            }
            else if(length < buffer_size)
            {
                *buffer_pointer = character;
                buffer_pointer++;
                length++;
                UART0_Output_Character(character);
            }
            character = UART0_Input_Character();
        }
        *buffer_pointer = 0;
}

void UART0_Output_String(char *pt)
{
        while(*pt)
        {
            UART0_Output_Character(*pt);
            pt++;
        }
}

uint32_t UART0_Input_Unsigned_Decimal(void)
{
        uint32_t number = 0;
```

```
	uint32_t length = 0;
char character = UART0_Input_Character();

	// Accepts until <enter> is typed
	// The next line checks that the input is a digit, 0-9.
	// If the character is not 0-9, it is ignored and not echoed
while(character != UART0_CR)
	{
		if ((character >= '0') && (character <= '9'))
		{
			// "number" will overflow if it is above 4,294,967,295
			number = (10 * number) + (character - '0');
			length++;
			UART0_Output_Character(character);
		}

		// If the input is a backspace, then the return number is
		// changed and a backspace is outputted to the screen
		else if ((character == UART0_BS) && length)
		{
			number /= 10;
			length--;
			UART0_Output_Character(character);
		}

		character = UART0_Input_Character();
	}

	return number;
```

```
}

void UART0_Output_Unsigned_Decimal(uint32_t n)
{
        // Use recursion to convert decimal number
        // of unspecified length as an ASCII string
  if (n >= 10)
        {
    UART0_Output_Unsigned_Decimal(n / 10);
    n = n % 10;
  }

        // n is between 0 and 9
  UART0_Output_Character(n + '0');
}

uint32_t UART0_Input_Unsigned_Hexadecimal(void)
{
        uint32_t number = 0;
        uint32_t digit = 0;
        uint32_t length = 0;
  char character = UART0_Input_Character();

        while(character != UART0_CR)
        {
                // Initialize digit and assume that the hexadecimal character is invalid
                digit = 0x10;

                if ((character >= '0') && (character <= '9'))
```

```
			{
				digit = character - '0';
			}
			else if ((character>='A') && (character <= 'F'))
			{
				digit = (character - 'A') + 0xA;
			}
			else if ((character >= 'a') && (character <= 'f'))
			{
				digit = (character - 'a') + 0xA;
			}

			// If the character is not 0-9 or A-F, it is ignored and not echoed
if (digit <= 0xF)
			{
  number = (number * 0x10) + digit;
  length++;
  UART0_Output_Character(character);
}

			// Backspace outputted and return value changed if a backspace is inputted
			else if((character == UART0_BS) && length)
			{
				number /= 0x10;
				length--;
				UART0_Output_Character(character);
			}

			character = UART0_Input_Character();
  }
  return number;
}

void UART0_Output_Unsigned_Hexadecimal(uint32_t number)
{
        // Use recursion to convert the number of
        // unspecified length as an ASCII string
        if (number >= 0x10)
        {
                UART0_Output_Unsigned_Hexadecimal(number / 0x10);
                UART0_Output_Unsigned_Hexadecimal(number % 0x10);
        }
        else
        {
                if (number < 0xA)
                {
                        UART0_Output_Character(number + '0');
                }
                else
                {
                        UART0_Output_Character((number - 0x0A) + 'A');
                }
        }
}

void UART0_Output_Newline(void)
{
        UART0_Output_Character(UART0_CR);
        UART0_Output_Character(UART0_LF);
}

void UART0_Clear_Terminal(void)
{
  UART0_Output_String("\033[2J\033[H");
}
```

PWM0.c

```
/**
 * @file PWM1_0.c
 *
 * @brief Source file for the PWM1_0 driver.
 *
 * This file contains the function definitions for the PWM1_0 driver.
 * It uses the Module 1 PWM Generator 0 to generate a PWM signal using the PD0 pin.
 *
 * @note This driver assumes that the system clock's frequency is 50 MHz.
 *
 * @note This driver assumes that the PWM_Clock_Init function has been called
 * before calling the PWM1_0_Init function.
 *
 * @author Aaron Nanas
 */

#include "PWM1_0.h"

void PWM1_0_Init(uint16_t period_constant, uint16_t duty_cycle)
{
	// Return from the function if the specified duty_cycle is greater than
	// or equal to the given period. The duty cycle cannot exceed 99%.
	if (duty_cycle >= period_constant) return;
//need to figure out parameters^^^^

	// Enable the clock to PWM Module 1 by setting the
	// R1 bit (Bit 1) in the RCGCPWM register
	SYSCTL->RCGCPWM |= 0x02;

	// Enable the clock to GPIO Port D by setting the
	// R1 bit (Bit 1) in the RCGCGPIO register
	SYSCTL->RCGCGPIO |= 0x08;

	// Configure the PD0 pin to use the alternate function (M1PWM0)
	// by setting Bit 0 in the AFSEL register
	GPIOD->AFSEL |= 0x01;

	// Clear the PMC0 field (Bits 3 to 0) in the PCTL register
	GPIOD->PCTL &= ~0x0000000F;

	// Configure the PD0 pin to operate as a Module 1 PWM0 pin (M1PWM0)
	// by writing 0x5 to the PMC0 field (Bits 3 to 0) in the PCTL register
	// The 0x5 value is derived from Table 23-5 in the TM4C123G Microcontroller Datasheet
	GPIOD->PCTL |= 0x00000005;

	// Enable the digital functionality for the PD0 pin
	// by setting Bit 0 in the DEN register
	GPIOD->DEN |= 0x01;
```

```
// Disable the Module 1 PWM Generator 0 block (PWM1_0) before
// configuration by clearing the ENABLE bit (Bit 0) in the PWM0CTL register
PWM1->_0_CTL &= ~0x01;

// Configure the counter for the PWM1_0 block to
// use Count-Down mode by clearing the MODE bit (Bit 1)
// in the PWM0CTL register. The counter will count from the load value
// to 0, and then wrap back to the load value
PWM1->_0_CTL &= ~0x02;

// Set the ACTCMPAD field (Bits 7 to 6) to 0x3 in the PWM0GENA register
// to drive the PWM signal high when the counter matches
// the comparator (i.e. the value in PWM0CMPA) while counting down
PWM1->_0_GENA |= 0xC0;

// Set the ACTLOAD field (Bits 3 to 2) to 0x2 in the PWM0GENA register
// to drive the PWM signal low when the counter matches the value
// in the PWM0LOAD register
PWM1->_0_GENA |= 0x08;

// Set the period by writing to the LOAD field (Bits 15 to 0)
// in the PWM0LOAD register. This determines the number of clock
// cycles needed to count down to zero
PWM1->_0_LOAD = (period_constant - 1);

// Set the duty cycle by writing to the COMPA field (Bits 15 to 0)
// in the PWM0CMPA register. When the counter matches the value in this register,
// the PWM signal will be driven high
PWM1->_0_CMPA = (duty_cycle - 1);
```

```
    // Enable the PWM1_0 block after configuration by setting the
    // ENABLE bit (Bit 0) in the PWM0CTL register
    PWM1->_0_CTL |= 0x01;

    // Enable the PWM1_0 signal to be passed to the PD0 pin (M1PWM0)
    // by setting the PWM0EN bit (Bit 0) in the PWMENABLE register
    PWM1->ENABLE |= 0x01;
}

void PWM1_0_Update_Duty_Cycle(uint16_t duty_cycle)
{
    // Set the duty cycle by writing to the COMPA field (Bits 15 to 0)
    // in the PWM0CMPA register. When the counter matches the value in this register,
    // the PWM signal will be driven high
    PWM1->_0_CMPA = (duty_cycle - 1);
}
```