Console Header

```h
#ifndef MEMCONSOLE_IS_ALIVE
#define MEMCONSOLE_IS_ALIVE 1

void memdmpDefault(uint8_t* const addr);
void memdmp(uint8_t* const addr, uint32_t len);
void memwrd(const uint32_t* addr);
void wmemwrd(uint32_t* dest, uint32_t contents);
int parseCommand(const char* input);
uint32_t* parseAddress(const char* input);
uint32_t parseArgument(const char* input);
void help(void);

#endif
```

Console API

```c
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#define DEFAULT_SIZE 16

/**
 * memdmpDefault
 * Called when the 'dm' command is called without a specified
 * length. Defaults to 16 bytes to print, prints in hex.
 *
 * Syntax: dm [address]
 *
 * Address must be hexadecimal, prefaced with '0x' and the letters
 * must be caps.
 */
void memdmpDefault(uint8_t* const addr) {
	// Empty string containing memory dump
	char output[100] = "";

	// Add hex dump
	for(int i = 0; i < DEFAULT_SIZE; i++) {
		uint8_t* target = addr + i;
		//delay_us(MEM_DELAY);
		uint8_t contents = *target;
		//delay_us(MEM_DELAY);
		char temp[10] = "";
		sprintf(temp, "%X", contents);
		strcat(temp, " ");
		strcat(output, temp);
	}

	// Print the formatted result
	printf("0x%X: %s\n\r", (unsigned int)addr, output);

	return;
}
```

```
/**
 * memdmp
 * Called when the 'dm' command is called with a specified length.
 * Prints in hex.
 *
 * Syntax: dm [address] [length]
 *
 * Address must be hexadecimal, prefaced with '0x' and the letters
 * must be caps.
 */
void memdmp(uint8_t* const addr, uint32_t len) {
	char output[100] = "";
	unsigned int newlines = 0;
	uint8_t* newaddr = 0;
	uint8_t contents = 0;

	// Hex dump
	for(int i = 0; i < len; i++) {
		newaddr = addr + i;
		contents = *newaddr;
		// New line every 16 bytes, clear output buffer after printing
		if((i%16 == 0) && i > 0) {
			printf("0x%X: %s\n\r", (unsigned int)newaddr-16, output);
			memset(output, 0, strlen(output));
			newlines++;

			char temp[10] = "";
			sprintf(temp, "%X", contents);

			// Append a space
			strcat(temp, " ");

			// Append contents to output
			strcat(output, temp);
		} else {
			// Current contents string
			char temp[10] = "";
			sprintf(temp, "%X", contents);

			// Append a space
			strcat(temp, " ");

			// Append contents to output
			strcat(output, temp);
		}
	}

	// If output is not empty, print it
	if(strnlen((const char*)output, 99) > 1) {
		unsigned int actualAddr = (unsigned int)addr + (newlines * 16);
		printf("0x%X: %s\n\r", actualAddr, output);
	}

	return;
}
```

```c
/**
 * memwrd
 * Reads a 32-bit word from memory in the provided address.
 * Prints in hexadecimal.
 *
 * Syntax: rmw [address]
 *
 * Address must be hexadecimal and word aligned, prefaced with '0x'
 * and the letters must be caps.
 */
void memwrd(const uint32_t* addr) {
	// One word in our system is 32 bits, so word alignment is every 32 bits
	unsigned int contents = 0;

	if((unsigned int)addr%32 == 0) {
		contents = *addr;
		printf("0x%X: 0x%X %d\n\r", (unsigned int)addr, contents, contents);
	} else {
		// Print the address and the text "Bad alignment" if it breaks word boundaries
		printf("0x%X: Bad alignment\n\r", (unsigned int)addr);
	}

	return;
}
```

```c
/**
 * wmemwrd
 * writes a 32-bit word to the provided address.
 *
 * Syntax: wmw [address] [value]
 *
 * Address must be hexadecimal and word aligned, prefaced with '0x'
 * and the letters must be caps.
 */
void wmemwrd(uint32_t* const dest, uint32_t contents) {
	if((unsigned int)dest % 32 == 0) {
		// Write value
		*dest = contents;

		// Print new value
		memwrd(dest);
	} else {
		// Error message
		printf("0x%X: Bad alignment, nothing written\n\r", (unsigned int)dest);
	}
	return;
```

```
/**
 * parseCommand
 * Takes a string which should represent a command and
 * attempts to parse it.
 *
 * '?' = 0
 * 'dm' = 1
 * 'rmw' = 2
 * 'wmw' = 3
 * 'music' = 4
 * invalid = -1
 */
int parseCommand(const char* input) {
	// Default command, -1 = invalid command
	int command = -1;

	// Switch statement based on first char
	switch(input[0]) {
	// Help command, return 0
	case '?':
		command = 0;
		break;

	// Dump memory command, return 1
	case 'd':
		if(input[1] == 'm') {
			command = 1;
		}
		break;

	// Read word command, return 2
	case 'r':
		if(input[1] == 'm' && input[2] == 'w') {
			command = 2;
		}
		break;

	// Write word command, return 3
	case 'w':
		if(input[1] == 'm' && input[2] == 'w') {
			command = 3;
		}
		break;

	case 'm':
		if(input[1] == 'u' && input[2] == 's' && input[3] == 'i' && input[4] == 'c') {
			command = 4;
		}
	}

	return command;
}
```

```c
/**
 * help
 * Prints a bunch of lines to stdout to help
 * with syntaxes of the commands
 */
void help() {
	printf("NOTE: All commands are case-sensitive!\n\n");

	// WMW
	printf("command \'wmw\' - write memory word - writes a provided 32-bit value into the specified address in memory\n\r");
	printf("\tsyntax: wmw [address] [value]\n\r");
	printf("the provided address must be hexadecimal with capital letters and prefaced with \'0x\'\n\r");
	printf("the value to be written can be provided in either hex or decimal, default is decimal, unless a \'0x\' is found\n\n\r");

	// RMW
	printf("command \'rmw\' - read memory word - reads a provided address and outputs the unsigned contents of that address in both hex and decimal\n\r");
	printf("\tsyntax: rmw [address]\n\r");
	printf("the provided address must be hexadecimal with capital letters and prefaced with \'0x\'\n\n\r");

	// DM
	printf("command \'dm\' - dump memory - reads memory starting at the provided address for the provided length in bytes, outputs byte-sized hex values\n\r");
	printf("\tsyntax: dm [address] [OPT:length]\n\r");
	printf("\tif no length is specified, default is 16 bytes\n\r");
	printf("the provided address must be hexadecimal with capital letters and prefaced with \'0x\'\n\r");
	printf("the length can be provided in either hex or decimal, default is decimal, unless a \'0x\' is found\n\r");

	// music
	printf("command \'music\' - Plays a song\n");
	printf("\tsyntax: music [song]\n");
	printf("current songs as of lab 5: \n\t\'doom\' - At Doom's Gate\n\t\'zelda\' - Legend of Zelda Main Theme");
	return;
}

/**
 * parseAddress
 * Attempts to parse a string containing an address into that address.
 * Uses sscanf
 * Input must be hex and prefaced with '0x'
 * All hex chars after the preface must be caps
 */
uint32_t* parseAddress(const char* input) {
	// Variable to store parsed address
	unsigned int address = 0;

	// Parse the address
	sscanf(input, "0x%X", &address);

	// Return the address
	return (uint32_t*)address;
}
```

```c
/**
 * parseArgument
 * Takes a string and attempts to convert it to a number.
 * Input must be in decimal and unsigned.
 */
uint32_t parseArgument(const char* input) {
	// Default value if the input can't be parsed
	uint32_t parsed = -1;

	// Attempt to parse
	sscanf(input, "%lu", &parsed);

	// Return the parsed value
	return parsed;
}
```

Music Header

```c
#ifndef MUSIC_API_ALIVE
#define MUSIC_API_ALIVE 1
#include <stdint.h>
// Ok for future reference,
// length:
//		1 = 32nd note, shortest
//		2 = 16th note
//		3 = 8th note
//		4 = qtr note
//		5 = half note
//		6 = full note, longest
typedef struct{
	uint32_t period;
	uint32_t octave;
	uint32_t length;
} note;

void music_Init(void);
void music_Play(const note song[], int tempoScale);
void note_Play(uint32_t period, uint32_t duration);
```

```
// 0th octave values
// In period instead of frequencies
// microseconds
// Side note, all of these values will fit into a uint16
// to get other octaves, divide by 2 * octave
// aka LSR  octave
#define C    61162
#define Cs   57736
#define Db   57736
#define D    54495
#define Ds   51413
#define Eb   51413
#define E    48543
#define F    45808
#define Fs   43252
#define Gb   43252
#define G    40816
#define Gs   38520
#define Ab   38520
#define A    36363
#define As   34317
#define Bb   34317
#define B    32393

// Rip and tear until it is done
static const int doomTempo = 63300;
static const note atDoomsGate[] = {
             {F, 2, 3},
             {F, 2, 3},
             {C, 5, 3},
             {F, 2, 3},
             {F, 2, 3},
             {B, 4, 3},
             {F, 2, 3},
             {F, 2, 3},
             {A, 4, 3},
             {F, 2, 3},
             {F, 2, 3},
             {Fs, 4, 3},
             {F, 2, 3},
             {F, 2, 3},
             {G, 4, 3},
             {A, 4, 3},
             {F, 2, 3},
             {F, 2, 3},
             {C, 5, 3},
             {F, 2, 3},
             {F, 2, 3},
             {B, 4, 3},
             {F, 2, 3},
             {F, 2, 3},
             {G, 4, 3},
             {F, 2, 3},
             {F, 2, 3},
             {Fs, 4, 3},
             {Fs, 4, 5},

             {F, 2, 3},
             {F, 2, 3},
             {C, 5, 3},
             {F, 2, 3},
```

```
            {F, 2, 3},
            {B, 4, 3},
            {F, 2, 3},
            {F, 2, 3},
            {A, 4, 3},
            {F, 2, 3},
            {F, 2, 3},
            {Fs, 4, 3},
            {F, 2, 3},
            {F, 2, 3},
            {G, 4, 3},
            {A, 4, 3},
            {F, 2, 3},
            {F, 2, 3},
            {C, 5, 3},
            {F, 2, 3},
            {F, 2, 3},
            {B, 4, 3},
            {F, 2, 3},
            {F, 2, 3},
            {G, 4, 3},
            {F, 2, 3},
            {F, 2, 3},
            {Fs, 4, 3},
            {Fs, 4, 6},
            {0, 0, 0}
};

static const int zeldaTempo = 60000;
static const note zelda[] = {
            {A, 4, 5},
            {0, 1, 4},
            {A, 4, 3},
            {A, 4, 3},
            {A, 4, 3},
            {A, 4, 3},
            {A, 4, 4},
            {B, 4, 3},
            {A, 4, 4},
            {0, 1, 4},
            {A, 4, 3},
            {A, 4, 3},
            {A, 4, 3},
            {A, 4, 3},
            {A, 4, 4},
            {B, 4, 3},
            {A, 4, 4},
            {0, 1, 4},
            {A, 4, 3},
            {A, 4, 3},
            {A, 4, 3},
            {A, 4, 3},
            {B, 4, 3},
            {E, 4, 2},
            {E, 4, 2},
            {E, 4, 3},
            {E, 4, 2},
            {E, 4, 2},
            {E, 4, 3},
            {E, 4, 2},
            {E, 4, 2},
```

```
            {E, 4, 3},
            {E, 4, 3},
            {A, 4, 4},
            {E, 4, 4},
            {E, 4, 3},
            {A, 4, 3},
            {A, 4, 2},
            {B, 4, 2},
            {C, 5, 2},
            {D, 5, 2},
            {E, 5, 5},
            {0, 1, 3},
            {D, 5, 3},
            {E, 5, 3},
            {F, 5, 3},
            {G, 5, 3},
            {B, 6, 5},
            {D, 6, 3},
            {B, 6, 3},
            {G, 5, 3},
            {F, 5, 5},
            {G, 5, 2},
            {0, 1, 2},
            {F, 5, 2},
            {E, 5, 5},
            {E, 5, 4},
            {D, 5, 3},
            {D, 5, 2},
            {E, 5, 2},
            {F, 5, 5},
            {E, 5, 3},
            {D, 5, 3},
            {D, 5, 3},
            {C, 5, 2},
            {D, 5, 2},
            {E, 5, 5},
            {0, 0, 0}
};

#endif
```

Music API

```c
// Kinda recursive-y since that's the header for this but
// I need the anonymous struct for notes
#include "music.h"
#include "registers.h"
#include "delay.h"
#include <stdio.h>

// Actual method to do notes
void note_Play(uint32_t period, uint32_t duration) {
	// If period = 0, just do a delay

	if(period == 0) {
		delay_us(duration);
	} else {
		// Load period / 2 into CCR
		volatile uint32_t* target = TIM3_CCR;
		*target = (period >> 1);

		// Load period - 1 into ARR
		target = TIM3_ARR;
		*target = period - 1;

		// Start playing the note
		target = TIM3_CR1;
		*target |= 1;

		// Delay for the proper time
		delay_us(duration);

		// Stop playing note
		*target &= ~(1);
	}

	return;
}

// Initialize note delay and piezo config
void music_Init() {
	// Enable GPIOB in RCC
	volatile uint32_t* target = RCC_AHB1ENR;
	*target |= RCC_GPIOBEN;

	// Enable TIM3 in RCC
	target = RCC_APB1ENR;
	*target |= RCC_TIM3EN;

	// Set PB4 to alternate function
	target = GPIOB_MODER;
	*target |= (GPIO_ALTFUN << 8);

	// Set AFRL such that PB4 is connected to TIM3
	target = GPIOB_AFRL;
	*target |= PB4_PIEZO;

	// Set TIM3 prescale to 16, AKA 1 count = 1us
	target = TIM3_PSC;
	*target = 16;
```

```
	// Prescale fix
	// Forces an event to be generated and then
	// clears it right away which tricks the timer
	// into applying the prescale somehow
	target = TIM3_EGR;
	*target = 1;
	target = TIM3_SR;
	*target &= ~(1);

	// Configure CCMR for PWM mode
	target = TIM3_CCMR;
	*target |= (OC1M_PWM | OC1M_PE);

	// Enable in CCER
	target = TIM3_CCER;
	*target |= CCER_CC1E;

	// Assert not counting in CR1
	target = TIM3_CR1;
	*target &= ~(1);

	return;
}

// Loops through the array of notes
void music_Play(const note song[], int tempoScale) {
	// Index counter
	int i = 0;

	// Loop through array until we find a note with 0 period and 0 length
	while(!(song[i].period == 0 && song[i].length == 0)) {
		register uint32_t length = song[i].length;
		length *= tempoScale;
		register uint32_t period = song[i].period;
		uint32_t octave = song[i].octave;
		period = period >> octave;
		note_Play(period, length);
		i++;
	}

	return;
}
```

```c
/*
	CE2812 Lab 5
	Songs
	Evan Heinrich
	1/14/2022
*/

#include <music.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "uart_driver.h"
#include "delay.h"
#include "music.h"
#include "memconsole.h"

#define F_CPU 16000000UL



// main
int main(void){
	init_usart2(115200,F_CPU);

	delay_Init();

	music_Init();

	// Blank string for input
	char input[30] = "";

	// Address to interact with
	uint32_t* address = 0;

	// Command variable
	int command = -1;

	// Last argument, either length to read or value to write
	uint32_t argument = 0;

	// Welcome message
	printf("Evan's Memory Management Console\n\r");
	printf("Type \'?\' for help\n\r");

	// Infinite loop for program
	while(1==1) {
		// Prompt
		printf("> ");
		fgets(input, 29, stdin);
		// First token, determines command
		char* token = strtok(input, " ");

		// Second token, determines address
		char* arg1 = strtok(NULL, " ");

		// Third token, optional third argument, required for wmw, optional for dm
		char* arg2 = strtok(NULL, " ");
```

```
// If there is an extracted command
if(token != NULL) {
	// Attempt to parse the command
	command = parseCommand(token);

	// Attempt to parse address
	if(arg1 != NULL) {
		address = parseAddress(arg1);
	}

	// Attempt to parse second argument
	if(arg2 != NULL) {
		argument = parseArgument(arg2);
	}

	// Switch case for reported commands
	switch (command) {
	// Help command
	case 0:
		help();
		break;

	// Dump memory command
	case 1:
		if(arg1 != NULL) {
			if(arg2 == NULL) {
				memdmpDefault((uint8_t*)address);
			} else {
				memdmp((uint8_t*)address, argument);
			}
		} else {
			printf("No address provided\n\r");
		}
		break;

	// Read word command
	case 2:
		if(arg1 != NULL) {
			memwrd(address);
		} else {
			printf("No address provided\n\r");
		}
		break;

	// Write word command
	case 3:
		if(arg1 != NULL) {
			if(arg2 != NULL) {
				wmemwrd(address, argument);
			} else {
				printf("No value to write provided\n\r");
			}
		} else {
			printf("No address provided\n\r");
		}
		break;
```

```
                case 4:
                        if(arg1 != NULL) {
                                if(strcmp(arg1, "doom\n") == 0) {
                                        for(int i = 0; i < 2; i++) {
                                                music_Play(atDoomsGate, doomTempo);
                                        }
                                } else if(strcmp(arg1, "zelda\n") == 0) {
                                        music_Play(zelda, zeldaTempo);
                                } else {
                                        printf("Invalid song name\n");
                                }
                        } else {
                                printf("No song provided\n");
                        }
                        break;

                default:
                        printf("Invalid command\n\r");
                }

        } else {
                printf("No input\n\r");
        }

        // fgets again because it will read the newline from previous entry
        fgets(input, 29, stdin);

        // Clear the input string
        memset(input, 0, strlen(input));

        }

    exit(EXIT_SUCCESS);
    return 0;
}
```