Slight refactor to main method

I put some of the parsing function calls into their respective case statements

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

	delay_Init();

	music_Init();

	period_Init();

	wave_Init();

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

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

	// Command variable
	int command = -1;

	uint32_t frequency = 0;
	uint32_t samples = 0;
	uint16_t* waveform;

	// 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* strnCommand = strtok(input, " ");

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

		// Third token
		char* arg2 = strtok(NULL, " ");

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

```
// Switch case for reported commands
switch (command) {

// Help command
case 0:
	help();
	break;

// Dump memory command
case 1:
	// Attempt to parse address
	if(arg1 != NULL) {
		address = parseAddress(arg1);
	}

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

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

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

	if(arg1 != NULL) {
		memwrd(address);
	} else {
		printf("No address provided\n\r");
	}
	break;

// Write word command
case 3:
	// Attempt to parse address
	if(arg1 != NULL) {
		address = parseAddress(arg1);
	}

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

```c
        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;

    // Music command
    case 4:
        // Determine song to be played
        if(strcmp(arg1, "doom") == 0 || strcmp(arg1, "doom\n") == 0) {

            // Play background/foreground accordingly
            if(strcmp(arg2, "background\n") == 0) {
                music_Background(atDoomsGate);
            } else {
                music_Play(atDoomsGate);
            }

        } else if(strcmp(arg1, "zelda") == 0 || strcmp(arg1, "zelda\n") == 0) {

            // Play background/foreground accordingly
            if(strcmp(arg2, "background\n") == 0) {
                music_Background(zelda);
            } else {
                music_Play(zelda);
            }

        } else {
            printf("Invalid song\n");
        }
        break;

    // Frequency Measurement
    case 5:
        if(arg1 != NULL) {
            if(strcmp(arg1, "frequency\n") == 0) {
                printf("\nMeasuring frequency...\n\n");
                double average = period_Measure();
                printf("Measured frequency was %.2f Hz\n", average);
            } else {
                printf("Invalid measurement\n");
            }
        } else {
            printf("Measurement type required\n");
        }
        break;
```

```c
// Sine wave command
case 6:
	// Parse Frequency
	if(arg1 != NULL) {
		frequency = parseArgument(arg1);
	} else {
		printf("No frequency provided\n");
	}

	// Parse Samples
	if(arg2 != NULL) {
		samples = parseArgument(arg2);
	} else {
		printf("No number of samples provided\n");
	}

	// Execute Command
	if(arg1 != NULL && arg2 != NULL) {
		waveform = sineWave(samples);
		wave_Start(waveform, frequency, samples);
	}
	break;

// Sawtooth wave command
case 7:
	// Parse Frequency
	if(arg1 != NULL) {
		frequency = parseArgument(arg1);
	} else {
		printf("No frequency provided\n");
	}

	// Parse Samples
	if(arg2 != NULL) {
		samples = parseArgument(arg2);
	} else {
		printf("No number of samples provided\n");
	}

	// Execute Command
	if(arg1 != NULL && arg2 != NULL) {
		waveform = sawtoothWave(samples);
		wave_Start(waveform, frequency, samples);
	}
	break;

// Triangle wave command
case 8:
	// Parse Frequency
	if(arg1 != NULL) {
		frequency = parseArgument(arg1);
	} else {
		printf("No frequency provided\n");
	}

	// Parse Samples
	if(arg2 != NULL) {
		samples = parseArgument(arg2);
	} else {
		printf("No number of samples provided\n");
	}
```

```
					// Execute Command
					if(arg1 != NULL && arg2 != NULL) {
						waveform = triWave(samples);
						wave_Start(waveform, frequency, samples);
					}
					break;

				// Stop waveform command
				case 9:
					wave_Stop();

					// Free the malloc
					free((void*) waveform);
					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;
}
```

Waveform header file

```c
#ifndef GENERATOR_IS_ALIVE
#define GENERATOR_IS_ALIVE 1

#include <stdint.h>

uint16_t* sineWave(uint32_t samples);
uint16_t* triWave(uint32_t samples);
uint16_t* sawtoothWave(uint32_t samples);
void wave_Init(void);
void wave_Start(uint16_t* samples, uint32_t frequency, uint32_t numSamples);
void wave_Stop(void);
void TIM6_DAC_IRQHandler(void);

#endif
```

Waveform code

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include "registers_new.h"

#define PI 3.1415926
#define DAC_MAX 4095

static uint8_t waveStatus;
static uint16_t* waveform;

/**
 * Generates a dynamically allocated float array containing
 * samples of a sine/cosine wave with samples on the interval
 * [0,4095]
 */
uint16_t* sineWave(uint32_t samples) {
	// Create a chunk of memory to dump the waveform with
	// one extra space for a terminator
	uint16_t* wave = (uint16_t*) malloc((samples+1) * sizeof(uint16_t));

	for(int i = 0; i < samples; i++) {
		// For speed purposes try to fit a float into a register or registers
		register float sample = (0.5) * cosf((2*PI*i)/(samples-1)) + (0.5);

		// Convert to a 16-bit int even though its artificially limited to 12-bit
		uint16_t conversion = sample * DAC_MAX;

		// Write
		wave[i] = conversion;
	}

	// Terminator
	wave[samples] = -1;

	return wave;
}
```

```c
/**
 * Generates a dynamically allocated float array containing
 * samples of a triangle wave, with samples being on
 * the interval of [0,4095]
 */
uint16_t* triWave(uint32_t samples) {
	// Create a chunk of memory to dump the waveform with
	// one extra space for a terminator
	uint16_t* wave = (uint16_t*) malloc((samples+1) * sizeof(uint16_t));

	for(int i = 0; i < samples; i++) {
		register float sample = (2/(2*PI)) * asinf(sinf((2*PI*i)/(samples-1))) + (0.5);

		uint16_t conversion = sample * DAC_MAX;

		wave[i] = conversion;
	}

	wave[samples] = -1;
	return wave;

}

/**
 * Generates a dynamically allocated float array containing
 * samples of a sawtooth wave, with samples being on the
 * interval of [0,4095]
 */
uint16_t* sawtoothWave(uint32_t samples) {
	// Create a chunk of memory to dump the waveform with
	// one extra space for a terminator
	uint16_t* wave = (uint16_t*) malloc((samples+1) * sizeof(uint16_t));

	for(int i = 0; i < samples; i++) {
		register float sample = ((-1 / PI) * atanf((1/tanf((PI*i/(samples-1)))))) + 0.5;

		uint16_t conversion = sample * DAC_MAX;

		wave[i] = conversion;
	}

	// Terminator
	wave[samples] = -1;

	return wave;
}
```

```c
void wave_Init() {
	// Use TIM6 (basic 16-bit timer) to sequence the waveform
	// NOTE: TIM6 is only an upcounter! Write period to ARR!
	// PA4 as analog output connected to DAC ch.1

	volatile RCC* RCC_Target = (RCC*) RCC_BASE;
	volatile GPIO* GPIOA = (GPIO*) GPIOA_BASE;
	volatile TIMER_BASIC* TIM6 = (TIMER_BASIC*) TIM6_BASE;
	volatile NVIC* NVIC_Target = (NVIC*) NVIC_BASE;
	volatile DAC* DAC_Target = (DAC*) DAC_BASE;

	// Debugging purposes, set TIM6 to freeze in debugging mode
	uint32_t* DBG_APB1 = (uint32_t*)0xE0042008;
	*DBG_APB1 |= 1<<4;

	// Enable GPIOA
	RCC_Target->AHB1ENR |= RCC_GPIOAEN;

	// Set PA4 as analog
	GPIOA->MODER |= (GPIO_ANALOG << 8);

	// Enable TIM6
	RCC_Target->APB1ENR |= RCC_TIM6EN;

	// Enable DAC
	RCC_Target->APB1ENR |= RCC_DACEN;

	// Prescale TIM6 to 1us
	TIM6->PSC = 15;

	// Prescale fix
	TIM6->EGR = 1;
	TIM6->SR &= ~(1);

	// Assert not one-pulse mode
	TIM6->CR1 &= ~(TIM_OPM);

	// Enable DAC Ch1 & its Trigger
	DAC_Target->CR |= DAC_CH1EN;
	DAC_Target->CR |= (DAC_SWTGR << 3);
	DAC_Target->CR |= DAC_TEN1;

	// Enable TIM6 interrupts in NVIC
	// NVIC_ISER1 bit 22
	NVIC_Target->ISER[1] |= 1<<22;

}

void wave_Start(uint16_t* samples, uint32_t frequency, uint32_t numSamples) {
	// If the generator is already running
	if(waveStatus != 0) {
		printf("Waveform generator is already running. Stop the current generator to 
continue\n");
	} else {
		volatile TIMER_BASIC* TIM6 = (TIMER_BASIC*) TIM6_BASE;

		// Set the waveform
		waveform = samples;

		// Set status flag to busy
		waveStatus = 1;
```

```c
		// Determine period from frequency
		double timePerSample = ((double)1) / frequency;

		// Convert period to microseconds
		timePerSample *= 10E5;

		// Divide period by number of samples for time per sample
		timePerSample = timePerSample / numSamples;

		// Push to ARR
		TIM6->ARR = (uint16_t)timePerSample;

		// Enable TIM6 interrupts
		TIM6->DIER |= TIM_UIE;

		// Set TIM6_CEN
		TIM6->CR1 |= TIM_CEN;
	}

	return;
}

void wave_Stop() {
	// If the generator is not running
	if(waveStatus != 1) {
		printf("Waveform generator is not currently running. No changes made.\n");
	} else {
		volatile TIMER_BASIC* TIM6 = (TIMER_BASIC*) TIM6_BASE;

		// Clear TIM6_CEN
		TIM6->CR1 &= ~(TIM_CEN);

		// Stop TIM6 interrupts
		TIM6->DIER &= ~(TIM_UIE);

		waveStatus = 0;
	}
	return;
}

void TIM6_DAC_IRQHandler(void) {
	// Clear status register
	volatile TIMER_BASIC* TIM6 = (TIMER_BASIC*) TIM6_BASE;
	TIM6->SR = 0;

	volatile DAC* DAC_Target = (DAC*) DAC_BASE;

	// Iterator
	static uint32_t i = 0;

	// Read sample
	register uint16_t sample = waveform[i];

	// Check for terminator
	if(sample != 65535) {
		// Push sample to DAC
		// Samples are already limited to 12-bit, so just write
		DAC_Target->CH1_R12 = sample;
```

```
		// Increment iterator
		i++;

		// Trigger DAC Ch1
		DAC_Target->SW_TRIGR |= 1<<0;
	} else {
		// Reset iterator
		i = 0;

		// Read 0th sample
		sample = waveform[i];

		// Push sample to DAC
		DAC_Target->CH1_R12 = sample;

		// Increment iterator
		i++;

		// Trigger DAC Ch1
		DAC_Target->SW_TRIGR |= 1<<0;
	}

	return;
}
```