Just a note, I did create a Box folder and invited you to it where I uploaded a short video showing this code working and a couple pictures to show the wiring of the AD2 to the devboard.

```
/*
 * Required range for input frequencies: 50Hz-10kHz
 *     ->     Periods: 20,000us to 100us
 *     ->     Maybe try a 32-bit timer scaled to 10us per count
 *            ->     This means 2,000 counts = 50Hz, 10 counts = 10kHz
 *            ->     At the longest, I need to record 10 count periods meaning max of 20,000 counts
for a measurement
 *            ->     I could do a 16-bit timer then. TIM4 is free, so I'll try that
 *
 * TIM4 is a 16-bit timer holding ranges 0-65535 and the largest count we would have is 20,000 for
a 50Hz input.
 *     ->     TIM4_CH1 maps to PB6 which maps to CN10-17
 *
 *     My Experience
 *     This lab was not too terribly difficult for me personally. The only thing
 *     that really tripped me up was that I still had the hardware implementation for
 *     floating point math enabled. Besides that, using the input PWM function on the timers
 *     made this lab easy in general. One interesting side effect is that PB6 is also
 *     connected to one of the LEDs on the green board, so the LED connected to PB6
 *     could be seen flashing at the same frequency as the input waveform.
 */

#include "registers_new.h"
#include <stdint.h>
#include <stdio.h>

static uint8_t period_Status;
static uint16_t readings[10];

void period_Init() {
	// Setup peripheral objects
	volatile RCC* RCC_Target = (RCC*) RCC_BASE;
	volatile TIMER* TIM4 = (TIMER*) TIM4_BASE;
	volatile NVIC* NVIC_Target = (NVIC*)NVIC_BASE;
	volatile GPIO* GPIOB = (GPIO*) GPIOB_BASE;

	// Enable GPIOB, even though it most likely already was
	RCC_Target->AHB1ENR |= RCC_GPIOBEN;

	// Enable TIM4 in APB1ENR
	RCC_Target->APB1ENR |= RCC_TIM4EN;

	// Set the 10us prescale on TIM4
	TIM4->PSC = 16;

	// Set PB6 MODER to alternate funct
	GPIOB->MODER |= (GPIO_ALTFUN << 12);

	// Configure PB6 to Alternate Funct 2; TIM4_CH1
	GPIOB->AFRL |= (2<<24);

	// Setup PWM Input mode for TIM4 CH1
	// CC1P and CC1NP in CCER stay defaults; active-high input
	TIM4->CCMR1 |= 0b01<<0;   // CC1S input mapped on TI1
	TIM4->SMCR |= (0b101 << 4);      // Set trigger to Timer Input Ch1
	TIM4->SMCR |= (0b100 << 0); // Set slave mode to reset on TI1 rising edge
```

```
	TIM4->CCER |= 1;                // Enable capture/compare Ch1
	TIM4->CR1 |= TIM_CEN;

	// Enable TIM4 interrupts in NVIC
	NVIC_Target->ISER[0] |= (1<<30);

	return;
}

double period_Measure() {
	volatile TIMER* TIM4 = (TIMER*)TIM4_BASE;

	double average = 0;

	// This method takes the main thread of execution, but still
	if(!period_Status) {
		// Enable interrupt and counter, set flag
		period_Status = 1;
		TIM4->DIER |= (1<<6);

		// Busy wait until measurements are done
		while(period_Status) {}

		double minimum = 131072;
		double maximum = 0;

		// Calculate the average period
		for(int i = 0; i < 10; i++) {
			uint16_t value = readings[i];

			// Record min/max
			if((double)value >= maximum) {
				maximum = value;
			}
			if((double)value <= minimum) {
				minimum = value;
			}

			average += value;
		}

		// Calculate average period in us
		average = average / 10;

		// Convert to period in seconds
		average *= (1E-6);

		// Convert period to frequency
		average = 1/average;
		printf("Smallest recorded pulse was %.2f nanoseconds\n", minimum);
		printf("Largest recorded pulse was %.2f nanoseconds\n", maximum);
	}

	return average;
}
```

```
void TIM4_IRQHandler(void) {
	// Timer object
	// CCR1 stores period in units of 1us
	volatile TIMER* TIM4 = (TIMER*) TIM4_BASE;

	// Clear status register so this doesn't keep triggering
	TIM4->SR = 0;

	// Static to retain count between calls, but not file-scope
	static uint8_t count;

	// Partially for debugging purposes to see what value was read
	register uint16_t value = TIM4->CCR1;

	if(count < 10) {
		// Store recorded value and increment counter
		readings[count] = value;
		count++;
	} else {
		// Disable interrupt and set flag accordingly once
		// buffer is full
		TIM4->DIER &= ~(1<<6);
		period_Status = 0;
		count = 0;
	}

	return;
}
```

Period API Header

```
#ifndef PERIOD_MEASURE_ALIVE
#define PERIOD_MEASURE_ALIVE 1

void period_Init(void);
void TIM4_IRQHandler(void);
double period_Measure(void);

#endif
```

Change to the main method

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

	delay_Init();

	music_Init();

	period_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\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;

// 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;
                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;

                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;
}
```