```c
/*----------------------------------------------------------------------------
 * Name:    ArbitoryFunc.c
 * Purpose: Code to ouput a specified resolution of a wave table to the DAC, using
            DMA requests to free up CPU time.
 * Note(s): adapted from example code found at
 *          http://00xnor.blogspot.co.uk/2014/01/6-stm32-f4-dac-dma-waveform-generator.html
 *----------------------------------------------------------------------------
 *
 *----------------------------------------------------------------------------*/

#include "STM32F4xx.h"
#include "main_2.h"

#define   DAC_DHR12R1_ADDR  0x40007408
#define   OUT_FREQ          5000                                 // Output waveform frequency
#define   WAVE_RES          128                                   // Waveform resolution
#define   CNT_FREQ          84000000                             // TIM6 counter clock (prescaled APB1)
#define   TIM_PERIOD        ((CNT_FREQ)/((WAVE_RES)*(OUT_FREQ))) // Autoreload reg value

// Sinc fucntion
const uint16_t waveForm[WAVE_RES] = { 3995, 3987, 3964, 3925, 3872, 3805, 3725, 3633, 3531, 3419,
                                      3300, 3176, 3047, 2915, 2784, 2653, 2524, 2400, 2282, 2171,
                                      2068, 1975, 1891, 1819, 1758, 1708, 1670, 1644, 1629, 1624,
                                      1630, 1646, 1669, 1700, 1738, 1780, 1827, 1876, 1926, 1977,
                                      2027, 2075, 2120, 2161, 2198, 2229, 2255, 2275, 2289, 2296,
                                      2297, 2293, 2282, 2267, 2247, 2223, 2195, 2165, 2134, 2101,
                                      2068, 2036, 2005, 1976, 1950, 1927, 1907, 1891, 1880, 1873,
                                      1870, 1871, 1877, 1886, 1899, 1916, 1935, 1956, 1979, 2003,
                                      2027, 2051, 2075, 2097, 2118, 2136, 2152, 2165, 2175, 2182,
                                      2185, 2185, 2182, 2175, 2166, 2154, 2140, 2124, 2106, 2087,
                                      2068, 2049, 2030, 2011, 1994, 1979, 1965, 1954, 1945, 1939,
                                      1935, 1935, 1937, 1941, 1948, 1957, 1969, 1981, 1996, 2011,
                                      2027, 2043, 2059, 2074, 2089, 2102, 2114, 212 };

void TIM5_Config(void)
{
  TIM_TimeBaseInitTypeDef TIM5_TimeBase;

  /* TIM5 Periph clock enable */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE);

  /* Time base configuration */
  TIM_TimeBaseStructInit(&TIM5_TimeBase);
  TIM5_TimeBase.TIM_Period        = (uint16_t)TIM_PERIOD;
  TIM5_TimeBase.TIM_Prescaler     = 0;
  TIM5_TimeBase.TIM_ClockDivision = 0;
  TIM5_TimeBase.TIM_CounterMode   = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM6, &TIM5_TimeBase);

  /* TIM5 TRGO selection */
  TIM_SelectOutputTrigger(TIM5, TIM_TRGOSource_Update);

  /* TIM5 enable counter */
  TIM_Cmd(TIM5, ENABLE);
}

void DAC_Ch1_ArbitoryConfig(void)
{
  DAC_InitTypeDef DAC_INIT;
  DMA_InitTypeDef DMA_INIT;

  /* DAC channel1 Configuration */
  DAC_INIT.DAC_Trigger        = DAC_Trigger_T5_TRGO;
  DAC_INIT.DAC_WaveGeneration = DAC_WaveGeneration_None;
  DAC_INIT.DAC_OutputBuffer   = DAC_OutputBuffer_Enable;
  DAC_Init(DAC_Channel_1, &DAC_INIT);

  /* DMA1_Stream5 channel7 configuration **************************************/
  DMA_DeInit(DMA1_Stream5);
  DMA_INIT.DMA_Channel            = DMA_Channel_7;
  DMA_INIT.DMA_PeripheralBaseAddr = (uint32_t)DAC_DHR12R1_ADDR;
  DMA_INIT.DMA_Memory0BaseAddr    = (uint32_t)&waveForm;
  DMA_INIT.DMA_DIR                = DMA_DIR_MemoryToPeripheral;
  DMA_INIT.DMA_BufferSize         = WAVE_RES;
  DMA_INIT.DMA_PeripheralInc      = DMA_PeripheralInc_Disable;
  DMA_INIT.DMA_MemoryInc          = DMA_MemoryInc_Enable;
  DMA_INIT.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
  DMA_INIT.DMA_MemoryDataSize     = DMA_MemoryDataSize_HalfWord;
  DMA_INIT.DMA_Mode               = DMA_Mode_Circular;
  DMA_INIT.DMA_Priority           = DMA_Priority_High;
  DMA_INIT.DMA_FIFOMode           = DMA_FIFOMode_Disable;
  DMA_INIT.DMA_FIFOThreshold      = DMA_FIFOThreshold_HalfFull;
  DMA_INIT.DMA_MemoryBurst        = DMA_MemoryBurst_Single;
  DMA_INIT.DMA_PeripheralBurst    = DMA_PeripheralBurst_Single;
  DMA_Init(DMA1_Stream5, &DMA_INIT);

  /* Enable DMA1_Stream5 */
  DMA_Cmd(DMA1_Stream5, ENABLE);

  /* Enable DAC Channel1 */
  DAC_Cmd(DAC_Channel_1, ENABLE);

  /* Enable DMA for DAC Channel1 */
  DAC_DMACmd(DAC_Channel_1, ENABLE);
}

void DAC_Arbitory_On(void)
{
  /* Enable DAC Channel1 */
  DAC_Cmd(DAC_Channel_1, ENABLE);
}

void DAC_Arbitory_Off(void)
{
  /* Disable DAC Channel1 */
  DAC_Cmd(DAC_Channel_1, DISABLE);
}

```

```c
/*----------------------------------------------------------------------------
 * Name:    DAC.c
 * Purpose: Functions to initilise the DAC, and subsequently provide triangle wave
            functionality, noise generation, and supports arbitory function generation.
 * Note(s): Example code taken from STMicroElectronics Application Teams,
            DAC_SignalsGeneration example project
 *----------------------------------------------------------------------------
 *
 *----------------------------------------------------------------------------*/

#include "STM32F4xx.h"
#include "main_2.h"
#include "DAC.h"
#include "ArbitoryFunc.h"

// CMSIS data structure for DAC
DAC_InitTypeDef  DAC_InitStructure;

void DACs_Init(void)
{
   /* Preconfiguration before using DAC------*/
  GPIO_InitTypeDef GPIO_InitStructure;

  /* DMA1 clock and GPIOA clock enable (to be used with DAC) */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1 | RCC_AHB1Periph_GPIOA, ENABLE);

  /* DAC Periph clock enable */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);

  /* DAC channel 1 & 2 (DAC_OUT1 = PA.4)(DAC_OUT2 = PA.5) configuration */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  /* TIM Configuration ----------*/
  TIM6_Config();
  TIM5_Config();

  /* Set DAC registers to default values */
  DAC_DeInit();
}

void TIM6_Config(void)
{
  TIM_TimeBaseInitTypeDef    TIM_TimeBaseStructure;

  /* TIM6 Periph clock enable */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6, ENABLE);

  /* Time base configuration */
  TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
  TIM_TimeBaseStructure.TIM_Period = 1;
  TIM_TimeBaseStructure.TIM_Prescaler = 0;
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM6, &TIM_TimeBaseStructure);

  /* TIM6 TRGO selection */
  TIM_SelectOutputTrigger(TIM6, TIM_TRGOSource_Update);

  /* TIM6 enable counter */
  TIM_Cmd(TIM6, ENABLE);
}

void DAC_Ch2_TriangleConfig(void)
{
 /* DAC channel2 Configuration */
  DAC_InitStructure.DAC_Trigger = DAC_Trigger_T6_TRGO;
  DAC_InitStructure.DAC_WaveGeneration = DAC_WaveGeneration_Triangle;
  DAC_InitStructure.DAC_LFSRUnmask_TriangleAmplitude = DAC_TriangleAmplitude_255;
  DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
  DAC_Init(DAC_Channel_2, &DAC_InitStructure);

  /* Set DAC channel2 DHR12RD register */
  DAC_SetChannel2Data(DAC_Align_12b_R, 0x100);
}

```

```c
void DAC_Ch1_NoiseConfig(void)
{
 /* DAC channel1 Configuration */
  DAC_InitStructure.DAC_Trigger = DAC_Trigger_T6_TRGO;
  DAC_InitStructure.DAC_WaveGeneration = DAC_WaveGeneration_Noise;
  DAC_InitStructure.DAC_LFSRUnmask_TriangleAmplitude = DAC_LFSRUnmask_Bits11_0;   // Max bits unmasked
  DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
  DAC_Init(DAC_Channel_1, &DAC_InitStructure);

  /* Set DAC Channel1 DHR12L register */
  DAC_SetChannel1Data(DAC_Align_12b_L, 0x7FF0);
}

void DAC_Noise_On(void)
{
  /* Enable DAC Channel1 */
  DAC_Cmd(DAC_Channel_1, ENABLE);
}

void DAC_Noise_Off(void)
{
  /* Disable DAC Channel1 */
  DAC_Cmd(DAC_Channel_1, DISABLE);
}

void DAC_Triangle_On(void) {
  /* Enable DAC Channel2 */
  DAC_Cmd(DAC_Channel_2, ENABLE);
}

void DAC_Traingle_Off(void) {
  /* Disable DAC Channel2 */
  DAC_Cmd(DAC_Channel_2, DISABLE);
}

```

```c
/*----------------------------------------------------------------------------
 * Name:    DDS.c
 * Purpose: Fucntions to initialise the DDS, set default data, and accept new
            frequencies from the user.
 * Note(s):
 *----------------------------------------------------------------------------*/

#include "STM32F4xx.h"
#include "DDS.h"
#include "main_2.h"
#include <math.h>

#define DDS_CLOCK 125000000
#define CLOCK 4 /* W_CLK pin */
#define LOAD   5 /* FQ_UP pin*/
#define DATA   3 /* DATA pin */

/*----------------------------------------------------------------------------
  initialize DDS for serial communication
 *----------------------------------------------------------------------------*/
void DDS_Init (void) {

  RCC->AHB1ENR    |= ((1UL <<  4));         /* Enable GPIOE clock */

  GPIOE->MODER    &= ~((3UL << 2* 3) |
                       (3UL << 2* 4) |
                       (3UL << 2* 5) );   /* PE.0,3-4 are outputs */
  GPIOE->MODER    |=  ((1UL << 2* 3) |
                       (1UL << 2* 4) |
                       (1UL << 2* 5) );
  GPIOE->OTYPER   &= ~((1UL <<    3) |
                       (1UL <<    4) |
                       (1UL <<    5) );   /* PE.0,3-4 are output Push-Pull */
  GPIOE->OSPEEDR  &= ~((3UL << 2* 3) |
                       (3UL << 2* 4) |
                       (3UL << 2* 5) );   /* PE.0,3-4 are 50MHz Fast Speed */
  GPIOE->OSPEEDR  |=  ((2UL << 2* 3) |
                       (2UL << 2* 4) |
                       (2UL << 2* 5) );
  GPIOE->PUPDR    &= ~((3UL << 2* 3) |
                       (3UL << 2* 4) |
                       (3UL << 2* 5) );   /* PE.0,3-4 are Pull up */
  GPIOE->PUPDR    |=  ((1UL << 2* 3) |
                       (1UL << 2* 4) |
                       (1UL << 2* 5) );
}

void Pulse_Clock() {
  GPIOE->ODR |= (1 << CLOCK);
  Delay(1);
  GPIOE->ODR &= ~(1 << CLOCK);
}

void Pulse_Frequency() {
  GPIOE->ODR |= (1 << LOAD);
  Delay(1);
  GPIOE->ODR &= ~(1 << LOAD);
}

void Data_Low() {
  GPIOE->ODR &= ~(1 << DATA);
}

void DDS_Write_Data(int input_data) {
  GPIOE->ODR |= (input_data << DATA);
}

/*----------------------------------------------------------------------------
  Function that set the DDS output to a default value
 *----------------------------------------------------------------------------*/
void DDS_Default_Init (void) {

  int i = 0;
  int Default_Data[40] = {0,0,0,1,1,1,0,0,0,1,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; //1KHz

  Pulse_Clock();
  Delay(1);
  Pulse_Frequency();
  Delay(1);

  // Send the data array 1 bit at a time to the DDS
  for(i = 0; i <40; i++){
    Data_Low();
    DDS_Write_Data(Default_Data[i]);
    Delay(1);
    Pulse_Clock();
  }

  Pulse_Frequency();
  Delay(1);
  Data_Low();

}
/*-----------------------------------------------------------------------------
  Function that sets the DDS frequency to a user provided value
 *-----------------------------------------------------------------------------*/
void DDS_Set (double frequency) {

  int j = 0;
  int k = 0;
  int tuningWord = 0;
  int Send_Data[40];

  // Calculate the new tuning word
  tuningWord = (int) ((frequency * pow(2, 32))/DDS_CLOCK);

  // Construct the data array ready to be sent to DDS
  for (j = 0; j < 40; j++) {
    // calculate each array position by bitwise anding the tuning word with 1
    Send_Data[j] = tuningWord & (1 << j) ? 1 : 0;
  }

  Pulse_Clock();
  Delay(1);
  Pulse_Frequency();
  Delay(1);

  // Send the data array 1 bit at a time to the DDS
  for(k = 0; k <40; k++){
    Data_Low();
    DDS_Write_Data(Send_Data[k]);
    Delay(1);
    Pulse_Clock();
  }

  Pulse_Frequency();
  Delay(1);
  Data_Low();
}
```

```
    /* TIM4 configuration: PWM Input mode ------------------------
```

```c
/*----------------------------------------------------------------------------
 * Name:    FSK.c
 * Purpose: Functions to provide frequency shift keying for an input waveform,
            by setting 2 different DDS frequencies appropiately.
 * Note(s): Example code taken from STMicroElectronics Application Teams,
            TIM_PWM_Input eample project.
 *----------------------------------------------------------------------------
 *
 *----------------------------------------------------------------------------*/

#include "STM32F4xx.h"
#include "main_2.h"
#include "FSK.h"
#include "DDS.h"

void FSK_Init(void)
{
  TIM_ICInitTypeDef  TIM_ICInitStructure;

  GPIO_InitTypeDef GPIO_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;

  /* TIM4 clock enable */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);

  /* GPIOB clock enable */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

  /* TIM4 chennel2 configuration : PB.07 */
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_7;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP ;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  /* Connect TIM pin to AF2 */
  GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_TIM4);

  /* Enable the TIM4 global Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  /* TIM4 configuration: PWM Input mode */
  TIM_ICInitStructure.TIM_Channel = TIM_Channel_2;
  TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_BothEdge;
  TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
  TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
  TIM_ICInitStructure.TIM_ICFilter = 0x0;

  TIM_PWMIConfig(TIM4, &TIM_ICInitStructure);

  /* Select the TIM4 Input Trigger: TI2FP2 */
  TIM_SelectInputTrigger(TIM4, TIM_TS_TI2FP2);

  /* Select the slave Mode: Reset Mode */
  TIM_SelectSlaveMode(TIM4, TIM_SlaveMode_Reset);
  TIM_SelectMasterSlaveMode(TIM4,TIM_MasterSlaveMode_Enable);

  /* TIM enable counter */
  TIM_Cmd(TIM4, ENABLE);

  /* Enable the CC2 Interrupt Request */
  TIM_ITConfig(TIM4, TIM_IT_CC2, ENABLE);
}
```

```c
/*----------------------------------------------------------------------------
 * Name:    main.c
 * Purpose:
 * Note(s):
 *----------------------------------------------------------------------------
 *
 *----------------------------------------------------------------------------*/

#include "STM32F4xx.h"
#include "stm32f4_discovery.h"
#include "main_2.h"
#include "LED.h"
#include "SWT.h"
#include "LCD.h"
#include "Sqaure.h"
#include "DAC.h"
#include "DDS.h"
#include "FreqMeter.h"
#include "hd44780.h"
#include "ArbitoryFunc.h"
#include "FSK.h"
#include <stdio.h>

volatile uint32_t msTicks;                      /* counts 1ms timeTicks       */
volatile double currentFrequency = 1000;
volatile double increment = 1;
volatile int function = WAVE_GENERATION;
volatile int freqRange = HUNDRED_TO_10K;
volatile unsigned char updateFlag = 1;
volatile int dutyCycle = 50;

/*----------------------------------------------------------------------------
  MAIN function
 *----------------------------------------------------------------------------*/
int main (void) {

  __disable_irq();
  SystemCoreClockUpdate();                      /* Get Core Clock Frequency   */

  if (SysTick_Config(SystemCoreClock / 1680)) {  /* SysTick 1 msec interrupts  */
    while (1);                                  /* Capture error              */
  }
  __enable_irq();

  // Initialise Required Pins
  BTN_Init();
  SWTS_Init();
  LED_Init();
  init_lcd_driver();
  hd44780_init(GPIOD, GPIOB, GPIO_Pin_0, GPIO_Pin_1, GPIO_Pin_2, GPIO_Pin_4,
                GPIO_Pin_5, GPIO_Pin_6, GPIO_Pin_7, HD44780_LINES_2, HD44780_FONT_5x8);
  DDS_Init();
  DACs_Init();

  //Initialise components to defaults
  DDS_Default_Init();
  Pulse_Config();

  // Turn on LCD display
  hd44780_display(true, false, false);

  // Set up intterupts for the blue user button - ie the menu
  //STM_EVAL_PBInit(BUTTON_USER, BUTTON_MODE_EXTI);
  Config_menu_interrupt();

  while(1)
  {

    while(function == WAVE_GENERATION)
    {
      uint32_t switchsState;

      if(updateFlag == 1)
      {
        updateFlag = 0;
        hd44780_clear();
        hd44780_position(0, 0);
        hd44780_print("WAVE GENERATION");
      }

      // Check for switch presses to change DDS fequency
      switchsState = SWT_Get();

      if (switchsState == (1UL << 8)) {
        LED_All_Off();
        LED_On(0);
        increment = 0.01;
        hd44780_print_lines("WAVE GENERATION", "Inc = 0.01    Hz");
      }
      else if (switchsState == (1UL << 9)) {
        LED_All_Off();
        LED_On(1);
        increment = 1;
        hd44780_print_lines("WAVE GENERATION", "Inc = 1       Hz");
      }
      else if (switchsState == (1UL << 10)) {
        LED_All_Off();
        LED_On(2);
        increment = 100;
        hd44780_print_lines("WAVE GENERATION", "Inc = 100     Hz");
      }
      else if (switchsState == (1UL << 11)) {
        LED_All_Off();
        LED_On(3);
        increment = 1000;
        hd44780_print_lines("WAVE GENERATION", "Inc = 1000    Hz");

      }
      else if (switchsState == (1UL << 12)) {
        LED_All_Off();
        LED_On(4);
        increment = 100000;
        hd44780_print_lines("WAVE GENERATION", "Inc = 10000   Hz");
      }
      else if (switchsState == (1UL << 13)) {
        LED_All_Off();
        LED_On(5);
        increment = 1000000;
        hd44780_print_lines("WAVE GENERATION", "Inc = 1000000 Hz");
      }
      else if (switchsState == (1UL << 14)) {
        char tmp_string[15];

        LED_On(6);

        currentFrequency = currentFrequency - increment;
        if(currentFrequency < 0.01)
          currentFrequency = 0.01;
        DDS_Set(currentFrequency);

        sprintf(tmp_string, "Freq = %.2f", currentFrequency);
        hd44780_print_lines("WAVE GENERATION", tmp_string);

        LED_Off(6);
      }
      else if (switchsState == (1UL << 15)) {
        char tmp_string[15];

        LED_On(7);

        currentFrequency = currentFrequency + increment;
        if(currentFrequency > 35000000)
            currentFrequency = 35000000;
        DDS_Set(currentFrequency);

        sprintf(tmp_string, "Freq = %.2f ", currentFrequency);
        hd44780_print_lines("WAVE GENERATION", tmp_string);

        LED_Off(7);
      }
    }

    while(function == FREQUENCY_METER)
    {
      uint32_t switchsState;

```

```c
      if(updateFlag == 1)
      {
        Freq_Meter_Init();
        updateFlag = 0;
        hd44780_clear();
        hd44780_position(0, 0);
        hd44780_print("FREQUENCY METER");
      }


      // Check for switch presses to capture frequency meter value
      switchsState = SWT_Get();

      if (switchsState == (1UL << 8)) {
        LED_All_Off();
        LED_On(0);
        freqRange = LESS_THAN_1;
        TIM4->PSC = 0xF000;
        hd44780_print_lines("FREQUENCY METER", "Range = <1");
      }
      else if (switchsState == (1UL << 9)) {
        LED_All_Off();
        LED_On(1);
        freqRange = ONE_TO_100;
        TIM4->PSC = 0x0F00;
        hd44780_print_lines("FREQUENCY METER", "Range = 1-100");
      }
      else if (switchsState == (1UL << 10)) {
        LED_All_Off();
        LED_On(2);
        freqRange = HUNDRED_TO_10K;
        TIM4->PSC = 0x000F;
        hd44780_print_lines("FREQUENCY METER", "Range = 100-10K");
      }
      else if (switchsState == (1UL << 11)) {
        LED_All_Off();
        LED_On(3);
        freqRange = MORE_THAN_10K;
        TIM4->PSC = 0x0000;
        hd44780_print_lines("FREQUENCY METER", "Range = > 10K");
      }
      else if (switchsState == (1UL << 15)) {
        char Freq_Tmp[15];
        char DC_Tmp[15];

        LED_On(7);

        if(freqRange == LESS_THAN_1){
          sprintf(Freq_Tmp, "Freq = %.2f", low_Frequency);
        }
        else {
          sprintf(Freq_Tmp, "Freq = %d", Frequency);
        }
        sprintf(DC_Tmp, "Duty = %d", DutyCycle);
        hd44780_print_lines(Freq_Tmp, DC_Tmp);

        LED_Off(7);
      }
    }

    while(function == NOISE_GENERATION)
    {
      if(updateFlag == 1)
      {
        DAC_Ch1_NoiseConfig();
        DAC_Noise_On();
        updateFlag = 0;
        hd44780_clear();
        hd44780_position(0, 0);
        hd44780_print("NOISE GENERATION");
      }

    }

    while(function == ARBITORY_FUNCTION)
    {
      if(updateFlag == 1)
      {
        DAC_Ch1_ArbitoryConfig();
        DAC_Arbitory_On();
        updateFlag = 0;
        hd44780_clear();
        hd44780_position(0, 0);
        hd44780_print("ARBITRARY FUNC");
      }

    }

    while(function == PULSE_GENERATOR)
    {
      uint32_t switchsState;

      if(updateFlag == 1)
      {
        updateFlag = 0;
        hd44780_clear();
        hd44780_position(0, 0);
        hd44780_print("PULSE GENERATOR");
      }

      // Check for switch presses to change duty cycle
      switchsState = SWT_Get();

      if (switchsState == (1UL << 14)) {
        char tmp_string[15];

        LED_On(6);

        dutyCycle--;

        if(dutyCycle < 0)
          dutyCycle = 0;

        PWM_SetDC(dutyCycle);

        sprintf(tmp_string, "Duty = %d %%", dutyCycle);
        hd44780_print_lines("PULSE GENERATOR", tmp_string);

        LED_Off(6);
      }
      else if (switchsState == (1UL << 15)) {
        char tmp_string[15];

        LED_On(7);

        dutyCycle++;

        if(dutyCycle > 100)
          dutyCycle = 100;

        PWM_SetDC(dutyCycle);

        sprintf(tmp_string, "Duty = %d %%", dutyCycle);
        hd44780_print_lines("PULSE GENERATOR", tmp_string);

        LED_Off(7);
      }
    }

    while(function == FREQUENCY_KEY_SHIFT)
    {
      if(updateFlag == 1)
      {
        FSK_Init();
        updateFlag = 0;
        hd44780_clear();
        hd44780_position(0, 0);
        hd44780_print("FREQ KEY SHIFT");
      }

      if(FSK_Change == true)
      {
        if(FSK_Freq == HIGH)
        {
          DDS_Set(1000);          //Output 1KHz wave if input wave is "high"
        }
        else if(FSK_Freq == LOW)
        {
          DDS_Set(1000000);           //Output 100Hz wave if input wave is "low"
        }
        FSK_Change = false;
      }
    }
  }
}

/*----------------------------------------------------------------------------
  SysTick_Handler
 *----------------------------------------------------------------------------*/
void SysTick_Handler(void) {
  msTicks++;
}

/*----------------------------------------------------------------------------
  delays number of tick Systicks (happens every 1 ms)
 *----------------------------------------------------------------------------*/
void Delay (uint32_t dlyTicks) {
  uint32_t curTicks;
  curTicks = msTicks;

  while ((msTicks - curTicks) < dlyTicks);
}

void Config_menu_interrupt_2 (void) {
  EXTI_InitTypeDef EXTI_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure;

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

  /* Configure GPIOs as as inputs */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15 | GPIO_Pin_14 | GPIO_Pin_13 | GPIO_Pin_12 | GPIO_Pin_11 | GPIO_Pin_10;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  /* Connect EXTI Lines 10-15 to GPIOB Pins 10-15*/
  SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB, EXTI_PinSource10);
  SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB, EXTI_PinSource11);
  SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB, EXTI_PinSource12);
  SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB, EXTI_PinSource13);
  SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB, EXTI_PinSource14);
  SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB, EXTI_PinSource15);

  /* Configure EXTI lines 8-15 */
  EXTI_InitStructure.EXTI_Line = EXTI_Line10 | EXTI_Line11 | EXTI_Line12 | EXTI_Line13 | EXTI_Line14 | EXTI_Line15;
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);

  /* Enable and set EXTI Lines 8-15 Interrupt to the lowest priority */
  NVIC_InitStructure.NVIC_IRQChannel = EXTI15_10_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}

void Config_menu_interrupt(void) {
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

    /* Connect EXTI Line0 to GPIOA Pin 0*/
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);

    /* Configure EXTI line0 */
    EXTI_InitStructure.EXTI_Line = EXTI_Line0;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    /* Enable and set EXTI Line0 Interrupt to the lowest priority */
    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; // changed from 0x01
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;        // changed from 0x01
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

 void EXTI0_IRQHandler(void) {

   LED_All_Off();
   updateFlag = 1;

   if (function == WAVE_GENERATION)
    {
      function = FREQUENCY_METER;
    }
  else if (function == FREQUENCY_METER)
    {
      function = NOISE_GENERATION;
    }
  else if (function == NOISE_GENERATION)
    {
      function = ARBITORY_FUNCTION;
      DAC_Noise_Off();
    }
  else if (function == ARBITORY_FUNCTION)
    {
      function = PULSE_GENERATOR;
      DAC_Arbitory_Off();
    }
  else if (function == PULSE_GENERATOR)
    {
      function = FREQUENCY_KEY_SHIFT;
    }
  else if (function == FREQUENCY_KEY_SHIFT)
    {
      function = WAVE_GENERATION;
    }
  else
    {
      function = WAVE_GENERATION;
      DAC_Noise_Off();
      DAC_Arbitory_Off();
    }

  EXTI_ClearITPendingBit(EXTI_Line0);       // Clear the pending bit to signal IRQ finished
}

void EXTI15_10_IRQHandler(void) {

  ITStatus line10, line11, line12, line13, line14, line15;

  LED_All_Off();
  updateFlag = 1;

  line10 = EXTI_GetITStatus(EXTI_Line10);
  line11 = EXTI_GetITStatus(EXTI_Line11);
  line12 = EXTI_GetITStatus(EXTI_Line12);
  line13 = EXTI_GetITStatus(EXTI_Line13);
  line14 = EXTI_GetITStatus(EXTI_Line14);
  line15 = EXTI_GetITStatus(EXTI_Line15);

  if(line10 == SET) {
    function = WAVE_GENERATION;
  }
  else if(line11 == SET) {
    function = FREQUENCY_METER;
  }
  else if(line12 == SET) {
    function = NOISE_GENERATION;
  }
  else if(line13 == SET) {
    function = ARBITORY_FUNCTION;
```

```c
    }
    else if(line14 == SET) {
      function = PULSE_GENERATOR;
    }
    else if(line15 == SET) {
      function = FREQUENCY_KEY_SHIFT;
    }

    DAC_Noise_Off();
    DAC_Arbitory_Off();

  }

```

```c
/*----------------------------------------------------------------------------
 * Name:    Square.c
 * Purpose: Pulse gerator, with a duty cyle variable by the user.
 * Note(s): Code modified from ST MicroElectronics Application Teams,
            TIM_PWM_Output example project.
 *----------------------------------------------------------------------------
 *
 *----------------------------------------------------------------------------*/

#include "STM32F4xx.h"
#include "LCD.h"
#include "Sqaure.h"

#define TIM3_CLK_OUT 42000
#define TIM3_CNT_CLK 28000000
#define TIM3_ARR 665 //((TIM3_CNT_CLK / TIM3_CLK_OUT) - 1)
#define FIFTY_PERCENT 333

void Pulse_Config (void) {
  // Run timer config and initialise pulses to 50:50 duty cycle
  TIM3_Config();
  PWM_Config(TIM3_ARR);
}

void TIM3_Config (void) {
  GPIO_InitTypeDef GPIO_InitStructure;

  /* TIM3 clock enable */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

  /* GPIOC clock enable */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

  /* GPIOC Configuration: TIM3 CH1 (PC6) */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 ;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP ;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  /* Connect TIM3 pins to AF2 */
  GPIO_PinAFConfig(GPIOC, GPIO_PinSource6, GPIO_AF_TIM3);
}

void PWM_Config(int period)
{
  TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
  TIM_OCInitTypeDef TIM_OCInitStructure;
  uint16_t PrescalerValue = 0;

/* Compute the prescaler value */
  PrescalerValue = (uint16_t) ((SystemCoreClock /2) / 28000000) - 1;

  /* Time base configuration */
  TIM_TimeBaseStructure.TIM_Period = 665;
  TIM_TimeBaseStructure.TIM_Prescaler = PrescalerValue;
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

  TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);

  /* PWM1 Mode configuration: Channel1 */
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStructure.TIM_Pulse = FIFTY_PERCENT;                  // 50:50 duty cyle
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;

  TIM_OC1Init(TIM3, &TIM_OCInitStructure);

  TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);

  TIM_ARRPreloadConfig(TIM3, ENABLE);

  /* TIM3 enable counter */
  TIM_Cmd(TIM3, ENABLE);
}

```

```c
void PWM_SetDC(uint16_t dutycycle)
{
  uint16_t newDutyCycle;

  // Calculate the new duty cycle
  newDutyCycle = (dutycycle * TIM3_ARR) / 100;

  // set the new duty cycle into the capture compare register
  TIM_SetCompare1(TIM3, newDutyCycle);
}

```