```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <math.h>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
#define NS 128  // Look up table size of 128
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;

DAC_HandleTypeDef hdac1;
DAC_HandleTypeDef hdac2;
DAC_HandleTypeDef hdac3;
DAC_HandleTypeDef hdac4;
DMA_HandleTypeDef hdma_dac1_ch2;
DMA_HandleTypeDef hdma_dac1_ch1;
DMA_HandleTypeDef hdma_dac2_ch1;

OPAMP_HandleTypeDef hopamp3;
OPAMP_HandleTypeDef hopamp4;
OPAMP_HandleTypeDef hopamp6;

TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;

UART_HandleTypeDef huart1;

/* USER CODE BEGIN PV */

uint32_t Sine_LUT[NS] = { //Sine lookup table
  2048, 2149, 2250, 2350, 2450, 2549, 2646, 2742, 2837, 2929, 3020, 3108, 3193, 3275, 3355, 3431,
  3504, 3574, 3639, 3701, 3759, 3812, 3861, 3906, 3946, 3982, 4013, 4039, 4060, 4076, 4087, 4094,
  4095, 4091, 4082, 4069, 4050, 4026, 3998, 3965, 3927, 3884, 3837, 3786, 3730, 3671, 3607, 3539,
  3468, 3394, 3316, 3235, 3151, 3064, 2975, 2883, 2790, 2695, 2598, 2500, 2400, 2300, 2199, 2098,
  1997, 1896, 1795, 1695, 1595, 1497, 1400, 1305, 1212, 1120, 1031, 944, 860, 779, 701, 627,
  556, 488, 424, 365, 309, 258, 211, 168, 130, 97, 69, 45, 26, 13, 4, 0,
  1, 8, 19, 35, 56, 82, 113, 149, 189, 234, 283, 336, 394, 456, 521, 591,
  664, 740, 820, 902, 987, 1075, 1166, 1258, 1353, 1449, 1546, 1645, 1745, 1845, 1946, 2047
};

/*
uint32_t ADSR_LUT[NS] = { //Sine lookup table
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
*/

uint32_t Square_LUT[NS] = { //Square wave look up table
  4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095,
  4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095,
  4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095,
  4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095, 4095,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

uint32_t Triangle_LUT[NS] = { //Triangle wave look up table
  63, 127, 191, 255, 319, 383, 447, 511, 575, 639, 703, 767, 831, 895, 959, 1023,
  1087, 1151, 1215, 1279, 1343, 1407, 1471, 1535, 1599, 1663, 1727, 1791, 1855, 1919, 1983, 2047,
  2111, 2175, 2239, 2303, 2367, 2431, 2495, 2559, 2623, 2687, 2751, 2815, 2879, 2943, 3007, 3071,
  3135, 3199, 3263, 3327, 3391, 3455, 3519, 3583, 3647, 3711, 3775, 3839, 3903, 3967, 4031, 4095,
  4032, 3968, 3904, 3840, 3776, 3712, 3648, 3584, 3520, 3456, 3392, 3328, 3264, 3200, 3136, 3072,
  3008, 2944, 2880, 2816, 2752, 2688, 2624, 2560, 2496, 2432, 2368, 2304, 2240, 2176, 2112, 2048,
  1984, 1920, 1856, 1792, 1728, 1664, 1600, 1536, 1472, 1408, 1344, 1280, 1216, 1152, 1088, 1024,
  960, 896, 832, 768, 704, 640, 576, 512, 448, 384, 320, 256, 192, 128, 64, 0
};

uint32_t last_1 = 1, last_2 = 1, last_3 = 1;  //for wave switching
uint32_t *pl1 = &last_1;  //pointers to be accessed by SetWaveState()
uint32_t *pl2 = &last_2;
uint32_t *pl3 = &last_3;
uint32_t adc_val = 1; //hold adc read value
uint32_t * LUTs[3] = {Sine_LUT, Square_LUT, Triangle_LUT};  //easy way to switch between waveforms
uint8_t Rx_data[10]; //Array to hold MIDI Rx data
struct Message {
   uint8_t type;  // Note on/off
   uint8_t channel; // Note channel
   uint8_t note;  // Note played
   uint8_t velocity;// Strength of note played
};

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_DAC1_Init(void);
static void MX_TIM2_Init(void);
static void MX_ADC1_Init(void);
static void MX_DAC2_Init(void);
static void MX_TIM3_Init(void);
static void MX_TIM4_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_DAC3_Init(void);
static void MX_OPAMP6_Init(void);
static void MX_DAC4_Init(void);
static void MX_OPAMP3_Init(void);
static void MX_OPAMP4_Init(void);
/* USER CODE BEGIN PFP */
void SetWaveState(uint32_t reset);  // Function to switch between waveforms being played
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void HAL_USART_RxCpltCallback(UART_HandleTypeDef *huart)  //MIDI receive interrupt handling
{
  HAL_UART_Receive_IT(&huart1, Rx_data, 3); //read 3 bytes from MIDI over UART
}

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
```

```
/* USER CODE BEGIN 1 */
//uint32_t i;
uint32_t temp = 0, env1 = 0, env2 = 0, env3 = 0;  //temp to hold ARR value, env for envelope amplitude
uint32_t reset = 1; //used to reset DMA when switching between CV and MIDI
float freq = 0; //for converting midi number to frequency
struct Message midi_in; //structure for midi data

/* USER CODE END 1 */

/* MCU Configuration--------------------------------------------------------*/

/* Reset of all peripherals, Initializes the Flash interface and the Systick. */
HAL_Init();

/* USER CODE BEGIN Init */

/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_DAC1_Init();
MX_TIM2_Init();
MX_ADC1_Init();
MX_DAC2_Init();
MX_TIM3_Init();
MX_TIM4_Init();
MX_USART1_UART_Init();
MX_DAC3_Init();
MX_OPAMP6_Init();
MX_DAC4_Init();
MX_OPAMP3_Init();
MX_OPAMP4_Init();
/* USER CODE BEGIN 2 */

HAL_TIM_Base_Start(&htim2); //start timer for oscillator 1
HAL_TIM_Base_Start(&htim3); //start timer for oscillator 2
HAL_TIM_Base_Start(&htim4); //start timer for oscillator 3

HAL_DAC_Start(&hdac3,DAC_CHANNEL_1);
HAL_DAC_Start(&hdac3,DAC_CHANNEL_2);
HAL_DAC_Start(&hdac4,DAC_CHANNEL_1);

HAL_OPAMP_Start(&hopamp3);  //OP amp follower to output internal DAC to external pin
HAL_OPAMP_Start(&hopamp4);  //OP amp follower to output internal DAC to external pin
HAL_OPAMP_Start(&hopamp6);  //OP amp follower to output internal DAC to external pin
```

```
  HAL_ADC_Start(&hadc1);    //start ADC 1 for inputs to control frequencies

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */

  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

    /*  MIDI Mode */
    if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_8)) {  //Pin C8 will switch between MIDI and CV
      if (reset) {  // Stop all DMA transfers when switching to MIDI mode
        HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_1);
        HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_2);
        HAL_DAC_Stop_DMA(&hdac2, DAC_CHANNEL_1);
        reset = 0;
      }
      SetWaveState(reset);  //On every loop, check pins and set waveform to be output
/*

      for (i = 0; i < NS; i ++) {
        Attack_LUT[i] = i * 32;
      }

      for (i = 0; i < NS; i++) {
        Decay_LUT[i] = 4096 - ((4096 - 1000) * i / NS);
      }

      for (i = 0; i < NS; i++) {
        Env_LUT[i] = env;
      }


      for (i = 0; i < NS; i++) {
        if (i <= attack) {
          ADSR_LUT[i] = i * (4096 / attack); //max value / attack gives # of steps
        }
        else if (i <= attack + decay) {
          ADSR_LUT[i] = 4096 - ((4096 - sustain) * (i - attack) / decay); // same as attack but negative slope, need to s
ubtract attack from i to "reset"
        }
        else if (i <= NS - release) {
          ADSR_LUT[i] = sustain;  // hold sustain
        }
        else {
          ADSR_LUT[i] = (sustain * (NS - i)) / release; //negative slope using remaining points
        }
      }
*/
      HAL_UART_Receive_IT(&huart1, Rx_data, 3); // Receive 3 bytes of MIDI data
      midi_in.type = Rx_data[0] & 0xF0;   // First 4 bits are either note on (9) or note off (8)
      midi_in.channel = Rx_data[0] & 0x0F;  // Next 4 bits are the channel being played on (0-15)
      midi_in.note = Rx_data[1];      // 2nd byte of data is the note being played (0-127)
      midi_in.velocity = Rx_data[2];    // Velocity of note being played (0-127)
      if(midi_in.type == 0x90) {
        freq = pow(2, (((float)midi_in.note-69)/12)) * 440; // Equation used to convert MIDI number to frequency as ta
ken from the University of New South Wales
        switch (midi_in.channel) {  // MIDI will be sent to one of 3 channels, for the 3 oscillators in this project
          case 0: // For each case, update timer frequency and output waveform to the corresponding channel
            HAL_DAC_SetValue(&hdac3, DAC_CHANNEL_1, DAC_ALIGN_12B_R, env1);
            htim2.Instance->ARR = 80000000/((freq) * 128);  // Set frequency of timer
            HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t*)LUTs[*pl1 - 1], 128, DAC_ALIGN_12
B_R);  // Start DMA with lookup table given by SetWaveState()
            //HAL_DAC_Start_DMA(&hdac3, DAC_CHANNEL_1, (uint32_t*)LUTs[0], 128, DAC_ALIGN_12B_R);
            break;
          case 1: // Same as case 0
            HAL_DAC_SetValue(&hdac3, DAC_CHANNEL_2, DAC_ALIGN_12B_R, env2);
            htim3.Instance->ARR = 80000000/((freq) * 128);
            HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_2, (uint32_t*)LUTs[*pl2 - 1], 128, DAC_ALIGN_12
B_R);
            break;
          case 2: // Same as case 0
            HAL_DAC_SetValue(&hdac4, DAC_CHANNEL_1, DAC_ALIGN_12B_R, env3);
            htim4.Instance->ARR = 80000000/((freq) * 128);
            HAL_DAC_Start_DMA(&hdac2, DAC_CHANNEL_1, (uint32_t*)LUTs[*pl3 - 1], 128, DAC_ALIGN_12
B_R);
            break;
        }
      }
      // Check if midi note off is received, or if note velocity is 0. Both ways of depicting note off
      if(midi_in.type == 0x80 || midi_in.velocity == 0x00) {
        switch (midi_in.channel) {
          case 0: // If note off received, stop DMA
            HAL_DAC_SetValue(&hdac3, DAC_CHANNEL_1, DAC_ALIGN_12B_R, 0);
            HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_1);
            break;
          case 1:
            HAL_DAC_SetValue(&hdac3, DAC_CHANNEL_2, DAC_ALIGN_12B_R, 0);
            HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_2);
            break;
          case 2:
            HAL_DAC_SetValue(&hdac4, DAC_CHANNEL_1, DAC_ALIGN_12B_R, 0);
            HAL_DAC_Stop_DMA(&hdac2, DAC_CHANNEL_1);
            break;
        }
      }

      // Poll ADCs for envelope amplitude levels
      if(HAL_ADC_PollForConversion(&hadc1, 1) == HAL_OK) {  //Check if first ADC conversion is ready for env
elope 1
        adc_val = HAL_ADC_GetValue(&hadc1);   //Read ADC value
        env1 = adc_val; // Set envelope amplitude
          HAL_ADC_Start(&hadc1);    //Start ADC to get next conversion
      }
```

```
      if(HAL_ADC_PollForConversion(&hadc1, 1) == HAL_OK) {  //Conversion for envelope 2
        adc_val = HAL_ADC_GetValue(&hadc1);
          env2 = adc_val;
          HAL_ADC_Start(&hadc1);
      }
      if(HAL_ADC_PollForConversion(&hadc1, 1) == HAL_OK) { //Conversion for envelope 3
        adc_val = HAL_ADC_GetValue(&hadc1);
          env3 = adc_val;
          HAL_ADC_Start(&hadc1);
      }

    }



    /*  Control Voltage Mode  */
    if (!HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_8)) {
      reset = 1;  // set reset high and start all DMA with the following waveforms

      HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t*)Sine_LUT, 128, DAC_ALIGN_12B_R);  //
OSCILLATOR 1 start as Sine
      HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_2, (uint32_t*)Square_LUT, 128, DAC_ALIGN_12B_R);
//OSCILLATOR 2 start as Square
      HAL_DAC_Start_DMA(&hdac2, DAC_CHANNEL_1, (uint32_t*)Triangle_LUT, 128, DAC_ALIGN_12B_R);
  //OSCILLATOR 3 start as Triangle

      SetWaveState(reset);  // Check pins for wave state

      //Three *almost* identical if statements to check ADC value and update frequency
      //ADC is configured to have 3 continuous conversions to read 3 different channels.
      if(HAL_ADC_PollForConversion(&hadc1, 1) == HAL_OK) {  //Check if first ADC conversion is ready for osci
llator 1
        adc_val = HAL_ADC_GetValue(&hadc1);   //Read ADC value
        temp =  80000000/((50 + adc_val) * 128);  //Do calculation to get ARR value for timer
        htim2.Instance->ARR = temp;   //Set ARR to change oscillator 1 frequency
        HAL_ADC_Start(&hadc1);    //Start ADC to get next conversion
      }
      if(HAL_ADC_PollForConversion(&hadc1, 1) == HAL_OK) {  //Conversion for oscillator 2
          adc_val = HAL_ADC_GetValue(&hadc1);
          temp =  80000000/((50 + adc_val) * 128);
          htim3.Instance->ARR = temp; //Update timer 3 to change oscillator 2 frequency
          HAL_ADC_Start(&hadc1);
      }
      if(HAL_ADC_PollForConversion(&hadc1, 1) == HAL_OK) { //Conversion for oscillator 3
          adc_val = HAL_ADC_GetValue(&hadc1);
          temp =  80000000/((50 + adc_val) * 128);
          htim4.Instance->ARR = temp; //update oscillator 3 frequency
          HAL_ADC_Start(&hadc1);
      }
    }
  }
  /* USER CODE END 3 */
}

/**
```

```
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

  /** Configure the main internal regulator output voltage
  */
  HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1);
  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = RCC_PLLM_DIV1;
  RCC_OscInitStruct.PLL.PLLN = 10;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the peripherals clocks
  */
  PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART1|RCC_PERIPHCLK_ADC12;
  PeriphClkInit.Usart1ClockSelection = RCC_USART1CLKSOURCE_PCLK2;
  PeriphClkInit.Adc12ClockSelection = RCC_ADC12CLKSOURCE_SYSCLK;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
```

```
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_MultiModeTypeDef multimode = {0};
  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */
  /** Common config
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.GainCompensation = 0;
  hadc1.Init.ScanConvMode = ADC_SCAN_ENABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc1.Init.LowPowerAutoWait = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.NbrOfConversion = 3;
  hadc1.Init.DiscontinuousConvMode = ENABLE;
  hadc1.Init.NbrOfDiscConversion = 1;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  hadc1.Init.OversamplingMode = DISABLE;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure the ADC multi-mode
  */
  multimode.Mode = ADC_MODE_INDEPENDENT;
  if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure Regular Channel
  */
  sConfig.Channel = ADC_CHANNEL_1;
  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_92CYCLES_5;
  sConfig.SingleDiff = ADC_SINGLE_ENDED;
  sConfig.OffsetNumber = ADC_OFFSET_NONE;
  sConfig.Offset = 0;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure Regular Channel
  */
  sConfig.Channel = ADC_CHANNEL_5;
  sConfig.Rank = ADC_REGULAR_RANK_2;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure Regular Channel
  */
  sConfig.Channel = ADC_CHANNEL_6;
  sConfig.Rank = ADC_REGULAR_RANK_3;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

/**
  * @brief DAC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_DAC1_Init(void)
{

  /* USER CODE BEGIN DAC1_Init 0 */

  /* USER CODE END DAC1_Init 0 */

  DAC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN DAC1_Init 1 */

  /* USER CODE END DAC1_Init 1 */
  /** DAC Initialization
  */
  hdac1.Instance = DAC1;
  if (HAL_DAC_Init(&hdac1) != HAL_OK)
  {
    Error_Handler();
  }
  /** DAC channel OUT1 config
  */
  sConfig.DAC_HighFrequency = DAC_HIGH_FREQUENCY_INTERFACE_MODE_AUTOMATIC;
  sConfig.DAC_DMADoubleDataMode = DISABLE;
  sConfig.DAC_SignedFormat = DISABLE;
  sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
  sConfig.DAC_Trigger = DAC_TRIGGER_T2_TRGO;
  sConfig.DAC_Trigger2 = DAC_TRIGGER_NONE;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
  sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_EXTERNAL;
  sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
  if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  /** DAC channel OUT2 config
  */
  sConfig.DAC_Trigger = DAC_TRIGGER_T3_TRGO;
  sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_EXTERNAL;
  if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN DAC1_Init 2 */

  /* USER CODE END DAC1_Init 2 */

}

/**
  * @brief DAC2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_DAC2_Init(void)
{

  /* USER CODE BEGIN DAC2_Init 0 */

  /* USER CODE END DAC2_Init 0 */

  DAC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN DAC2_Init 1 */

  /* USER CODE END DAC2_Init 1 */
  /** DAC Initialization
  */
  hdac2.Instance = DAC2;
  if (HAL_DAC_Init(&hdac2) != HAL_OK)
  {
    Error_Handler();
  }
  /** DAC channel OUT1 config
  */
  sConfig.DAC_HighFrequency = DAC_HIGH_FREQUENCY_INTERFACE_MODE_AUTOMATIC;
  sConfig.DAC_DMADoubleDataMode = DISABLE;
  sConfig.DAC_SignedFormat = DISABLE;
  sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
  sConfig.DAC_Trigger = DAC_TRIGGER_T4_TRGO;
  sConfig.DAC_Trigger2 = DAC_TRIGGER_NONE;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
  sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_EXTERNAL;
  sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
  if (HAL_DAC_ConfigChannel(&hdac2, &sConfig, DAC_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN DAC2_Init 2 */

  /* USER CODE END DAC2_Init 2 */

}

/**
  * @brief DAC3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_DAC3_Init(void)
{

  /* USER CODE BEGIN DAC3_Init 0 */

  /* USER CODE END DAC3_Init 0 */

  DAC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN DAC3_Init 1 */

  /* USER CODE END DAC3_Init 1 */
  /** DAC Initialization
  */
  hdac3.Instance = DAC3;
  if (HAL_DAC_Init(&hdac3) != HAL_OK)
  {
    Error_Handler();
  }
  /** DAC channel OUT1 config
  */
  sConfig.DAC_HighFrequency = DAC_HIGH_FREQUENCY_INTERFACE_MODE_AUTOMATIC;
  sConfig.DAC_DMADoubleDataMode = DISABLE;
  sConfig.DAC_SignedFormat = DISABLE;
  sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
  sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
  sConfig.DAC_Trigger2 = DAC_TRIGGER_NONE;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_DISABLE;
  sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_INTERNAL;
  sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
  if (HAL_DAC_ConfigChannel(&hdac3, &sConfig, DAC_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  /** DAC channel OUT2 config
  */
  sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_INTERNAL;
  if (HAL_DAC_ConfigChannel(&hdac3, &sConfig, DAC_CHANNEL_2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN DAC3_Init 2 */

  /* USER CODE END DAC3_Init 2 */

}

/**
  * @brief DAC4 Initialization Function
  * @param None
  * @retval None
  */
static void MX_DAC4_Init(void)
{

  /* USER CODE BEGIN DAC4_Init 0 */

  /* USER CODE END DAC4_Init 0 */

  DAC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN DAC4_Init 1 */

  /* USER CODE END DAC4_Init 1 */
  /** DAC Initialization
  */
  hdac4.Instance = DAC4;
  if (HAL_DAC_Init(&hdac4) != HAL_OK)
  {
    Error_Handler();
  }
  /** DAC channel OUT1 config
  */
  sConfig.DAC_HighFrequency = DAC_HIGH_FREQUENCY_INTERFACE_MODE_AUTOMATIC;
  sConfig.DAC_DMADoubleDataMode = DISABLE;
  sConfig.DAC_SignedFormat = DISABLE;
  sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
  sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
  sConfig.DAC_Trigger2 = DAC_TRIGGER_NONE;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_DISABLE;
  sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_INTERNAL;
  sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
  if (HAL_DAC_ConfigChannel(&hdac4, &sConfig, DAC_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN DAC4_Init 2 */

  /* USER CODE END DAC4_Init 2 */

}

/**
  * @brief OPAMP3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_OPAMP3_Init(void)
{

  /* USER CODE BEGIN OPAMP3_Init 0 */

  /* USER CODE END OPAMP3_Init 0 */

  /* USER CODE BEGIN OPAMP3_Init 1 */

  /* USER CODE END OPAMP3_Init 1 */
  hopamp3.Instance = OPAMP3;
  hopamp3.Init.PowerMode = OPAMP_POWERMODE_NORMAL;
  hopamp3.Init.Mode = OPAMP_FOLLOWER_MODE;
  hopamp3.Init.NonInvertingInput = OPAMP_NONINVERTINGINPUT_DAC;
  hopamp3.Init.InternalOutput = DISABLE;
  hopamp3.Init.TimerControlledMuxmode = OPAMP_TIMERCONTROLLEDMUXMODE_DISABLE;
  hopamp3.Init.UserTrimming = OPAMP_TRIMMING_FACTORY;
  if (HAL_OPAMP_Init(&hopamp3) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN OPAMP3_Init 2 */

  /* USER CODE END OPAMP3_Init 2 */

}

/**
  * @brief OPAMP4 Initialization Function
  * @param None
  * @retval None
  */
static void MX_OPAMP4_Init(void)
{

  /* USER CODE BEGIN OPAMP4_Init 0 */

  /* USER CODE END OPAMP4_Init 0 */

  /* USER CODE BEGIN OPAMP4_Init 1 */

  /* USER CODE END OPAMP4_Init 1 */
  hopamp4.Instance = OPAMP4;
  hopamp4.Init.PowerMode = OPAMP_POWERMODE_NORMAL;
  hopamp4.Init.Mode = OPAMP_FOLLOWER_MODE;
  hopamp4.Init.NonInvertingInput = OPAMP_NONINVERTINGINPUT_DAC;
  hopamp4.Init.InternalOutput = DISABLE;
  hopamp4.Init.TimerControlledMuxmode = OPAMP_TIMERCONTROLLEDMUXMODE_DISABLE;
  hopamp4.Init.UserTrimming = OPAMP_TRIMMING_FACTORY;
  if (HAL_OPAMP_Init(&hopamp4) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN OPAMP4_Init 2 */

  /* USER CODE END OPAMP4_Init 2 */

}

/**
  * @brief OPAMP6 Initialization Function
  * @param None
  * @retval None
  */
static void MX_OPAMP6_Init(void)
{

  /* USER CODE BEGIN OPAMP6_Init 0 */

  /* USER CODE END OPAMP6_Init 0 */

  /* USER CODE BEGIN OPAMP6_Init 1 */

  /* USER CODE END OPAMP6_Init 1 */
  hopamp6.Instance = OPAMP6;
  hopamp6.Init.PowerMode = OPAMP_POWERMODE_NORMAL;
  hopamp6.Init.Mode = OPAMP_FOLLOWER_MODE;
  hopamp6.Init.NonInvertingInput = OPAMP_NONINVERTINGINPUT_DAC;
  hopamp6.Init.InternalOutput = DISABLE;
  hopamp6.Init.TimerControlledMuxmode = OPAMP_TIMERCONTROLLEDMUXMODE_DISABLE;
  hopamp6.Init.UserTrimming = OPAMP_TRIMMING_FACTORY;
  if (HAL_OPAMP_Init(&hopamp6) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN OPAMP6_Init 2 */

  /* USER CODE END OPAMP6_Init 2 */

}

/**
  * @brief TIM2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM2_Init(void)
```

```
{

  /* USER CODE BEGIN TIM2_Init 0 */

  /* USER CODE END TIM2_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM2_Init 1 */

  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 0;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 624;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM2_Init 2 */

  /* USER CODE END TIM2_Init 2 */

}

/**
  * @brief TIM3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM3_Init(void)
{

  /* USER CODE BEGIN TIM3_Init 0 */

  /* USER CODE END TIM3_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  /* USER CODE BEGIN TIM3_Init 1 */

  /* USER CODE END TIM3_Init 1 */
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 0;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 624;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM3_Init 2 */

  /* USER CODE END TIM3_Init 2 */

}

/**
  * @brief TIM4 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM4_Init(void)
{

  /* USER CODE BEGIN TIM4_Init 0 */

  /* USER CODE END TIM4_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM4_Init 1 */

  /* USER CODE END TIM4_Init 1 */
  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 0;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 624;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM4_Init 2 */

  /* USER CODE END TIM4_Init 2 */

}

/**
  * @brief USART1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART1_UART_Init(void)
{

  /* USER CODE BEGIN USART1_Init 0 */

  /* USER CODE END USART1_Init 0 */

  /* USER CODE BEGIN USART1_Init 1 */

  /* USER CODE END USART1_Init 1 */
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 31250;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart1.Init.ClockPrescaler = UART_PRESCALER_DIV1;
  huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_UARTEx_SetTxFifoThreshold(&huart1, UART_TXFIFO_THRESHOLD_1_8) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_UARTEx_SetRxFifoThreshold(&huart1, UART_RXFIFO_THRESHOLD_1_8) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_UARTEx_DisableFifoMode(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART1_Init 2 */

  /* USER CODE END USART1_Init 2 */

}

/**
  * Enable DMA controller clock
  */
static void MX_DMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_DMAMUX1_CLK_ENABLE();
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Channel1_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel1_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel1_IRQn);
  /* DMA1_Channel2_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel2_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel2_IRQn);
  /* DMA1_Channel3_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel3_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel3_IRQn);

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : LPUART1_TX_Pin LPUART1_RX_Pin */
  GPIO_InitStruct.Pin = LPUART1_TX_Pin|LPUART1_RX_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  GPIO_InitStruct.Alternate = GPIO_AF12_LPUART1;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pins : PC8 PC9 */
  GPIO_InitStruct.Pin = GPIO_PIN_8|GPIO_PIN_9;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /*Configure GPIO pins : PA8 PA9 PA10 PA11
                           PA12 */
  GPIO_InitStruct.Pin = GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10|GPIO_PIN_11
                          |GPIO_PIN_12;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /* EXTI interrupt init*/
  HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

}

/* USER CODE BEGIN 4 */
void SetWaveState(uint32_t reset) {

    uint32_t wave1 = 0, wave2 = 0, wave3 = 0;   //for wave switching
    //Three *almost* identical if statements to check the GPIO pins for
    //each waveform. Two switches for each give us four possible values.
    //wave values -
    //3: Triangle
    //2: Square
    //1: Sine
    //0: No change in waveform

    if(!HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_12)) { //check pin status, pull-up resistor means switch will pull pi
n low
        if(!HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_11)) {
          wave1 = 3;  //If both switches are on (triangle)
        } else {
          wave1 = 2;  //If only first switch is on (Square)
        }
      } else if(!HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_11)) {
        wave1 = 1;    //If only last switch is on (Sine)
      } else {
        wave1 = 0;    //If no switch is on (no change)
      }
      if ((*pl1 != wave1) && wave1) {   //Check that wave isn't 0, and that the value has changed
        HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_1);  //Stop DMA
        if(reset) {
          HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t*)LUTs[wave1 - 1], 128, DAC_ALIGN_12
B_R); //Start DMA with new look up table
        }
        *pl1 = wave1; //set value to remember and to be used in MIDI if needed
      }

      //Same if statement, but for oscillator 2
      if(!HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_10)) {
        if(!HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_9)) {
          wave2 = 3;
        } else {
          wave2 = 2;
        }
      } else if(!HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_9)) {
        wave2 = 1;
      } else {
        wave2 = 0;
      }
      if ((*pl2 != wave2) && wave2) {
        HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_2);
        if (reset) {
          HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_2, (uint32_t*)LUTs[wave2 - 1], 128, DAC_ALIGN_12
B_R);
        }
        *pl2 = wave2;
      }

      //same if statement for oscillator 3
      if(!HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_8)) {
        if(!HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_9)) {
          wave3 = 3;
        } else {
          wave3 = 2;
        }
      } else if(!HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_9)) {
        wave3 = 1;
      } else {
        wave3 = 0;
      }
      if ((*pl3 != wave3) && wave3) {
        HAL_DAC_Stop_DMA(&hdac2, DAC_CHANNEL_1);
        if (reset) {
          HAL_DAC_Start_DMA(&hdac2, DAC_CHANNEL_1, (uint32_t*)LUTs[wave3 - 1], 128, DAC_ALIGN_12
B_R);
        }
        *pl3 = wave3;
      }
```

```
  return;
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */

  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     tex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```