

**Kauno technologijos universitetas**

Elektros ir elektronikos fakultetas

**STM32F4 DSP AUDIO signalo tyrimas**

1 Laboratorinis darbas

|  |
| --- |
|  |
| **Žygimantas Marma, EMEI-2 gr.**  Studentas |
|  |
| **Doc. Šarūnas Kilius**  Dėstytojas |
|  |

**Kaunas, 2023**

**Kaunas, 2023**

Turinys

[1. Dažnių persidengimas (angl. aliasing) 4](#_Toc146647851)

[1.1. Praktinė dalis 4](#_Toc146647852)

[1.2. Rezultatų analizė 5](#_Toc146647853)

[2. Stačiakampės formos signalo tyrimas 6](#_Toc146647854)

[2.1. Praktinė dalis 6](#_Toc146647855)

[2.2. Tyrimas esant 16 kHz diskretizavimo dažniui ir 16 masyvo reikšmių 7](#_Toc146647856)

[2.3. Filtro atsako tyrimas 8](#_Toc146647857)

[3. Išvados 10](#_Toc146647858)

[4. Priedai 11](#_Toc146647859)

**Darbo tikslas**: Susipažinti su DSP signalų generavimu ir analize įterptinėse sistemose panaudojant Audio signalą.

**Laaboratorinio darbo uždaviniai:**

1. Atlikti dažnių persidengimo tyrimą.
2. Sugeneruoti stačiakampio formos signalus.
3. Atlikti analizę keičiant stačiakampio formos signalo parametrus

# Dažnių persidengimas (angl. aliasing)

Patobulinkite pateiktame pavyzdyje esančią sinusinio signalo skaičiavimui skirtą funkciją, įtraukdami kintamajį arba konstantą dažniui nurodyti.

## Praktinė dalis

Matematiškai sinuso banga yra aprašoma šia forma:

,

kur A yra amplitudė, nustatanti bangos maksimalią reikšmę.

f yra dažnis, nustatantis, kiek ciklų atsiranda per vienetą laiko.

t yra laiko kintamasis.

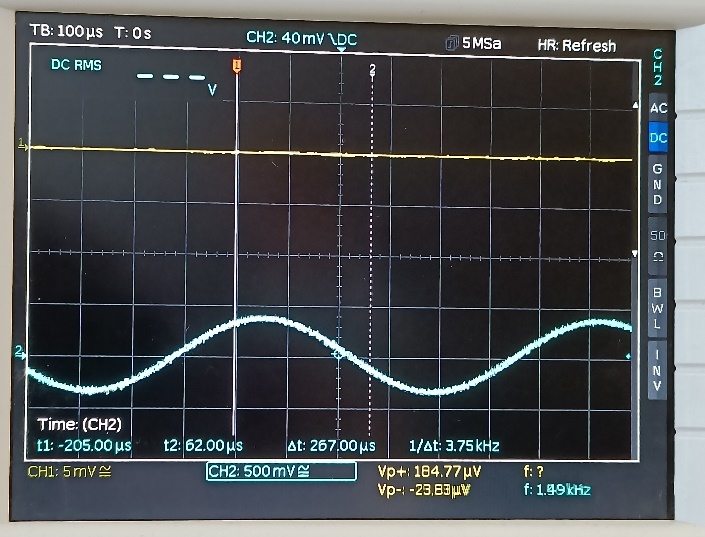
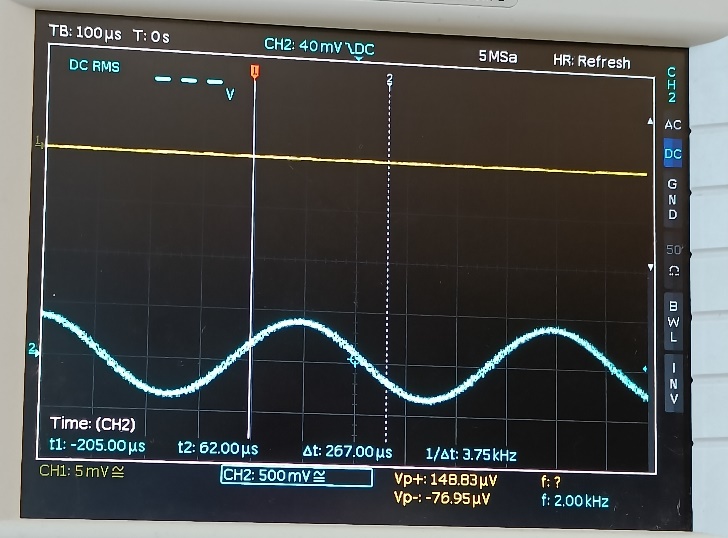
ϕ (fi) yra fazinė kampas, kuris nurodo horizontalų poslinkį bangos laiko ašyje. Ši funkcija realizuota 1 lentelėje pateiktu programiniu kodu.

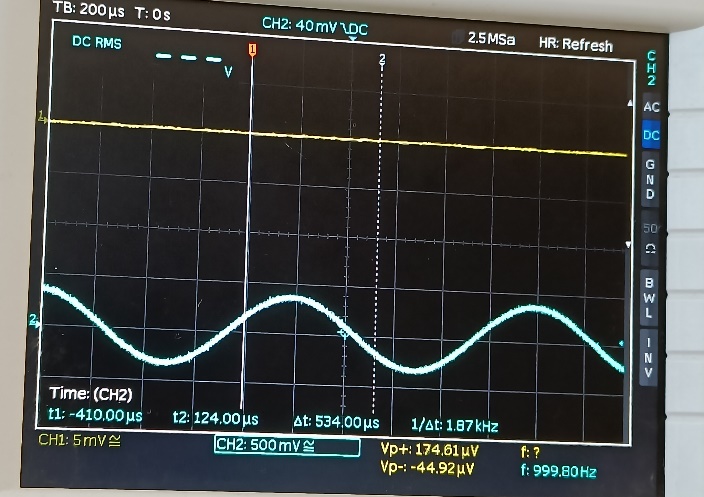
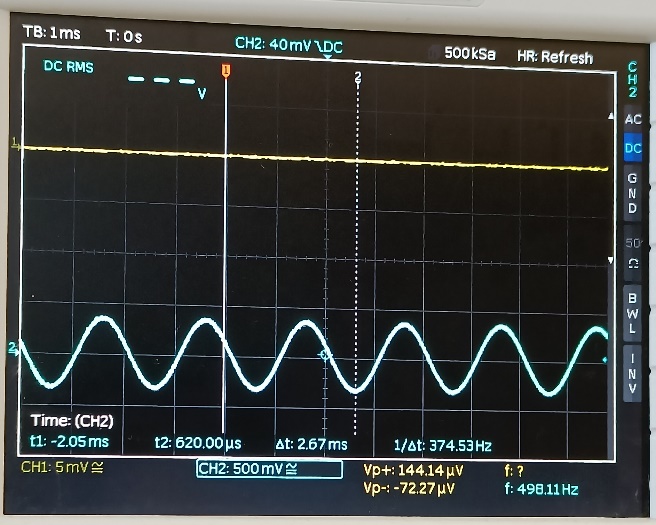
1 lentelė. Programinis kodas generuoti sinusinį signalą

|  |
| --- |
| #define FREQUENCY 1000.0 //Generated signal frequency  #define SAMPLING\_FREQ 8000.0  ...  /\*\*\* Test Signal generation \*\*\*/  for(int index = 0; index < SAMPLES\_QTY; index++)  {  double phase = 2 \* PI \* FREQUENCY \* index / (2 \* SAMPLING\_FREQ);  Left\_out\_buffer[index] = (int16\_t)(30000 \* sin(phase));  Right\_out\_buffer[index] = (int16\_t)(30000 \* sin(2 \* phase));  }  /\*\*\* Fill Output Buffer \*\*\*/  for(int i = 0; i < SAMPLES\_QTY; i++)  {       OutputBuffer[i<<1] = Left\_out\_buffer[i];       OutputBuffer[(i<<1)+1] = Right\_out\_buffer[i];  } |

2 lentelė. Dažnių persidengimo tyrimo rezultatai

|  |  |
| --- | --- |
| **Nurodytas signalo dažnis (Hz)** | **Analoginio išėjimo signalo dažnis (Hz)** |
| 1500 | 1500 |
| 2000 | 2000 |
| 3500 | 3500 |
| 4500 | 3500 |
| 7000 | 1000 |
| 8500 | 500 |

1 pav. Sugeneruoti sinusiniai signalai, viršuje kairėje nustačius 1,5k Hz dažnį, viršuje dešinėje nustačius 2 kHz, apačioje kairėje nustačius 7kHz ir apačioje dešinėje nustačius 8 kHz dažnį.

## Rezultatų analizė

Kaip galima matyti ši problema susijusi su dažnių persidengimu, kuris vadinamas aliasingu. Persidengimas atsiranda, kai diskretizuojant analoginį signalą, t.y., paverčiant jį į diskretųjį laiką naudodami diskretaus signalo apribojimus. Pagrindinė aliasingo priežastis yra nepakankamas diskretizavimo dažnis. **Diskretizuojant signalą reikia užtikrinti, kad diskretizavimo dažnis būtų didesnis nei dvigubai didesnis už didžiausią signalo dažnį, kurį norite išsaugoti. Tai vadinama Nyquisto teorema.** Kitu atveju, signalas gali būti neteisingai atkuriamas dėl dažnių persidengimo.

Pagal anksčiau pateiktą kodą ir tyrimo rezultatus, galima pastebėti, kad dažnių persidengimas atsiranda, kai norima sugeneruoti signalą, kurio dažnis yra didesnis nei pusė diskretinio laiko signalo dažnio (SAMPLING\_FREQ/2).

Šiame tyrime dalis analizuotų signalų, turi didesnius dažnius nei pusė diskretinio laiko signalo dažnio: **4500** Hz ir **7000** Hz dažniai turi viršytą Nyquisto ribą ir yra persidengiantys. Signalas su **8500** Hz dažniu yra virš Nyquisto ribos, todėl jis taip pat persidengia.

# Stačiakampės formos signalo tyrimas

Sugeneruokite stačiakampės formos 1 kHz periodinį signalą su šiomis signalo amplitudės reikšmėmis: {20000, 20000, 20000, 20000, -20000, -20000, -20000, -20000}.

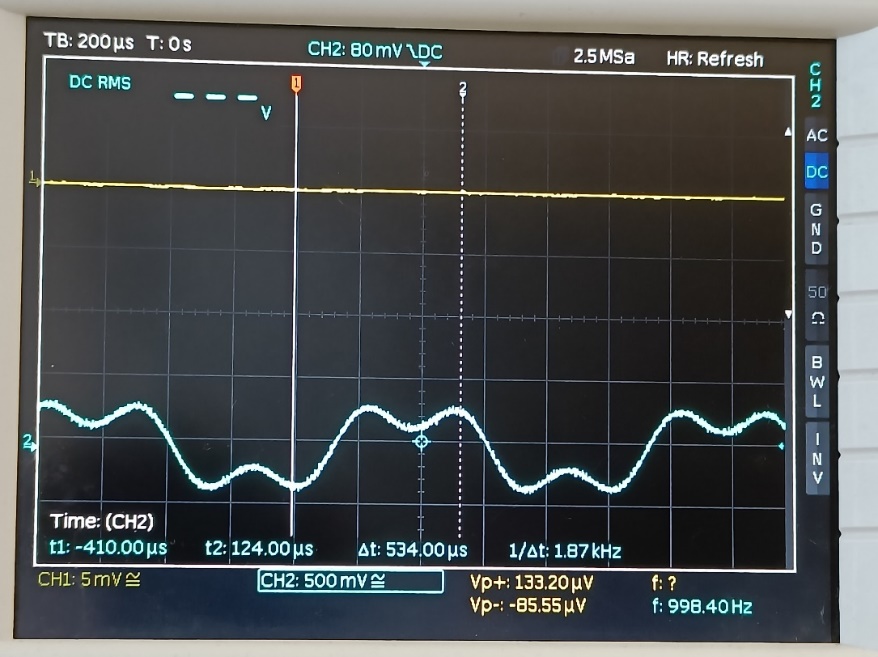
## Praktinė dalis

Stačiakampio signalo generavimas yra pateikiamas 3 lentelėje.

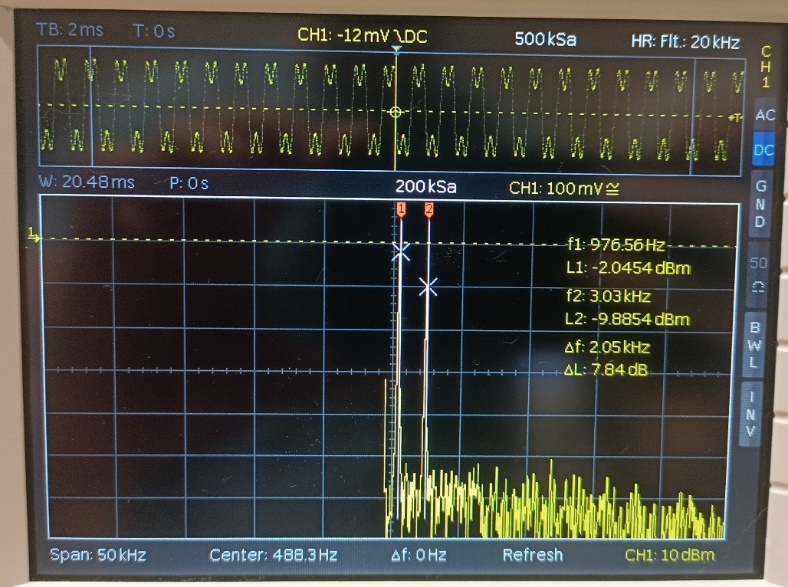
3 lentelė. Programinis kodas generuoti stačiakampį signalą

|  |
| --- |
| #define SAMPLING\_FREQ 8000.0 //BSP\_audio\_out sampling frequency  #define SAMPLES\_QTY 8 //Signal buffer for one channel  int16\_t Square\_wave[SAMPLES\_QTY];  ...  for (int index = 0; index < SAMPLES\_QTY / 2; index++)  {      Square\_wave[index] = 20000;      Square\_wave[index + SAMPLES\_QTY / 2] = -20000;  }  /\*\*\* Test Square Wave generation \*\*\*/  for (int index = 0; index < SAMPLES\_QTY; index++)  {     OutputBuffer[index << 1] = Square\_wave[index];     OutputBuffer[(index << 1) + 1] = Square\_wave[index];  } |

Sugeneruotas signalas matomas 2 paveikslėlyje. Nors signalas yra panašus į stačiakampį, jame vis dar galima matyti bangavimų. Tai lemia per mažas signalo masyvo reikšmių kiekis – šiuo atveju tik 8 elementai.



2 pav. Stačiakampis signalas, signalo masyvo ilgis 8

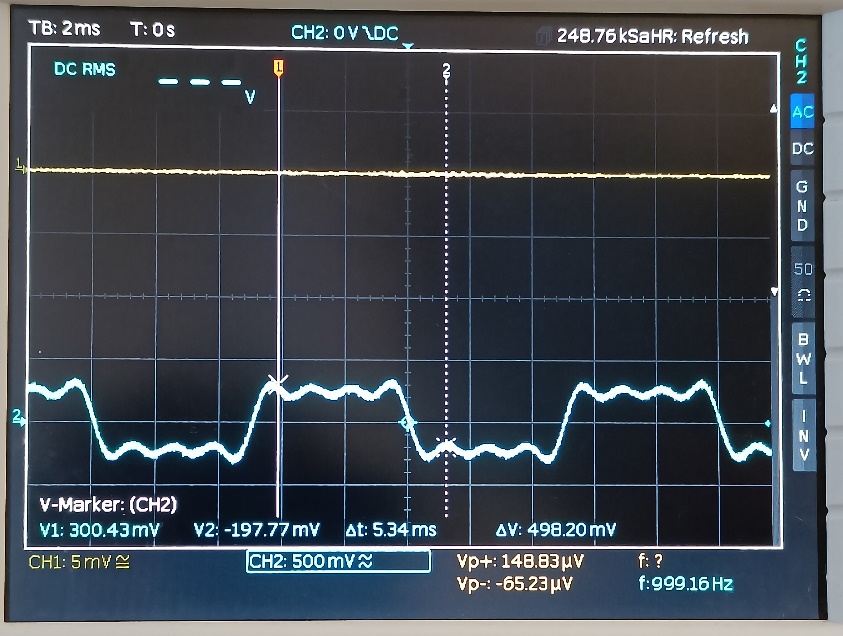


3 pav. Stačiakampio signalo, masyvo ilgis 8, spektras

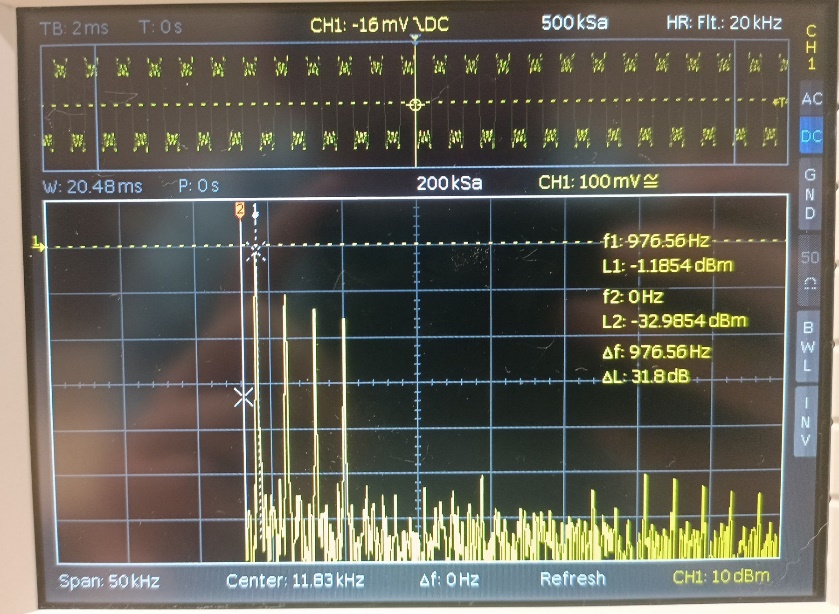
Iš signalo spektro (3 pav.) galima matyti, jog svarbiausios yra harmonikos nelyginės harmonikos pasireiškiančios ties 1kHz, 3kHz, 5kHz.

## Tyrimas esant 16 kHz diskretizavimo dažniui ir 16 masyvo reikšmių

Atlikus šį tyrimą buvo pakeistos diskretizavimo dažnio reikšmę į 16‘000 ir masyvo reikšmių kiekis į 16. Rezultatus galima matyti 4 ir 5 paveikslėliuose. Galima matyti, jog signalas yra labiau panašus į stačiakampį.



4 pav. Stačiakampis signalas, signalo masyvo ilgis 16

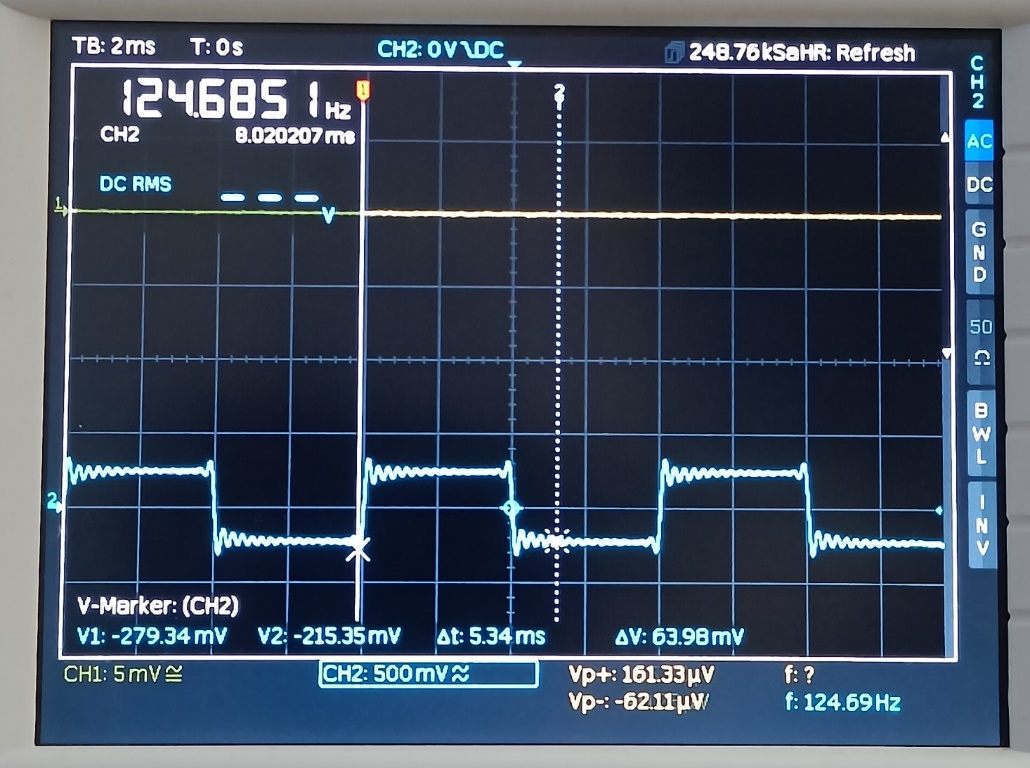


5 pav. Stačiakampio signalo, masyvo ilgis 16, spektras

Palyginus spektrus matomos tos pačios harmonikos (1kHz. 3kHz, 5kHz), tačiau antru atveju (kai signalo masyvo ilgis 16) galima matyti mažiau triukšmo.

## Filtro atsako tyrimas

Sugeneruokite stačiakampės formos signalą, diskretizavimo dažnis 8 kHz, reikšmių kiekis – 64, pirmosios pusės reikšmių amplitudė lygi 20000, likusios pusės amplitudė lygi -20000.



6 pav. Stačiakampis signalas, signalo masyvo ilgis 64, diskretizavimo dažnis 8 kHz

A screen with a graph on it

Description automatically generated

7 pav. Stačiakampio signalo, kurio masyvo ilgis 64 diskretizavimo dažnis 8 kHz, spektras

Iš 6 ir 7 paveikslėlių galima matyti jog šis signalas jau yra praktiškai stačiakampio formos. Taip pat spektre galima matyti jog harmonikos yra labiau išreikštos ties: 125 Hz, 370 Hz (teoriškai turėjo būti 375Hz), 625Hz, 875Hz.

# Išvados

1. Atlikus darbą įsitikinome Nyquisto teorema, kuri nurodo, kad diskretizuojant signalą, diskretizavimo dažnis turi būti dvigubai didesnis už didžiausią signalo dažnį, kurį norite išsaugoti.
2. Tyrimas parodė, kad stačiakampio formos signalas, kurį generuojate, turi bangavimus dėl per mažo signalo masyvo ilgio (šiuo atveju tik 8 elementai). Pakeitus signalo masyvo ilgį į 16 ir diskretizavimo dažnį į 16 kHz, signalas tampa labiau panašus į stačiakampį ir mažiau triukšmo matyti jo spektre.
3. Filtro atsako tyrimas parodė, kad generuojant stačiakampio formos signalą su pakankamai ilgu masyvu (64 elementai) ir pakankamai aukštu diskretizavimo dažniu (8 kHz), signalas tampa praktiškai stačiakampio formos. Šio 125 Hz dažnio signalo spektre galima matyti išreikštą harmonikas, kurių dažniai yra 125 Hz, 375 Hz, 625 Hz ir 875 Hz.

# Priedai

4 lentelė pagrindinis laboratorinio darbo programins kodas

|  |
| --- |
| /\* Includes ------------------------------------------------------------------\*/  #include "main.h"  #include "usb\_device.h"  /\* Private includes ----------------------------------------------------------\*/  /\* USER CODE BEGIN Includes \*/  #include "stm32f4\_discovery\_audio.h"  #include "arm\_math.h"  /\* USER CODE END Includes \*/  /\* Private define ------------------------------------------------------------\*/  /\* USER CODE BEGIN PD \*/  #define SAMPLING\_FREQ 8000      //BSP\_audio\_out sampling frequency  #define SAMPLES\_QTY 32          //Signal buffer for one channel  /\* USER CODE END PD \*/  /\* Private macro -------------------------------------------------------------\*/  /\* Private variables ---------------------------------------------------------\*/  I2C\_HandleTypeDef hi2c1;  I2S\_HandleTypeDef hi2s3;  DMA\_HandleTypeDef hdma\_spi3\_tx;  SPI\_HandleTypeDef hspi1;  /\* USER CODE BEGIN PV \*/  static int16\_t OutputBuffer[SAMPLES\_QTY\*2];         //Buffer to BSP\_AUDIO\_OUT, left+right channels  static int16\_t Left\_out\_buffer[SAMPLES\_QTY];  static int16\_t Right\_out\_buffer[SAMPLES\_QTY];  /\* 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\_I2C1\_Init(void);  static void MX\_I2S3\_Init(void);  static void MX\_SPI1\_Init(void);  /\* USER CODE BEGIN PFP \*/  /\* USER CODE END PFP \*/  /\* Private user code ---------------------------------------------------------\*/  /\* USER CODE BEGIN 0 \*/  /\* USER CODE END 0 \*/  /\*\*    \* @brief  The application entry point.    \* @retval int    \*/  int main(void)  {    /\* USER CODE BEGIN 1 \*/    /\* 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\_I2C1\_Init();    MX\_I2S3\_Init();    MX\_SPI1\_Init();    MX\_USB\_DEVICE\_Init();    /\* USER CODE BEGIN 2 \*/    /\*\*\* Initialize the Audio codec and all related peripherals (I2S, I2C, IOExpander, IOs...) \*\*\*/          if(BSP\_AUDIO\_OUT\_Init(OUTPUT\_DEVICE\_AUTO, 80, SAMPLING\_FREQ) != AUDIO\_OK)          {              Error\_Handler();          }    /\*\*\* Test Signal generation \*\*\*/         for(int index = 0; index < SAMPLES\_QTY; index++)         {            Left\_out\_buffer[index] = (int16\_t)(30000\*sin(2\*PI\*index/SAMPLES\_QTY));            Right\_out\_buffer[index] = (int16\_t)(30000\*sin(4\*2\*PI\*index/SAMPLES\_QTY));         }    /\*\*\* Fill Output Buffer \*\*\*/         for(int i = 0; i < SAMPLES\_QTY; i++)         {            OutputBuffer[i<<1] = Left\_out\_buffer[i];            OutputBuffer[(i<<1)+1] = Right\_out\_buffer[i];         }    /\*\*\* Start Audio play \*\*\*/         BSP\_AUDIO\_OUT\_Play((uint16\_t\*)&OutputBuffer[0], SAMPLES\_QTY\*2);    //Test - application is running         HAL\_GPIO\_WritePin(LD4\_GPIO\_Port, LD4\_Pin, GPIO\_PIN\_SET);      /\* USER CODE END 2 \*/    /\* Infinite loop \*/    /\* USER CODE BEGIN WHILE \*/    while (1)    {        HAL\_GPIO\_TogglePin(LD4\_GPIO\_Port, LD4\_Pin);        HAL\_Delay(500);      /\* USER CODE END WHILE \*/      /\* USER CODE BEGIN 3 \*/    }    /\* USER CODE END 3 \*/  } |