

DSP Education Kit

LAB 1

Analog Input and Output

Issue 1.0

Contents

1	Introduction.....	1
1.1	Lab overview	1
2	Requirements	1
2.1.1	STM32F746G Discovery board	1
3	Basic Digital Signal Processing System.....	2
4	Basic Analogue Input and Output Using the STM32F746G Discovery Board	3
4.1	Program operation of stm32f7_loop_DMA.c	7
4.2	Running the program	7
5	Delaying the Signal	8
6	Creating a Fading Echo Effect	12
6.1.1	Exercise	16
7	Real-Time Sine Wave Generation.....	17
7.1	Program operation.....	17
7.2	Exercise—Modifying the sine wave	20
7.3	Viewing program output using MATLAB.....	21
8	Conclusions.....	23
9	Additional References.....	23

1 Introduction

1.1 Lab overview

The STM32F746G Discovery board is a low-cost development platform featuring a 212 MHz Arm Cortex-M7 floating-point processor. It connects to a host PC via a USB A to mini-b cable and uses the ST-LINK/V2 in-circuit programming and debugging tool. The Keil MDK-Arm development environment, running on the host PC, enables software written in C to be compiled, linked, and downloaded to run on the STM32F746G Discovery board. Real-time audio I/O is provided by a Wolfson WM8994 codec included on the board.

This laboratory exercise introduces the use of the STM32F746G Discovery board and several of the procedures and techniques that will be used in subsequent laboratory exercises.

2 Requirements

To carry out this lab, you will need:

- An STM32F746G Discovery board
- A PC running Keil MDK-Arm
- MATLAB
- An oscilloscope
- Suitable connecting cables
- An audio frequency signal generator
- Optional: External microphone, although you can also use the microphones on the board

2.1.1 STM32F746G Discovery board

An overview of the STM32F746G Discovery board can be found in the Getting Started Guide.

3 Basic Digital Signal Processing System

A basic DSP system that is suitable for processing audio frequency signals comprises a digital signal processor and analogue interfaces as shown in Figure 1. The STM32F746G Discovery board provides such a system, using a Cortex-M7 floating point processor and a WM8994 codec.

The term codec refers to the *coding* of analogue waveforms as digital signals and the *decoding* of digital signals as analogue waveforms. The WM8994 codec performs both the Analogue to Digital Conversion (ADC) and Digital to Analogue Conversion (DAC) functions shown in Figure 1.

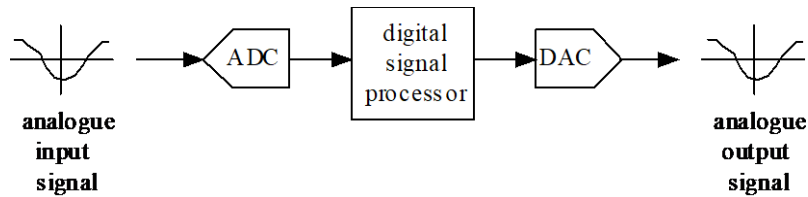


Figure 1: Basic digital signal processing system

Program code may be developed, downloaded, and run on the STM32F746G Discovery board using the *Keil MDK-Arm* integrated development environment. You will not be required to write C programs from scratch, but you will learn how to compile, link, download, and run the example programs provided, and in some cases, make minor modifications to their source files.

You will learn how to use a subset of the features provided by MDK-Arm in order to do this (using the full capabilities of MDK-Arm is beyond the scope of this set of laboratory exercises). The emphasis of this set of laboratory exercises is on the digital signal processing concepts implemented by the programs.

Most of the example programs are quite short, and this is typical of real-time DSP applications. Compared with applications written for general purpose microprocessor systems, DSP applications are more concerned with the efficient implementation of relatively simple algorithms. In this context, efficiency refers to speed of execution and the use of resources such as memory.

The examples in this document introduce some of the features of *MDK-Arm* and the STM32F746G Discovery board. In addition, you will learn how to use *MATLAB* in order to analyze audio signals.

4 Basic Analogue Input and Output Using the STM32F746G Discovery Board

The example source file implements a simple audio “loop-through” using two DMA transfers. First, it programs the on-board digital microphone to sample at 16 kHz and hand its data off to memory via DMA. As soon as one buffer’s worth of samples has arrived, an interrupt-driven callback sets a flag to indicate “record done.” The code then immediately launches a second DMA transfer that feeds that exact buffer into the WM8994 codec’s headphone output. Again, once playback of the buffer completes, another interrupt signals “playback done,” and the cycle repeats. By configuring the SAI peripheral in 2-slot TDM mode, each buffer exactly matches one period of the 16 kHz frame, simplifying the buffer management to “record → wait → play → wait.”

Direct Memory Access (DMA) decouples data movement from the CPU by giving a hardware DMA controller control of the memory bus. You configure DMA channels to shuttle data between peripherals (like microphones or codecs) and RAM without CPU intervention, freeing the core to run other tasks or enter low-power states. In systems like this audio example, DMA dramatically reduces CPU load: rather than copying samples one word at a time in software, the DMA engine automatically handles whole blocks of audio. This concept is demonstrated in the block diagram below:

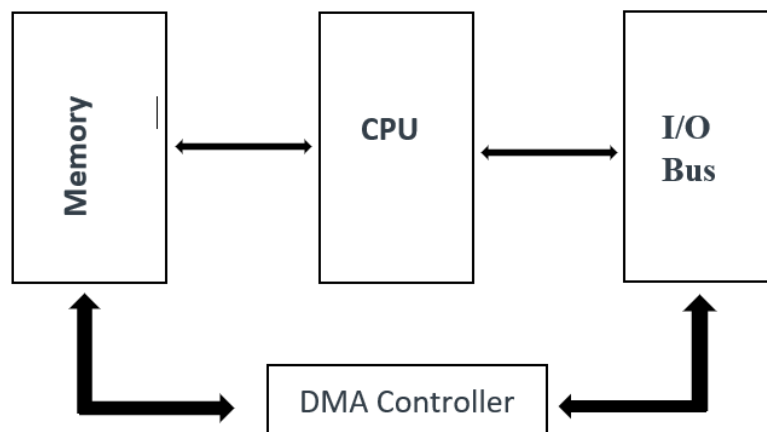


Figure 2: Block diagram representation DMA-Based I/O

```
// stm32f7_loop_DMA.c

/* Includes -----*/
#include "stm32f7_loop_DMA.h"

/* Private typedef -----*/
```

```

/* Private define -----*/
#define SOURCE_FILE_NAME      "stm32f7_loop_DMA.c"

#define AUDIO_FREQ            16000u
#define AUDIO_IN_BIT_RES      16u
#define AUDIO_IN_CHANNEL_NBR  2u
#define BLOCK_SAMPLES_PER_CH  512u
#define BLOCK_SAMPLES_TOTAL    (BLOCK_SAMPLES_PER_CH * AUDIO_IN_CHANNEL_NBR)
#define BUF_SAMPLES            (BLOCK_SAMPLES_TOTAL * 2u)

/* Private macro -----*/
/* Private variables -----*/
static __attribute__((aligned(32))) uint16_t InBuf[BUF_SAMPLES];
static __attribute__((aligned(32))) uint16_t OutBuf[BUF_SAMPLES];

static __IO uint8_t InHalfComplete = 0;
static __IO uint8_t InFullComplete = 0;
static __IO uint8_t OutHalfComplete = 0;
static __IO uint8_t OutFullComplete = 0;

/* Private function prototypes -----*/
static void MPU_Config(void);
static void SystemClock_Config(void);
static void CPU_CACHE_Enable(void);
static void Error_Handler(void);
void BSP_AUDIO_OUT_ClockConfig(SAI_HandleTypeDef *hsai, uint32_t AudioFreq, void *Params);

/* Private functions -----*/
void BSP_AUDIO_IN_HalfTransfer_CallBack(void)
{
    InHalfComplete = 1;
}

void BSP_AUDIO_IN_TransferComplete_CallBack(void)
{
    InFullComplete = 1;
}

```

```

void BSP_AUDIO_OUT_TransferComplete_CallBack(void)
{
    OutFullComplete = 1;
}

int main(void)
{
    /* Configure the MPU attributes */
    MPU_Config();

    /* Enable the CPU Cache */
    CPU_CACHE_Enable();

    HAL_Init();

    /* Configure the System clock to have a frequency of 216 MHz */
    SystemClock_Config();

    stm32f7_LCD_init(AUDIO_FREQ, SOURCE_FILE_NAME, NOGRAPH);

    if (BSP_AUDIO_IN_OUT_Init(INPUT_DEVICE_DIGITAL_MICROPHONE_2,
                              OUTPUT_DEVICE_HEADPHONE,
                              AUDIO_FREQ,
                              AUDIO_IN_BIT_RES,
                              AUDIO_IN_CHANNEL_NBR) != AUDIO_OK)
    {
        Error_Handler();
    }

    BSP_AUDIO_OUT_SetAudioFrameSlot(CODEC_AUDIOFRAME_SLOT_02);

    if (BSP_AUDIO_OUT_Play(OutBuf, sizeof(OutBuf)) != AUDIO_OK)
    {
        Error_Handler();
    }
}

```

```

}

/* Start IN with ping-pong buffer. Size is in HALF-WORDS (uint16_t). */
if (BSP_AUDIO_IN_Record(InBuf, BUF_SAMPLES) != AUDIO_OK)
{
    Error_Handler();
}

/* Forever loopback: copy halves from IN -> OUT as they become ready */
while (1)
{
    /* Copy first half when recorded */
    if (InHalfComplete)
    {
        InHalfComplete = 0;

        memcpy(OutBuf, InBuf, BLOCK_SAMPLES_TOTAL * sizeof(uint16_t));

    }

    /* Copy second half when recorded */
    if (InFullComplete)
    {
        InFullComplete = 0;

        memcpy(OutBuf + BLOCK_SAMPLES_TOTAL,
               InBuf + BLOCK_SAMPLES_TOTAL,
               BLOCK_SAMPLES_TOTAL * sizeof(uint16_t));

    }
}
}

```


4.1 Program operation of stm32f7_loop_DMA.c

In `stm32f7_loop_DMA.c`, the program configures the MPU, enables instruction and data caches, calls `HAL_Init()`, and sets the system clock to 216 MHz. The LCD is initialized to display the source filename and 16 kHz sample rate. Audio is brought up full-duplex: `BSP_AUDIO_IN_OUT_Init()` selects the WM8994's digital-microphone pair (`INPUT_DEVICE_DIGITAL_MICROPHONE_2`) as input and headphones as output at 16 kHz, 16-bit, stereo. Headphone TDM is forced to the 0&2 slots with `BSP_AUDIO_OUT_SetAudioFrameSlot(CODEC_AUDIOFRAME_SLOT_02)`. Playback is primed with `BSP_AUDIO_OUT_Play()` on a ping-pong output buffer, then capture starts with `BSP_AUDIO_IN_Record()` on a matching ping-pong input buffer (512 samples per channel per half). From there, DMA does the work: when the input buffer is half full, `BSP_AUDIO_IN_HalfTransfer_CallBack()` copies that half into the output buffer; when the input buffer is full, `BSP_AUDIO_IN_TransferComplete_CallBack()` copies the second half. The main loop remains idle while the callbacks continuously shuttle audio, yielding a steady 16 kHz → 16 kHz loopback with block-level latency and minimal CPU load—an ideal scaffold for inserting per-block real-time DSP inside the two callbacks.

4.2 Running the program

The following steps assume that you have followed all the steps described in the **Getting Started Guide** provided with the labs.

To run the `stm32f7_loop_DMA.c` program, follow these steps:

1. Open µVision 5 project `Digital-Signal-Processing-Labs\Lab01_AnalogIO\Projects\STM32746G-Discovery\Lab01_AnalogIO\MDK-ARM\Project.uvprojx` by double-clicking on its icon, similar to the one used in the **Getting Started Guide**.
2. You should now see a project structure like that shown in the following figure.

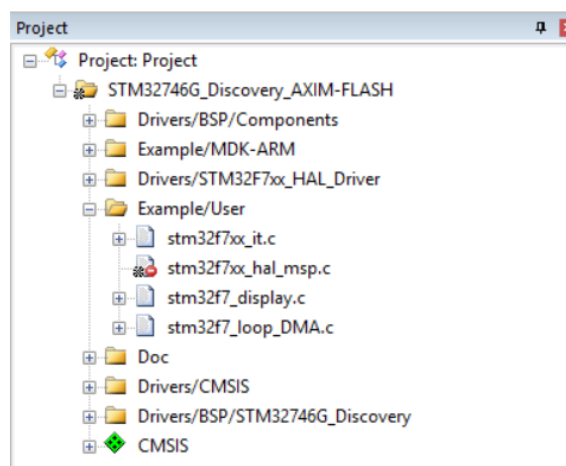


Figure 2: Screenshot of MDK-Arm showing the project

3. Connect the STM32F746 Discovery board to the host PC using a USB A to mini-b cable.
4. Plug the headphones into the headset jack socket (CN10) on the board.




5. Build the project by selecting the **Project > Build target** or by clicking on the **Build** toolbar button .
6. After successfully building the project with no errors, switch to the debugger mode (and download the executable code into flash memory) by clicking on the **Start/Stop Debug Session** toolbar button .
7. Once the **Debugger View** has appeared, click on the **Run** toolbar button .
8. Once the program is running, you should see a start screen on the LCD on the board as shown in Figure 3. You should be able to hear sounds picked up by the digital microphones on the STM32F746 Discovery board (micro right and micro left on the right side of the LCD screen as shown in Figure 3). Depending on the characteristics of the headphones you are using, the sound may be loud or quiet. If you cannot hear anything, try blowing gently onto the microphones.



Figure 3: Start screen for program `stm32f7_loop_DMA.c`

5 Delaying the Signal

By simply delaying and feeding back past samples, you can create echo-like effects with very little code. In our updated `stm32f7_delay.c`, we record audio at **16 kHz (stereo)** into a circular delay line, mix each new sample with the one from **~64 ms** ago, and play the result back—all via DMA for minimal CPU load.

A fixed-size array `DelayBuf` of

`BUF_SAMPLES = (16 000 Hz × 64 ms × 2 ch) / 1000 = 2 048 samples`

holds the delayed audio. As each sample arrives (in the DMA half/full callbacks), the code:

1. Reads the current input sample (`in = buffer[i]`).
2. Retrieves the oldest sample from `DelayBuf[buflptr]`.
3. Sums them, clamping the result to the 16-bit range.

4. Writes the mixed sample to the output half-buffer (out[i]).
5. Stores the current input into DelayBuf[bufptr] (overwriting the oldest).
6. Increments bufptr modulo BUF_SAMPLES so the line wraps around.

This Process-Delay step is invoked by the DMA half/full transfer callbacks (effectively between record and play), giving a consistent **~64 ms single-tap delay** on the Discovery board's headphone output.

```

/* Includes -----
----*/

#include "stm32f7_delay.h"

/* Private typedef -----
----*/

/* Private define -----
----*/

#define SOURCE_FILE_NAME "stm32f7_delay.c"
#define AUDIO_FREQ          16000u
#define AUDIO_IN_BIT_RES    16u
#define AUDIO_IN_CHANNEL_NBR 2u
#define BLOCK_SAMPLES_PER_CH 512u
#define BLOCK_SAMPLES_TOTAL  (BLOCK_SAMPLES_PER_CH *
AUDIO_IN_CHANNEL_NBR)
#define BUF_SAMPLES          (BLOCK_SAMPLES_TOTAL * 2u)

/* Private macro -----
----*/

/* Private variables -----
----*/

static __attribute__((aligned(32))) uint16_t InBuf[BUF_SAMPLES];
static __attribute__((aligned(32))) uint16_t OutBuf[BUF_SAMPLES];
static __attribute__((aligned(32))) uint16_t DelayBuf[BUF_SAMPLES];

static __IO uint8_t InHalfComplete = 0;
static __IO uint8_t InFullComplete = 0;

```

```

static __IO uint8_t OutHalfComplete = 0;
static __IO uint8_t OutFullComplete = 0;

static uint32_t bufptr;

/* Private function prototypes -----
----*/
static void MPU_Config(void);
static void SystemClock_Config(void);
static void CPU_CACHE_Enable(void);
static void Error_Handler(void);
static void ProcessDelay(const uint16_t *in, uint16_t *out, uint32_t
length);
void BSP_AUDIO_OUT_ClockConfig(SAI_HandleTypeDef *hsai, uint32_t
AudioFreq, void *Params);

/* Private functions -----
----*/
void BSP_AUDIO_IN_HalfTransfer_CallBack(void)
{
    ProcessDelay(InBuf, OutBuf, BLOCK_SAMPLES_TOTAL);
}

void BSP_AUDIO_IN_TransferComplete_CallBack(void)
{
    ProcessDelay(InBuf + BLOCK_SAMPLES_TOTAL, OutBuf + BLOCK_SAMPLES_TOTAL,
BLOCK_SAMPLES_TOTAL);
}

static void ProcessDelay(const uint16_t *in, uint16_t *out, uint32_t
length)
{
    for (uint32_t i = 0; i < length; i++) {

```

```

        int16_t delayed = DelayBuf[bufptr];
        int32_t sum = (int32_t)in[i] + (int32_t)delayed;

        out[i] = (int16_t)sum;
        DelayBuf[bufptr] = in[i];
        bufptr = (bufptr + 1) % BUF_SAMPLES;
    }
}

int main(void)
{
    /* Configure the MPU attributes */
    MPU_Config();

    /* Enable the CPU Cache */
    CPU_CACHE_Enable();

    HAL_Init();

    /* Configure the System clock to have a frequency of 216 MHz */
    SystemClock_Config();

    stm32f7_LCD_init(AUDIO_FREQ, SOURCE_FILE_NAME, NOGRAPH);

    if (BSP_AUDIO_IN_OUT_Init(INPUT_DEVICE_DIGITAL_MICROPHONE_2,
                              OUTPUT_DEVICE_HEADPHONE,
                              AUDIO_FREQ,
                              AUDIO_IN_BIT_RES,
                              AUDIO_IN_CHANNEL_NBR) != AUDIO_OK)
    {
        Error_Handler();
    }
}

```

```

}

BSP_AUDIO_OUT_SetAudioFrameSlot(CODEC_AUDIOFRAME_SLOT_02);

if (BSP_AUDIO_OUT_Play(OutBuf, sizeof(OutBuf)) != AUDIO_OK)
{
    Error_Handler();
}

/* Start IN with ping-pong buffer. Size is in HALF-WORDS (uint16_t). */
if (BSP_AUDIO_IN_Record(InBuf, BUF_SAMPLES) != AUDIO_OK)
{
    Error_Handler();
}

/* Infinite loop */
while (1);
}

```

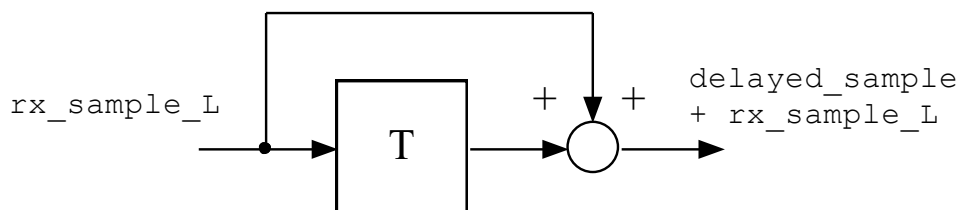


Figure 4: Block diagram representation of program *stm32f7_delay_intr.c*

6 Creating a Fading Echo Effect

By feeding back a fraction of the output of the delay line to its input, a fading echo effect can be realized. Program *stm32f7_echo.c*, shown in the following code snippet, does this.

```

/* Includes -----
----*/

#include "stm32f7_echo.h"

```

```

/* Private typedef -----
----*/

/* Private define -----
----*/

#define SOURCE_FILE_NAME "stm32f7_echo.c"

#define AUDIO_FREQ          16000u
#define AUDIO_IN_BIT_RES    16u
#define AUDIO_IN_CHANNEL_NBR 2u
#define BLOCK_SAMPLES_PER_CH 512u
#define BLOCK_SAMPLES_TOTAL  (BLOCK_SAMPLES_PER_CH *
AUDIO_IN_CHANNEL_NBR)
#define BUF_SAMPLES          (BLOCK_SAMPLES_TOTAL * 2u)

#define DELAY_BUF_SIZE      2000u
#define GAIN                 0.6f

/* Private macro -----
----*/

/* Private variables -----
----*/

static __attribute__((aligned(32))) uint16_t InBuf[BUF_SAMPLES];
static __attribute__((aligned(32))) uint16_t OutBuf[BUF_SAMPLES];

static __IO uint8_t InHalfComplete = 0;
static __IO uint8_t InFullComplete = 0;
static __IO uint8_t OutHalfComplete = 0;
static __IO uint8_t OutFullComplete = 0;

static int16_t DelayBuf[DELAY_BUF_SIZE];
static uint32_t bufptr;

```

```

/* Private function prototypes -----
----*/

static void MPU_Config(void);
static void SystemClock_Config(void);
static void CPU_CACHE_Enable(void);
static void Error_Handler(void);
static void ProcessDelay(const uint16_t *in, uint16_t *out, uint32_t
length);

void BSP_AUDIO_OUT_ClockConfig(SAI_HandleTypeDef *hsai, uint32_t
AudioFreq, void *Params);

/* Private functions -----
----*/

void BSP_AUDIO_IN_HalfTransfer_CallBack(void)
{
    ProcessDelay(InBuf, OutBuf, BLOCK_SAMPLES_TOTAL);
}

void BSP_AUDIO_IN_TransferComplete_CallBack(void)
{
    ProcessDelay(InBuf + BLOCK_SAMPLES_TOTAL, OutBuf + BLOCK_SAMPLES_TOTAL,
BLOCK_SAMPLES_TOTAL);
}

static void ProcessDelay(const uint16_t *in, uint16_t *out, uint32_t
length)
{
    for (uint32_t i = 0; i < length; i++) {
        int16_t delayed = DelayBuf[bufptr];
        int32_t sum = (int32_t)in[i] + (int32_t)delayed * GAIN;

        out[i] = (int16_t)sum;
        DelayBuf[bufptr] = sum;
    }
}

```



```

        bufptr = (bufptr + 1) % DELAY_BUF_SIZE;
    }
}

int main(void)
{
    /* Configure the MPU attributes */
    MPU_Config();

    /* Enable the CPU Cache */
    CPU_CACHE_Enable();

    HAL_Init();

    /* Configure the System clock to have a frequency of 216 MHz */
    SystemClock_Config();

    stm32f7_LCD_init(AUDIO_FREQ, SOURCE_FILE_NAME, NOGRAPH);

    if (BSP_AUDIO_IN_OUT_Init(INPUT_DEVICE_DIGITAL_MICROPHONE_2,
                              OUTPUT_DEVICE_HEADPHONE,
                              AUDIO_FREQ,
                              AUDIO_IN_BIT_RES,
                              AUDIO_IN_CHANNEL_NBR) != AUDIO_OK)
    {
        Error_Handler();
    }

    BSP_AUDIO_OUT_SetAudioFrameSlot(CODEC_AUDIOFRAME_SLOT_02);

    if (BSP_AUDIO_OUT_Play(OutBuf, sizeof(OutBuf)) != AUDIO_OK)

```

```

{
    Error_Handler();
}

/* Start IN with ping-pong buffer. Size is in HALF-WORDS (uint16_t). */
if (BSP_AUDIO_IN_Record(InBuf, BUF_SAMPLES) != AUDIO_OK)
{
    Error_Handler();
}

/* Infinite loop */
while (1);
}

```

6.1.1 Exercise

Experiment with different values of the constants `DELAY_BUF_SIZE` and `GAIN`. What would happen if the value of `GAIN` were made greater than or equal to 1?

1. Study the program listing in `stm32f7_echo.c` and, with reference to Figure 4, draw a block diagram of the system it implements in the space provided below. In the space below that, sketch what you think its response to a unit impulse at time $t = 0$ would be (with a `GAIN` of 0.6 and a `DELAY_BUF_SIZE` size of 2000 samples).

Block diagram representation of program `stm32f7_echo.c`:

Impulse response of program `stm32f7_echo.c` (`DELAY_BUF_SIZE = 2000`, `GAIN = 0.6`):

7 Real-Time Sine Wave Generation

7.1 Program operation

The C source file `stm32f7_sine_lut.c`, shown in the code snippet below, generates a sinusoidal signal using interrupts and a table lookup method.

```
// stm32f7_sine_lut.c

/* Includes -----*/
#include "stm32f7_sine_lut.h"    // your board/HAL declarations

/* Private defines -----*/
#define SOURCE_FILE_NAME      "stm32f7_sine_lut.c"
#define AUDIO_FREQ            8000u
#define LOOPLength            8u

/* Private variables -----*/
static __IO uint8_t  PlayComplete = 0;
static int16_t       sine_table[LOOPLength] = {0, 7071, 10000, 7071, 0, -7071, -10000, -7071};
static int16_t       stereo_buf[LOOPLength * 2];

/* Private function prototypes -----*/
static void MPU_Config(void);
static void SystemClock_Config(void);
static void CPU_CACHE_Enable(void);
static void Error_Handler(void);

int main(void)
{
    /* Configure MPU, enable cache, HAL init, system clock */
    MPU_Config();
    CPU_CACHE_Enable();
    HAL_Init();
```

```

SystemClock_Config();

/* LCD feedback */
stm32f7_LCD_init(AUDIO_FREQ, SOURCE_FILE_NAME, GRAPH);

/* Plot the raw 8-sample LUT on the LCD */
plotSamples(sine_table, LOOPLength, 32);

/* Build interleaved stereo buffer */
for (uint32_t i = 0; i < LOOPLength; i++){
    stereo_buf[2*i] = sine_table[i];          // left slot
    stereo_buf[2*i + 1] = sine_table[i];    // right slot
}

/* Init audio out @8 kHz */
if (BSP_AUDIO_OUT_Init(OUTPUT_DEVICE_HEADPHONE, 70, AUDIO_FREQ) != AUDIO_OK)
    Error_Handler();

/* Force 2-slot (mono/stereo) mode */
BSP_AUDIO_OUT_SetAudioFrameSlot(CODEC_AUDIOFRAME_SLOT_02);

/* Play the 16-sample (8x2) buffer = 8 frames @ 1 kHz tone */
if (BSP_AUDIO_OUT_Play((uint16_t*)stereo_buf, LOOPLength * 2 *
sizeof(int16_t)) != AUDIO_OK){
    Error_Handler();
}

/* Done, just sit and toggle LED on transfer-complete */
while (1){
    if (PlayComplete){
        PlayComplete = 0;
    }
}
}

```

An eight-point lookup table is initialized using the array `sine_table` such that the value of `sine_table[i]` is equal to

$$\text{sine_table}[i] = 10000 \sin((2\pi i/8) + \phi)$$

where in this case, $\phi = 0$. The `LOOPLength` values in array `sine_table` are samples of exactly one cycle of a sinusoid.

In `main()`, after configuring the MPU, enabling caches, initializing HAL, and switching the system clock to 216 MHz, the LCD is brought up to display the filename and sample rate, and `plotSamples()` draws the eight discrete values as a bar graph. Next, we build a simple interleaved stereo buffer by copying each mono sample into both left and right channels. We then initialize the WM8994 codec for 8 kHz output and immediately force a 2-slot TDM frame so that our eight-sample buffer spans eight audio frames (rather than the default four).

Because we output eight samples at an 8 000 Hz rate, the result is a 1 000 Hz sine wave ($8000 \text{ Hz} \div 8 = 1000 \text{ Hz}$). The WM8994's onboard reconstruction filter smooths the discrete steps into a continuous analog waveform, exactly as sketched in the original figure. When you load and run this program, you should first see the start screen and bar-graph of sample values. If you connect the headphone jack to an oscilloscope you will observe a clean 1 kHz sine tone in both the time and frequency domains.

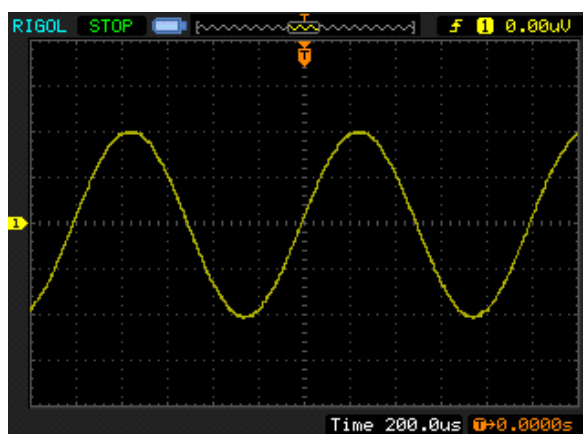


Figure 5: Analog output generated by program `stm32f7_sine_lut.c`

When you run the program, you should see a start screen on the LCD as shown in Figure 6. Press the blue user pushbutton to continue, and you should see on the LCD a graphical representation of the sequence of discrete sample values being written to the DAC (Figure 7). The sample values are represented as bars in the graph on the LCD to emphasize that it is the discrete sample values written to the DAC that are being shown and not the continuous-time signal output by the DAC. Connect one channel of the audio card HEADPHONE OUT output to an oscilloscope and verify that the output signal is a 1 kHz sinusoid using both time-domain and frequency-domain oscilloscope displays.



Figure 6: Start screen for program `stm32f7_sine_lut.c`

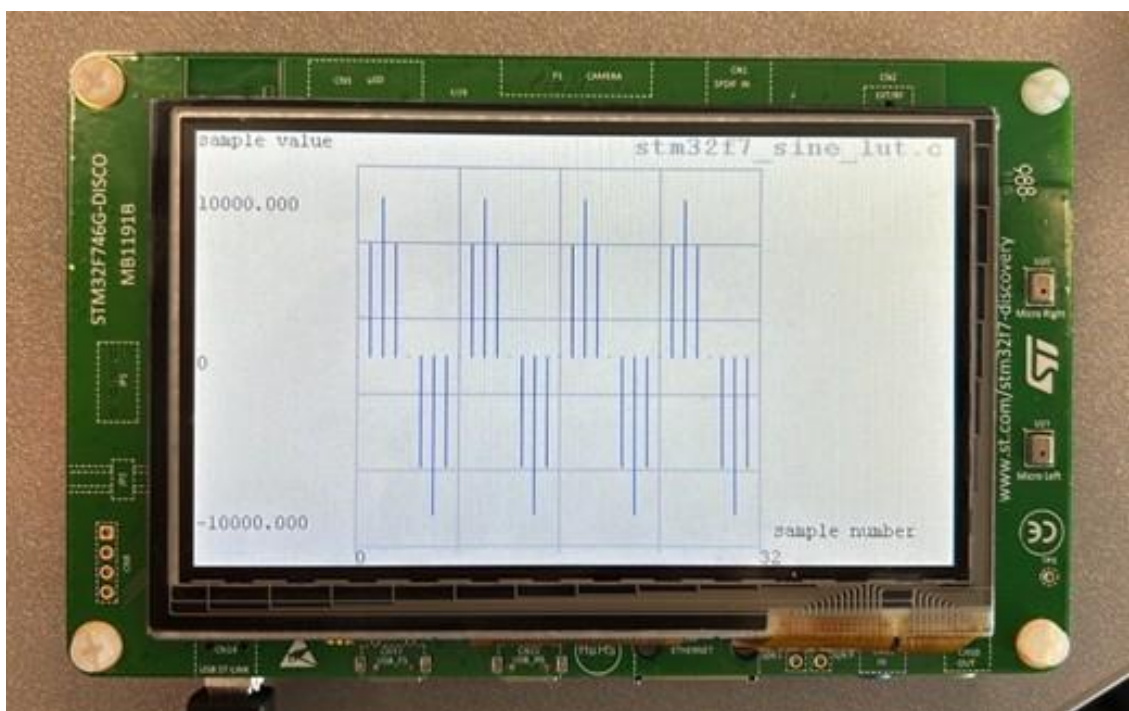


Figure 7: Graphical representation of first 32 sample values output by program `stm32f7_sine_lut.c`

7.2 Exercise—Modifying the sine wave

Edit the source file `stm32f7_sine_lut.c` to generate

1. A 500 Hz sinusoid

2. A 2000 Hz sinusoid

3. A 3000 Hz sinusoid

You should be able to achieve these simply by changing the initialized contents of the array `sine_table` (and by changing the value of the constant `LOOPLENGTH` accordingly). **Do not change any other program statements.** Record the combinations of `LOOPLENGTH` and `sine_table` with which you achieve these results in the space below.

500 Hz sinewave

`LOOPLENGTH =`

`sine_table =`

2000 Hz sinewave

`LOOPLENGTH =`

`sine_table =`

3000 Hz sinewave

`LOOPLENGTH =`

`sine_table =`

7.3 Viewing program output using MATLAB

To view your program output in Matlab, you can first store the output values into a file and then use Matlab to load the values from the saved file.

`stm32f7_sine_lut_buf.c` shows how to store the output values, it is very similar to program `stm32f7_sine_lut.c`, but it also stores the most recent `BUFFER_LENGTH` number of output values in the array `buffer`. Array `buffer` is of type `float32_t` for compatibility with the *MATLAB* function that will be used to view its contents.

To save the program output into a file and view them in Matlab, follow these steps:

1. Run the program and press the user button to start the program.
2. Halt it by clicking on the **Stop** toolbar button in the MDK_Arm debugger.
3. Type the variable name **buffer** as the **Address** in the debugger's **Memory 1** window. Right-click on the **Memory 1** window and set the displayed data type to **Decimal** and **Float** as shown in Figure 8.

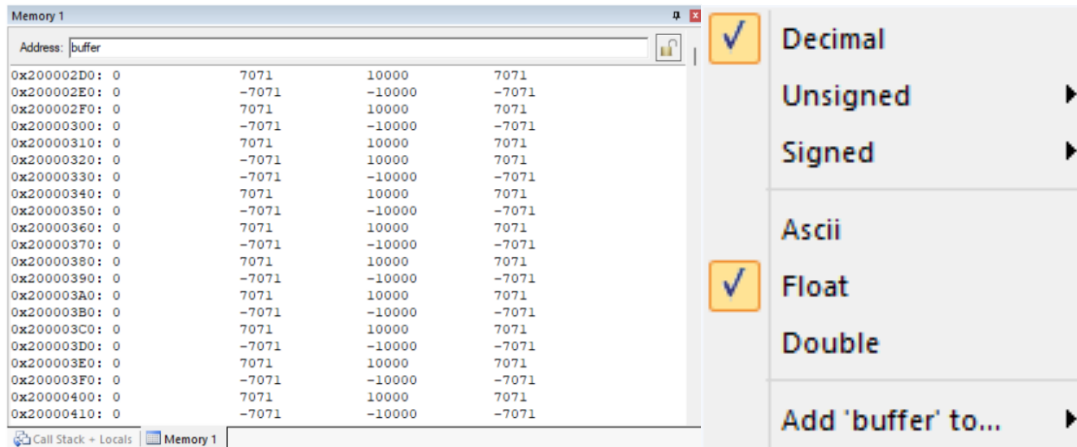


Figure 8: Memory 1 window showing the contents of array `buffer`

The start address of array `buffer` will be displayed in the top left-hand corner of the window.

- Use the following command at the prompt in the debugger's **Command** window to save the contents of array `buffer` to a file in your project folder.

SAVE <filename> <start address>, <end address>

The end address should be the start address plus 0x190 (bytes) representing 100 32-bit sample values. For example,

SAVE sinusoid.dat 0x200002D0, 0x20000460

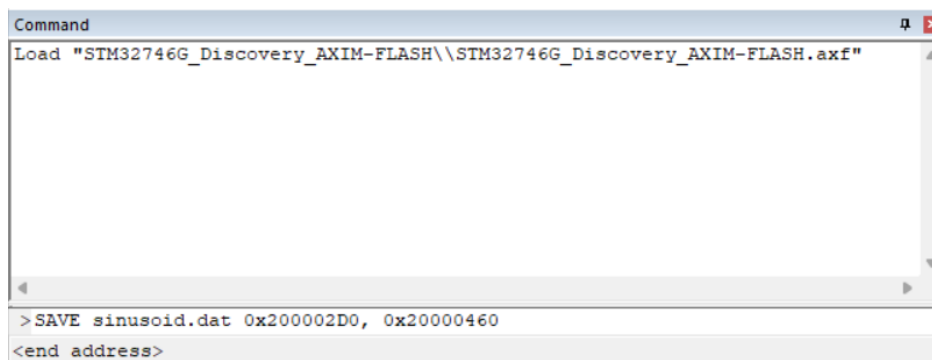


Figure 9: Saving data to file in MDK-Arm

- Launch **MATLAB** and run the **MATLAB** function `stm32f7_logfft.m` (provided with the DSP Education Kit in **Lab01_AnalogIO\Projects\STM32746G-Discovery\Lab05_MATLAB\Matlab_Lab_Files\stm32f7_logfft.m**) to obtain a graphical representation of the contents of the buffer. The **MATLAB** function will require you to input some information, such as the saved `.dat` filename (full path) and sampling frequency.

8 Conclusions

At the end of this exercise, you should have become familiar with several of the tools and techniques that you will use in subsequent lab exercises.

9 Additional References

Link to Board information and resources:

<https://www.st.com/en/evaluation-tools/32f746gdiscovery.html#overview>

Using DMA controllers in STM Discovery boards:

https://www.st.com/content/ccc/resource/technical/document/application_note/27/46/7c/ea/2d/91/40/a9/DM00046011.pdf/files/DM00046011.pdf/jcr:content/translations/en.DM00046011.pdf

For more details about DMA:

<http://cires1.colorado.edu/jimenez-group/QAMSResources/Docs/DMAFundamentals.pdf>