***DSP Education Kit***

**LAB 1   
Analog Input and Output**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc28962171)

[1.1 Lab overview 1](#_Toc28962172)

[2 Requirements 1](#_Toc28962173)

[2.1.1 STM32F746G Discovery board 1](#_Toc28962174)

[3 Basic Digital Signal Processing System 2](#_Toc28962175)

[4 Basic Analogue Input and Output Using the STM32F746G Discovery Board 3](#_Toc28962176)

[4.1 Program operation of stm32f7\_loop\_DMA.c 4](#_Toc28962177)

[4.2 Running the program 5](#_Toc28962178)

[5 Delaying the Signal 8](#_Toc28962181)

[6 Creating a Fading Echo Effect 10](#_Toc28962182)

[6.1.1 Exercise 12](#_Toc28962183)

[7 Real-Time Sine Wave Generation 13](#_Toc28962184)

[7.1 Program operation 13](#_Toc28962185)

[7.2 Exercise—Modifying the sine wave 16](#_Toc28962186)

[7.3 Viewing program output using MATLAB 17](#_Toc28962187)

[8 Conclusions 23](#_Toc28962189)

[9 Additional References 23](#_Toc28962190)

# Introduction

## 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.

# 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

### STM32F746G Discovery board

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

# 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 conceptsimplemented 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.

# 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:

A diagram of a computer hardware system

AI-generated content may be incorrect.

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

// stm32f7\_loop\_DMA.c

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

#include "stm32f7\_loop\_intr.h"

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

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

#define SOURCE\_FILE\_NAME "stm32f7\_loop\_intr.c"

#define AUDIO\_FREQ 16000u

#define AUDIO\_IN\_BIT\_RES 16u

#define AUDIO\_IN\_CHANNEL\_NBR 1u

#define RECORD\_DURATION 10u

#define RECORD\_SAMPLES (AUDIO\_FREQ \* RECORD\_DURATION \* AUDIO\_IN\_CHANNEL\_NBR)

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

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

static uint16\_t RecordBuffer[RECORD\_SAMPLES];

static \_\_IO uint8\_t RecordComplete = 0;

static \_\_IO uint8\_t PlayComplete = 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\_TransferComplete\_CallBack(void)

{

RecordComplete = 1;

}

void BSP\_AUDIO\_OUT\_TransferComplete\_CallBack(void)

{

PlayComplete = 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);

/\* Infinite loop \*/

while (1)

{

/\* Start record \*/

BSP\_AUDIO\_IN\_InitEx(INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2, AUDIO\_FREQ, AUDIO\_IN\_BIT\_RES, AUDIO\_IN\_CHANNEL\_NBR);

BSP\_AUDIO\_IN\_Record(RecordBuffer, RECORD\_SAMPLES);

while (RecordComplete == 0);

BSP\_AUDIO\_IN\_Stop(CODEC\_PDWN\_SW);

RecordComplete = 0;

/\* Start playback \*/

BSP\_AUDIO\_OUT\_Init(OUTPUT\_DEVICE\_HEADPHONE, 70, AUDIO\_FREQ);

/\* Force 2-slot TDM \*/

BSP\_AUDIO\_OUT\_SetAudioFrameSlot(CODEC\_AUDIOFRAME\_SLOT\_02);

BSP\_AUDIO\_OUT\_Play(RecordBuffer, RECORD\_SAMPLES \* sizeof(uint16\_t));

while (PlayComplete == 0);

BSP\_AUDIO\_OUT\_Stop(CODEC\_PDWN\_SW);

PlayComplete = 0;

}

}

## Program operation of stm32f7\_loop\_DMA.c

In **stm32f7\_loop\_dma.c**, the program begins by configuring the MPU, enabling both instruction and data caches, calling HAL\_Init(), and setting the system clock to 216 MHz. The LCD is then initialized to show the source filename and sample rate (without graphing). Audio I/O is performed via DMA: when the record buffer is completely filled, BSP\_AUDIO\_IN\_TransferComplete\_CallBack() fires, and sets a “record complete” flag; similarly, when playback of that buffer finishes, BSP\_AUDIO\_OUT\_TransferComplete\_CallBack() sets a “play complete” flag. In the main loop, the WM8994 codec is first initialized for mono, 16-bit, 16 kHz digital-microphone input, BSP\_AUDIO\_IN\_Record() is called, and the code blocks until the record-complete flag is set, after which the input is stopped. It then re-initializes the codec for headphone output at 16 kHz, forces two-slot TDM mode by calling BSP\_AUDIO\_OUT\_SetAudioFrameSlot(CODEC\_AUDIOFRAME\_SLOT\_02), and invokes BSP\_AUDIO\_OUT\_Play(), blocking until the play-complete flag is set before stopping the output. This sequence repeats indefinitely, delivering a true 16 kHz→16 kHz loopback with minimal CPU intervention and forming a solid template for further real-time DSP work.

## 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.uvprojxby 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.

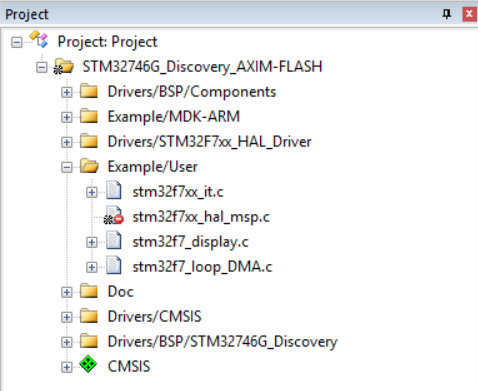


Figure 3: Screenshot of MDK-Arm showing the project

1. Connect the STM32F746 Discovery board to the host PC using a USB A to mini-b cable.
2. Plug the headphones into the headset jack socket (CN10) on the board.
3. Build the project by selecting the **Project > Build target** or by clicking on the ***Build*** toolbar button .
4. 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 .
5. Once the ***Debugger View*** has appeared, click on the ***Run*** toolbar button .
6. Once the program is running, you should see a start screen on the LCD on the board as shown in Figure 4. 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 4). 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.

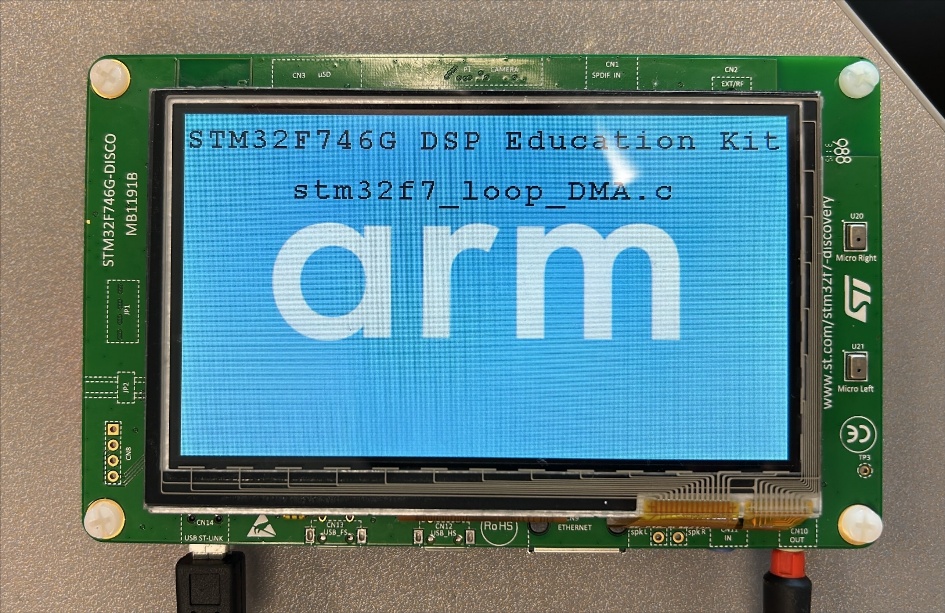


Figure 4: Start screen for program stm32f7\_loop\_DMA.c

# 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 into a circular delay line, mix each new sample with the one from 500 ms ago, and play the result back—all via DMA for minimal CPU load.

A fixed‐size array DelayBuffer of **DELAY\_BUF\_SIZE = (16 000 Hz×500 ms)/1000 = 8 000 samples** holds the delayed audio. As each sample arrives, the code:

1. Reads the current input sample (in = buffer[i]).
2. Retrieves the oldest sample from DelayBuffer[bufptr].
3. Sums them, clamping the result to the 16-bit range.
4. Writes the mixed sample back into buffer[i] for output.
5. Stores the current input into DelayBuffer[bufptr] (overwriting the oldest).
6. Increments bufptr modulo DELAY\_BUF\_SIZE so the line wraps around.

This Process-Delay step sits between DMA record and DMA play in the main loop, giving a consistent 500 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"

/\* Audio parameters \*/

#define AUDIO\_FREQ 16000u

#define AUDIO\_IN\_BIT\_RES 16u

#define AUDIO\_IN\_CHANNEL\_NBR 1u

/\* Delay parameters \*/

#define DELAY\_MS 500u

#define DELAY\_BUF\_SIZE ((AUDIO\_FREQ \* DELAY\_MS) / 1000u)

/\* Recording buffer (for entire capture) \*/

#define RECORD\_DURATION 5u

#define RECORD\_SAMPLES (AUDIO\_FREQ \* RECORD\_DURATION)

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

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

static int16\_t RecordBuffer[RECORD\_SAMPLES];

static int16\_t DelayBuffer[DELAY\_BUF\_SIZE];

static uint32\_t bufptr = 0;

static \_\_IO uint8\_t RecordComplete = 0;

static \_\_IO uint8\_t PlayComplete = 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);

static void ProcessDelay(int16\_t \*buffer, uint32\_t length);

void BSP\_AUDIO\_OUT\_ClockConfig(SAI\_HandleTypeDef \*hsai, uint32\_t AudioFreq, void \*Params);

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

void BSP\_AUDIO\_IN\_TransferComplete\_CallBack(void)

{

RecordComplete = 1;

}

void BSP\_AUDIO\_OUT\_TransferComplete\_CallBack(void)

{

PlayComplete = 1;

}

static void ProcessDelay(int16\_t \*buffer, uint32\_t length)

{

for (uint32\_t i = 0; i < length; i++) {

int16\_t in = buffer[i];

int16\_t delayed = DelayBuffer[bufptr];

int32\_t sum = (int32\_t)in + delayed;

/\* clamp to 16-bit \*/

if (sum > 32767) sum = 32767;

if (sum < -32768) sum = -32768;

buffer[i] = (int16\_t)sum;

DelayBuffer[bufptr] = in;

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);

/\* Infinite loop \*/

while (1)

{

/\* 1) DMA-record RECORD\_SAMPLES @16 kHz from digital mic \*/

BSP\_AUDIO\_IN\_InitEx(INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2, AUDIO\_FREQ, AUDIO\_IN\_BIT\_RES, AUDIO\_IN\_CHANNEL\_NBR);

BSP\_AUDIO\_IN\_Record((uint16\_t\*)RecordBuffer, RECORD\_SAMPLES);

while (!RecordComplete);

BSP\_AUDIO\_IN\_Stop(CODEC\_PDWN\_SW);

RecordComplete = 0;

/\* 2) apply 500 ms delay+feedback \*/

ProcessDelay(RecordBuffer, RECORD\_SAMPLES);

/\* 3) DMA-play back the processed buffer at 16 kHz \*/

BSP\_AUDIO\_OUT\_Init(OUTPUT\_DEVICE\_HEADPHONE, 70, AUDIO\_FREQ);

/\* force 2-slot TDM so buffer aligns 1:1 with audio frames \*/

BSP\_AUDIO\_OUT\_SetAudioFrameSlot(CODEC\_AUDIOFRAME\_SLOT\_02);

BSP\_AUDIO\_OUT\_Play((uint16\_t\*)RecordBuffer, RECORD\_SAMPLES \* sizeof(int16\_t));

while (!PlayComplete);

BSP\_AUDIO\_OUT\_Stop(CODEC\_PDWN\_SW);

PlayComplete = 0;

}

}



Figure 5: Block diagram representation of program stm32f7\_delay\_intr.c

# 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.

// stm32f7\_echo.c

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

#include "stm32f7\_echo.h"

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

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

#define SOURCE\_FILE\_NAME "stm32f7\_echo.c"

/\* Audio parameters \*/

#define AUDIO\_FREQ 16000u

#define AUDIO\_IN\_BIT\_RES 16u

#define AUDIO\_IN\_CHANNEL\_NBR 1u

/\* Delay parameters \*/

#define DELAY\_MS 500u

#define DELAY\_BUF\_SIZE ((AUDIO\_FREQ \* DELAY\_MS) / 1000u)

/\* Echo parameters \*/

#define GAIN 0.3f

/\* Recording buffer (for entire capture) \*/

#define RECORD\_DURATION 5u

#define RECORD\_SAMPLES (AUDIO\_FREQ \* RECORD\_DURATION)

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

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

static int16\_t RecordBuffer[RECORD\_SAMPLES];

static int16\_t DelayBuffer[DELAY\_BUF\_SIZE];

static uint32\_t bufptr = 0;

static \_\_IO uint8\_t RecordComplete = 0;

static \_\_IO uint8\_t PlayComplete = 0;

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

static void MPU\_Config(void);

static void SystemClock\_Config(void);

static void Error\_Handler(void);

static void CPU\_CACHE\_Enable(void);

static void ProcessDelay(int16\_t \*buffer, uint32\_t length);

void BSP\_AUDIO\_OUT\_ClockConfig(SAI\_HandleTypeDef \*hsai, uint32\_t AudioFreq, void \*Params);

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

void BSP\_AUDIO\_IN\_TransferComplete\_CallBack(void)

{

RecordComplete = 1;

}

void BSP\_AUDIO\_OUT\_TransferComplete\_CallBack(void)

{

PlayComplete = 1;

}

static void ProcessDelay(int16\_t \*buffer, uint32\_t length)

{

for (uint32\_t i = 0; i < length; i++) {

int16\_t in = buffer[i];

int16\_t delayed = DelayBuffer[bufptr];

int32\_t sum = (int32\_t)in + delayed\*GAIN;

/\* clamp to 16-bit range \*/

if (sum > 32767) sum = 32767;

if (sum < -32768) sum = -32768;

buffer[i] = (int16\_t)sum;

DelayBuffer[bufptr] = in;

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);

/\* Infinite loop \*/

while (1)

{

BSP\_AUDIO\_IN\_InitEx(INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2, AUDIO\_FREQ, AUDIO\_IN\_BIT\_RES, AUDIO\_IN\_CHANNEL\_NBR);

BSP\_AUDIO\_IN\_Record((uint16\_t\*)RecordBuffer, RECORD\_SAMPLES);

while (!RecordComplete);

BSP\_AUDIO\_IN\_Stop(CODEC\_PDWN\_SW);

RecordComplete = 0;

/\* apply delay effect to the captured buffer \*/

ProcessDelay(RecordBuffer, RECORD\_SAMPLES);

BSP\_AUDIO\_OUT\_Init(OUTPUT\_DEVICE\_HEADPHONE, 70, AUDIO\_FREQ);

BSP\_AUDIO\_OUT\_SetAudioFrameSlot(CODEC\_AUDIOFRAME\_SLOT\_02);

BSP\_AUDIO\_OUT\_Play((uint16\_t\*)RecordBuffer, RECORD\_SAMPLES \* sizeof(uint16\_t));

while (!PlayComplete);

BSP\_AUDIO\_OUT\_Stop(CODEC\_PDWN\_SW);

PlayComplete = 0;

}

}

### Exercise

Experiment with different values of the constants DELAY\_BUF\_SIZE and GAIN (the delay in seconds is equal to DELAY\_BUF\_SIZE divided by the sampling frequency in Hz, and the fraction of the delayed signal fed back is equal to GAIN.)

1. What would happen if the value of GAIN were made greater than or equal to 1?
2. Study the program listing in stm32f7\_echo.c and, with reference to Figure 5, 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):

# Real-Time Sine Wave Generation

## 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 (8�2) 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

where in this case, . 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 Hz ÷ 8 = 1000 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.

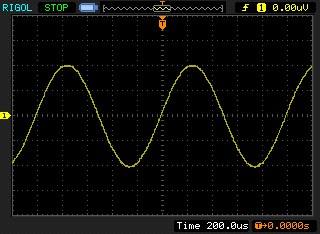


Figure 6: Analog output generated by program stm32f7\_sine\_lut.c

When you run the program, you should see a start screen on the LCD as shown in Figure 7. 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 8). 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 7: Start screen for program stm32f7\_sine\_lut.c

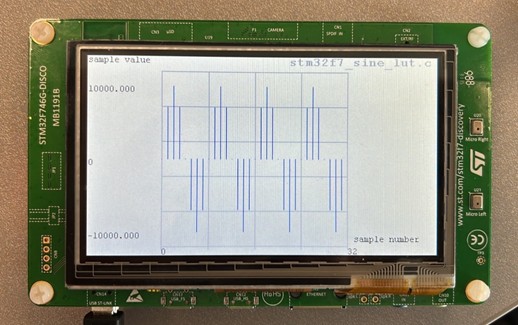


Figure 8: Graphical representation of first 32 sample values output by program stm32f7\_sine\_lut.c

## 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 =

## 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 9.

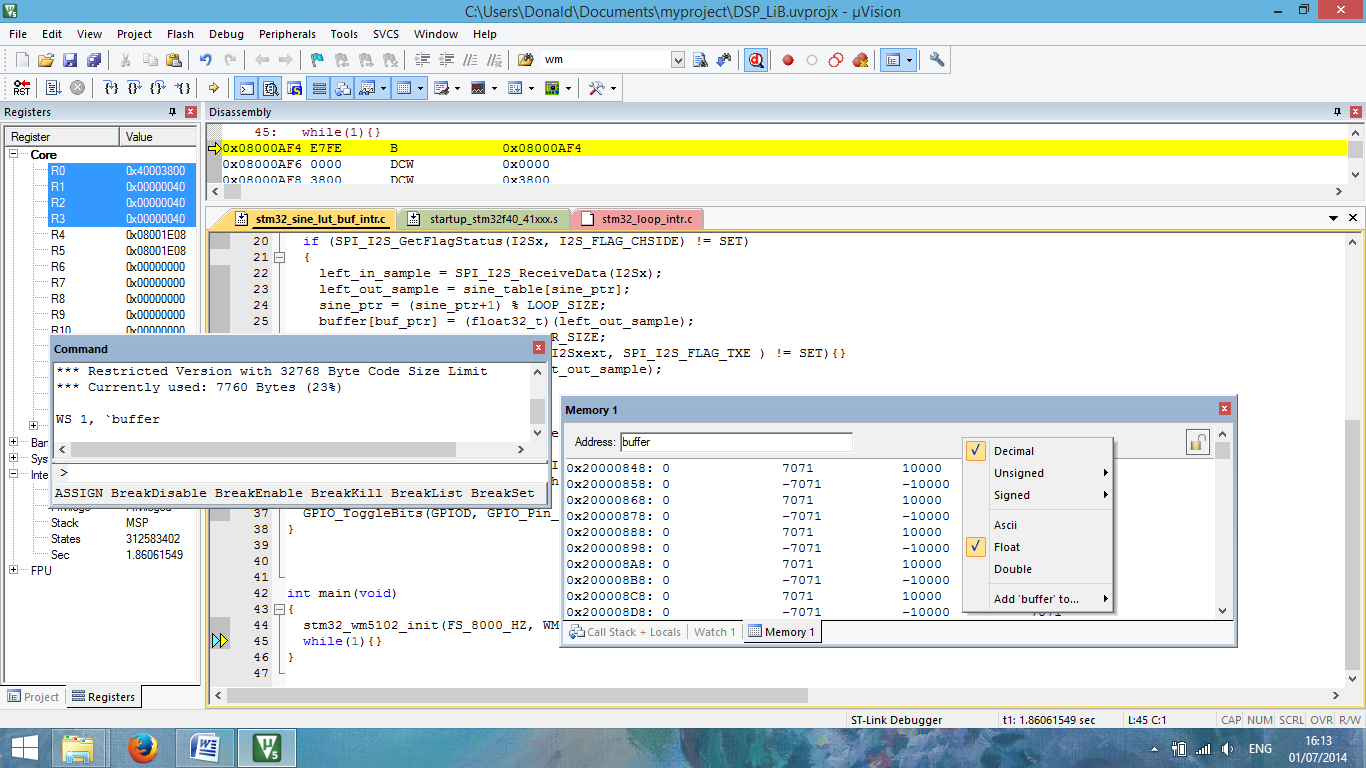
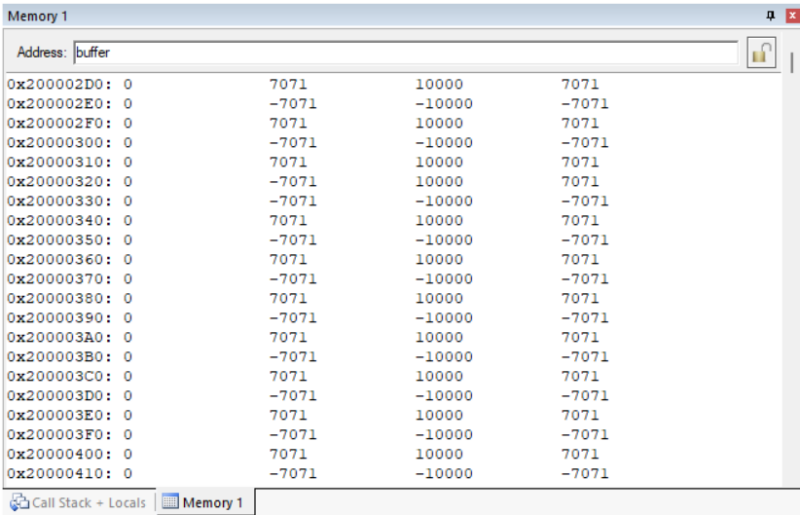


Figure 9: Memory 1 window showing the contents of array buffer

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

1. 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 0×190 (bytes) representing 100 32-bit sample values. For example,

SAVE sinusoid.dat 0×200002D0, 0×20000460

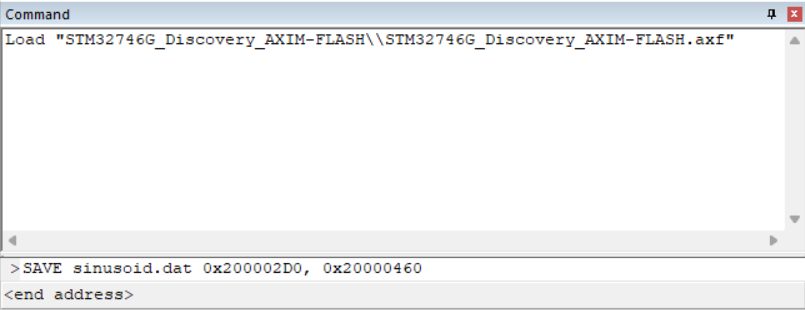


Figure 10: Saving data to file in MDK-Arm

1. 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.

# 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.

# Additional References

**Link to Board information and resources:**

[https://www.st.com/en/evaluation-tools/32f746gdiscovery.html#overview](https://www.st.com/en/evaluation-tools/32f746gdiscovery.html%23overview)

**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>