***DSP Education Kit***

**LAB 1: INSTRUCTOR VERSION  
Analog Input and Output**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc29213579)

[1.1 Lab overview 1](#_Toc29213580)

[2 Requirements 1](#_Toc29213581)

[2.1.1 STM32F746G Discovery board 1](#_Toc29213582)

[3 Basic Digital Signal Processing System 2](#_Toc29213583)

[4 Basic Analogue Input and Output Using the STM32F746G Discovery Board 3](#_Toc29213584)

[4.1 Program operation of stm32f7\_loop\_intr.c 4](#_Toc29213585)

[4.2 Running the program 5](#_Toc29213586)

[4.3 Use of GPIO pin for timing indication 8](#_Toc29213587)

[4.3.1 Connect an oscilloscope probe to pin D13 on connector CN7 to confirm timing 8](#_Toc29213588)

[5 Delaying the Signal 8](#_Toc29213589)

[6 Creating a Fading Echo Effect 10](#_Toc29213590)

[6.1.1 Exercise 12](#_Toc29213591)

[7 Real-Time Sine Wave Generation 13](#_Toc29213592)

[7.1 Program operation 13](#_Toc29213593)

[7.2 Exercise—Modifying the sine wave 17](#_Toc29213594)

[7.3 Viewing program output using MATLAB 17](#_Toc29213595)

[8 DMA-Based Example Program 20](#_Toc29213596)

[9 Conclusions 24](#_Toc29213597)

[10 Additional References 24](#_Toc29213598)

# 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.

Instructor note: Text in red and images in red boxes are for instructors only.

# 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 code snippet below shows a source file for a program that simply copies input samples read from two digital microphones mounted on the board and connected to the WM8994 codec and the WM8994 DAC. In effect, the program connects the digital microphones to the headphone output socket on the board. This simple program is important because many of the other example programs that will be used in subsequent laboratory exercises use the same *interrupt-based*, *real-time* structure. It is worth taking time to ensure that you understand how program **stm32f7\_loop\_intr.c** works, which will also be explained in this document.

In addition, this example introduces the *MDK-Arm* development environment and the editing, compiling, linking, and downloading procedures that you will use again for subsequent examples.

// stm32f7\_loop\_intr.c

#include "stm32f7\_wm8994\_init.h"

#include "stm32f7\_display.h"

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

extern int16\_t rx\_sample\_L;

extern int16\_t rx\_sample\_R;

extern int16\_t tx\_sample\_L;

extern int16\_t tx\_sample\_R;

void BSP\_AUDIO\_SAI\_Interrupt\_CallBack()

{

// when we arrive at this interrupt service routine

// the most recent input sample values are (already) in global

// variables rx\_sample\_L and rx\_sample\_R

// this routine should write new output sample values in

// global variables tx\_sample\_L and tx\_sample\_R

tx\_sample\_L = rx\_sample\_L;

tx\_sample\_R = rx\_sample\_R;

BSP\_LED\_Toggle(LED1);

return;

}

int main(void)

{

stm32f7\_wm8994\_init(AUDIO\_FREQUENCY\_48K,

IO\_METHOD\_INTR,

INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2,

OUTPUT\_DEVICE\_HEADPHONE,

WM8994\_HP\_OUT\_ANALOG\_GAIN\_6DB,

WM8994\_LINE\_IN\_GAIN\_0DB,

WM8994\_DMIC\_GAIN\_17DB,

SOURCE\_FILE\_NAME,

NOGRAPH);

while(1) {}

}

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

The operation of program stm32f7\_loop\_intr.c is as follows.

In function main(), an initialization function stm32f7\_wm8994\_init() is called. This configures the STM32F746G processor and WM8994 codec such that the codec will read (left and right channel) sample values from the digital microphones and interrupt the processor at a sampling frequency determined by the parameter AUDIO\_FREQUENCY\_48K passed to the function.

Parameter INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2 specifies that input to the WM8994 will come from the digital microphones on the STM32F746G Discovery board.

Parameter IO\_METHOD\_INTR passed to function stm32f7\_wm8994\_init()determines that interrupt, as opposed to DMA-based I/O, will be used by the program.

Parameter OUTPUT\_DEVICE\_HEADPHONE is redundant insofar as the headphone socket (CN10) is the only audio output currently supported by the DSP Education Kit.

Parameters WM8994\_HP\_OUT\_ANALOG\_GAIN\_6DB, WM8994\_LINE IN\_GAIN\_0DB, and WM8994\_DMIC\_GAIN\_17DB concern the configuration of programmable gain blocks in the signal path through the codec.

Parameters SOURCE\_FILE\_NAME and NOGRAPH influence what will be shown on the Discovery board’s LCD.

There is no need to understand every detail of the initialization carried out by function stm32f7\_wm8994\_init(). After it has been called, interrupts generated by the Serial Audio Interface (SAI) peripheral in the STM32F746G microcontroller (to which the WM8994 codec is connected) will be enabled, and each time an interrupt occurs, the interrupt service routine function BSP\_AUDIO\_SAI\_Interrupt\_CallBack()will be called. One interrupt will occur per sampling period, and both left and right channel samples are processed in one call to function BSP\_AUDIO\_SAI\_Interrupt\_CallBack().

Following initialization, function main()enters an endless while() loop, doing nothing but waiting for interrupts.

When function BSP\_AUDIO\_SAI\_Interrupt\_CallBack() is called, new input sample values (from the WM8994 codec1) may be read as variables rx\_sample\_L and rx\_sample\_R, and sample values written to variables tx\_sample\_L and tx\_sample\_R will be written to the WM8994 DAC at the next sampling instant.

*1Input sample values may have come either from the analogue LINE IN socket (CN11) on the Discovery board, via the WM8994 ADC, or from the two digital microphones on the Discovery board, via a digital interface on the WM8994.*

## 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\_intr.c program, follow these steps:

1. Open µVision 5 project **DSP\_Education\_Kit** by double-clicking on its icon, similar to the one used in the **Getting Started Guide**.
2. Right-click on the **STM32F746\_DISCOVERY** folder in the **Project** pane and select **Manage Project Items**, after which you should get a window like the one shown below:

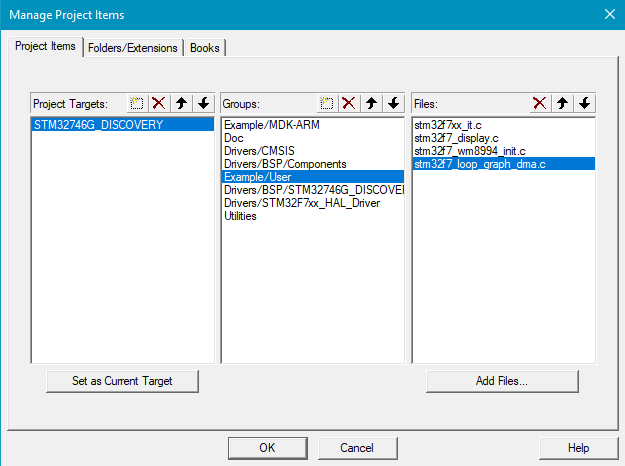


Figure 2: Screenshot of Manage Project Items

1. Delete **stm32f7\_sine\_lut\_intr.c** using the delete icon  on the top right of the **Files** pane and then click on **Add Files.**
2. Find stm32f7\_loop\_intr.c in the **DSP Education Kit\Src** folder and add it to the project. Click **OK**. You should now see a project structure like that shown in the following figure.

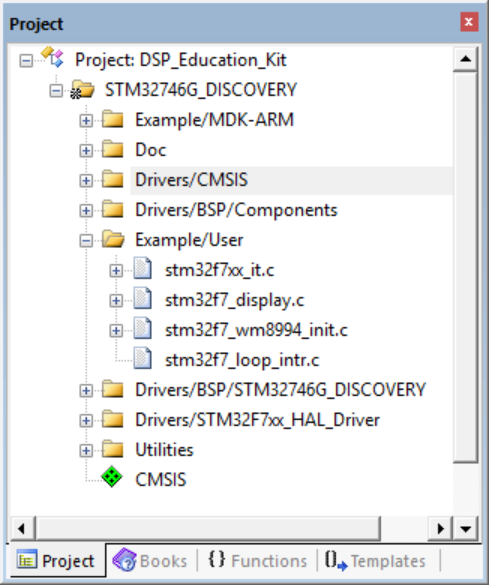


Figure 3: Screenshot of MDK-Arm showing the DSP\_Education\_Kit project

**Note**: Files **stm32f7\_loop\_intr.c**, **stm32f7\_wm8994\_init.c**, **stm32f7xx\_it.c**, and **stm32f7\_display.c** are supplied as part of the DSP Education Kit. Other files making up the project shown in Figure 3 are part of the STM32F746 Discovery board DFP Software Pack.

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.

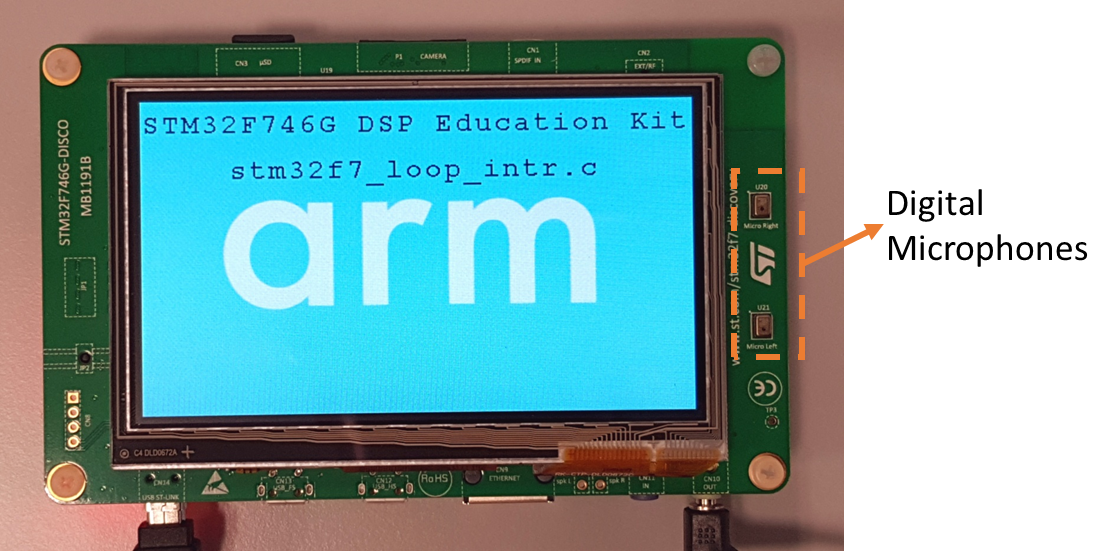


Figure 4: Start screen for program stm32f7\_loop\_intr.c

**Optional**: If you would like to use an external microphone instead of the microphones on the board, you can pass the parameter INPUT\_DEVICE\_INPUT\_LINE\_1 (instead of INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2) to function stm32f7\_wm8994\_init(),you can listen to a signal input either via the LINE IN (CN11) socket on the board or via the digital microphones on the Discovery board. You can do this by editing the source file stm32f7\_loop\_intr.c, re-building the project, downloading, and running the program.

## Use of GPIO pin for timing indication

In several example programs, including stm32f7\_loop\_intr.c, the state (high or low) of a GPIO pin is used so that by connecting an oscilloscope to that pin an indication of the execution of a program may be obtained.

In the case of program stm32f7\_loop\_intr.c, GPIO pin PI1 is toggled each time an interrupt occurs, i.e., in the interrupt service routine BSP\_AUDIO\_SAI\_Interrupt\_CallBack(), using program statement BSP\_LED\_Toggle(LED1).

GPIO pin PI1 is accessible via pin D13 on connector CN7 and drives the green LED (LD1) next to the black reset pushbutton (B2).

Since interrupts should occur once per sampling period, the expected signal on GPIO pin PI1 is a square wave of frequency 24 kHz (if the sampling rate is 48 kHz).

### **Connect an oscilloscope probe to pin D13 on connector CN7 to confirm** timing

GPIO pin PI1 may be set (HIGH) or reset (LOW) using program statements

BSP\_LED\_On(LED1);

BSP\_LED\_Off(LED1);

Alternatively, GPIO pin PI2, accessible via pin D8 on connector CN7 but not connected to an LED, may be set (HIGH), reset (LOW), or toggled using program statements

BSP\_GPIO\_On();

BSP\_GPIO\_Off();

BSP\_GPIO\_Toggle();

# Delaying the Signal

Some simple, yet striking, effects can be achieved simply by delaying the samples as they pass from input to output. Program stm32f7\_delay\_intr.c demonstrates this.

A delay line is implemented using the array buffer to store samples as they are read from the digital microphones. Once the array is full, the pointer bufptr is reset and program overwrites the oldest stored input sample with the newest input sample. Just prior to overwriting the oldest stored input sample in buffer, that sample is retrieved, added to the current input, and written to the WM8994 DAC. The length of the delay is determined by the value of the constant DELAY\_BUF\_SIZE. As supplied, this is equal to 24000 samples, corresponding to a delay of 500 ms at a sampling rate of 48 kHz.

The following code snippet shows the source code of stm32f7\_delay\_intr.c.

#include "stm32f7\_wm8994\_init.h"

#include "stm32f7\_display.h"

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

#define DELAY\_BUF\_SIZE 24000

extern int16\_t rx\_sample\_L;

extern int16\_t rx\_sample\_R;

extern int16\_t tx\_sample\_L;

extern int16\_t tx\_sample\_R;

int16\_t buffer[DELAY\_BUF\_SIZE];

int16\_t bufptr = 0;

void BSP\_AUDIO\_SAI\_Interrupt\_CallBack()

{

// when we arrive at this interrupt service routine (callback)

// the most recent input sample values are (already) in global variables

// rx\_sample\_L and rx\_sample\_R

// this routine should write new output sample values in

// global variables tx\_sample\_L and tx\_sample\_R

int16\_t delayed\_sample;

delayed\_sample = buffer[bufptr];

tx\_sample\_L = delayed\_sample + rx\_sample\_L;

buffer[bufptr] = rx\_sample\_L;

bufptr = (bufptr+1) % DELAY\_BUF\_SIZE;

tx\_sample\_R = tx\_sample\_L;

BSP\_LED\_Toggle(LED1);

return;

}

int main(void)

{

stm32f7\_wm8994\_init(AUDIO\_FREQUENCY\_48K,

IO\_METHOD\_INTR,

INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2,

OUTPUT\_DEVICE\_HEADPHONE,

WM8994\_HP\_OUT\_ANALOG\_GAIN\_6DB,

WM8994\_LINE\_IN\_GAIN\_0DB,

WM8994\_DMIC\_GAIN\_17DB,

SOURCE\_FILE\_NAME,

NOGRAPH);

while(1){}

}



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\_intr.c, shown in the following code snippet, does this.

// stm32f7\_echo\_intr.c

#include "stm32f7\_wm8994\_init.h"

#include "stm32f7\_display.h"

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

#define DELAY\_BUF\_SIZE 6000

#define GAIN 0.6f

extern int16\_t rx\_sample\_L;

extern int16\_t rx\_sample\_R;

extern int16\_t tx\_sample\_L;

extern int16\_t tx\_sample\_R;

int16\_t buffer[DELAY\_BUF\_SIZE];

int16\_t bufptr = 0;

void BSP\_AUDIO\_SAI\_Interrupt\_CallBack()

{

// when we arrive at this interrupt service routine (callback)

// the most recent input sample values are (already) in global variables

// rx\_sample\_L and rx\_sample\_R

// this routine should write new output sample values in

// global variables tx\_sample\_L and tx\_sample\_R

int16\_t delayed\_sample;

delayed\_sample = buffer[bufptr];

tx\_sample\_L = delayed\_sample + rx\_sample\_L;

buffer[bufptr] = rx\_sample\_L + delayed\_sample\*GAIN;

bufptr = (bufptr+1) % DELAY\_BUF\_SIZE;

tx\_sample\_R = 0;

BSP\_LED\_Toggle(LED1);

return;

}

int main(void)

{

stm32f7\_wm8994\_init(AUDIO\_FREQUENCY\_48K,

IO\_METHOD\_INTR,

INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2,

OUTPUT\_DEVICE\_HEADPHONE,

WM8994\_HP\_OUT\_ANALOG\_GAIN\_6DB,

WM8994\_LINE\_IN\_GAIN\_0DB,

WM8994\_DMIC\_GAIN\_17DB,

SOURCE\_FILE\_NAME,

NOGRAPH);

while(1){}

}

### 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?

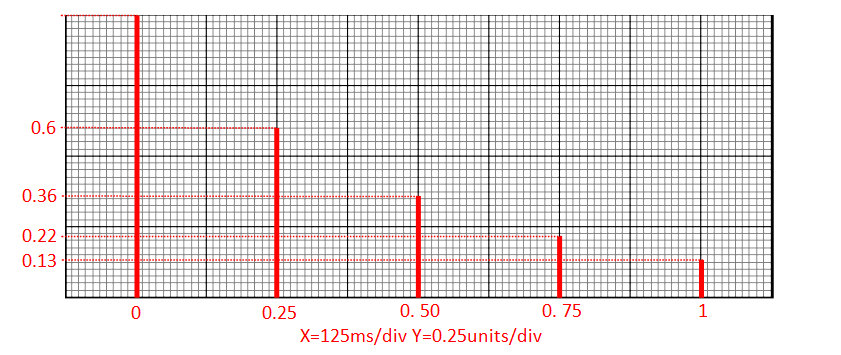
If GAIN=1, then the echo will not fade, it remains the same (repeated). If GAIN>1, then it gets louder at each echo.

1. Study the program listing in stm32f7\_echo\_intr.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\_intr.c:



Impulse response of program stm32f7\_echo\_intr.c (DELAY\_BUF\_SIZE = 2000, GAIN = 0.6):



# Real-Time Sine Wave Generation

## Program operation

The C source file stm32f7\_sine\_lut\_intr.c, shown in the code snippet below, generates a sinusoidal signal using interrupts and a table lookup method.

// stm32f7\_sine\_lut\_intr.c

#include "stm32f7\_wm8994\_init.h"

#include "stm32f7\_display.h"

#define SOURCE\_FILE\_NAME "stm32f7\_sine\_lut\_intr.c"

#define LOOPLENGTH 8

extern int16\_t rx\_sample\_L;

extern int16\_t rx\_sample\_R;

extern int16\_t tx\_sample\_L;

extern int16\_t tx\_sample\_R;

int16\_t sine\_table[LOOPLENGTH] = {0, 7071, 10000, 7071, 0, -7071, -10000, -7071};

int16\_t sine\_ptr = 0; // pointer into lookup table

void BSP\_AUDIO\_SAI\_Interrupt\_CallBack()

{

// when we arrive at this interrupt service routine (callback)

// the most recent input sample values are (already) in global variables

// rx\_sample\_L and rx\_sample\_R

// this routine should write new output sample values in

// global variables tx\_sample\_L and tx\_sample\_R

BSP\_LED\_On(LED1);

tx\_sample\_L = sine\_table[sine\_ptr];

sine\_ptr = (sine\_ptr+1)%LOOPLENGTH;

tx\_sample\_R = tx\_sample\_L;

BSP\_LED\_Off(LED1);

return;

}

int main(void)

{

stm32f7\_wm8994\_init(AUDIO\_FREQUENCY\_8K,

IO\_METHOD\_INTR,

INPUT\_DEVICE\_INPUT\_LINE\_1,

OUTPUT\_DEVICE\_HEADPHONE,

WM8994\_HP\_OUT\_ANALOG\_GAIN\_0DB,

WM8994\_LINE\_IN\_GAIN\_0DB,

WM8994\_DMIC\_GAIN\_9DB,

SOURCE\_FILE\_NAME,

GRAPH);

plotSamples(sine\_table, LOOPLENGTH, 32);

while(1){}

}

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.

Just as in the previous examples, in function main(), initialization function stm32f7\_wm8994\_init() is called. This configures processor and codec such that the WM8994 will sample, and interrupt the processor, at a frequency determined by the parameter value AUDIO\_FREQUENCY\_8K, i.e., in this case at 8 kHz. Interrupts will occur every 0.125 ms.

Following the call to function stm32f7\_wm8994\_init(), function main() enters an endless loop, doing nothing but waiting for interrupts (which will occur once per sampling period).

On interrupt, the interrupt service routine function BSP\_AUDIO\_SAI\_Interrupt\_CallBack() is called, and in that routine, the most important program statements are executed: the sample values read from array sine\_table are written to both channels to the DAC and the index variable sine\_ptr is incremented to point to the next value in the array.

The 1 kHz frequency of the sinusoidal output signal corresponds to the eight samples per cycle output at a rate of 8 kHz.

The WM8994 DAC is effectively a low pass reconstruction filter that interpolates between output sample values to give a continuous sinusoidal analogue output signal as shown in Figure 6. This will be explained further in the next lab exercise.

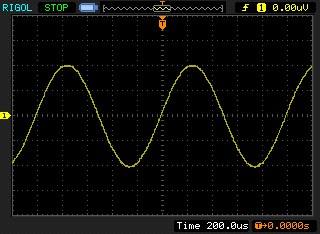


Figure 6: Analog output generated by program stm32f7\_sine\_lut\_intr.c

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.

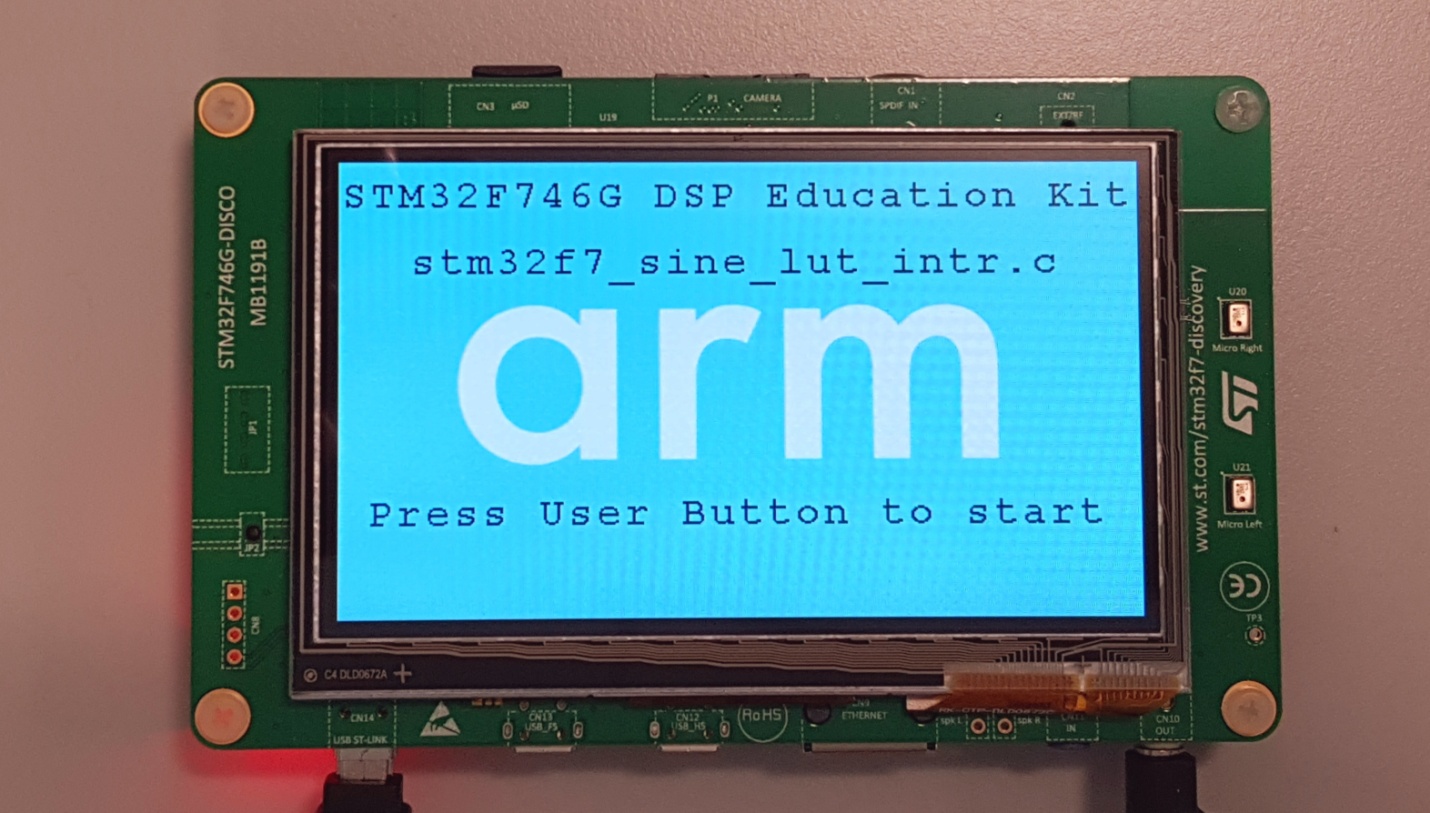


Figure 7: Start screen for program stm32f7\_sine\_lut\_intr.c

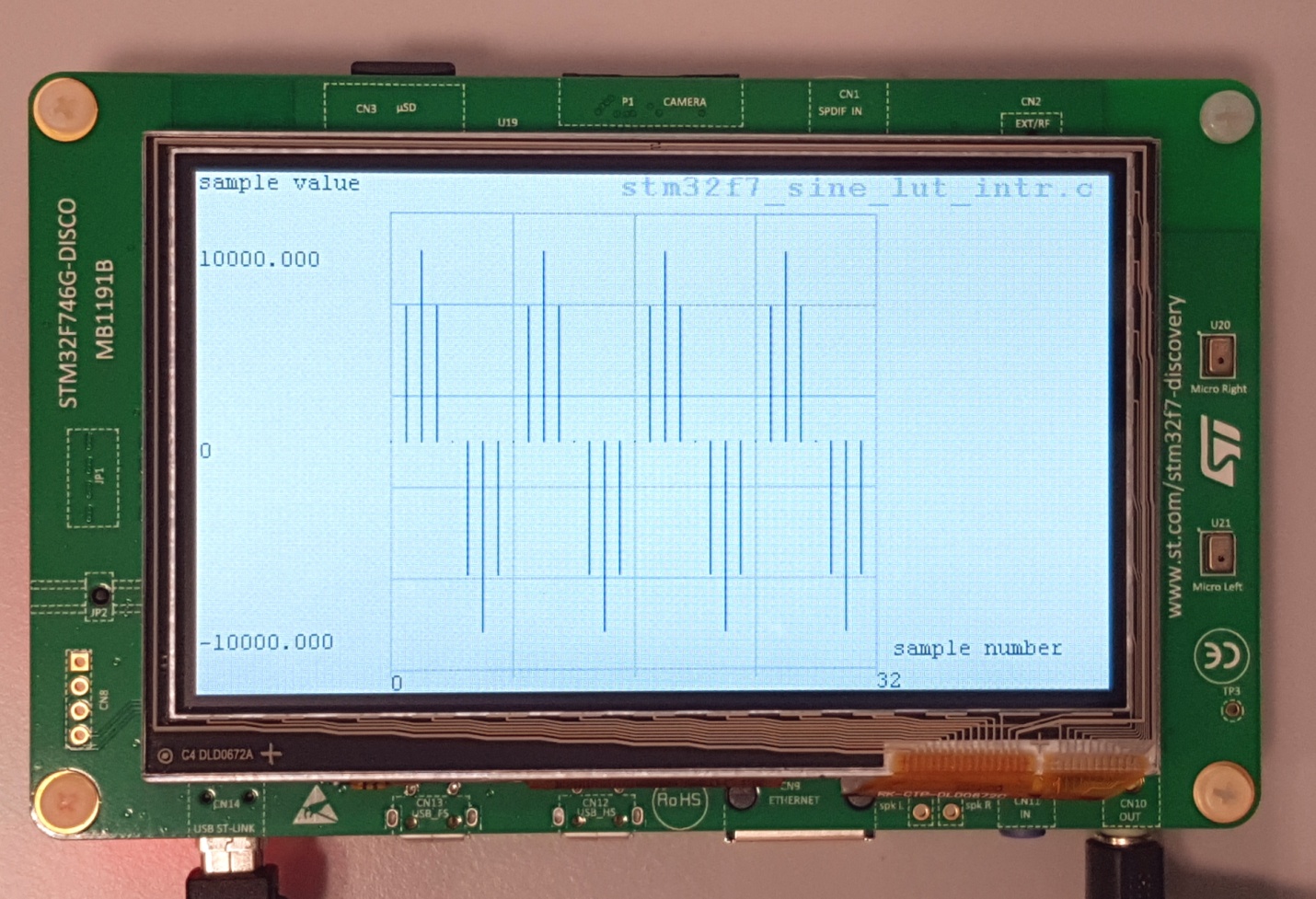


Figure 8: Graphical representation of first 32 sample values output by program stm32f7\_sine\_lut\_intr.c

## Exercise—Modifying the sine wave

Edit the source file stm32f7\_sine\_lut\_intr.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 = 16

sine\_table = {0, 3827, 7071, 9239, 10000, 9239, 7071, 3827, 0, -3827, -7071, -9239, -10000, -9239, -7071, -3827}

**2000 Hz sinewave**

LOOPLENGTH = 4

sine\_table = {0, 10000, 0, -10000}

An infinite number of different correct solutions to this and to the other two problems set here are possible, corresponding to different values of phi. For example, in this case, another (slightly less intuitive) solution is:

{7071, 7071, -7071, -7071}

**3000 Hz sinewave**

LOOPLENGTH = 8

sine\_table = {0, 7071, -10000, 7071, 0, -7071, 10000, -7071}

## 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\_intr.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.

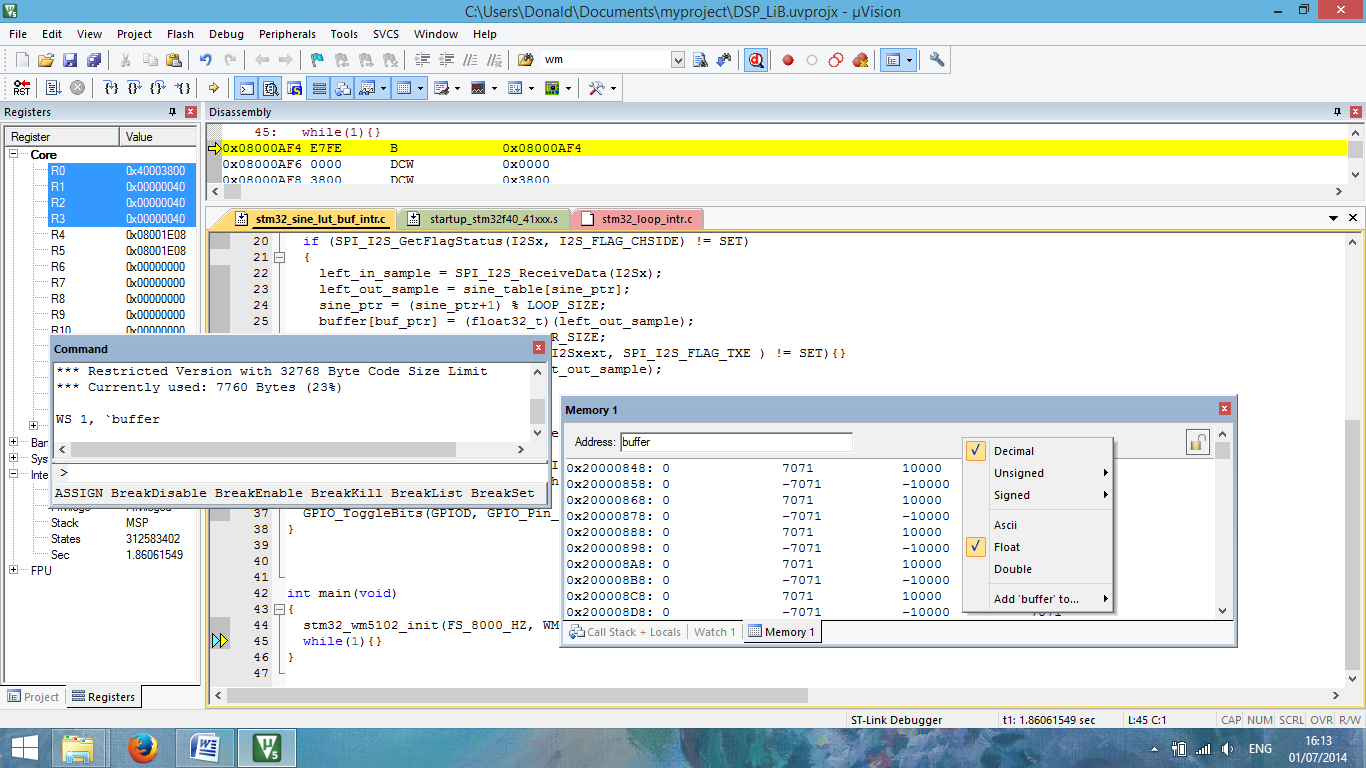
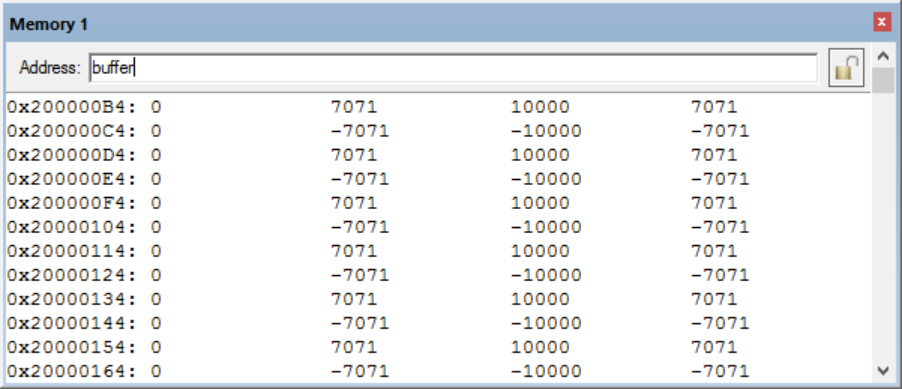


Figure 9: Memory 1 window showing the contents of array buffer

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 0×200000B4, 0×20000244

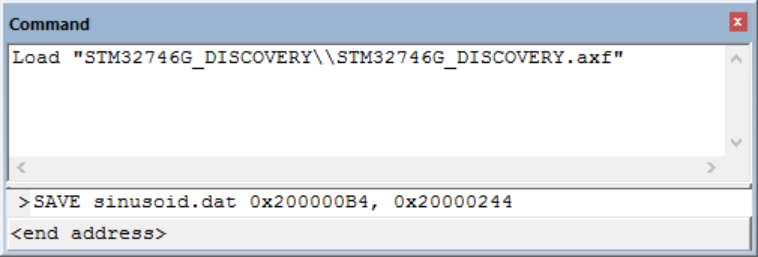
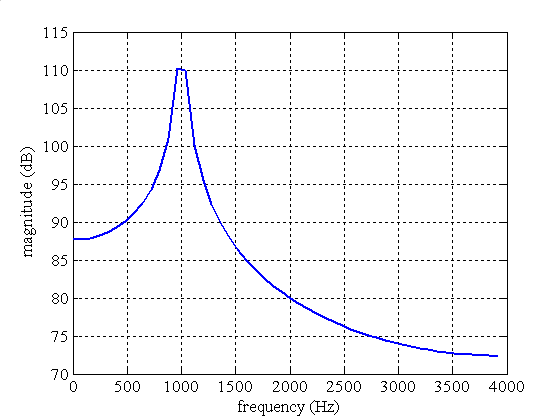


Figure 10: Saving data to file in MDK-Arm

1. Launch *MATLAB* and run the *MATLAB* function stm32f7\_logfft.m (provided with the DSP Education Kit in **General\_Matlab\_Files\**) 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.

There are some subtleties here, linked to the simplicity of function stm32f7\_logfft(). It is designed to read 32-bit floating point values from a file saved in MDK-Arm. Rather than modify the MATLAB function, which is used again in other laboratory exercises, it was deemed easier to convert the 16-bit integer sample values written to the DAC to 32-bit floating point values in program stm32f7\_sine\_lut\_buf\_intr.c. The size of the buffer used to store output sample values has deliberately been chosen such that the buffer will not hold an integer number of cycles of the 1 kHz sinusoid being generated. This leads to spectral leakage in the frequency domain representation of the data plotted in MATLAB. The important feature of the magnitude frequency response plot is the 1 kHz center frequency of the single peak rather than its shape. If BUFFER\_LENGTH is adjusted to be equal to an integer multiple of LOOPLENGTH, then function stm32f7\_logfft() will run into problems computing the logs of zero values in the FFT. In general, zero FFT values will not be encountered by function stm32f7\_logfft().

MATLAB resulting plot using stm32f7\_logfft.m:



# DMA-Based Example Program

Direct Memory Access (DMA) is a method in which a hardware component of a computer gains access to the Memory Bus and controls the transfer. DMA controllers can be configured to handle data transfers between memories, memory to peripherals, and vice versa, enabling the processor deal with other processes. Essentially the main benefit of this method is to reduce strain on the CPU. This concept is demonstrated in the block diagram below:

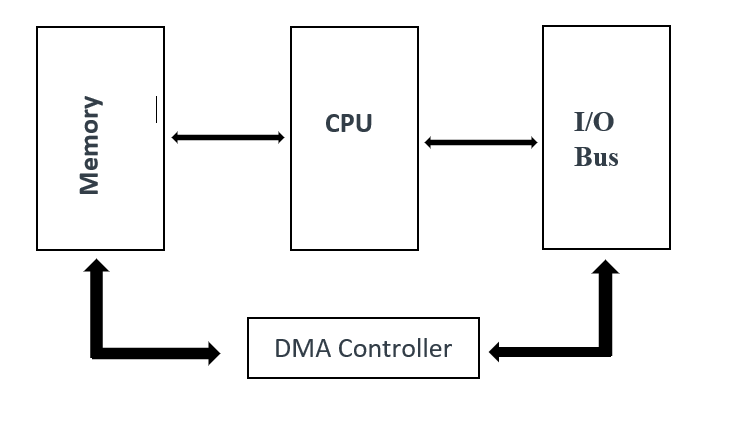


Figure 11: Block diagram representation DMA-Based I/O

Program stm32f7\_loop\_graph\_dma.c. has similar functionality to program stm32f7\_loop\_intr.c except that it uses the DMA I/O, as opposed to interrupt-based. DMA-based I/O is investigated in more detail in lab exercise 5.

Run program stm32f7\_loop\_graph\_dma.c and confirm that input to the digital microphones is passed to the headphones. A major difference between programs stm32f7\_loop\_intr.c and stm32f7\_loop\_graph\_dma.c is that the latter program plots the sample values it writes to the WM8994 DAC as a graph on the LCD. Pressing the blue user pushbutton toggles between time-domain and frequency-domain representations of those sample values.

// stm32f7\_loop\_graph\_dma.c

#include "stm32f7\_wm8994\_init.h"

#include "stm32f7\_display.h"

#define PLOTBUFSIZE 128

#define BLOCK\_SIZE 1

#define SOURCE\_FILE\_NAME "stm32f7\_loop\_graph\_dma.c"

extern volatile int32\_t TX\_buffer\_empty; // these may not need to be int32\_t

extern volatile int32\_t RX\_buffer\_full; // they were extern volatile int16\_t in F4 version

extern int16\_t rx\_buffer\_proc, tx\_buffer\_proc; // will be assigned token values PING or PONG

float32\_t x[PING\_PONG\_BUFFER\_SIZE];

float32\_t cmplx\_buf[2\*PING\_PONG\_BUFFER\_SIZE];

float32\_t outbuffer[PING\_PONG\_BUFFER\_SIZE] = { 0.0f };

void process\_buffer(void) // this function processes one DMA transfer block of data

{

int i;

int16\_t \*rx\_buf, \*tx\_buf;

if (rx\_buffer\_proc == PING) {rx\_buf = (int16\_t \*)PING\_IN;}

else {rx\_buf = (int16\_t \*)PONG\_IN;}

if (tx\_buffer\_proc == PING) {tx\_buf = (int16\_t \*)PING\_OUT;}

else {tx\_buf = (int16\_t \*)PONG\_OUT;}

for (i=0 ; i<(PING\_PONG\_BUFFER\_SIZE) ; i++)

{

x[i] = (float32\_t)(\*rx\_buf);

\*tx\_buf++ = \*rx\_buf++;

\*tx\_buf++ = \*rx\_buf++;

cmplx\_buf[i\*2] = x[i]; // real part

cmplx\_buf[(i\*2)+1] = 0.0; // imaginary part

}

RX\_buffer\_full = 0;

TX\_buffer\_empty = 0;

}

int main(void)

{

int i;

int button = 0;

stm32f7\_wm8994\_init(AUDIO\_FREQUENCY\_8K,

IO\_METHOD\_DMA,

INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2,

OUTPUT\_DEVICE\_HEADPHONE,

WM8994\_HP\_OUT\_ANALOG\_GAIN\_0DB,

WM8994\_LINE\_IN\_GAIN\_0DB,

WM8994\_DMIC\_GAIN\_9DB,

SOURCE\_FILE\_NAME,

GRAPH);

while(1)

{

while(!(RX\_buffer\_full && TX\_buffer\_empty)){}

BSP\_LED\_On(LED1);

process\_buffer();

button = checkButtonFlag();

if(button == 1)

{

for(i=0; i<PING\_PONG\_BUFFER\_SIZE; i++)

{

cmplx\_buf[2\*i] = x[i];

cmplx\_buf[2\*i + 1] = 0.0;

}

arm\_cfft\_f32(&arm\_cfft\_sR\_f32\_len256, (float32\_t \*)(cmplx\_buf), 0, 1);

arm\_cmplx\_mag\_f32((float32\_t \*)(cmplx\_buf),(float32\_t \*)(outbuffer), PING\_PONG\_BUFFER\_SIZE);

plotLogFFT(outbuffer, PING\_PONG\_BUFFER\_SIZE, LIVE);

}

else

{

plotWave(x, PLOTBUFSIZE, LIVE, ARRAY);

}

BSP\_LED\_Off(LED1);

}

}

Figure 12: Listing of program stm32f7\_loop\_graph\_dma.c

The DMA-based I/O method introduces a delay in the signal path equal to two DMA transfer blocks, or buffers, of samples. The number of sampling periods represented by one DMA transfer block is determined by the value of PING\_PONG\_BUFFER\_SIZE, defined in header file stm32f7\_wm8994\_init.h. If you change the sampling frequency in program stm32f7\_loop\_graph\_dma.c from 48 kHz to 8 kHz, you may be able to discern a slight delay between the signal input to the digital microphones and the signal output to the headphones.

Change the parameter INPUT\_DEVICE\_DIGITAL\_MICROPHONE\_2 to INPUT\_DEVICE\_INPUT\_LINE\_1 and the sampling frequency back to 8 kHz in program stm32f7\_loop\_graph\_dma.c and use a signal generator to input a sinusoid of frequency 250 Hz to the LINE IN input. You should see graphs on the LCD similar to those shown in Figures 13 and 14.

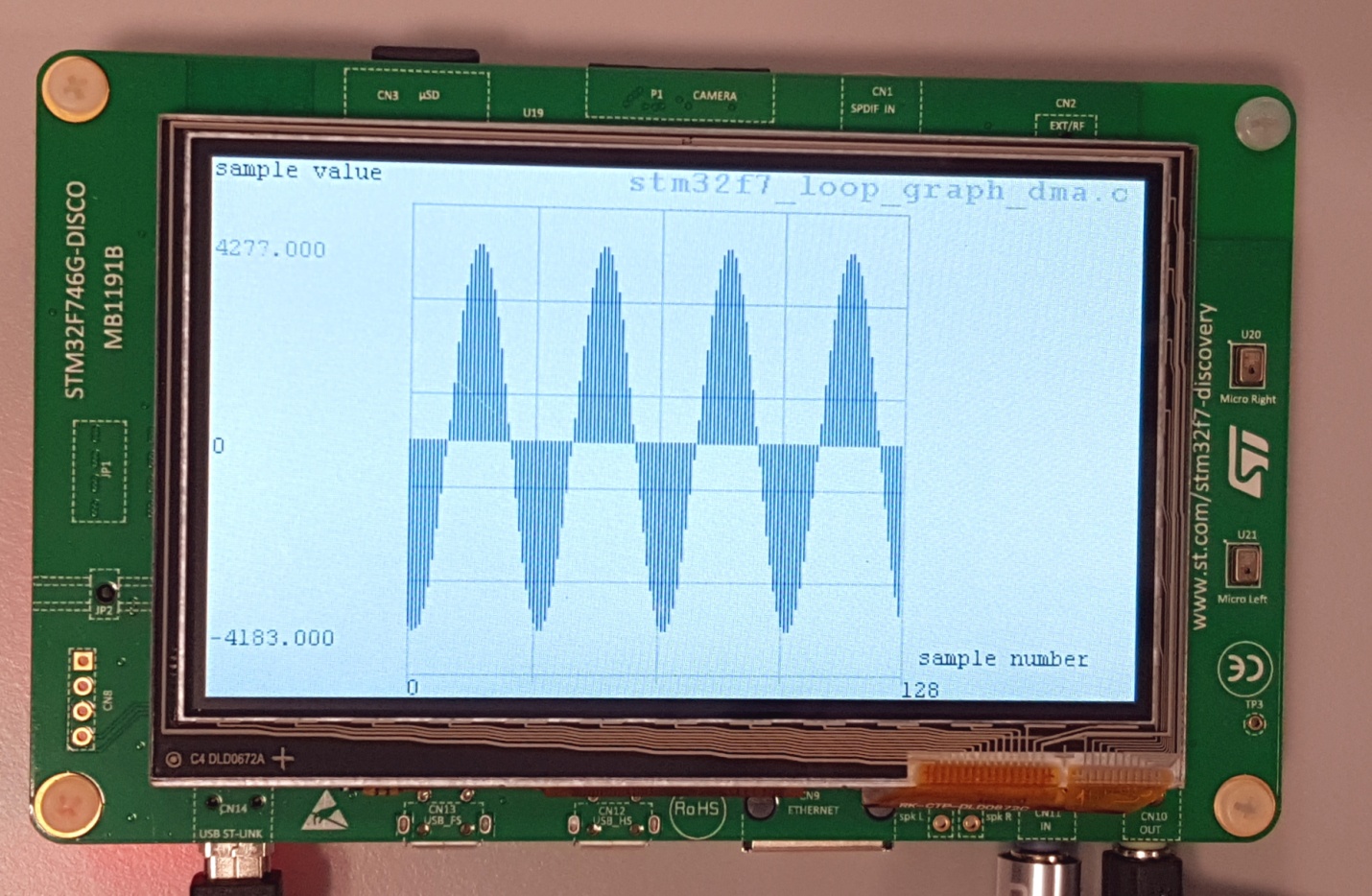


Figure 13: Graphical representation of 250 Hz sinusoidal signal input to program stm32f7\_loop\_graph\_dma.c in the time-domain. Sampling frequency is 8 kHz

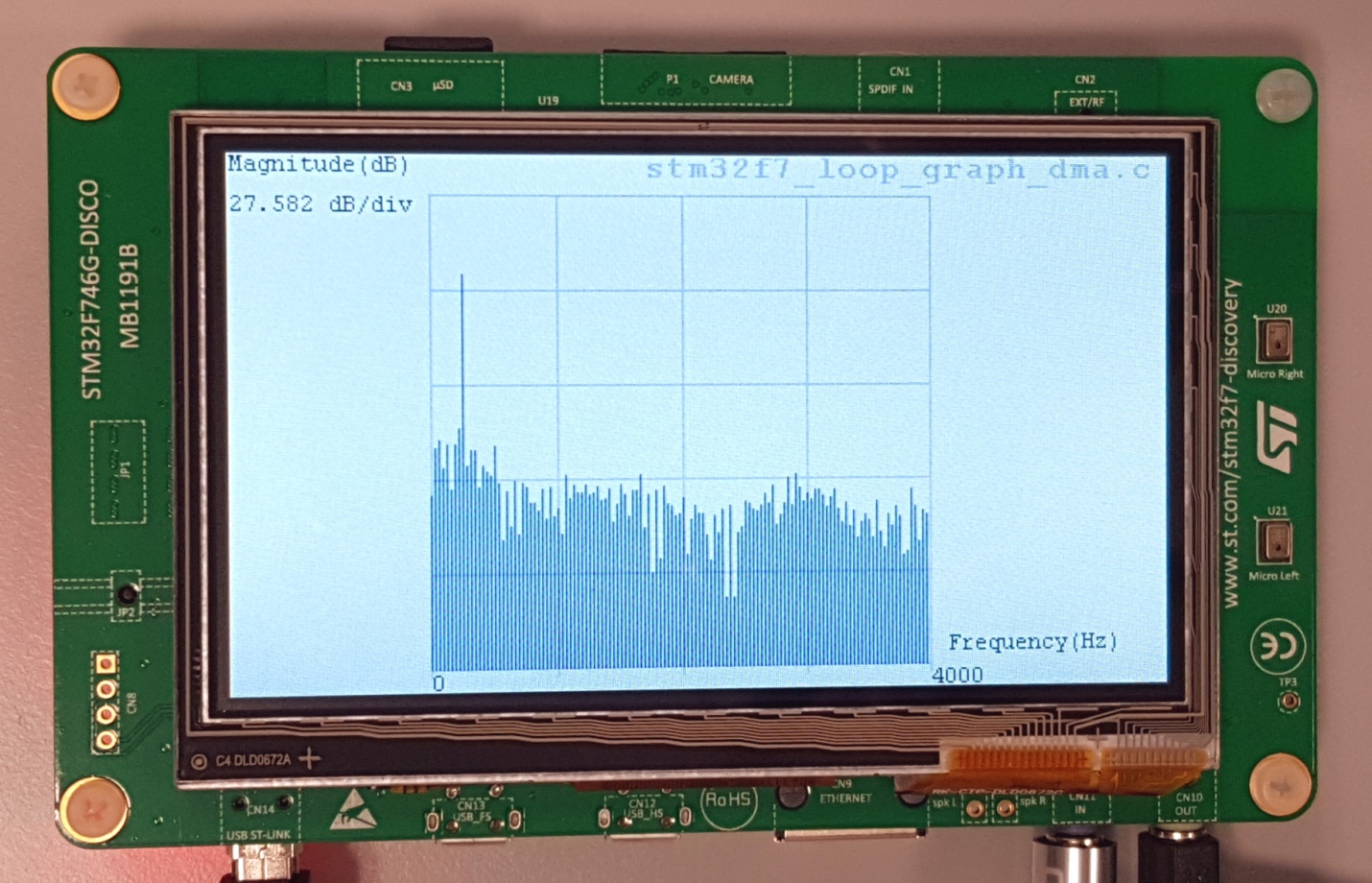


Figure 14: Graphical representation of 250 Hz sinusoidal signal input to program stm32f7\_loop\_graph\_dma.c in the frequency-domain. Sampling frequency is 8 kHz

# 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>