***DSP Education Kit***

**LAB 5: INSTRUCTOR VERSION**

**Fast Fourier Transform**

**Issue 1.0**

­

Contents

[1 Introduction 1](#_Toc29195684)

[1.1 Lab overview 1](#_Toc29195685)

[2 Requirements 1](#_Toc29195686)

[3 Discrete Fourier Transform (DFT) of a Sequence of Real Numbers 1](#_Toc29195687)

[3.1 Exercise 2](#_Toc29195688)

[4 Twiddle Factors 4](#_Toc29195689)

[4.1 Exercise 4](#_Toc29195690)

[5 Testing Execution Times for Different DFT Functions 5](#_Toc29195691)

[5.1 Exercise 5](#_Toc29195692)

[6 Frame-based Processing 6](#_Toc29195693)

[6.1 DMA-based I/O on the STM32F746G 6](#_Toc29195694)

[6.1.1 stm32f7\_wm8994\_init() DMA operation 6](#_Toc29195695)

[6.1.2 stm32f7\_loop\_dma.c operation 7](#_Toc29195696)

[6.1.3 Exercise 7](#_Toc29195697)

[6.2 FFT of a signal in real time 8](#_Toc29195698)

[6.2.1 Running program stm32f7\_fft256\_dma.c 9](#_Toc29195699)

[7 Spectral Leakage 10](#_Toc29195700)

[7.1 Modifying the program to reduce spectral leakage 13](#_Toc29195701)

[7.2 Real-time performance of program stm32f7\_fft256\_dma.c 16](#_Toc29195702)

[7.2.1 Exercise 16](#_Toc29195703)

[8 Conclusions 17](#_Toc29195704)

[9 Additional References 17](#_Toc29195705)

# Introduction

## Lab overview

The examples in this exercise introduce some of the concepts behind the Fast Fourier Transform (FFT). You will write a C function to implement the Discrete Fourier transform (DFT) and assess its computational efficiency. Next, you will modify that function to use pre-computed “twiddle factors” and measure the time taken to execute the modified function. You will compare this with the time taken to execute fast Fourier transforms written in C and implemented using the CMSIS DSP library. Finally, you will embed these functions in a real-time program that acts as a simple spectrum analyzer.

Instructor note: Items (texts, images, and plots) in bright red 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
* 3.5 mm audio jack
* BNC T-Piece connector
* Male to male BNC coaxial cable
* An audio frequency signal generator

# Discrete Fourier Transform (DFT) of a Sequence of Real Numbers

This example illustrates the DFT of an N-point, real-valued sequence. Program stm32f7\_dft.c will be used to calculate the complex DFT

 (1)

As supplied, the program does not do this. You must write the definition of function dft(). Program stm32f7\_dft.c is written so that an N-point complex sequence is stored in array samples and a pointer to this array is passed to the function dft(). That function is required to replace the complex, time-domain sequence passed to it with its complex, frequency domain representation, that is, its DFT. A structure, COMPLEX, intended for the representation of complex numbers in rectangular form is defined in the program. Recall that Euler’s formula describes the relationship between polar (exponential) and rectangular representations of complex quantities as

 (2)

As supplied, the complex time-domain sample values written to array samples represent a real-valued sinusoid with a frequency of 1800 Hz, sampled at 8000 Hz.

The complex result of the DFT calculation (the contents of array samples after function dft() has been called) is plotted as a bar graph on the LCD with both the real parts (blue) and the imaginary parts (red) of X(k) on the same axes as shown in Figure 1.

## Exercise

Write the dft() function and verify your results.

The code solution for dft() is in dft\_soln.c provided in the Lab solutions folder.

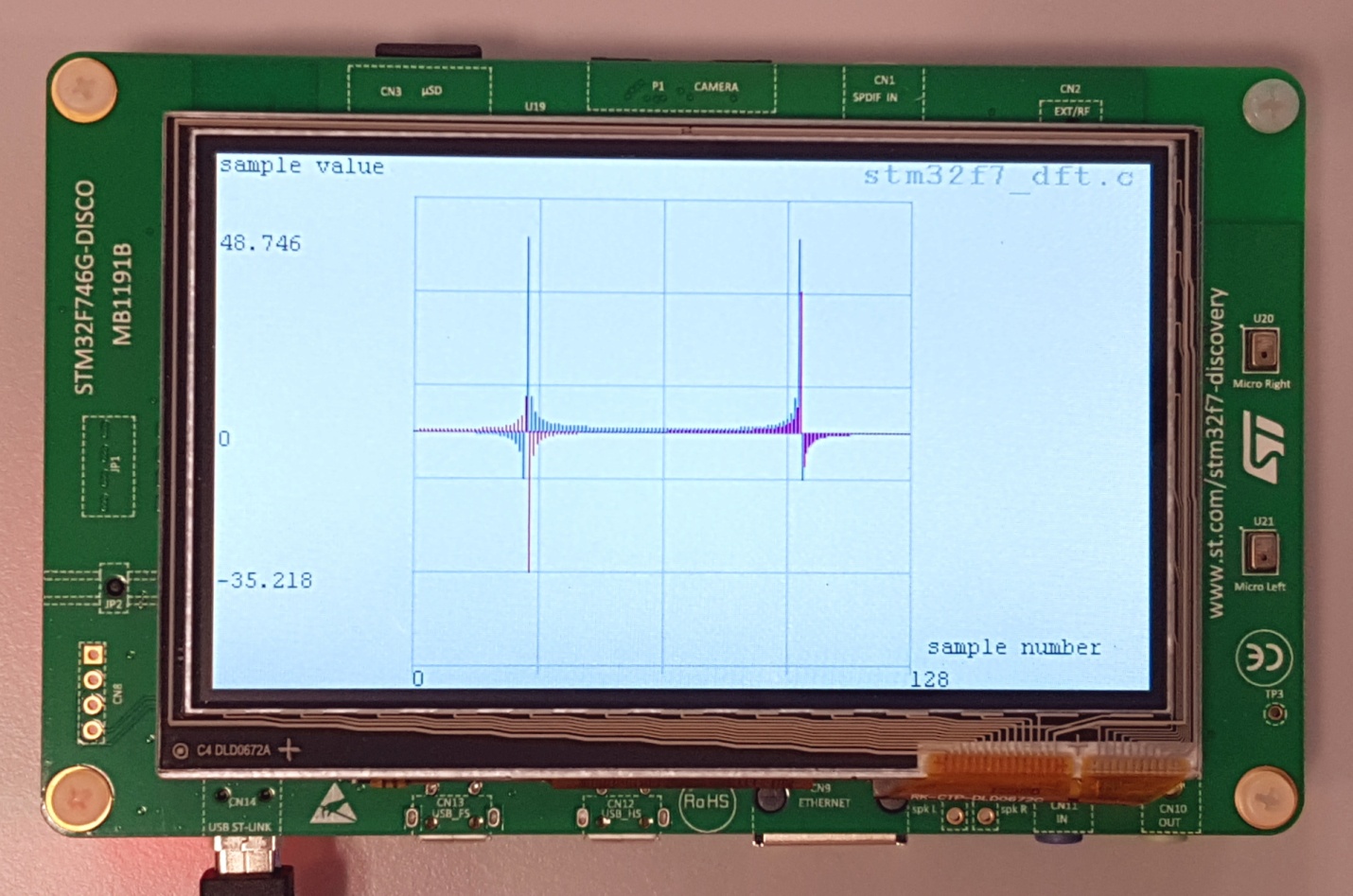
If your results do not look similar to those shown in Figure 1, you will need to correct the dft() function definition that you have written.

Figure 1: LCD screen showing result of DFT calculation made by program stm32f7\_dft.c. TESTFREQ = 1800.0, SAMPLING\_FREQ = 8000.0, N = 128

An alternative method of checking the results of calling function dft() is, after halting the program, saving the contents of array samples to a file and plotting them using MATLAB function stm32f7\_bar\_complex(). Figure 2 shows the result of doing this.

It is a good practice to test DFT or FFT functions using simple input sequences precisely because the results are straightforward to interpret. Different time domain input sequences can be used to test your function by changing the value of the constant TESTFREQ and by editing program statements

samples[n].real = cos(2\*PI\*TESTFREQ\*n/SAMPLING\_FREQ);

samples[n].imag = 0.0f;

It is important that you test your function using complex as well as real-valued input data.



Figure 2: Complex contents of array samples (TESTFREQ = 1800.0) plotted using MATLAB function stm32f7\_bar\_complex()

**Test your function using BOTH imaginary and complex time-domain data.**

Common difficulties encountered by students.

Computation of the DFT involves multiplication of complex quantities. Make sure that students are clear about how to multiply two complex numbers together, i.e. , and that the real and imaginary parts of the result each involve both the real and imaginary parts of both operands. Computation of all values of the DFT will involve two loops. Each value of the DFT is a sum of products ( thru ) and there are values to compute ( thru ). An array to hold the computed DFT values will have to be declared. Computation of each value of the DFT involves all values of the input sequence—you can’t start overwriting those values with DFT values before all of the DFT values have been computed. It is important that students test their code with complex input data. It is possible to get away with an incorrect function definition if only real-valued input is used.

# Twiddle Factors

The chances are that if you have coded the computation of the DFT in a straightforward manner, then it will involve repeated calls to functions cos() and sin(). These function calls are extremely computationally expensive and are out of place in a real-time program.

But the time taken to execute function dft() may be reduced more significantly by pre-computing and storing *twiddle factors* (the N different values of  used in the DFT calculation).

## Exercise

Remove source file stm32f7\_dft.c from your project and replace it with stm32f7\_dftw.c. Program stm32f7\_dftw.c is supplied in a form similar to that in which program stm32f7\_dft.c was provided. **You must write the definition of function dftw() such that it computes the DFT of the complex data passed to it in array samples.** However, you must not make calls to functions sin() or cos()within function dftw(). Instead, at the start of function main(), you should initialize the contents of array twiddle. Its contents should be used within function dftw() as a lookup table of twiddle factors.

**Test program STM32f7\_dftw.c to check that function dftw() returns the same results as function dft().**

The code solution for dftw() is in dftw\_soln.c provided in the Lab solutions folder.

The use of pre-computed twiddle factors is one influence on the computational efficiency of the Fast Fourier transform (FFT). Programs stm32f7\_fft.c and stm32f7\_fft\_CMSIS.c compute the complex DFT of an array of sample values using an FFT function written in C and an optimized FFT function from the CMSIS DSP library, respectively.

# Testing Execution Times for Different DFT Functions

The execution times of the different DFT functions can be compared using an oscilloscope to measure the time that GPIO pin PI1 is high. It is set high just before calling a DFT function and reset low just after that function has been executed. Since a DFT function is called only once by each program, only one timing pulse will be generated by each program. Use single trigger mode on the oscilloscope in order to capture those pulses.

## Exercise

Measure the durations of the pulses output by programs stm32f7\_dft.c, stm32f7\_dftw.c, stm32f7\_fft.c, and stm32f7\_fft\_CMSIS.c for the values of N indicated in Table 1.

**To accommodate a limitation in function plotWave(), if the value of N is equal to 256, pass the parameter value N/2, as opposed to N, to that function. Only the first 128 complex values stored in array samples will be plotted on the LCD.**

|  |  |  |  |
| --- | --- | --- | --- |
| Program name | Function name | N | Execution time (ms) |
| stm32f7\_dft.c | dft() | 128 | 2288.6778 |
| stm32f7\_dft.c | dft() | 256 | N/A |
| stm32f7\_dftw.c | dftw() | 128 | 2.0613 |
| stm32f7\_dftw.c | dftw() | 256 | 8.218 |
| stm32f7\_fft.c | fft() | 128 | 0.08818 |
| stm32f7\_fft.c | fft() | 256 | 0.19451 |
| stm32f7\_fft\_CMSIS.c | arm\_cfft\_f32() | 128 | 0.03602 |
| stm32f7\_fft\_CMSIS.c | arm\_cfft\_f32() | 256 | 0.03641 |

Table 1: Experimentally measured execution times for functions dft(), dftw(),fft() and arm\_cfft\_f32()

# Frame-based Processing

Rather than processing one sample at a time, the DFT algorithm is applied to blocks, or frames, of samples. Using the DFT in a real-time program therefore requires a slightly different approach to that used for input and output in most of the previous lab exercises. You will have noticed that example programs using CMSIS DSP library functions, for example, arm\_fir\_f32(), did process blocks of samples and used DMA-based I/O. It is possible to implement buffering, and to process blocks of samples, using interrupt-based I/O. However, DMA-based I/O is more intuitive for frame-based processing and will be used here.

## DMA-based I/O on the STM32F746G

Arm Cortex-M7 processors from different manufacturers implement DMA slightly differently from each other (although the underlying principles are similar).

Program stm32f7\_loop\_dma.c makes use of the ping-pong mode of multi-buffering possible on the STM32F746G to implement frame-based processing. This program is a cut-down version of program stm32f7\_loop\_graph\_dma.c, used in Laboratory Manual 1. It does not plot any graphs on the LCD.

### stm32f7\_wm8994\_init() DMA operation

The DMA in the STM32F746G is organized into unidirectional streams, two of which are used for this application. In function stm32f7\_wm8994\_init(), DMA stream 7 is configured to make DMA transfers between the Synchronous Audio Interface (SAI) peripheral and input buffers (arrays) in memory (alternately PING\_IN and PONG\_IN). It generates an interrupt when a transfer of PING\_PONG\_BUFFER\_SIZE 32-bit words has completed. Each 32-bit word comprises two 16-bit sample values (left and right channels). The value PING\_PONG\_BUFFER\_SIZE is therefore equivalent to the number of sampling instants represented by one DMA transfer. The value of PING\_PONG\_BUFFER\_SIZE is defined in file stm32f7\_wm8994\_init.h.

DMA stream 3 is configured to make DMA transfers between output buffers in memory (alternately PING\_OUT and PONG\_OUT) and the SAI peripheral. It too generates an interrupt when a transfer of PING\_PONG\_BUFFER\_SIZE 32-bit words has completed.

Four different interrupt service routines (functions) are involved in the DMA ping-pong buffering process. BSP\_AUDIO\_OUT\_TransferComplete\_CallBack(), BSP\_AUDIO\_IN\_TransferComplete\_CallBack(), BSP\_AUDIO\_OUT\_TransferCompleteM1\_CallBack(), and BSP\_AUDIO\_IN\_TransferCompleteM1\_CallBack(), defined in file stm32f7\_wm8994\_init.c, are associated with completion of DMA transfers from array PING\_OUT to the SAI peripheral, from the SAI peripheral to array PING\_IN, from array PONG\_OUT to the SAI peripheral, and from the SAI peripheral to array PONG\_IN, respectively.

### stm32f7\_loop\_dma.c operation

The actions carried out in these routines are simply to toggle the values of variables rx\_buffer\_proc and tx\_buffer\_proc between PING and PONG and to set flags RX\_buffer\_full and TX\_buffer\_empty. Switching between buffers PING\_IN, PONG\_OUT and PING\_OUT and PONG\_OUT in the DMA streams is handled automatically by the DMA multi-buffering mechanism.

Function main()waits until both RX\_buffer\_full and TX\_buffer\_empty flags are set, that is, until both DMA transfers have completed, before calling function process\_buffer(). In program stm32f7\_loop\_dma.c, function process\_buffer() simply copies the contents of the most recently filled input buffer (PING\_IN or PONG\_IN) to the most recently emptied output buffer (PING\_OUT or PONG\_OUT), according to the values of variables rx\_buffer\_proc and tx\_buffer\_proc. In general, frame-based processing will be carried out in function process\_buffer() using the contents of the most recently filled input buffer as input and writing output sample values to the most recently emptied output buffer.

DMA transfers will complete, and function process\_buffer()will be called every PING\_PONG\_BUFFER\_SIZE sampling instants, and therefore any processing must be completed within PING\_PONG\_BUFFER\_SIZE /fs seconds, that is, before the next DMA transfer completion.

Figure 3 was generated by a signal generator using a rectangular pulse of duration 1 ms repeated at intervals of 5 ms. The sampling rate was 48 kHz and PING\_PONG\_BUFFER\_SIZE was equal to 128.

The expected delay between input and output signals is PING\_PONG\_BUFFER\_SIZE\*2/fs seconds.

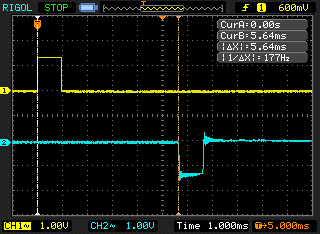


Figure 3: Input pulse (yellow–channel 1) and delayed pulse (blue–channel 2) generated using program stm32f7\_loop\_dma.c, PING\_PONG\_BUFFER\_SIZE = 128

### Exercise

The default value of PING\_PONG\_BUFFER\_SIZE (defined in header file stm32f7\_wm8994\_init.h) is 256 and the sampling rate fs (determined by the parameter passed to function stm32f7\_wm8994\_init() in program stm32f7\_loop\_dma.c) is 48000 Hz. Hence, the expected delay is PING\_PONG\_BUFFER\_SIZE\*2/fs = 10667 µs. **Temporarily** change the value of PING\_PONG\_BUFFER\_SIZE defined in header file stm32f7\_wm8994\_init.h and repeat the experiment in order to fill in the remainder of Table 2. Rebuild the project for the altered values of PING\_PONG\_BUFFER\_SIZE to take effect. If your signal generator is a part of your oscilloscope, it would be helpful to use a BNC T-piece in order to split the output of the generator.

**Measure the delay introduced by program loop\_intr.c for comparison.**

After completing this experiment, change the value of PING\_PONG\_BUFFER\_SIZE defined in header file stm32f7\_wm8994\_init.h back to its default value of 256.

|  |  |  |
| --- | --- | --- |
| PING\_PONG\_BUFFER\_SIZE | Measured delay (µs) | PING\_PONG\_BUFFER\_SIZE\*2/fs (µs) |
| 256 | 966 | 10667 |
| 128 | 626 | 5333 |
| 64 | 966 | 2667 |
| 32 | 634 | 1333 |
| 16 | 938 | 667 |

Table 2: *delays introduced by using different sizes of DMA ping-pong buffer in program stm32f7\_loop\_dma.c. Sampling frequency fs = 48 kHz*

## FFT of a signal in real time

Program stm32f7\_fft256\_dma.c combines function fft()with a real-time program to implement a very simple spectrum analyzer. The program uses the DMA-based frame-processing mechanism implemented in program stm32f7\_loop\_dma.c and calls function fft()from function process\_buffer().

Blocks of PING\_PONG\_BUFFER\_SIZE = 256 real-valued input samples (for both left and right channels) are transferred from the WM8994 ADC, via the SAI peripheral, to memory using DMA, and their 256-point complex DFTs are computed using function fft(). Left channel input samples are written to buffer inbuffer for later plotting using MATLAB. The scaled magnitudes of the elements of the frequency-domain representations are written to the DAC (again via the SAI peripheral and using DMA) and to buffer outbuffer for plotting. The value of PING\_PONG\_BUFFER\_SIZE is set in file stm32f7\_wm8994\_init.h and is equal to the number of sampling instants over which data in one DMA transfer block correspond. Since the WM8994 is a stereo codec, each DMA transfer block comprises PING\_PONG\_BUFFER\_SIZE 16-bit left channel sample values plus PING\_PONG\_BUFFER\_SIZE 16-bit right channel sample values.

### Running program stm32f7\_fft256\_dma.c

Use a signal generator connected to LINE IN on the Discovery board to input a sinusoidal test signal of magnitude approximately 600 mVpp and connect an oscilloscope to HEADPHONE OUT. The input signal (stored in inbuffer) and the magnitude of its FFT (stored in outbuffer) are output on L and R channels, respectively.

Vary the frequency of the input signal between 100 Hz and 4000 Hz. Figure 4 shows an example of what you should see on an oscilloscope. The blue trace (channel 2) represents 256-sample blocks of the input signal and the yellow trace (channel 1) represents the magnitudes of their FFTs. In the case illustrated in Figure 4: Oscilloscope display produced using program stm32f7\_fft256\_dma.c corresponding to a 1750 Hz sinusoidal input signal, the input signal was a sinusoid with frequency 1750 Hz. The two large pulses correspond to trigger values added to the output signal every 256 samples, replacing the magnitude of sample X(0). One block of data (256 samples) extends from one trigger pulse to the next. The two smaller pulses in between the two larger pulses represent frequency components at +/- 1750 Hz, corresponding to the sinusoidal input signal. Due to the characteristics of the WM8994 DAC (as examined in Laboratory Manual 2), the pulses are highly oscillatory and decay relatively slowly. You will need to look past this aspect of the output waveform to see the underlying frequency domain information.

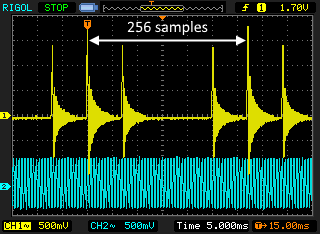


Figure 4: Oscilloscope display produced using program stm32f7\_fft256\_dma.c corresponding to a 1750 Hz sinusoidal input signal

The data in each output block are ordered such that the first value |X(0)| corresponds to a frequency of 0 Hz. The next 128 (*N*/2–1) values (|X(1)| through |X(128)|) correspond to frequencies 31.25 Hz (fs/N) to 4000.00 Hz (fs/2), inclusive, in steps of 31.25 Hz. The next 127 values (|X(129)| through |X(255)|) correspond to frequencies of -3968.75 Hz to -31.25 Hz inclusive, in steps of 31.25 Hz.

Shown on the LCD are the first 128 values in array outbuffer, that is, the FFT magnitudes corresponding to frequencies 0 to 3968.75 Hz. Figure 5 shows the LCD graph corresponding to the oscilloscope display of Figure 4. The data plotted on the LCD do not include a trigger value.

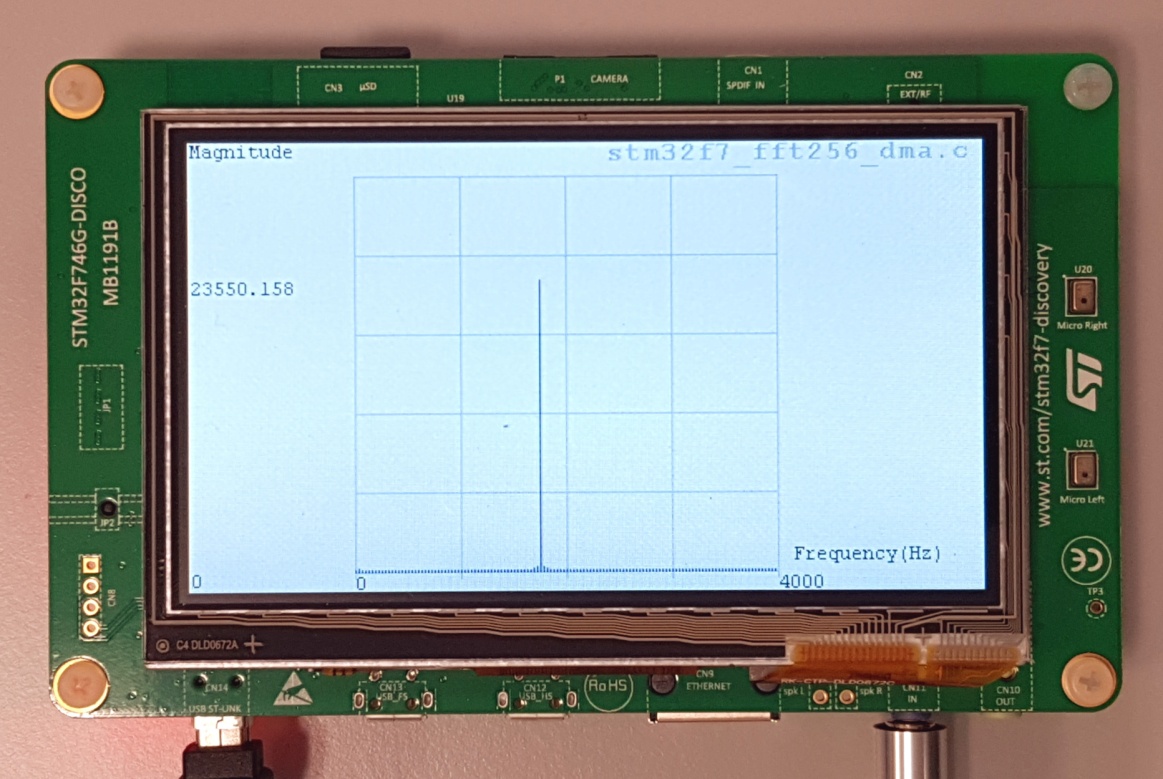


Figure 5: LCD graph corresponding to oscilloscope display shown in Figure 4

Increase the frequency of the input signal, and as it approaches 4 kHz, you should see the two smaller pulses move together toward a point halfway between consecutive trigger pulses. On the LCD, you should see one pulse moving further right as the frequency of the input signal is increased. Increase the frequency of the input signal beyond 4 kHz and the two smaller pulses on the oscilloscope trace, and the one pulse on the LCD, should disappear.

As the frequency of the input signal is varied between 0 and 4 kHz, not only should the two smaller pulses in the oscilloscope display, and the one pulse on the LCD, move (relative to the trigger pulses), the shape of the pulses should change, too.

# Spectral Leakage

If the frequency of the input signal is exactly 1750 Hz, the magnitude of the DFT of a block of samples should in theory be equal to zero except at two points (|X(56)| and |X(200)|), corresponding to frequencies of +/- 1750 Hz. This may be verified by adjusting the frequency of the input signal to 1750 Hz and looking at the graph on the LCD or by halting the program (stm32f7\_fft256\_dma.c) and viewing the contents of array outbuffer in the *MDK-Arm Debugger Memory* window. Alternatively, the contents of array outbuffer may be saved to a file and plotted using MATLAB function stm32f7\_bar\_real(). The results of doing this, for an input frequency of 1750 Hz, are shown below, along with corresponding oscilloscope trace.



Figure 6: Partial contents of array outbuffer, plotted using MATLAB function stm32f7\_bar\_real(). Input frequency = 1750 Hz

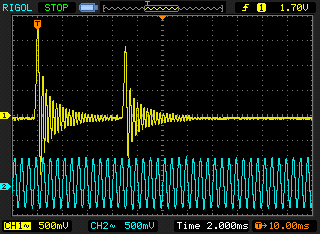


Figure 7: Oscilloscope trace corresponding to data plotted in Figure 6. Input frequency = 1750 Hz. Both the trigger pulse and the smaller pulse representing the magnitude of X(56) are essentially impulse responses from the WM8731 DAC

Next, adjust the frequency of the input signal to 1765 Hz and repeat the experiment. You should find that the shape of the smaller pulse in the oscilloscope trace has changed, and this is due to a change in the data sequence being written to the DAC (as shown in Figure 8). Figure 10 shows the corresponding graph on the LCD.



Figure 8: Partial contents of array outbuffer, plotted using MATLAB function plot\_int16(). Input frequency = 1765 Hz

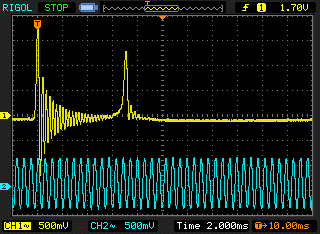


Figure 9: Oscilloscope trace corresponding to data plotted in Figure 8. Input frequency = 1765 Hz

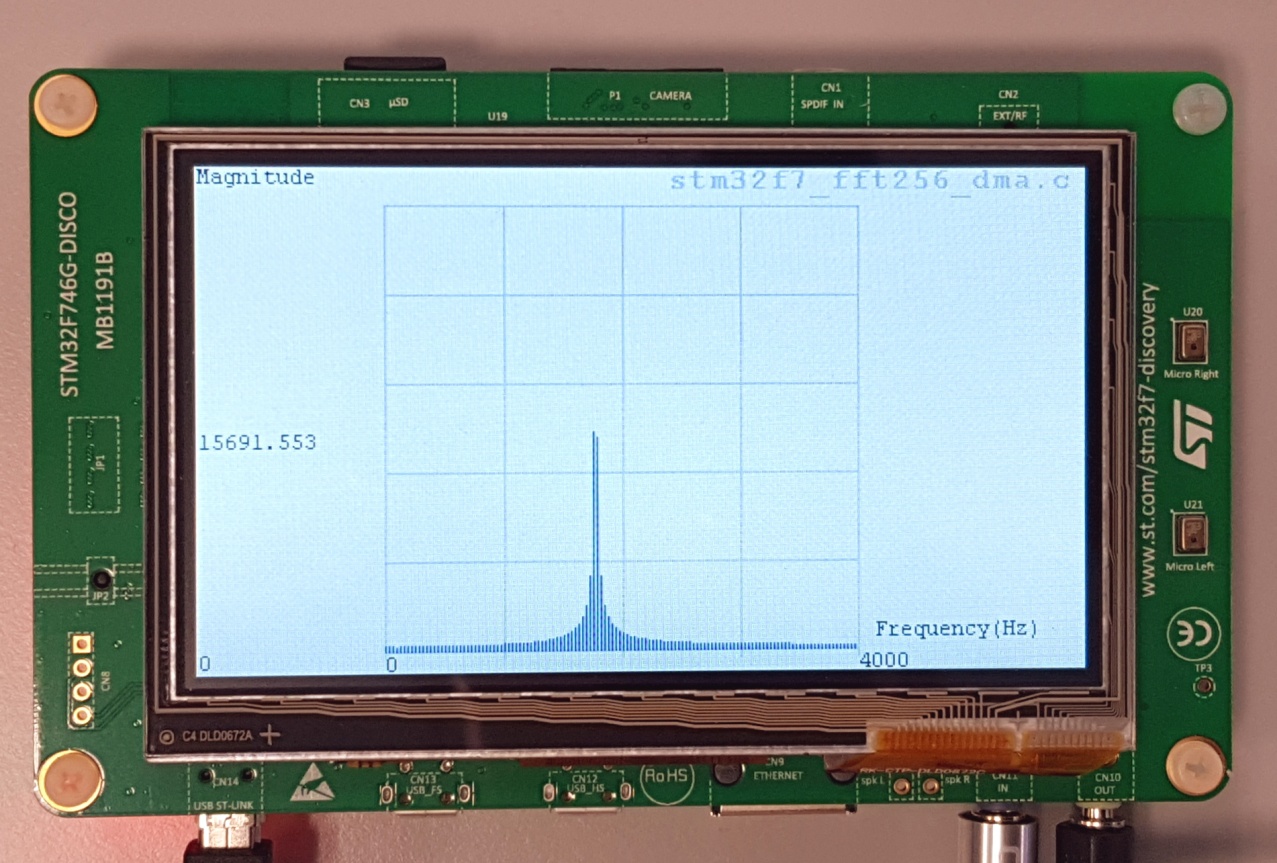


Figure 10: LCD graph corresponding to oscilloscope display shown in Figure 9

## Modifying the program to reduce spectral leakage

One method of reducing spectral leakage is to multiply each block of input samples by a tapered window function prior to computing its DFT. Alter the line of program stm32f7\_fft256\_dma.c that reads

cbuf[i].real = ((float)left\_sample);

to

cbuf[i].real = ((float)left\_sample)\*hamming[i];

and add the line

#include “hamming256.h”

File hamming256.h contains the declaration of an array hamming, initialized to contain a 256-point hamming window. Figure 11 through Figure 14 show the effect of windowing the blocks of time domain samples before computing their FFT.

You can investigate the effects of several other window functions using header files bartlett256.h, hann256.h, blackman256.h, and kaiser256.h.



Figure 11: Partial contents of array outbuffer, plotted using MATLAB function plot\_int16(). Input frequency = 1750 Hz. Hamming window used

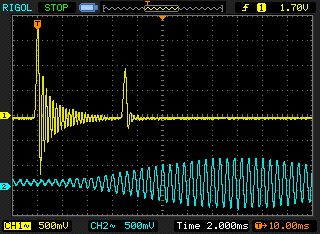


Figure 12: Oscilloscope trace corresponding to data plotted in Figure 11. Input frequency = 1750 Hz



Figure 13: Partial contents of array outbuffer, plotted using MATLAB function plot\_int16(). Input frequency = 1765 Hz. Hamming window used

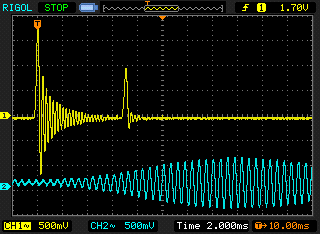


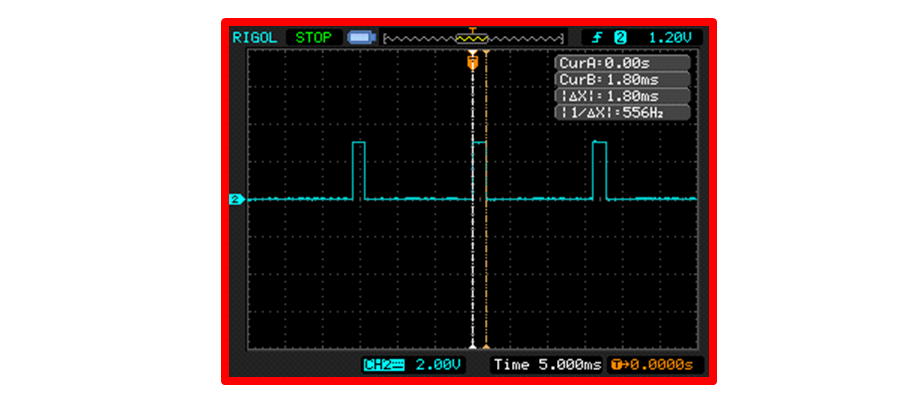
Figure 14: Oscilloscope trace corresponding to data plotted in Figure 13. Input frequency = 1765 Hz

## Real-time performance of program stm32f7\_fft256\_dma.c

The time taken to execute function process\_buffer() in program stm32f7\_ft256\_dma.c may be assessed by connecting an oscilloscope probe to GPIO pin PI1 on the Discovery board. This is taken high just before the call to function process\_buffer() and then low again just afterward.

### Exercise

1. Measure the execution time for function process\_buffer() and compare this with the corresponding measurement recorded in Table 1.
2. Can you account for the discrepancy in these values?



Oscilloscope trace showing output from the test pin during execution of program stm32f7\_fft256\_dma.c.

Function ProcessBuffer () takes approximately 1.8 ms to execute. It is called every 16 ms.

Much of the execution time is spent computing the magnitude of the FFT result, using calls to function sqrt().

The execution time may be reduced significantly by substituting a call to CMSIS function arm\_magnitude\_f32().

Using CMSIS-DSP function arm\_cfft\_f32() will, of course, reduce execution time even further.

Now look back at Table 1.

**How long did your dftw() function take to compute a 256-point DFT? Should it work in real-time?**

You can also ask students to modify program stm32f7\_fft256\_dma.c, by replacing function fft() with the function dftw(). Students can then confirm that the modified program produces a similar output signal in response to sinusoidal inputs and measure the time taken to execute function process\_buffer(). Note that the students will have to initialize the twiddle factors used by function dftw()exactly as how it was done in program dftw.c and **not** as they are initialized for function fft()in program stm32f7\_fft256\_dma.c.

# Conclusions

In this exercise, you have demonstrated the relative computational efficiency of the Fast Fourier Transform algorithm. The use of DMA-based I/O for frame-based processing has been introduced.

# Additional References

**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