B39SB 2019 Lab Exercise #3:   
FAST FOURIER TRANSFORM

# 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.

# Details

## Hardware

To carry out this exercise you will need a Cypress FM4 board, an oscilloscope, an audio frequency signal generator, a PC running *Keil MDK-ARM*, *GoldWave* and M*ATLAB*, and suitable connecting cables.

### Discrete Fourier transform (dFT) of a SEQUENCE OF REAL NUMBERS

This example illustrates the DFT of an *N*-point, real-valued sequence. Program fm4\_dft\_gpio.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 fm4\_dft\_gpio.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, i.e. 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, or exponential, and rectangular representations of complex quantities as

 (2)

As supplied, the complex time-domain sample values written to array samples represent exactly ten cycles of a real-valued sinusoid. The DFT of this sequence should be equal to zero for all *k* except for *k*=10 and *k*=90. These two real-valued samples in the frequency domain correspond to positive and negative frequency components at +/- fs/10 Hz. Source file fm4\_dft\_gpio.c is quite long because it contains a significant number of program statements pertaining to setting up GPIO pin DIAGNOSTIC\_PIN. You can ignore these and concentrate on function dft() and main(). Program fm4\_dft\_gpio.c does not require source file fm4\_wm8731\_init.c as part of its MDK-ARM project.

// fm4\_dft\_gpio.c

#define ARM\_MATH\_CM4

#include <s6e2cc.h>

#include "arm\_math.h"

#include <stdint.h>

#include "arm\_const\_structs.h"

#include <math.h>

//#define PI 3.14159265358979

#define N 100

#define TESTFREQ 800.0

#define SAMPLING\_FREQ 8000.0

// offsets in bit-banded memory for GPIO pins used by example programs

#define DIAGNOSTIC\_PIN 0x0080

#define PFR\_BASE (0x42DE0000)

#define PCR\_BASE (0x42DE2000)

#define DDR\_BASE (0x42DE4000)

#define PDIR\_BASE (0x42DE6000)

#define PDOR\_BASE (0x42DE8000)

#define HIGH 1

#define LOW 0

typedef struct

{

float real;

float imag;

} COMPLEX;

COMPLEX samples[N];

void dft(COMPLEX \*x)

{

}

// find addresses of GPIO register bits in bit-banded memory space

// based on base addresses and on offset values that identify port and pin

// e.g. port C pin 9 has offset value (0x0C x 0x80) + (0x09 x 0x04) = 0x624

// and hence the address in bit-banded memory corresponding to the PCR for

// port C pin 9 is 0x42DE2624

//

// PFR is port function setting register - 0 for GPIO, 1 for peripheral function

#define GET\_PFR(pin\_ofs) ((volatile unsigned char\*) (PFR\_BASE + pin\_ofs))

// PCR is port pull-up setting register - 0 for pull-up, 1 for no pull-up

#define GET\_PCR(pin\_ofs) ((volatile unsigned char\*) (PCR\_BASE + pin\_ofs))

// PDDR is port direction setting register - 0 for GPIO in, 1 for GPIO out

#define GET\_DDR(pin\_ofs) ((volatile unsigned char\*) (DDR\_BASE + pin\_ofs))

// PDIR is port input data register - read input pin status here

#define GET\_PDIR(pin\_ofs) ((volatile unsigned char\*) (PDIR\_BASE + pin\_ofs))

// PDOR is port output data register - write GPIO output data here

#define GET\_PDOR(pin\_ofs) ((volatile unsigned char\*) (PDOR\_BASE + pin\_ofs))

void gpio\_set(int pin\_ofs, int value)

{

\*GET\_PDOR(pin\_ofs) = value;

}

int main()

{

int n;

bFM4\_CLK\_GATING\_CKEN0\_GIOCK |= 0x01; // supply clock GPIO module

// switch off analog input function on those GPIO pins used by example programs

bFM4\_GPIO\_ADE\_AN00 = 0x00; // P10 DIAGNOSTIC\_PIN

\*GET\_PFR(DIAGNOSTIC\_PIN) &= ~0u; // set pin function as GPIO

\*GET\_DDR(DIAGNOSTIC\_PIN) = 1u ; // set pin direction as output

\*GET\_PCR(DIAGNOSTIC\_PIN) &= ~0u; // set pin to have pull-up

for(n=0 ; n<N ; n++)

{

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

samples[n].imag = 0.0;

}

// call DFT function once only, setting DIAGNOSTIC\_PIN for the duration of the call

gpio\_set(DIAGNOSTIC\_PIN, HIGH);

dft(samples); //call DFT function

gpio\_set(DIAGNOSTIC\_PIN, LOW);

while(1){}

}

Figure 1. Listing of program fm4\_dft\_gpio.c.

It is 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, most readily 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 = 0.0;

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

In order to test the program, assuming that it has compiled and linked successfully and that you have launched the debugger,

1. Place breakpoints at the following two program statements

dft(samples);

and

while(1){}

1. Run the program to the first breakpoint. At this point the array samples should contain a real-valued time-domain input sequence. The contents of the array samples may be viewed in a memory window as shown in figure 2.
2. Save the contents of array samples (100 values \* 2(real + imag)\*type float32\_t) to a file by typing

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

in the *MDK-ARM Debugger Command* window. filename is chosen by you. start address is the address of array samples, and end address is equal to (start address + 0x320).

1. The contents of the data file you have created may be viewed using MATLAB function fm4\_plot\_complex().
2. Run the program again. It should halt at the second breakpoint. At this point, function dft() has been called, and array samples should contain the complex DFT of the time-domain input sequence.
3. Save the contents of array samples to a second file and plot the complex frequency-domain data X(k) using MATLAB function fm4\_plot\_complex(). You should see something similar to figure 3.

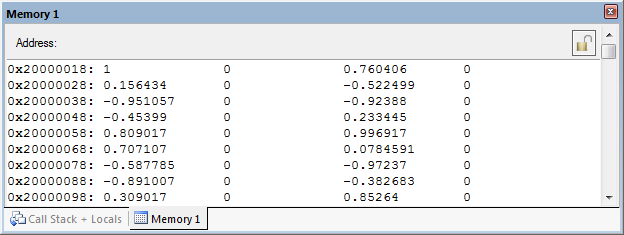


Figure 2. Initial contents of array samples displayed in *MDK-ARM Memory* window.



Figure 3. Complex contents of array samples (TESTFREQ = 800.0) plotted using MATLAB function fm4\_plot\_complex().

Note the very small magnitude of the imaginary part of X(k). Theoretically this should be equal to zero, but function dft() has introduced small arithmetic rounding errors.

If your results do not look similar to those in figure 3, you will need to correct the dft() function definition that you have written.

Change the frequency of the time-domain input sequence x(n) to 900 Hz by editing the definition of the constant TESTFREQ to read

#define TESTFREQ 900.0

and repeat the previous steps. The complex DFT result this time should appear as shown in figure 4.



Figure 4. Complex contents of array samples (TESTFREQ = 900.0) plotted using MATLAB function fm4\_plot\_complex().

If your results do not look similar to those in figure 4, you will need to correct the dft() function definition that you have written.

**Show graphs similar to those in Figure 4 to a demonstrator and get them to sign here**

……………………………………………………………………………….  **(15 marks)**

The effect that you are seeing (more than two non-zero real values and many significant imaginary values in the frequency domain representation of a 900 Hz sinusoid) is referred to as spectral leakage. The discrete values in X(k) correspond to frequencies at integer multiples of fs/*N*.

The first time that the program was tested, the frequency of the input sequence x(n) was equal to 800 Hz which is an integer multiple of fs/*N*. Intuitively this results in a single non-zero value in X(k)at *k* = 10 (and slightly less intuitively in another non-zero value in X(k)at *k* = 90).

The second time the program was tested, the frequency of the input sequence x(n) was equal to 900 Hz which is not an integer multiple of fs/*N* and there is no single value of *k* that corresponds exactly to that frequency.

Test your function further, using imaginary and complex time-domain data.

### 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. One way of reducing significantly the execution time of your function is to pre-compute and store *twiddle factors* (the *N* different values of  used in the DFT calculation). Remove source file fm4\_dft\_gpio.c from your project and replace it with fm4\_dftw\_gpio.c. Program fm4\_dftw\_gpio.c is supplied in a form similar to that in which program fm4\_dft\_gpio.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 will be used within function dftw() as a lookup table of twiddle factors.

Test program FM4\_dftw\_GPIO.c to check that function dftw() returns the same results as function dft().

**Show this to a demonstrator and get them to sign here**

……………………………………………………………………………….  **(15 marks)**

The use of pre-computed twiddle factors is one factor in the computational efficiency of the fast Fourier transform (FFT). Programs fm4\_fft\_gpio.c and fm4\_fft\_CMSIS\_gpio.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 computational efficiency of different DFT functions can be compared by measuring the time taken to process each function by toggling a GPIO pin while the function is being processed and using an oscilloscope to measure how long the pin is on. Carry out the following steps for programs fm4\_dft\_gpio.c, fm4\_dftw\_gpio.c, fm4\_fft\_gpio.c and fm4\_fft\_CMSIS\_gpio.c. Change the number of points N, in the programs fm4\_dft\_gpio.c and fm4\_dftw\_gpio.c to 128. For each of the four programs

1. Run the program.
2. Connect an oscilloscope probe to GPIO pin P10 on the FM4 S6E2CCA.
3. Measure the length of time that the output is high. The program generates only one pulse so you will need to set the trigger on your oscilloscope accordingly. You can run the program repeatedly by pressing the reset switch on the board.
4. Enter your results in table 1 in the column headed *Execution time (ms) oscilloscope***.**

|  |  |  |  |
| --- | --- | --- | --- |
| Function name | *N* | Execution time (ms)  oscilloscope | Execution time (ms)  MDK-ARM |
| dft() | 128 |  |  |
| dftw() | 128 |  |  |
| fft() | 128 |  |  |
| arm\_cfft\_f32() | 128 |  |  |

Table 1. Execution times for functions dft(), dftw(),fft() and arm\_cfft\_f32().

Another method of assessing the execution times of the functions is to set breakpoints at the the following two program statements

dft(samples);

and

while(1){}

Run the program to the first breakpoint and note the value of *Internal ->* *States* in the *Registers* window on the left hand side of the MDK-ARM debugger display. Run the program to the second breakpoint and note the value of *States* again. The difference between the two values of *States* is equal to the number of instruction cycles taken to execute the program segment between the two breakpoints. Divide that number of instruction cycles by the CPU clock rate (200 MHz) to find the time in seconds taken to execute the function (and to reset the GPIO pin).

Enter the execution times determined this way in the final column of table 1.  **(16 marks)**

### 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 program examples 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 a more intuitive for frame-based processing and will be used here.

### dma-based i/o on the FM4 S6E2CCA

Direct Memory Access (DMA) is a function that is used to transfer data at high speed without using the CPU. Using DMA improves the system performance as the data is moved through a bus which is independent from the CPU bus; therefore, it allows the transfer operation even when the CPU bus is accessed. ARM Cortex-M4 processors from different manufacturers implement DMA slightly differently from each other (although the underlying principles are similar). The Cypress FM4 S6E2CCA MCU has two peripherals that have direct memory access for transferring data, the Descriptor System data Transfer Controller (DSTC) and the more conventional Direct Memory Access Controller (DMAC). The DSTC is a descriptor based DMA which means that instead of save the characteristics of each transfer into registers (size of the transfer, source address, destination address…), all the parameters are packed in 32bits descriptors and stored in the RAM memory. This reduces the size of the peripheral and allows more channels (up to 256 DSTC channels compared to 8 DMAC channels).

To transfer data from and to the I2S peripheral using Direct Memory Access we have to use the DSTC peripheral.

We will use to channels for our application. In function fm4\_wm8731\_init(), one DSTC channel is configured to make DMA transfers between the output buffers arrays, (alternately dma\_tx\_buffer\_ping and dma\_tx\_buffer\_pong) and the I2S peripheral. It generates an interrupt when a transfer of DMA\_BUFFER\_SIZE 32-bit samples has completed.

Another DSTC channel is configured to make DMA transfers between the I2S peripheral and the input buffers in memory (alternately dma\_rx\_buffer\_ping and dma\_rx\_buffer\_pong). It too generates an interrupt when a transfer of DMA\_BUFFER\_SIZE 32-bit samples has completed.

The same interrupt service routine is used for both aforementioned DMA processes. The actions carried out in this routine are simply to assign to pointers rx\_proc\_buffer and tx\_proc\_buffer the values PING or PONG, switch between buffers dma\_tx\_buffer\_ping, dma\_tx\_buffer\_pong, dma\_rx\_buffer\_ping and dma\_rx\_buffer\_pong and to set flags rx\_buffer\_full and tx\_buffer\_empty. These variables are used in function proc\_buffer(). If rx\_proc\_buffer is equal to PING, this indicates that the most recently completed DSTC1 transfer has filled buffer dma\_rx\_buffer\_ping and this data is available to be processed. If tx\_proc\_buffer is equal to PING, this indicates that the most recently completed stream DSTC0 transfer has written the contents of buffer dma\_tx\_buffer\_ping to the I2S peripheral and this buffer is available to be filled with new data.

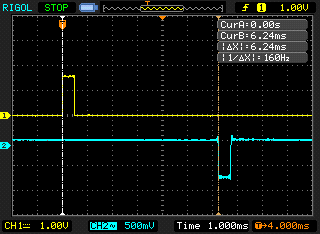
Function main() simply waits until both rx\_buffer\_full and tx\_buffer\_empty flags are set, that is when both DMA transfers have completed, before calling function process\_dma\_buffer(). In program fm4\_loop\_dma.c, function process\_dma\_buffer() simply copies the contents of the most recently filled input buffer (dma\_rx\_buffer\_ping or dma\_rx\_buffer\_pong ) to the most recently emptied output buffer (dma\_tx\_buffer\_ping or dma\_tx\_buffer\_pong ), according to the values of pointers rx\_proc\_buffer and tx\_proc\_buffer. In general, frame-based processing will be carried out in function process\_dma\_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\_dma\_buffer() will be called, every DMA\_BUFFER\_SIZE sampling instants and therefore any processing must be completed within DMA\_BUFFER\_SIZE/fs seconds (or, more strictly speaking, before the next DMA transfer completion).

Run program fm4\_loop\_dma.c and verify its operation using a signal source and oscilloscope or headphones. As supplied, the program reads input from the MIC IN socket on the audio card.

Use a SIGNAL generator and oscilloscope in order to measure the delay introduced by program FM4\_LOOP\_DMA.c

Change the settings of the WM8731 to read data from the LINE IN. Set the oscilloscope to see both input and output channel and the trigger for the input channel. Input and (delayed) output pulses as they appeared on an oscilloscope are shown in figure 5. Notice in Figure 5 that a) the output pulse is inverted (this is a feature of the WM8731 that had probably escaped your attention so far), b) the edges of the output pulse show the ‘ringing’ effect due to the low pass reconstruction filter, and c) the amplitude of the output pulse is approximately 6.5 dB down on that of the input pulse. All three of these effects are expected.



**Figure 5. Example of 0.5 ms duration rectangular pulse delayed by 6.24 ms using program fm4\_loop\_dma.c**

**DMA\_BUFFER\_SIZE = 128, sampling frequency 48 kHz.**

Enter the ANTICIPATED AND MEASURED delays in table 2.

The default value of DMA\_BUFFER\_SIZE (defined in header file fm4\_wm8731\_init.h) is 128 and the sampling rate fs (determined by the parameter passed to function fm4\_wm8731\_init() in program fm4\_loop\_dma.c) is 48 kHz. Hence, the expected delay is DMA\_BUFFER\_SIZE\*2/fs = 5333 µs. **Temporarily** change the value of DMA\_BUFFER\_SIZE defined in header file fm4\_wm8731\_init.h and repeat the experiment in order to fill in the remainder of table 2. Rebuild the project for the altered values of DMA\_BUFFER\_SIZE to take effect. **(16 marks)**

|  |  |  |
| --- | --- | --- |
| DMA\_BUFFER\_SIZE | Measured delay (µs) | DMA\_BUFFER\_SIZE\*2/fs (µs) |
| 256 |  |  |
| 128 |  |  |
| 64 |  |  |
| 32 |  |  |
| 16 |  |  |

**Table 2: delays introduced by using different sizes of DMA ping pong buffer in program fm4\_loop\_dma.c.**

**Sampling frequency fs = 48 kHz.**

**Measure the delay introduced by program fm4\_loop\_intr.c for comparison:**

Delay introduced by program fm4\_loop\_intr.c = **(4 marks)**

After completing this experiment, change the value of DMA\_BUFFER\_SIZE defined in header file fm4\_wm8731\_init.h back to its default value of 128.

### FFT of a signal in real-time

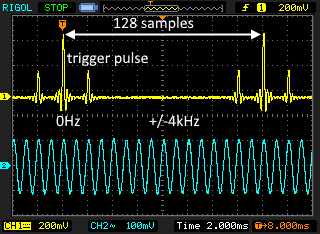
Program fm4\_fft128\_dma.c combines function fft()with a real-time program in order to implement a very simple spectrum analyzer. The program uses the dma-based frame-processing mechanism implemented in program fm4\_loop\_dma.c and calls function fft()from function process\_dma\_buffer(). The while loop in function main()is identical to that in fm4\_loop\_dma.c.

Blocks of *N* = DMA\_BUFFER\_SIZE = 128 real-valued samples are transferred from the ADC, via I2S, to memory using DMA and their *N*-point complex DFTs are computed using function fft(). The scaled magnitudes of the elements of the frequency-domain representations are written to the DAC (again via I2S and using DMA) and also to buffer outbuffer for plotting. The value of DMA\_BUFFER\_SIZE is set in file fm4\_wm8731\_init.h.

Build and run the program.

Use a signal generator connected to LINE IN on the audio card to input a sinusoidal test signal of magnitude approximately 600 mVpp and connect an oscilloscope to HEADPHONE OUT. The input signal and the magnitude of the FFT are output on L and R channels.

Vary the frequency of the input signal between 100 Hz and 4000 Hz. Figure 6 shows an example of what you should see on the oscilloscope (magnitude of the FFT (channel 1) and input to the FFT function (channel 2)). The two smaller pulses correspond to the magnitude of the frequency content of the input signal, computed using the FFT. The larger pulses correspond to trigger values added to the output signal every 128 samples, replacing the magnitude of sample X(0). The smaller pulses correspond also to values in the FFT magnitude data written to the DAC. Due to the characteristics of the DAC (as explored in laboratory #2) the pulses are shaped approximately as sinc functions. You will need to look past this aspect of the output waveform to see the underlying frequency domain information. Figure 6 corresponds to an input signal frequency of 1 kHz.



**Figure 6. Oscilloscope display produced using program fm4\_fft128\_dma.c.**

The data in an output block is ordered such that the first value |X(0)| corresponds to a frequency of 0 Hz. The next 63 (*N*/2 – 1) values (|X(1)| through |X(63)|) correspond to frequencies 62.5 Hz (*fs*/*N*) to 4000 Hz(fs/2), inclusive, in steps of 62.5 Hz. The next 63 values (|X(64)| through |X(127)|) correspond to frequencies of -3937.5 Hz to -62.5 Hz inclusive.

Increase the frequency of the input signal and as it approaches 4 kHz you should see the two smaller pulses move together towards a point halfway between consecutive trigger pulses. Increase the frequency of the input signal beyond 4 kHz and the two smaller pulses should disappear.

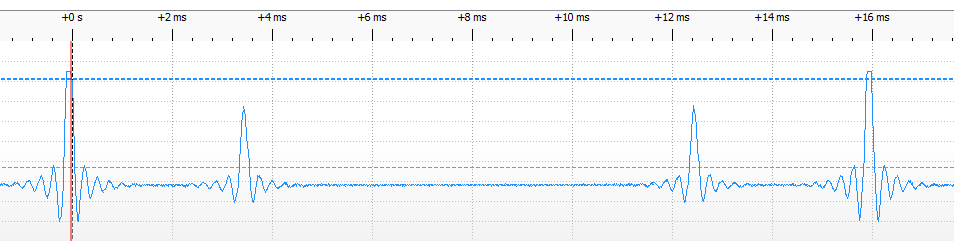
As the frequency of the input signal is varied between 0 and 4 kHz, not only should the two smaller pulses in the oscilloscope display move (relative to the trigger pulses), the shape of the pulses should change too.

### Spectral leakage

If the frequency of the input signal is exactly 1750 Hz, the magnitude of the the DFT of a block of samples should in theory be equal to zero except at two points (|X(28)| through |X(100)|), corresponding to frequencies of +/- 1750Hz. This may be verified by adjusting the frequency of the input signal to 1750 Hz, halting the program, 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 fm4\_plot\_int16(). The results of doing this, for input frequencies of 1750 Hz and 1781 Hz are shown below, along with corresponding oscilloscope traces (essentially the data plotted written to the DAC).



**Figure 7. Partial contents of array outbuffer, plotted using MATLAB function plot\_int16(). Input frequency = 1750 Hz.**

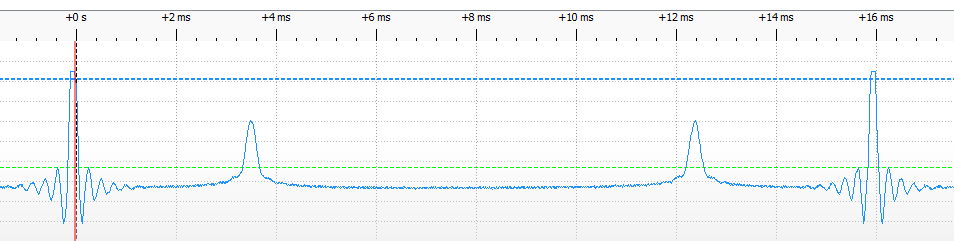


**Figure 8. Oscilloscope trace corresponding to data plotted in figure 7. Input frequency = 1750 Hz. Both the trigger pulse and the smaller pulse representing the magnitude of X(28) are essentially impulse responses from theWM8731 DAC.**

Now adjust the frequency of the input signal to 1781 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 9).



**Figure 9. Partial contents of array outbuffer, plotted using MATLAB function plot\_int16().. Input frequency = 1781 Hz.**



**Figure 10. Oscilloscope trace corresponding to data plotted in figure 9. Input frequency = 1781 Hz.**

### Modifying the program to reduce spectral leakage.

One method of reducing spectral leakage is to multiply the blocks of input samples by a tapered window function prior to computing their DFT. Pressing the user switch on the FM4 board will toggle the program between using program statement

cbuf[i].real = (float)(sample.uint16bit[LEFT]);

and using program statement

cbuf[i].real = (float)(sample.uint16bit[LEFT])\*hamming[i];

The second program statement multiplies a block of 128 real-valued input samples by a 128-point hamming window, read from header file hamm128.h. Figures 11 through 14 show the effect of windowing the blocks of time domain samples before computing their FFT.



**Figure 11. Partial contents of array outbuffer, plotted using MATLAB function fm4\_plot\_int16(). Input frequency = 1750 Hz. Hamming window used.**

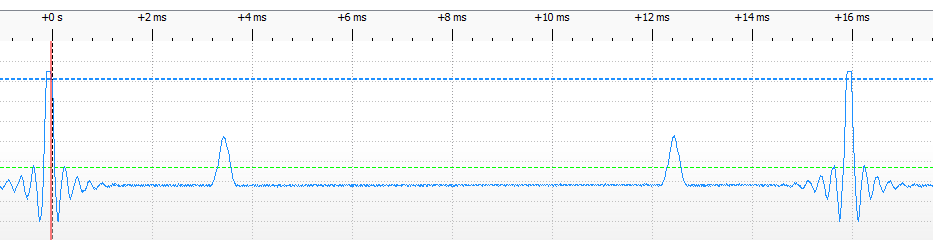


Figure 12. Oscilloscope trace corresponding to data plotted in figure 11. Input frequency = 1750 Hz.



**Figure 13. Partial contents of array outbuffer, plotted using MATLAB function fm4\_plot\_int16(). Input frequency = 1781 Hz. Hamming window used.**

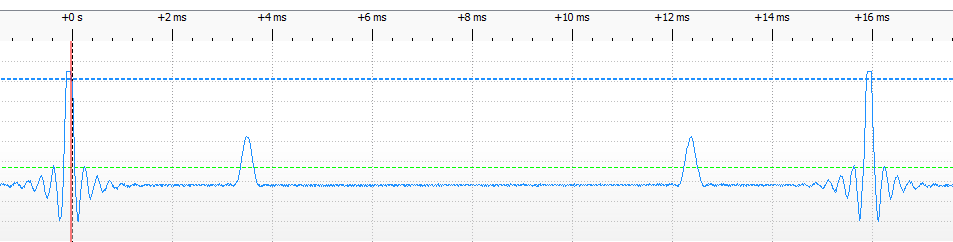


Figure 14. Oscilloscope trace corresponding to data plotted in figure 13. Input frequency = 1781 Hz.

It’s instructive to display both the output (left channel) from program fm4\_fft128\_dma.c and the windowed blocks of input samples (right channel) on the oscilloscope at the same time as shown in figure 15. You may also experiment with other window functions by substituting for header file hamm128.h in program fm4\_fft128\_dma.c.

Header files bartlett128.h, blackman128.h, and kaiser128.h are supplied for this purpose. Note that the arrays declared in these header files have identifiers different to hamming, e.g. bartlett.

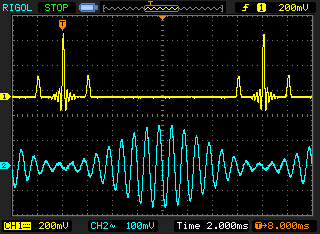


Figure 15. Oscilloscope trace showing effect of Hamming window.

### Real-time performance of program FM4\_fFT128\_dma.c

The time taken to execute function process\_dma\_buffer() in program fm4\_fft128\_dma.c may be assessed by connecting an oscilloscope probe to GPIO pin P10. This GPIO pin is set just before the call to function process\_dma\_buffer() and reset after its execution.

Measure the execution time for function process\_dma\_buffer() and compare this with the corresponding measurement recorded in table 1.

Can you account for the discrepancy in these values?

**(2 marks)**

How might the execution time for function process\_dma\_buffer() be reduced?

**(4 marks)**

How long did your dftw() function take to compute a 128-point DFT?

**(4 marks)**

Ought that to work in real-time?

**(4 marks)**

**Modify program fm4\_ft128\_dma.c**, replacing function fft() with your function dftw(). Confirm that the modified program produces a similar output signal in response to sinusoidal inputs, and measure the time taken to execute function proc\_dma\_buffer().

You will have to initialize the twiddle factors used by function dftw()exactly as you did in program fm4\_dftw.c and not as they are initialized for function fm4\_fft()in program fft128\_dma.c.

**Demonstrate real-time operation of your program incorporating function dftw() to a demonstrator and get them to sign here**

…………………………………………………………… **(20 marks)**

# conclusions

In this exercise you have demonstrated the relative computational efficiency of the fast Fourier transform algorithm. The use of DMA-based i/o and frame-based processing have been introduced.