B39SB 2019 Lab Exercise #1:   
Introduction to the CypresS FM4 Starter Kit

# Overview

The Cypress FM4 Pioneer board (FME-176L-S6E2CC-ETH Starter Kit) is a low cost development platform featuring a 200MHz ARM Cortex-M4 based processor (FM4 S6E2CCA). It connects to a host PC via a USB cable using a CMSIS programming and debugging tool (CMSIS-DAP). 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 FM4 S6E2CCA. Real-time audio i/o is provided by a Wolfson WM8731 codec included on the development board. This first lab exercise introduces the use of the Cypress FM4 board and several of the procedures and techniques that will be used in subsequent lab exercises.

# 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* and M*ATLAB*, and suitable connecting cables.

GPIO pin P10

DIAGNOSTIC\_PIN

DAC12

output

HP OUT

LINE IN

MIC IN

USB to host PC

GND

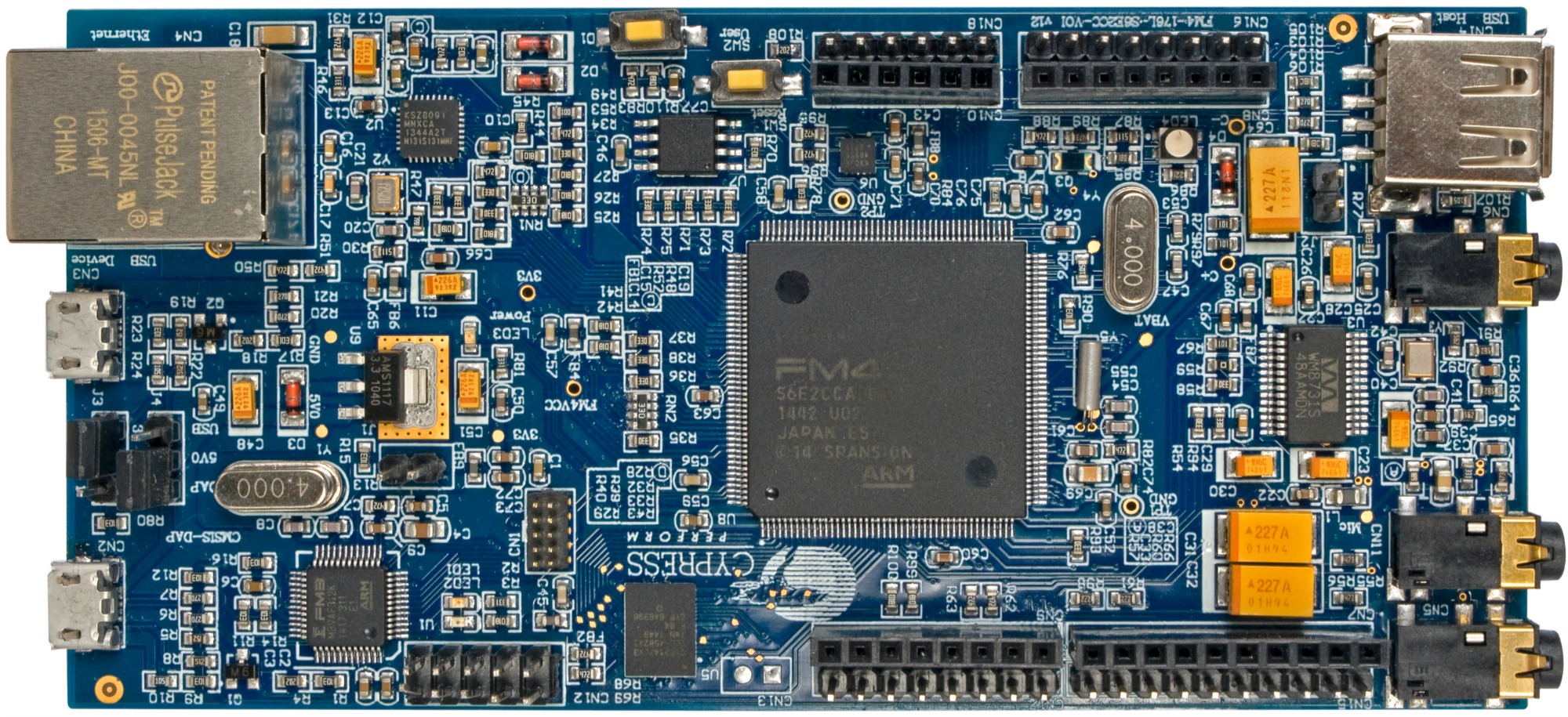


Figure 1: Cypress FM4-176L-S6E2CC-ETH Starter Kit

### Basic digital signal processing system

A basic DSP system, suitable for processing audio frequency signals comprises a digital signal processor and analogue interfaces as shown in figure 2. The FM4 board provide just such a system, using the Cortex-M4 floating point processor and the WM8731 codec. The term codec refers to the *coding* of analogue waveforms as digital signals and the *decoding* of digital signals as analogue waveforms. The WM8731 codec performs both the analogue to digital conversion (ADC) and digital to analogue conversion (DAC) functions shown in figure 2.



Figure 2: Basic Digital Signal Processing System

Program code may be developed, downloaded and run on the FM4 board using the *Keil MDK-ARM* integrated development environment. You will not be required to write C programs from scratch but will learn how to compile, link, download and run the example programs provided and in some cases make minor modifications to their source files. You will learn how to use a subset of the features provided by MDK-ARM in order to do this (Using the full capabilities of MDK-ARM is beyond the scope of this set of laboratory exercises). The emphasis of this set of laboratory exercises is on the digital signal processing *concepts* implemented by the aforementioned programs.

Most of the example programs are 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 following examples introduce some of the features of *MDK-ARM* and the Cypress FM4 board. In addition you will learn how to use *MATLAB* in order to generate, observe and analyse audio signals.

### Basic analogue input and output using the Cypress FM4 board

The C language source file for a program that simply copies input samples read from the WM8731 ADC back to the WM8731 DAC is listed in figure 3. In effect, the program connects the microphone to the headphone output socket on the same board. This simple program is important because many of the other example programs that will be used are based on the same interrupt-driven real-time model. It is worth taking time to ensure that you understand how program fm4\_loop\_intr.c works.

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

// fm4\_loop\_intr.c

#include "fm4\_wm8731\_init.h"

void PRGCRC\_I2S\_IRQHandler(void)

{

union WM8731\_data sample;

gpio\_toggle(DIAGNOSTIC\_PIN);

sample.uint32bit = i2s\_rx();

i2s\_tx(sample.uint32bit);

NVIC\_ClearPendingIRQ(PRGCRC\_I2S\_IRQn);

}

int main(void)

{

fm4\_wm8731\_init (FS\_48000\_HZ, WM8731\_MIC\_IN, IO\_METHOD\_INTR,

WM8731\_HP\_OUT\_GAIN\_0\_DB, WM8731\_LINE\_IN\_GAIN\_0\_DB);

while(1){}

}

Figure 3: Listing of program fm4\_loop\_intr.c

### PROGRAM OPERATION

The C source file fm4\_loop\_intr.c listed in figure 3 looks more complicated than it really is. Its operation is as follows.

In the function main(), an initialisation function fm4\_wm8731\_init() is called. This sets up i/o and interrupts such that the WM8731 codec will sample the analogue input signal, and the processor will be interrupted, at the sampling frequency determined by the parameter FS\_48000\_HZ passed to the function. Additionally, the parameter WM8731\_MIC\_IN specifies that input to the WM8731 ADC will come from the microphone input on the FM4 board. Parameter IO\_METHOD\_INTR , passed to function fm4\_wm8731\_init() determines that interrupt-based (as opposed to polling- or DMA-based) i/o will be used by the program.

There is no need to understand the details of the initialisation carried out by function fm4\_wm8731\_init(). Suffice to say that after it has been called, interrupts generated by the I2S peripheral block in the S6E2CC processor connected to the WM8731 will be enabled, and each time an interrupt occurs the interrupt service routine function PRGCRC\_I2S\_IRQHandler()will be called. One interrupt per sampling period will occur and both left and right channels are processed in the same interrupt service routine. Following initialisation, the function main()enters an endless while() loop, doing nothing but waiting for interrupts.

Function PRGCRC\_I2S\_IRQHandler() reads 32 bits of data from the I2S peripheral using the function i2s\_rx(). This function copies data from the I2S input FIFO to the sample variable. The data comprises 16 bits for the right channel and 16 bits for the left channel, i.e. a total of 32 bits. sample is a variable of type WM8731\_data, the union of of a single 32-bit variable and an array of two 16-bit variables, allowing both left and right channel data to be copied (to the I2S transmit FIFO using function i2s\_tx()) as a single entity in this case or, in other programs, for 16-bit left and right channel sample values to be accessed individually.

### Running the Program

1. Open µVision 5 project fm4\_project by double clicking on its icon in fm4\_folder. You should see a project structure similar to that in figure 4.

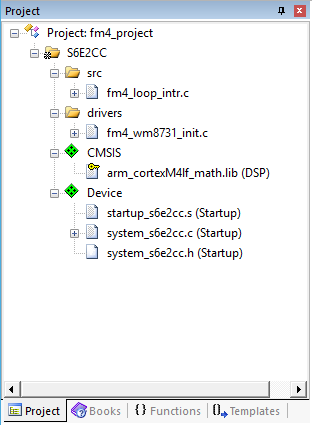


Figure 4. Screenshot of Keil µVision (MDK-ARM) showing initial project structure.

1. Files fm4\_loop\_intr.c , fm4\_wm8731\_init.c and fm4\_wm8731\_init.h were written specifically for this lab exercise. All other files are part of the *Keil MDK-ARM* package.
2. Connect the FM4 board to the host PC using a USB cable.
3. Plug headphones and a microphone into the HP OUT and MIC IN sockets on the FM4 Starter Kit. The socket on the corner (CN5) is the headphones socket. The one next to it (CN11) is the microphone socket. The third one, (CN6) is the line in socket.
4. Build the project by selecting *Build target* from the *Project* menu or by clicking on the *Build* toolbar button.
5. Switch to the debugger (and download the executable code into flash memory) by clicking on the *Start/Stop Debug Session* toolbar button.
6. Once the debugger windows have appeared, click on the *Run* toolbar button.
7. Once the program is running, you should be able to hear the sounds picked up by the microphone in the headphones. Depending on the characteristics of the microphone and headphones you are using, the sound may be louder or quieter. Using some inexpensive headphones the sound may be quite quiet. If you are uncertain whether you are hearing anything through the headphones, try blowing gently onto the microphone.

By passing parameter WM8731\_LINE\_IN or WM8731\_MIC\_IN to function fm4\_wm8731\_init() (by editing source file fm4\_loop\_intr.c and re-building, downloading and running) you can listen to a signal input via either the LINE IN socket or the MIC IN socket.

### Use of GPIO pin for timing indication

In several example programs 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 fm4\_loop\_intr.c, GPIO pin P10 is toggled each time an interrupt occurs.

gpio\_toggle(DIAGNOSTIC\_PIN);

Since interrupts should occur once per sampling period, the expected signal on GPIO pin P10 is a square wave of frequency 24 kHz (sampling rate is 48 kHz)

Connect an oscilloscope probe to GPIO pin P10 on the FM4 board to confirm this.

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

gpio\_set(DIAGNOSTIC\_PIN, HIGH);

gpio\_set(DIAGNOSTIC\_PIN, LOW);

### Delaying the signal

Some simple, yet striking, effects can be achieved simply be delaying the samples as they pass from input to output. Program fm4\_delay\_intr.c, listed in figure 5, demonstrates this. A delay line is implemented using the array buffer to store samples as they are read from the ADC. Once the array is full, the program overwrites the oldest stored input sample with the current or newest, input sample. Just prior to overwriting the oldest stored input sample in buffer, that sample is retrieved, added to the current input sample and written to the DAC. The length of the delay is determined by the value of the constant DELAY\_BUF\_SIZE. As supplied, this is equal to 16000 samples, corresponding to a delay of 333 ms at a sampling rate of 48 kHz.

// fm4\_delay\_intr.c

#include "fm4\_wm8731\_init.h"

#define DELAY\_BUF\_SIZE 16000

uint32\_t buffer[DELAY\_BUF\_SIZE];

int16\_t buf\_ptr = 0;

void PRGCRC\_I2S\_IRQHandler(void)

{

union WM8731\_data input\_sample, delayed\_sample;

gpio\_toggle(DIAGNOSTIC\_PIN);

input\_sample.uint32bit = i2s\_rx();

delayed\_sample.uint32bit = buffer[buf\_ptr];

buffer[buf\_ptr] = input\_sample.uint32bit;

buf\_ptr = (buf\_ptr+1)%DELAY\_BUF\_SIZE;

i2s\_tx(delayed\_sample.uint32bit);

NVIC\_ClearPendingIRQ(PRGCRC\_I2S\_IRQn);

}

int main(void)

{

fm4\_wm8731\_init (FS\_48000\_HZ, WM8731\_MIC\_IN, IO\_METHOD\_INTR,

WM8731\_HP\_OUT\_GAIN\_0\_DB, WM8731\_LINE\_IN\_GAIN\_0\_DB);

while(1){}

}

Figure 5: Listing of program fm4\_delay\_intr.c



Figure 6: Block Diagram representation of program fm4\_delay\_intr.c

### ADDING FEEDBACK

By feeding back a fraction of the output of the delay line to its input, a fading echo effect can be realised. Program fm4\_echo\_intr.c, listed in figure 7, does this. Experiment with different values of the constants BUF\_SIZE and GAIN (the delay in seconds is equal to BUF\_SIZE divided by the sampling frequency in Hz and the fraction of the delayed signal fed back is equal to GAIN.)

What would happen if the value of GAIN were made greater than or equal to 1?

// fm4\_echo\_intr.c

#include "fm4\_wm8731\_init.h"

#define DELAY\_BUF\_SIZE 16000

#define GAIN 0.6f

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

int16\_t buf\_ptr = 0;

void PRGCRC\_I2S\_IRQHandler(void)

{

union WM8731\_data sample;

int16\_t delayed\_sample, output\_sample, input\_sample;

gpio\_toggle(DIAGNOSTIC\_PIN);

sample.uint32bit = i2s\_rx();

input\_sample = sample.uint16bit[LEFT];

delayed\_sample = buffer[buf\_ptr];

output\_sample = input\_sample + delayed\_sample;

buffer[buf\_ptr] = input\_sample + delayed\_sample\*GAIN;

buf\_ptr = (buf\_ptr+1)%DELAY\_BUF\_SIZE;

sample.uint16bit[LEFT] = output\_sample;

sample.uint16bit[RIGHT] = output\_sample;

i2s\_tx(sample.uint32bit);

NVIC\_ClearPendingIRQ(PRGCRC\_I2S\_IRQn);

}

int main(void)

{

fm4\_wm8731\_init (FS\_48000\_HZ, WM8731\_MIC\_IN, IO\_METHOD\_INTR,

WM8731\_HP\_OUT\_GAIN\_0\_DB, WM8731\_LINE\_IN\_GAIN\_0\_DB);

while(1){}

}

Figure 7: Listing of program fm4\_echo\_intr.c

Study the program listing in figure 7 and draw a block diagram, in the space provided for figure 8, of the system it implements. In the space provided for figure 9, sketch what you think its response to a unit impulse would be (with a gain of 0.6 and a buffer size of 2000 samples).

Figure 8: Block diagram representation of program fm4\_echo\_intr.c

Figure 9: Impulse response of program fm4\_echo\_intr.c (BUF\_SIZE = 2000, GAIN = 0.6)

### Real-Time Sine Wave Generation

Program Operation

The C source file fm4\_sine8\_intr.c listed in figure 5 generates a sinusoidal signal using interrupts and a table lookup method. Its operation is as follows. An eight point lookup table is initialised in the array sine\_table such that the value of sine\_table[i] is equal to



Where, in this case, . The LOOP\_LENGTH values in array sine\_table are samples of exactly one cycle of a sinusoid.

Just as in the previous examples, in function main(), initialisation function fm4\_wm8731\_init() is called. This sets up i/o and interrupts such that the WM8731 codec will sample the analogue input signal, and interrupt the processor, at a frequency determined by the parameter value FS\_8000\_HZ.

**In this example, a sampling rate of 8kHz has been specified and interrupts will occur every 0.125ms.**

Following the call to function fm4\_wm8731\_init(), function main() enters an endless loop, doing nothing but waiting for interrupts (which will occur once per sampling period).

At each sampling instant, the interrupt service routine function PRGCRC\_I2S\_IRQHandler() is called and in that routine the most important program statements are executed. Sample values read from array sine\_table are written to both channels of 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 is due to the eight samples per cycle output at a rate of 8 kHz.**

As will be investigated in more detail in exercise #2, the WM8731 DAC contains a digital low pass reconstruction filter which interpolates between output sample values to give a smooth sinusoidal analogue output signal as shown in figure 6.

// fm4\_sine8\_intr.c

#include "fm4\_wm8731\_init.h"

#define LOOPLENGTH 8

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

static int sine\_ptr = 0;

void PRGCRC\_I2S\_IRQHandler(void)

{

union WM8731\_data sample;

gpio\_toggle(DIAGNOSTIC\_PIN);

sample.uint16bit[LEFT] = sine\_table[sine\_ptr];

sample.uint16bit[RIGHT] = sine\_table[sine\_ptr];

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

i2s\_tx(sample.uint32bit);

NVIC\_ClearPendingIRQ(PRGCRC\_I2S\_IRQn);

}

int main(void)

{

fm4\_wm8731\_init (FS\_8000\_HZ, WM8731\_MIC\_IN, IO\_METHOD\_INTR,

WM8731\_HP\_OUT\_GAIN\_0\_DB, WM8731\_LINE\_IN\_GAIN\_0\_DB);

while(1){}

}

Figure 5: Listing of program fm4\_sine8\_intr.c

Connect one channel of the audio card LINE OUT output to an oscilloscope, and verify that the output signal is a 1kHz sinusoid using both time-domain and frequency-domain (Math FFT function) oscilloscope displays.

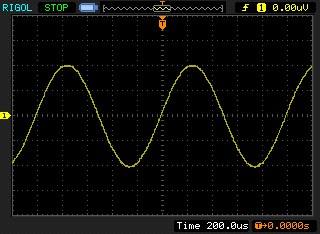


Figure 6: Analogue output generated using program fm4\_sine8\_intr.c.

### Modifying program FM4\_sine8\_intr.c

Edit the source file fm4\_sine8\_intr.c so as 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 initialised contents of the array sine\_table (and by changing the value of the constant LOOP\_SIZE accordingly) in the following lines in the source file.

#define LOOPLENGTH 8

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

**Do not change any other program statements**. Record the combinations of LOOP\_SIZE and sine\_table with which you achieve these results in the spaces below.

500 Hz sinewave

LOOPLENGTH =

sine\_table =

2000 Hz sinewave

LOOPLENGTH =

sine\_table =

3000 Hz sinewave

LOOPLENGTH =

sine\_table=

### VIEWING PROGRAM OUTPUT USING MATLAB

Program fm4\_sine8\_buf\_intr.c is very similar to program fm4\_sine8\_intr.c but it stores the most recent BUFFER\_SIZE 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.

Run the program and then halt it by clicking on the *Stop* toolbar button. Type the variable name buffer as the *Address* in the debugger's *Memory 1* window and set the displayed data type to *Decimal* and *Float* as shown in figure 8.

// fm4\_sine8\_buf\_intr.c

#include "fm4\_wm8731\_init.h"

#define LOOPLENGTH 8

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

static int sine\_ptr = 0;

#define BUFFER\_SIZE 100

float32\_t buffer[BUFFER\_SIZE];

static int buf\_ptr=0;

void PRGCRC\_I2S\_IRQHandler(void)

{

union WM8731\_data sample;

gpio\_toggle(DIAGNOSTIC\_PIN);

sample.uint16bit[LEFT] = sine\_table[sine\_ptr];

sample.uint16bit[RIGHT] = sine\_table[sine\_ptr];

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

buffer[buf\_ptr] = sample.uint16bit[LEFT];

buf\_ptr = (buf\_ptr+1) % BUFFER\_SIZE;

i2s\_tx(sample.uint32bit);

NVIC\_ClearPendingIRQ(PRGCRC\_I2S\_IRQn);

}

int main(void)

{

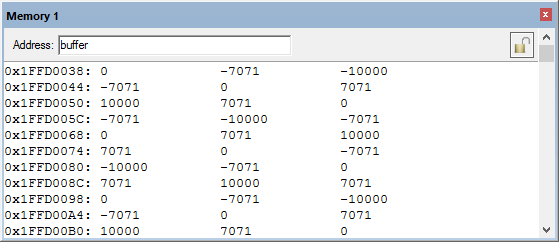
fm4\_wm8731\_init (FS\_48000\_HZ, WM8731\_MIC\_IN, IO\_METHOD\_INTR,

WM8731\_HP\_OUT\_GAIN\_0\_DB, WM8731\_LINE\_IN\_GAIN\_0\_DB);

while(1){}

}

Figure 7: Listing of program fm4\_sine8\_buf\_intr.c



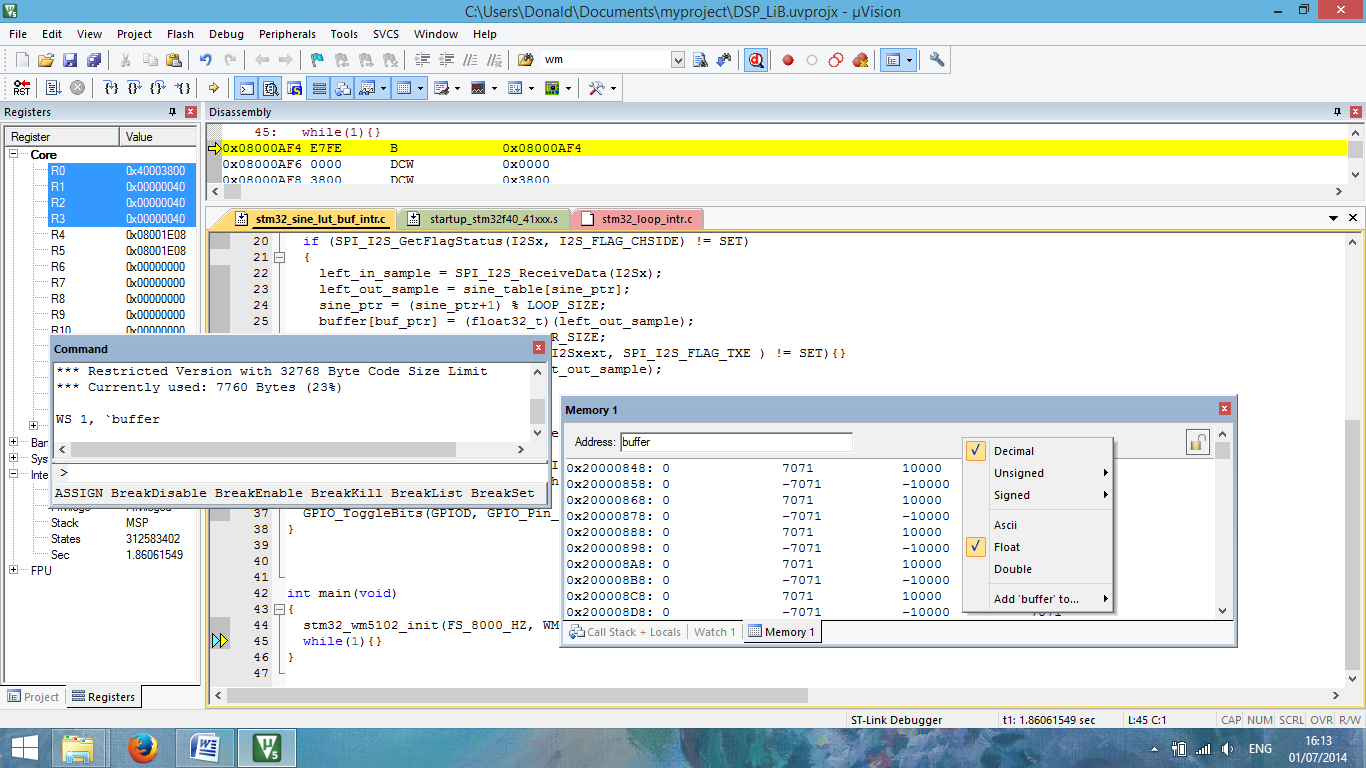


Figure 8: *Memory 1* window showing the contents of array buffer.

The start address of array buffer will be displayed in the top left hand corner of the window. The end address should be the start address plus 0x190 (bytes) representing 100 32-bit sample values.

Type 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>

for example, SAVE sinusoid.dat 0x1FFD0038, 0x1FFD01C8

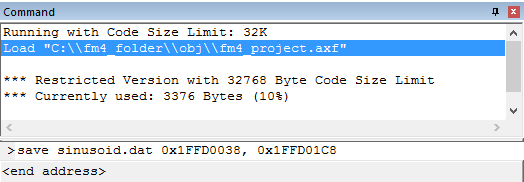


Figure 9: Saving data to a file in *MDK-ARM* debugger *Command* window.

You can then use MATLAB function fm4\_logfft.m to obtain a graphical representation of the contents of the buffer that have been stored in the file.

# 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 exercises.