Lab Exercise #1: instructor version  
Introduction to the FREESCALE FRDM-K64F and Wolfson Pi Audio Card

# Overview

The Freescale FRDM-K64F is a low cost development platform featuring a 120 MHz an ARM Cortex-M4 based processor. It connects to a host PC via a USB cable using a CMSIS 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 FRDM-K64F. Real-time audio i/o is provided by the Wolfson Pi Audio Card which may be easily connected to the FRDM-K64F. This first lab exercise introduces the use of the FRDM-K64F and Wolfson Pi Audio Card and several of the procedures and techniques that will be used in subsequent lab exercises.

Instructor note: Items in bright red are solutions to be deleted before posting for students.

# Details

## Hardware

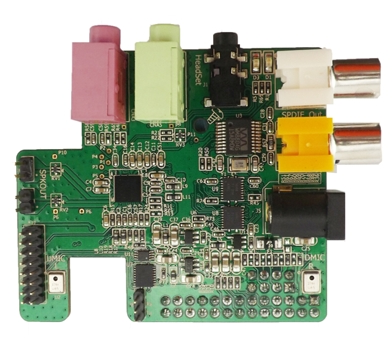
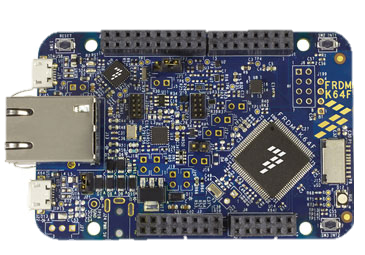
To carry out this exercise you will need a Freescale FRDM-K64F, a Wolfson Pi Audio Card, an oscilloscope, an audio frequency signal generator, a PC running *Keil MDK-ARM*, *GoldWave* and M*ATLAB*, and suitable connecting cables.

Open SDA USB

Headphone Out /Mic in

Line Out

Line In



Reset button

SPDIF In

SPDIF Out

AUX Power In

ARM Cortex M4

Dig. Microphone (R)

Dig. Microphone (L)

User button

RGB LED

Figure 1: Freescale FRDM-K64F and Wofson Audio Card

### 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 FRDM-K64F and Wolfson Pi Audio Card provide just such a system, using the Cortex-M4 floating point processor on the FRDM-K64F and the WM5102 codec on the Wolfson Pi Audio Card. The term codec refers to the *coding* of analogue waveforms as digital signals and the *decoding* of digital signals as analogue waveforms. The WM5102 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 developed, downloaded and run on the FRDM-K64F 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 succinct, 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*, the FREESCALE FRDM-K64F and Wolfson Pi Audio Card. In addition you will learn how to use *MATLAB* and *GoldWave* in order to generate, observe and analyse audio signals.

### Basic analogue input and output using the FREESCALE FRDM-K64F and Wolfson Pi Audio Card

The C language source file for a program that simply copies input samples read from the WM5102 ADC back to the WM5102 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 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.

1. // loop\_intr.c
2. #include "audio.h"
3. volatile int16\_t audio\_chR=0;
4. volatile int16\_t audio\_chL=0;
5. void I2S\_HANDLER(void) {
6. gpio\_toggle(TEST\_PIN);
7. audio\_IN = i2s\_rx(); //32-bits; 16-bits channel left + 16-bits channel right
8. audio\_chL = (audio\_IN & 0x0000FFFF);
9. audio\_chR = ((audio\_IN >>16)& 0x0000FFFF);
11. audio\_OUT = ((audio\_chR<<16 & 0xFFFF0000)) + (audio\_chL & 0x0000FFFF);
12. i2s\_tx(audio\_OUT);
13. }
14. int main(void)
15. {
16. gpio\_set\_mode(TEST\_PIN,Output);
17. audio\_init ( hz48000, dmic\_in, intr, I2S\_HANDLER);
19. while(1){}
20. }

Figure 3: Listing of program loop\_intr.c

### PROGRAM OPERATION

The C source file 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 audio\_init() is called. This sets up i/o and interrupts such that the WM5102 codec will sample the analogue input signal, and interrupt the processor, at the sampling frequency determined by the parameter hz4800 passed to the function. Additionally, the parameter mic\_in specifies that input to the WM5102 ADC will come from the microphone line IN on the Wolfson Pi Audio Card. Parameter intr and I2S\_HANDLER passed to function audio\_init()determines that interrupt-based (as opposed to polling- or DMA-based) i/o will be used by the program, and the name of the interruption.

There is no need to understand the details of the initialization carried out by function audio\_init(). Suffice to say that after it has been called, FRDM-K64F core interrupts generated by the I2S peripheral connected to the WM5102 will be enabled and each time an interrupt occurs, the interrupt service routine function I2S\_HANDLER()will be called. One interrupt per sampling period will occur – both left and right channel are processed at the same interruption. Following initialization, the function main()enters an endless while() loop, doing nothing but waiting for interrupts.

Function I2S\_HANDLER() reads an input sample from the I2S peripheral using the function i2s\_rx() at line 12. This function reads the data from the I2S input FIFO and it is moved into the audio\_IN variable. The data is composed of 16 bits for the right channel and 16 bits for the left channel. To be able to process the data, the channels need to be separate, e.g.

audio\_chL = (audio\_IN & 0x0000FFFF);

audio\_chR = ((audio\_IN >>16)& 0x0000FFFF);

It writes that value as an output sample to the I2S peripheral, e.g.

left\_out\_sample = left\_in\_sample;

SPI\_I2S\_SendData(I2Sxext, left\_out\_sample);

To bring them back together we use,

audio\_OUT = ((audio\_chR<<16 & 0xFFFF0000)) + (audio\_chL & 0x0000FFFF);

Finally it is sent back to the I2S peripheral using the i2s\_tx() that copies the data into the output FIFO

### Running the Program

The following steps assume that you have installed *Keil MDK-ARM* and extracted the LAB\_1 project folder from the DSP\_LiB file exactly as described in the document *Before Starting with the DSP LiB*.

1. Open µVision 5 project *DSP\_LiB* by double clicking on its icon in the LAB\_1 folder. You should see a project structure similar to that in figure 4.

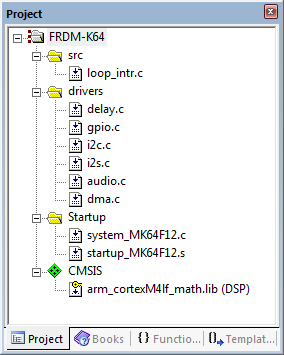


Figure 4. Snapshot of Keil µVision showing the project structure

1. Files loop\_intr.c and the files inside the drivers folder are supplied with the DSP LiB files. All other files are part of the *MDK-ARM* package.
2. Connect the FRDM-K64F to the host PC using a USB cable (it is assumed that you have already connected the Wolfson Pi Audio Card to the FRDM-K64F).
3. Plug headphones or a headset and a microphone into the headset jack socket on the Wolfson Pi Audio Card.
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 headset or headphones. Depending on the characteristics of the microphone and headset or headphones you are using, the sound may be louder or quieter. Using some smartphone headsets, the sounds from the environment surrounding the audio card may be clearly audible. Using some inexpensive headphones the sound is quite quiet. If you cannot hear anything, try blowing gently onto the microphone.

By passing parameters line\_in, mic\_in or dmic\_in to function audio\_init() (by editing source file loop\_intr.c and re-building, downloading and running) you can listen to a signal input via the LINE IN socket on the audio card, via the microphone in a headset or capturing the sound with the on-board digital microphones. By default, analogue output is routed simultaneously to the headset and to the LINE OUT socket.

### Use of GPIO pin for timing indication

In several example programs the state (high or low) of one GPIO pin is used so that by connecting an oscilloscope to that pin an indication of the execution of a program may be obtained. We use for this purpose the pin PTE 24 that can be easily read from J2 #24 on our board.

In the case of program loop\_intr.c, the pin is toggled each time an interrupt occurs (line 10).

gpio\_toggle(TEST\_PIN);

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

Connect an oscilloscope probe to the selected pin on the FRDM-K64F to confirm this.

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

gpio\_set(TEST\_PIN, HIGH);

gpio\_set(TEST\_PIN, LOW);

The characteristics of the GPIO pin are configured with function gpio\_set\_mode() at line 22. All the GPIOs need to be configured with the function gpio\_set\_mode() before being use.

### Delaying the signal

Some simple, yet striking, effects can be achieved simply be delaying the samples as they pass from input to output. Program 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 output 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 24000 samples, corresponding to a delay of 500 ms at a sampling rate of 48 kHz.

1. // delay\_intr.c
2. #include "audio.h"
3. volatile int16\_t audio\_chR=0;
4. volatile int16\_t audio\_chL=0;
5. #define DELAY\_BUF\_SIZE 24000
6. int16\_t buffer[DELAY\_BUF\_SIZE];
7. int16\_t i = 0;
9. void I2S\_HANDLER(void) {
10. int16\_t delayed\_sample;
11. int16\_t audio\_out\_chL = 0;
13. audio\_IN = i2s\_rx();
14. audio\_chL = (audio\_IN & 0x0000FFFF);
15. audio\_chR = ((audio\_IN >>16)& 0x0000FFFF);
16. delayed\_sample = buffer[i];
17. audio\_out\_chL = delayed\_sample + audio\_chL;
18. buffer[i] = audio\_chL;
19. i = (i+1) % DELAY\_BUF\_SIZE;
21. audio\_OUT = ((audio\_chR<<16 & 0xFFFF0000)) + (audio\_out\_chL & 0x0000FFFF);
22. i2s\_tx(audio\_OUT);
23. }
24. int main(void)
25. {
26. audio\_init ( hz48000, dmic\_in, intr, I2S\_HANDLER);
28. while(1){}
29. }

Figure 5: Listing of program delay\_intr.c



Figure 6: Block Diagram representation of program 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 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?

If the value of GAIN is made equal to or greater than 1.0, the amplitude of the output signal will increase unstably.

1. // echo\_intr.c
2. #include "audio.h"
3. volatile int16\_t audio\_chR=0;
4. volatile int16\_t audio\_chL=0;
5. #define DELAY\_BUF\_SIZE 16000
6. #define GAIN 0.6f
7. int16\_t buffer[DELAY\_BUF\_SIZE];
8. int16\_t buf\_ptr = 0;
9. void I2S\_HANDLER(void) {
10. int16\_t delayed\_sample;
11. int16\_t audio\_out\_chL = 0;
13. audio\_IN = i2s\_rx();
14. audio\_chL = (audio\_IN & 0x0000FFFF);
15. audio\_chR = ((audio\_IN >>16)& 0x0000FFFF);
16. delayed\_sample = buffer[buf\_ptr];
17. audio\_out\_chL = delayed\_sample + audio\_chL;
18. buffer[buf\_ptr] = audio\_chL + delayed\_sample\*GAIN;
19. buf\_ptr = (buf\_ptr+1)%DELAY\_BUF\_SIZE;
21. audio\_OUT = ((audio\_chR<<16 & 0xFFFF0000)) + (audio\_out\_chL & 0x0000FFFF);
22. i2s\_tx(audio\_OUT);
23. }
24. int main(void)
25. {
26. audio\_init ( hz48000, dmic\_in, intr, I2S\_HANDLER);
27. while(1){}
28. }

Figure 7: Listing of program echo\_intr.c

Study the program listing in figure 7 and draw a block diagram, in the space for figure 8, of the system it implements. In the space 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 echo\_intr.c



1

0. 75

0. 50

0.25

0

0.13

0.22

0.36

0.6

X=125ms/div Y=0.25units/div

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

### Real-Time Sine Wave Generation

Program Operation

The C source file sine\_lut\_intr.c listed in figure 10 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 audio\_init() is called. This sets up i/o and interrupts such that the WM5102 codec will sample the analogue input signal, and interrupt the processor, at a frequency determined by the parameter value hz8000.

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

Following the call to function audio\_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 I2S\_HANDLER() is called and in that routine the most important program statements are executed. The sample values read from array sine\_table are written into 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 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 WM5102 DAC contains a low pass reconstruction filter which interpolates between output sample values to give a smooth sinusoidal analogue output signal as shown in figure 11.

1. // sine\_lut\_intr.c
2. #include "audio.h"
3. volatile int16\_t audio\_chR=0;
4. volatile int16\_t audio\_chL=0;
5. #define LOOP\_SIZE 8
6. int16\_t sine\_table[LOOP\_SIZE] = {0, 7071, 10000, 7071, 0, -7071, -10000, -7071};
7. static int sine\_ptr = 0;
9. void I2S\_HANDLER(void) {
10. audio\_IN = i2s\_rx();
11. audio\_chL = (audio\_IN & 0x0000FFFF);
12. audio\_chR = ((audio\_IN >>16)& 0x0000FFFF);
13. audio\_chL = sine\_table[sine\_ptr];
14. audio\_chR = sine\_table[sine\_ptr];
15. sine\_ptr = (sine\_ptr+1) % LOOP\_SIZE;
17. audio\_OUT = ((audio\_chR<<16 & 0xFFFF0000)) + (audio\_chL & 0x0000FFFF);
18. i2s\_tx(audio\_OUT);
19. }
20. int main(void)
21. {
22. audio\_init ( hz8000, dmic\_in, intr, I2S\_HANDLER);
24. while(1){}
25. }

Figure 10: Listing of program sine\_lut\_intr.c

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

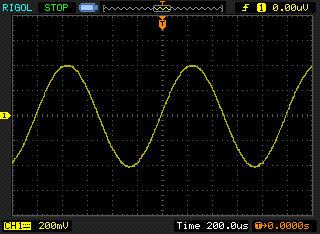


Figure 11: Analogue output generated by program sine\_lut\_intr.c.

### Modifying program sine8\_intr.c

Edit the source file sine\_lut\_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) on lines 8 and 9. **Do not change any other program statements**. Record the combinations of LOOP\_SIZE 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

Program sine\_lut\_buf.c is very similar to program sine\_lut\_intr.c but it also 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. Set the displayed data type to *Decimal* and *Float* as shown in figure 13.

1. // sine\_lut\_buf\_intr.c
2. #include "audio.h"
3. volatile int16\_t audio\_chR=0;
4. volatile int16\_t audio\_chL=0;
5. #define LOOP\_SIZE 8
6. #define BUFFER\_SIZE 100
7. int16\_t sine\_table[LOOP\_SIZE] = {0, 7071, 10000, 7071, 0, -7071, -10000, -7071};
8. static int sine\_ptr = 0;
9. float32\_t buffer[BUFFER\_SIZE];
10. static int buf\_ptr=0;
11. void I2S\_HANDLER(void) {
12. audio\_IN = i2s\_rx();
13. audio\_chL = (audio\_IN & 0x0000FFFF);
14. audio\_chR = ((audio\_IN >>16)& 0x0000FFFF);
15. audio\_chL = sine\_table[sine\_ptr];
16. audio\_chR = sine\_table[sine\_ptr];
17. sine\_ptr = (sine\_ptr+1) % LOOP\_SIZE;
18. buf\_ptr = (buf\_ptr+1) % BUFFER\_SIZE;
20. audio\_OUT = ((audio\_chR<<16 & 0xFFFF0000)) + (audio\_chL & 0x0000FFFF);
21. i2s\_tx(audio\_OUT);
22. }
23. int main(void)
24. {
25. audio\_init ( hz8000, dmic\_in, intr, I2S\_HANDLER);
27. while(1){}
28. }

Figure 12: Listing of program sine\_lut\_buf\_intr.c

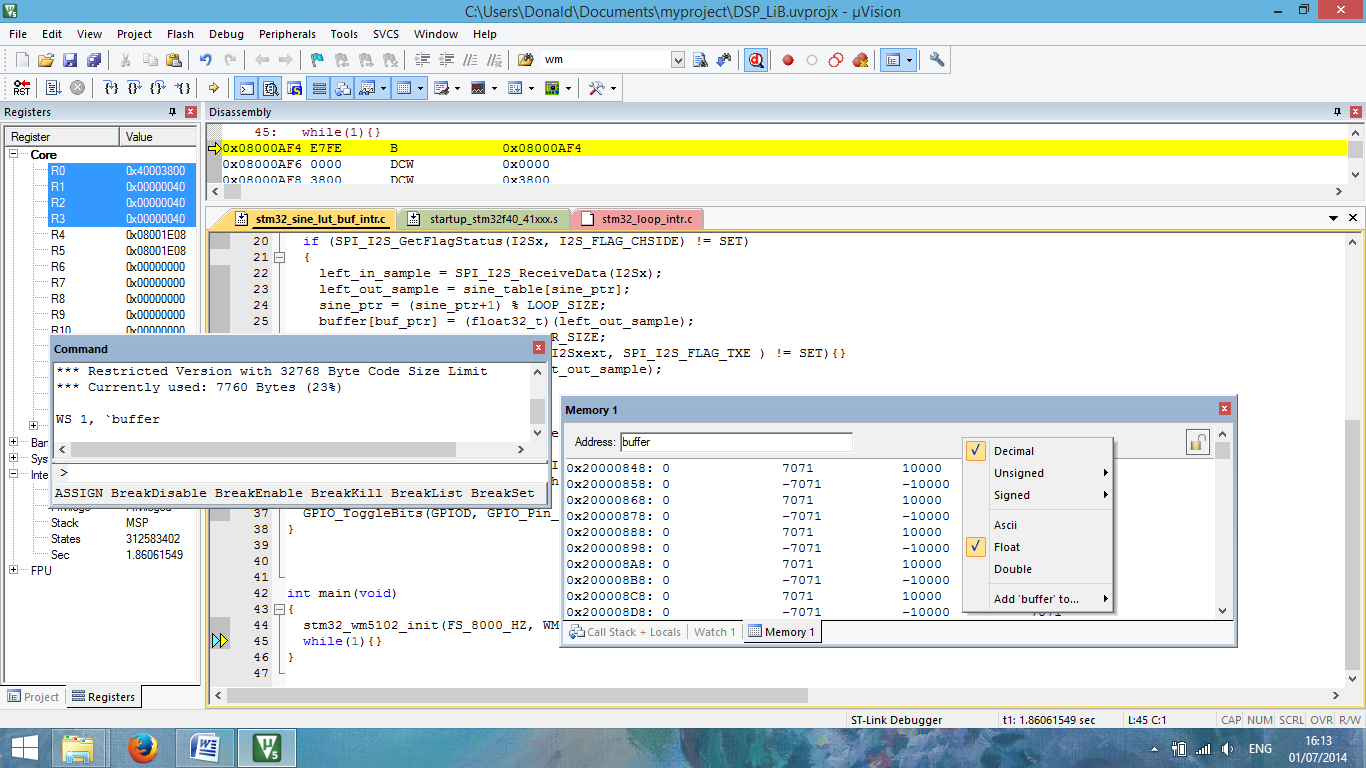
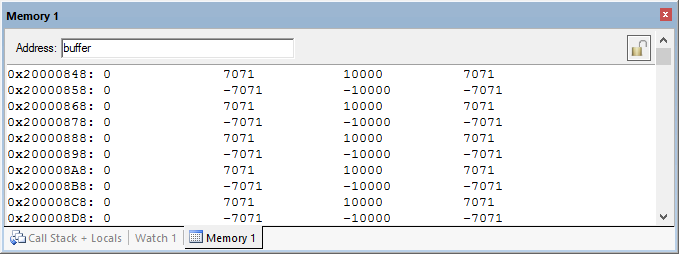


Figure 13: *Memory 1* window showing the contents of array buffer.

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 0x20000848, 0x200009D8

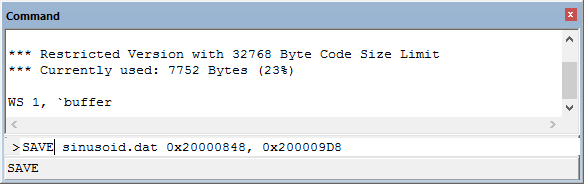


Figure 14: Saving data to file in *MDK-ARM*.

You can use MATLAB function logfft.m (provided with the DSP LiB) to obtain a graphical representation of the contents of the buffer.

There are some subtleties here, linked to the simplicity of function 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 WM5102 DAC to 32-bit floating point values in program 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 centre frequency of the single peak rather than its shape. If BUFFER\_SIZE is adjusted to be equal to an integer multiple of LOOPLENGTH then function 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 logfft().

# 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.

MATLAB resulting plot using logfft.m

