EE-3220 LABORATORY
Weeks 9 & 10

Real-time DSP Using the Cypress FM4 Board

Goal – Design, verification, and implementation of DSP filtering that runs in real time

Background – To perform real-time DSP, we'll use the C language, an Integrated Development Environment (IDE) called MDK-ARM, and hardware called the Cypress (Spansion) FM4 board (FM4-176L-S6E2CC-ETH Pioneer Kit). This can be checked out from TSC as the **Cypress FM4 Starter Kit**.

Part I: IDE Install and Setup

1. Begin at Dr. Prust's FM4 resources website. Follow all steps in "Laboratory1.pdf" through Section 6, "Building and Running Programs." This will take you through installing the development environment and running a program that passes microphone input to the headphone out jack. You may complete additional steps from this document if you wish, but it is not required.

https://faculty-web.msoe.edu/prust/armdsp/

The line in jack on the FM4 should only receive a maximum of 894 mV peak-peak; this is the consumer audio line level. Monitor any signal that is put into this jack by using an oscilloscope. Adjust the level of the signal before connecting it to the line-in jack. To be safe, keep the voltage around 500 mV peak-peak.

2. Test the sample program as outlined in Laboratory1.pdf. Connect the headphone out jack on the FM4 to the second channel of the scope to monitor the signals.

3. When the sine wave is being generated, use the scope to verify the waveform. Observe the signal in both the time domain and frequency domain (use the FFT feature).

Part II: Writing your own code

1. To try other programs provided in the sample project, right-click on "src" and select "Add Existing Files to Group 'src'…" You should then right-click the unneeded .c file and select the "Remove File" option; there can be only one main() function per project.

2. When you are ready to begin your own code, base it on one of the examples. Save an example file under a different name (e.g., notch.c) and then add it as described in the previous step.

3. Compile your code by pressing F7 or selecting Project → Build Target. In almost all cases, you should address all warnings; ask the professor for help if needed. Of course, you must fix all errors.

4. Connect the FM4 unit, start a Debug session, and Run the program as you did in the previous section. Test your program with a sinusoid in the range 0–24 kHz – keep the voltage around 500 mV peak-to-peak. The sampling rate is 48 kHz. Test it with music from your laptop.

Part III: Moving Average Filter

Analyze, implement, and validate a 5-point moving average FIR filter. That is, implement the filter described by the following difference equation:

$$y(n) = [x(n) + x(n-1) + x(n-2) + x(n-3) + x(n-4)] / 5$$

Analyze the predicted phase and magnitude response via Matlab (perhaps using fvtool or freqz).

Follow the same steps as before to build and debug the program.

Begin by measuring the filter magnitude and phase response in 50 Hz steps from 100 Hz to 1000 Hz. Use the function generator signal as both the line in input on the FM4 and channel 1 on the oscilloscope. Connect line out to channel 2. You might find setting the oscilloscope to simultaneously measure peak-peak voltage on both channels plus the delay from channel 1 to channel 2 useful. Setting the scope to reject HF (high frequency) noise should improve the triggering, giving you a steady-in-time view of the sinusoids.

Continue measuring just the magnitude response at several frequencies up to 24 kHz. Your goal is to collect enough data to confirm that the actual response contains all the key features predicted by theory.

1. Submit a description of the filter design, including phase and magnitude plots in hertz for the predicted filter response, and a pole/zero plot.
2. Submit the measured phase response of the filter to 1000 Hz. `unwrap(phaseList, 180)` may be useful if you converted time delays to phase delays in degrees. Does the phase delay appear linear? If so, how much time delay is shown? How much time delay is predicted by theory? Note that the parts of the DSP system outside of the algorithm are responsible for some delay.
3. Submit the magnitude measurements for the filter – plot the predicted response and the measured response on the same graph for comparison.
4. Submit source code for this filter.

Part IV: Notch Filter

Analyze, implement, and validate a notch filter that will remove a 1200 Hz tone. Design the notch filter to have a -3 dB width of approximately ±2.5 Hz. Use a sampling rate of 48 kHz. You can zoom in on a frequency response using either the GUI by restricting the range at computation time `freqz(b,a,1100:1300,fs)`

Use Matlab to predict the magnitude and phase response of the filter. In the IDE, implement the filter and test on the FM4 board (see the Appendix for implementation hints). You can create a new source file, but it may be easier to modify your existing, moving average file, commenting out your previous code or putting it in a separate ma() function that will be unused for now. You only need to process one channel. You may find it helpful to output the unfiltered signal on the left channel and the filtered signal on the right channel.

You can generate a signal in Matlab that will quickly test whether your notch filter seems to be working before moving on to detailed measurements:

```
fs = 48000;
f = 1200 + [-20 0 +20]; % frequencies to generate in series
T = 3; % seconds, duration of each frequency
t = (0:fs*T-1)/fs;
x = reshape(cos(t'*f*2*pi),1,[]); % outer product, reshape works in column order
sound(x,fs)
```

5. Submit a description of the filter design, including filter coefficients, phase and magnitude plots in hertz for the predicted filter response, and a pole/zero plot.
6. Submit a plot of the measured magnitude response for your filter from 20 Hz below to 20 Hz above your notch frequency.
7. Submit source code for this filter.
8. Demonstrate the notch behavior of this filter to the instructor, using a test signal similar to the above.

Appendix: Needed Details and Hints

You will find 3.5mm to banana cables, available in TSC, convenient for connecting the FM4's line in and headphone out jacks to the laboratory equipment.

You may find a BNC to banana cable, available in TSC, to provide a more reliable connection to the function generator than the spring-loaded probe leads connected to the laboratory equipment.

Note in main() that the sample programs usually take audio from the "mic_in" port. You will want to change this to "line_in" to use the line in port. This is suitable for receiving music from an audio player and from the function generator. Unpowered microphones operate at much lower voltages than line level devices.

In your moving average filter, it is recommended that you work with just integers; as you know from the sample programs, inputs/outputs from the DSP hardware are 16-bit, 2's complement signed integers. Note that you're adding 5, 16-bit signed integers together, so the result could take up to 19 bits. This is okay since the native int size on this platform is 32 bits. This is taken care of automatically if you write something like (x0+x1+x2+x3+x4)/5, where the x values have type int16_t. If you want to use a variable to hold the intermediate sum, it should be of type int32_t.

For your notch filter, your calculations will be done in single-precision floating point (float). This happens mostly automatically in C since arithmetic operations between ints like int16_t and floating point types like float are automatically converted to the floating point type. When you assign the final float result to your int16_t output, truncation occurs, which is okay. In the recursive filter, you should store your previous y values in float values for better precision even though they are converted to int16_t when you output them.

The FM4 uses standard 3-conductor 3.5mm (1/8 inch) audio connectors. From the base, the conductors are ground, right, and left. Some device-specific headsets (e.g., for cell phones) have a mic and either 1 or 2 audio output channels and might not behave exactly correctly on the headphone out jack of the FM4. For example, 2015 Apple Earpods (stereo with a mic, requiring 4 conductors) only seem to work properly when the pause button is depressed.

You will need to allocate memory to remember past samples of x and y values. In DSP systems, when memory is needed for the life of the program and does not vary in size, it is normal to allocate memory statically (for the life of the program), not dynamically. Thus, in a **function**, you might write `static float x[5];` to allocate your memory.

Although you could use global variables (declared outside of any function), it is bad style to give variables wider scope than needed. Note that making a variable declared outside of a function static disables linking to it from other modules; the variable becomes visible to the whole compilation module (typically one C file), but not others.

You may write your filter directly in the interrupt service routine I2S_HANDLER, or you may call a function you write to do the filtering. This ISR runs every time a pair of L/R samples is received and it is responsible for outputting a pair of L/R samples. You may want to use an `int16_t*` for the parameter type when you need to return a result (e.g., filter output). Recall that you can write to memory pointed to by a pointer by using `*ptr` on the left hand side of an assignment (=) statement.

Although you could hard-code your IIR filter coefficients directly into your implementation for this assignment, it will generally be easier to store them in an array, for example, `static const`

`float b[] = {1, 2.3, 4.6};`. This results in b[0], b[1], and b[2]; note how this matches our notation from class, but differs from Matlab array notation.

Matlab normally only shows you a few decimal places of precision. Tell Matlab `format long` to see greater precision, which is necessary given the high r value (poles quite close to the unit circle) necessary to meet your specification.

This is ANSI C, so variables must be declared at the ***start*** of blocks of code (functions, {} within for loops, etc.). Also, this isn't C99, so if you wanted to write a for loop (may be helpful, but not required for this assignment), you can't write `for(unsigned int idx = 2;` …, but have to declare the index variable before the loop.

For an extra challenge, once everything else is working, you may consider storing x and y using circular indexing. In this case, new samples always go in the next slot, and when you fall off the end, you go back to [0]. This eliminates the need to move the values on every sample (which can be very expensive for high order filters), but makes the indexing logic more complex since you need to keep track of where the last sample was deposited. The % operator is useful to achieve wraparound.