Experimentation with NXP's LPC4350 ARM Cortex M4 for filtering of I²S (audio) data

By Brewster LaMacchia, Momemtum Data Systems

Being new to the Keil tools, the LPC4350, and the Cortex M4 ecosystem, I chose a simple I²S audio processing project in order to get some experience before tackling the "real" project. However that initial project expanded in scope, so it became the focus of my evaluation.

Momentum Data Systems (MDS), for whom I work, has offered its QEDesign filter design tool for 20 years; it was the first graphically oriented tool for practical DSP filter design. Initially developed for Sun workstations (PCs were in the DOS era back then), it offers advanced features for the expert while retaining a simple interface for the novice.

The goal of this project was to first create an analog audio in-to-out example (using the UDA1380 codec on the EVM board), and then using QEDesign to create filters that the ARM CMSIS DSP library routines could use.

Since the LPC4350 supports native floating point operation it was decided to target that for testing since there are fewer implementation issues to consider (using floating emulation the filters would not run in real time). The task list was:

- Create a new KEIL project that has floating point enabled and runs the CPU at 204 MHz
- Program the LPC4350 I²S block
- Program the UDA1380
- Create an interrupt based circular buffer routine for input and output and a loopback test
- Add the filter routines
- Test and benchmark

Some ancillary tasks were added to make running the code easier:

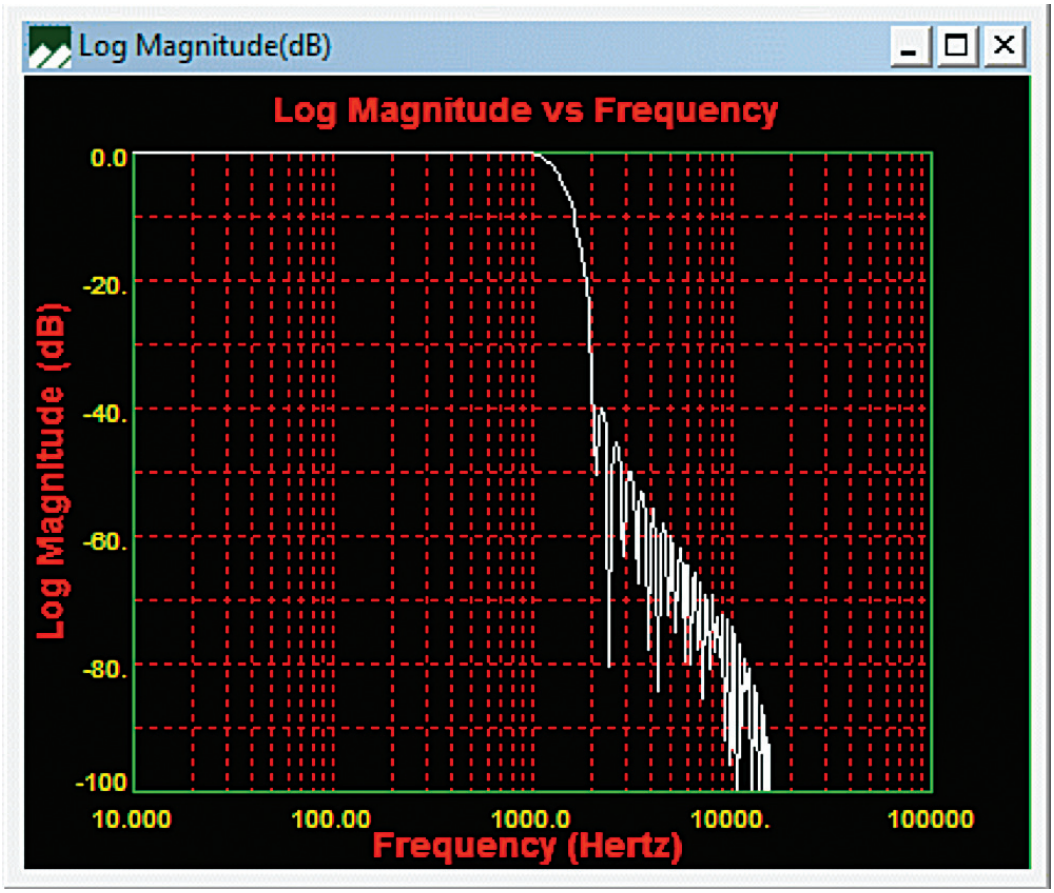
- Implement support for touch buttons and LEDs
- Add a menu to the eval board LCD
- Use the Keil Logic Analyzer to benchmark execution time

Download the sample code here

Full documentation of the project can be found by downloading the project from http://www.lpcware.com/content/contribproj/lamacchiab-lpc4300ex

General steps to filter implementation

The QEDesign tool can create C code output. The frequency response of a typical filter design (a simple 64-tap FIR low-pass filter (LPF) with a 1 kHz cutoff and a system sample rate of 32 kHz) looks like this:

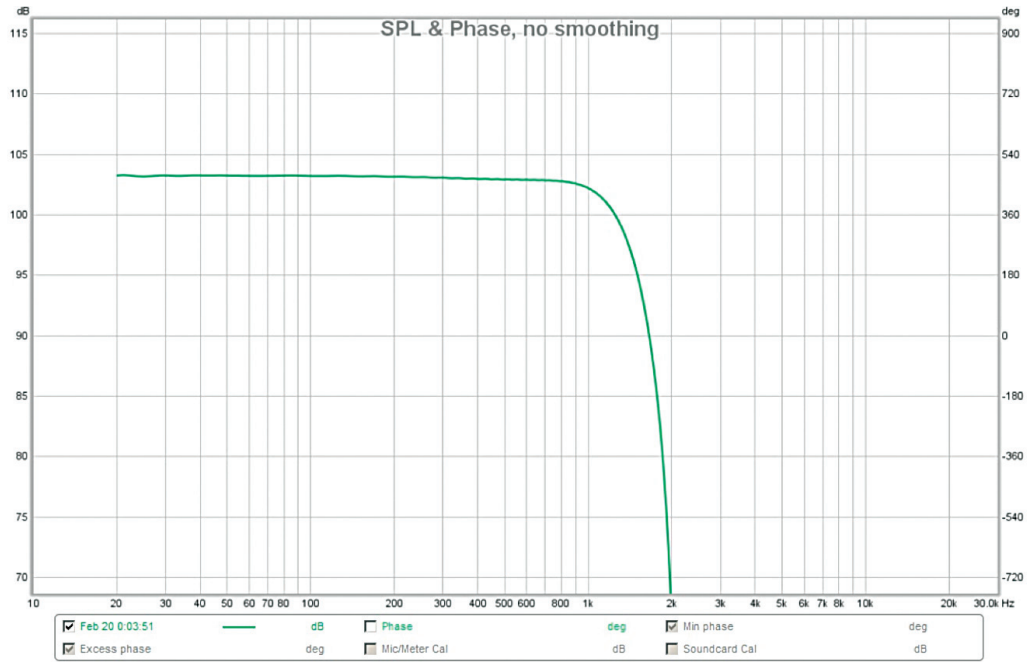


and results in coefficients that were copy/pasted into the source code:

```
const float lp_fir_coeff[64] = {
    2.481690607965e-004F,   /* filter tap #     0 */
    8.727381937206e-004F,   /* filter tap #     1 */
    1.631523948163e-003F,   /* filter tap #     2 */
...
```

The CMSIS library provides the `arm_fir_init_f32()` function to initialize their filter control structure for a floating point FIR filter, once this is done the `arm_fir_f32()` is used to process a block of samples.

Running the filter with a swept frequency input we can verify it is performing as expected:



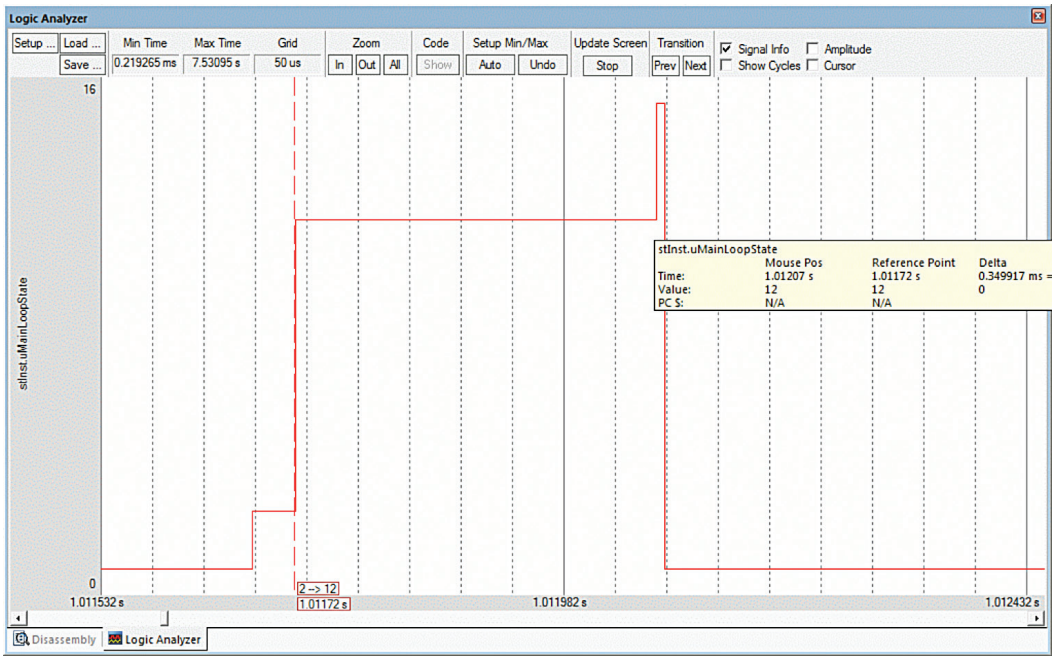
Performance results

Basic performance measurements were performed to provide a guide for future projects with the LPC4350. The techniques used were meant to give a rough feel (maybe +/- 10%) vs. exact numbers for comparison.

The loading from the ISR that services the I^2S port data FIFOs was checked by setting and clearing a GPIO bit on entry and exit (which misses the stack save/restore). This averaged about 4 µsec every 140 µSec, or about 3% overhead. DMA should lower that, but it's already so small that it wouldn't be the first thing to optimize further.

With the 32 kHz sample rate a buffer to process data (@128 samples/channel) is created every 4 msec.

For the filter routines Keil's Logic Analyzer was useful; it allows the value of a variable to be graphed over time. By setting the variable to different values at different places in the processing it's easy to measure the time taken. The next figure shows the timing for the FIR filter:



The small steps before and after the bigger time block are the time to convert the 16-bit integer input (stereo, 128 samples/channel) to floats and vice versa.

The times were as follows (per channel) for the filter:

- 64 tap FIR: 175 µsec
- 365 tap FIR: 1 msec
- 8th order IIR (4 biquads): 45 µsec

Remember that the I²S interrupt service routine is also running during this period so actual time on the filter code would be less.

Looking at the 64 tap FIR and the 175 usec/channel: with 128 samples it took 1.4 µsec/sample, with a 64-tap filter this is 20 nsec/tap. With the 204 MHz CPU clock this is 4 CPU clock cycles average per sample. The M4 FPU takes 3 cycles for a multiply-accumulate so this again is a reasonable result, with the other time going to loading/saving from memory, loop overhead, etc.

Summary

The M4 offers good DSP performance in a low cost part. The ability to use the M0 core for "everything" else opens up interesting possibilities for applications that might have otherwise needed an RTOS or other context switching mechanism on a single CPU to handle other tasks while allowing real-time processing to occur without data loss.

Date of release: April 2012
Published in the USA