Fast Fourier Transform on a TI C5502 DSP Core

Real Time Digital Signal Processing - University of Nebraska

Daniel Shchur
62096122

Contents

List of Figures

List of source codes

1 Introduction

The Fast Fourier Transform (FFT) is an optimized algorithm to perform a Fourier transform on an input signal. Performing the transform allows us to perform spectral analysis on signals and relay back an estimated frequency spectrum of a given signal, among other things.

In this project, an FFT algorithm is implemented on the TI C5502 DSP Core using DSP BIOS to delegate different functionality as threaded tasks as to allow for multitasking on the system. This project builds upon the previous projects that allow for user input and control of filtering and muting while also relaying system state to LEDs and the dot matrix display. Those components are left untouched and must still be able to function with the addition of the FFT.

2 Fast Fourier Transform

A Fourier transform is defined by an infinite integration of multiplications. As this is extremely computationally expensive to perform, a shortcut method was devised called the Fast Fourier Transform which allows for the Fourier Transform to be through the use of multiple Discrete Fourier Transforms in quick fashion at the expense of overall accuracy.

2.1 Implementation

The implementation was performed using MathWork's Simulink simulation and prototyping system. Using the CodeGen package of Simulink allowed for the FFT algorithm to be generated in optimized C for the DSP core at hand. This effectively gives a highly accurate and efficient implementation of the FFT without the possibility of implementation errors by the programmer. As such, the expectation is that the mean squared error between simulation in MATLAB and the actual implementation should be minimal.

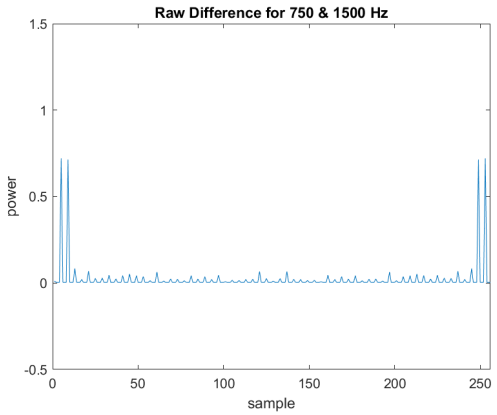
Within the actual task, a BIOS mailbox sends messages of 256 samples form the audio processing task to the FFT task for processing. On receive, the mailbox copies a 256 sample message into the global memory required by the FFT function. After it completes, global memory is copied from the FFT function in other system global memory which can later be used for other purposes. Listing 1 shows this implementation.

Two sets of test signals were chosen to test the implementation: 750 with 1500 Hz, and 1000 with 1200 Hz. These were chosen to test the system with arbitrary frequencies as well as frequencies derived from the sampling rate, since signals derived from the sampling rate have no overlap and therefore should produce cleanly identifiable peaks with minimal error, while the arbitrary signals may not be as clear due to their closeness and the low resolution of the FFT implemented.

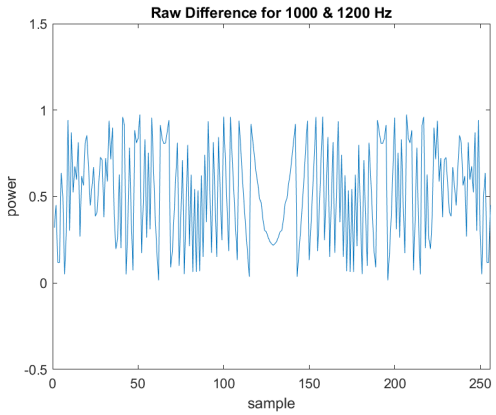
Figure 1 presents the raw difference of every sample between the MATLAB simulation and real time implementation. The difference between samples is always less than 1 and comes from the lack of floating point support on the hardware.

The mean squared error of the simulation versus implementation is 0.00839 and 0.34756 for the 750 with 1500 Hz and 1000 with 1200 Hz signals respectively.

As expected, the difference is very small between the simulation and implementation. The 750 with 1500 Hz signal produced a smaller error as was expected due to the signals being derived



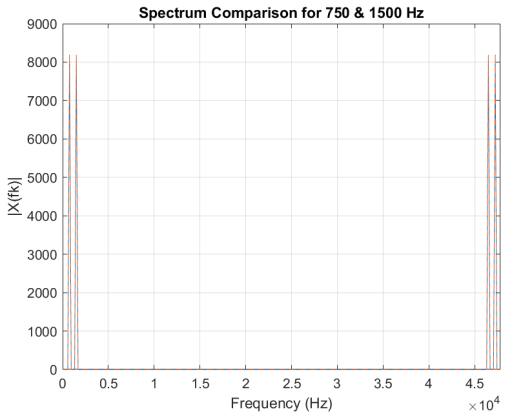
(a) 750 & 1500 Hz Raw Difference



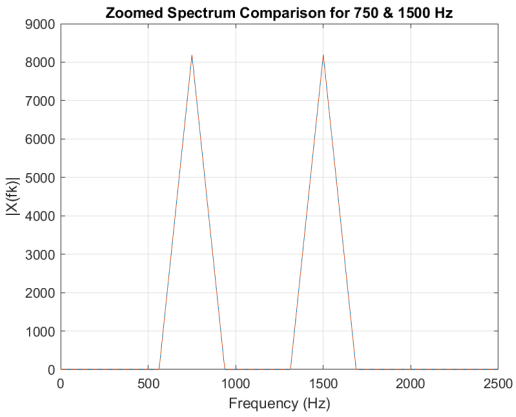
(b) 1000 & 1200 Hz Raw Difference

Figure 1: Raw difference of different signal groups

from the sampling rate. Flooring the MATLAB simulated signals results in the exact same output as the implemented FFT which shows that all errors come from loss of accuracy due to lack of rounding.



(a) Full signal



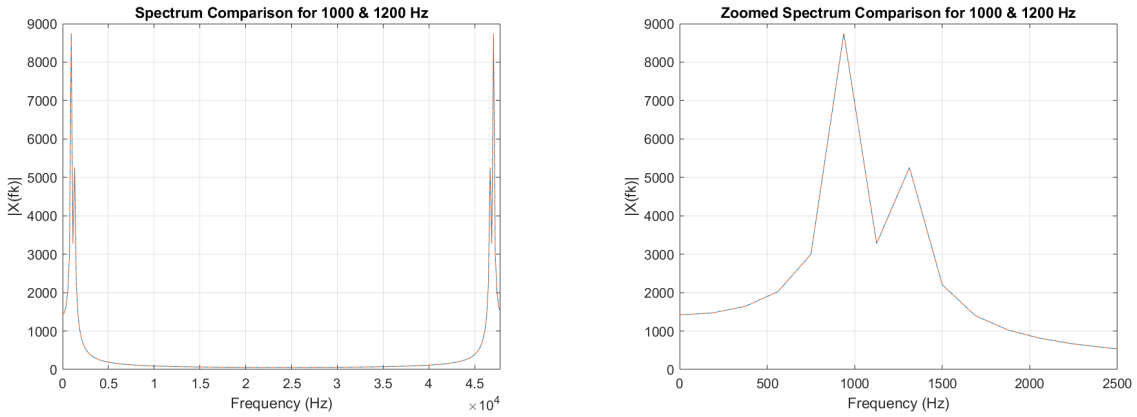
(b) Zoomed in

Figure 2: FFT comparison and presentation of 750 with 1500 Hz

Figures 2 and 3 both show the signal output of the simulated FFTs overlayed with the implementations. As can be seen, no discernible differences are visible. One thing to note is that the 750 with 1500 as both accurate in analysis showing the frequencies exactly as they are, as expected, while the 1000 with 1200 show up a little bit off from their actual frequencies showing up as 973 and 1312 Hz. This is primarily due to the low resolution of the FFT being at 256 bins, severely limiting the accuracy achievable of the FFT.

2.2 Efficiency

Using the built in clock feature in the implementation software, measurement of processing cycles was possible over a large range. On average, after disabling other tasks and interrupts to measure only the FFT being performed on 256 samples, it was found that the function



(a) Full signal (b) Zoomed in

Figure 3: FFT comparison and presentation of 1000 with 1200 Hz

would take 3,521,820 or 3,523,450 cycles to complete. Most runs would present the higher cycle count, but some would be the lower value. This could be due to overhead from the BIOS. At a clock rate of 300 MHz, the FFT takes a total of 1.174% of the total CPU cycle count to run. With cycles dedicated to running other tasks such as read and writing on the audio codec, filtering samples, and taking in user input, the effective cycle count is much higher due to context switching between tasks as the FFT task is prioritized lower than the filtering and user interface tasks. It is set as the lowest priority thus far to make sure the filter is still real time and to also make sure the user interface task is given time to read inputs as the FFT is computationally expensive and can end up taking over any cycles the user interface task might has needed.

3 Summary

Fast Fourier Transform implementation was explored on the TI C5502 DSP core for future use of signal analysis and presentation. The FFT was implemented with a high degree of accuracy when compared to the simulation due to the use of MathWork's CodeGen feature of Simmulink. The transform is able to be performed relatively quickly which gives plenty of room for the rest of the system to perform other tasks while keeping everything real time. This is especially true as BIOS threads are utilized to allow for multitasking.

4 Appendix

```
void TSK_FFT()
{
    while(1)
    {
        MBX_pend(&mbx_processedAudio, FFT_U.In1, SYS_FOREVER);
        FFT_step();
        memcpy(fftProcessed, FFT_Y.Out1, 256);
    }
}
```

Listing 1: FFT Task Implementation