ECE 4220

Real Time Embedded Systems

Final Project

Spectrum Analyzer

by:

Matt Mazzola

12222670

Abstract

The design of a spectrum analyzer on an embedded device is presented. The device achieves minimum error of 2% and a maximum of 42%. Input is given by a signal generator and output is displayed in a console vial serial cable. The goal of the project is to have the spectrum data be updated at a rate so the user notices no lag, i.e. faster than 0.3 seconds which is the average human reaction time.

Introduction

For this project I am designing a spectrum analyzer to be implemented on the Technologic Systems 7250 board with a 200MHz ARM9 CPU and MAX197 8 channel 12-bit A/D Converter. The spectrum analyzer will be used to analyze the audio spectrum and output a frequency intensity graph. This information could then be sent to other machines that rely on sound information but don't have the computational power to handle it internally. To implement the spectrum analyzer I plan to use three threads, one to collect samples, one to compute the ftt, and one to display the information. Due to time constraints I will assume that it is not necessary to develop a microphone, amp and filter since we won't be receive credit for electrical component design. The goal of this project is to get the most accurate results out of the limited power of the hardware and display the data in an easily read form.

Problem Statement

The problem is that we cannot naturally measure all types of radiation. We need a device such as a spectrum analyzer to capture the data and process to a more useful form such as taking time domain signals and outputting in the frequency domain. This information is then used to determine if the device generating the signal is operating correctly or if adjustments need to be made. One of the main applications for spectrum analyzers is to measure the spectral purity of multiplexed signals and modulation characteristics of AM and FM waves to monitor the operation of cell towers. This ensures that the multiplexed signals stay in the expected frequency range so when they are filtered out at the receiving end data loss is minimized. In my application I will focus on the audio spectrum defined by the human hearing range from 20 Hz to 22kHz. This type of analyzer is often much cheaper since I can use a microphone instead of an antenna to capture the input signals however it performs the same basic function.

Design Considerations

Sampling Data:

The human hearing range is approximately 100Hz to 22kHz. By the Nyquist theorem I need to sample at least twice the maximum frequency of my input signal so I will sample at 44.1kHz which is CD quality. To determine minimum samples required to catch even the lowest frequencies I use the following equation: 2 (T_max/T_min)=2((1/100)/(1/44100)) = 441 ≈ 512. From this I can estimate the run time of my get_samples() thread as: 512(1/44100)=0.0116 sec . Based on this design requirements I first verified that the MAX 197 chip could perform at this speed.

FFT Computation:
The FFT algorithm requires the samples to be a factor of 2^n which is why I rounded up my samples from 441 to 512. The complexity of this algorithm is much better than the discrete fourier transform and is derived as follows: M log$M$, since M=2^ , then 2^N log 2^N -> N2^N log2 .

Display:
Because there is no VGA controller or LCD module to use I will have to use the console to display my frequency intensity plot.

Proposed Approach

Hardware:
Since I am focusing on processing audio, I attempted to use a microphone to capture an incoming signal which would then go through a low-pass filter and be amplified up to the maximum range of my analogue to digital converter; however, I could not generate a clean signal from the microphone and chose replace the microphone and filter/amp setup with a signal generator. Although this means my end product cannot be used as originally intended it will allow me to record more accurate results since I can guarantee what the incoming signal will be and compare this with the detected frequency output. The new block diagram is shown below:

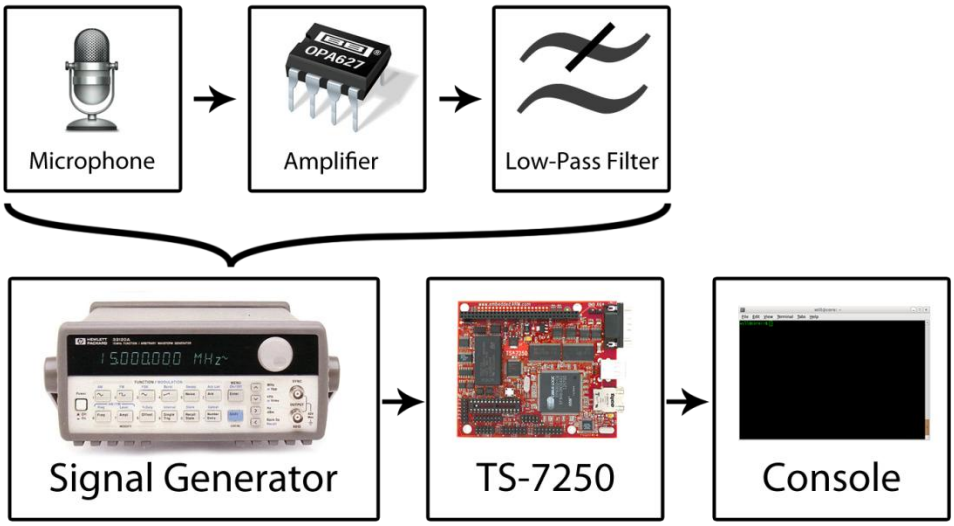


Figure 1: Block Diagram of Hardware

Software:
For software, I have three pthreads (get_samples(), compute_fft(), and display_data()), each with a real-time task created in user space, synchronized together by semaphores to form a consumer-producer model on a ring of 10 buffers. All threads have the same priority and are run with a time-slice round robin scheduling algorithm. The number of samples to collect is given as a command line parameter and this program will use malloc() to dynamically allocate the space for each buffer at runtime. The block diagram is shown in figure 2.

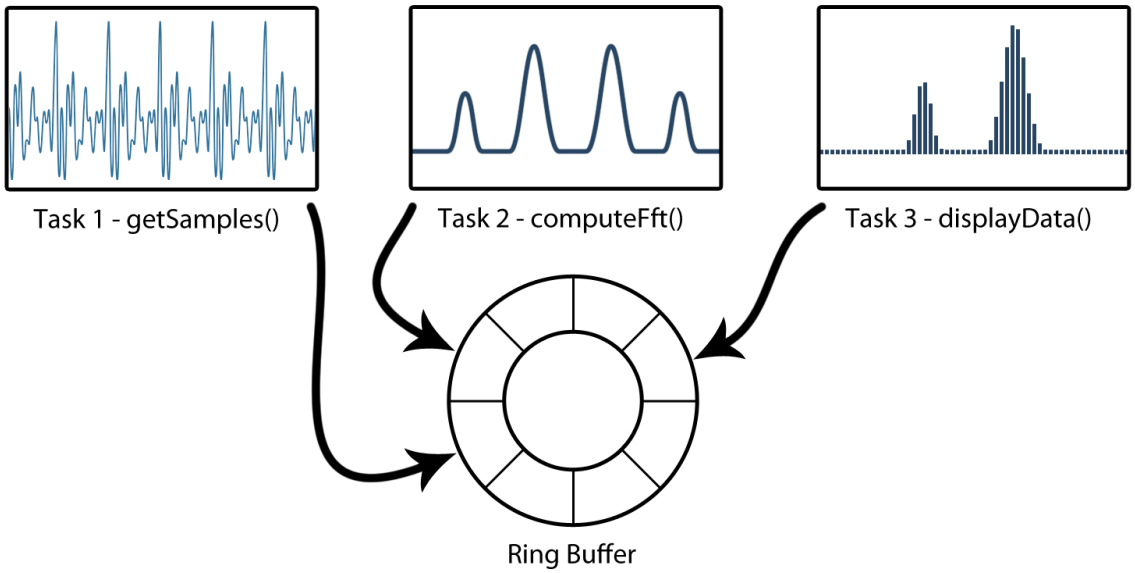


Figure 2: Block/UML Diagram of Software

Real Time Constraint:
Because I am performing data acquisition I need to use a periodic real time task to ensure that the converter on the MAX 197 chip is initiated at the same interval. If this was not a real time task there could be conflicts with other Linux OS processes that are of no concern to my program like network interrupts other system monitoring processes. If the samples are not taken at the same interval there would be inherent error in the data I feed to the FFT program which would decrease the precision of my software.

Consumer –Producer Model and Round Robin Scheduling:
Instead of having the three tasks simply execute one after another on a single buffer, I have improved the efficiency by implementing the consumer producer model. I chose to do this because the MAX197 chip is guaranteed to take longer than 12uS for the conversion to occur and this is a significant amount of time which the CPU could be used to run the FFT or Display tasks. I also implement the Round Robin scheduling algorithm since RTAI defaults to FIFO which would always cause the display program to wait until the FFT program was completely finished. With Round Robin the display can come in after the FFT's over runs the time slice and this increases the responsiveness of the system.

Semaphores:

As I mentioned previously the threads are synchronized by the use of three counting semaphores which denote the number of available resources each thread could potentially run on. The get_samples() task signals the compute_fft() task indicating it has filled a buffer with signal data and the fft may now be calculated on it. In turn, the compute_fft() task signals the display_data() function indicating that the fft information has been calculated and that information can now be displayed in the console. Lastly, the display_data() function will signal the get_samples() semaphore to indicate that this buffer has been display and reset so it is ready to be refilled with new samples. A visual diagram explains the process below:

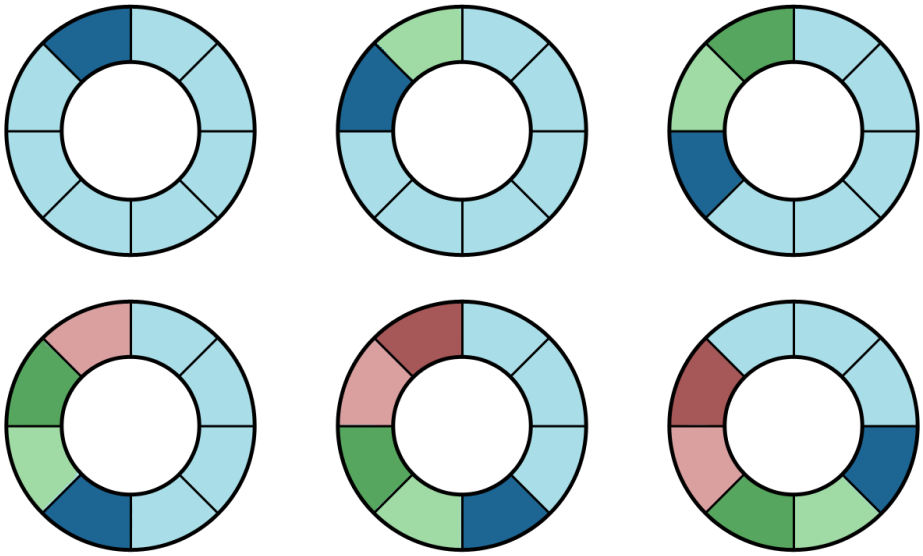


Figure 3: Visual Diagram of Semaphore signaling

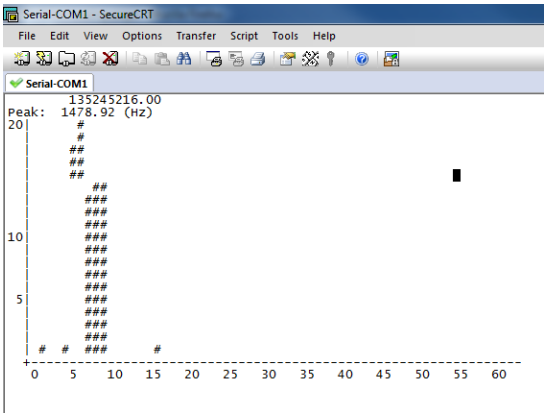
Blue: generate_samples(), Green: compute_fft(), Red: display_data()
Light colors note availability of buffers, Dark Shades of same color mean the buffer is being processed

Results

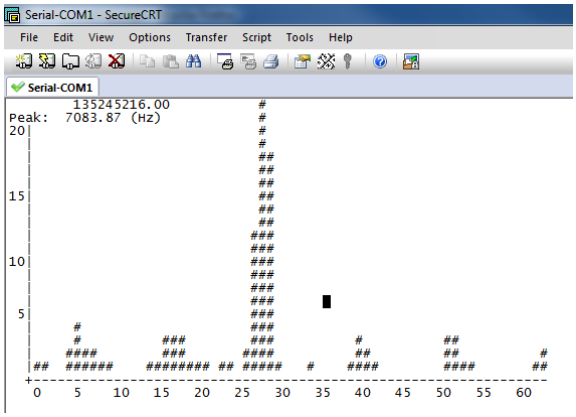
Samples	Avg. Delay (ms)
64	75.87
128	158.23
256	N/A
512	N/A
1024	N/A
2048	N/A

Input Frequency (Hz)	Measured Frequency (Hz)
1000	970-1005
2000	1970-2010
4000	3900-4050
6000	5900-6070
8000	5900-8050
10000	6700-10100
12000	7000-12100
14000	8000-14200

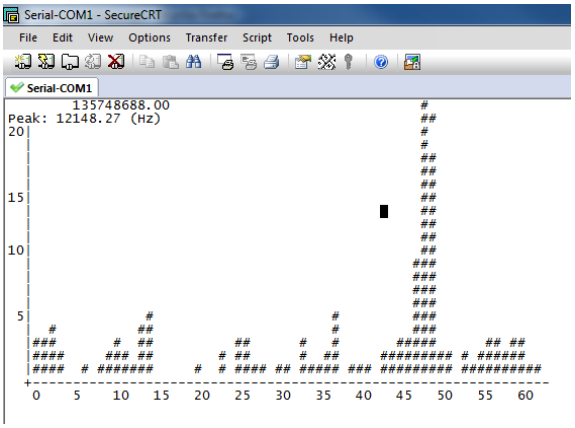
Pictures of Output:



Low Frequency (~ 3kHz)



Mid-Range Frequency (~ 8kHz)



High Frequency (~ 12kHz)

Conclusions

In conclusion I believe my project was not a failure because I could successfully take a time domain signal and display the frequency intensity graph in the console. The results show that the algorithm does have some glitches which cause it to show at most 42% error during the high frequencies however the high of the detected frequency ranges was always within 2%. It must also be noted that in a real scenario using a microphone which will carry much more noise than the signal generator therefore the expected error would likely increase to approximately 10% at minimum. An important note is that the error calculated was only for the difference between the detected peaks. For instance, if I input a sign wave of 1kHz and the software outputs the largest peak at 1100 Hz I would calculate 10% error however this can be misleading because there may be other minor peaks and other high noise shown on the graph that are not taken into account in the error calculations. There are also many improvements that could be made to increase the accuracy of the software. The first and most problematic error was that my fft task would hang when trying to calculate on inputs over 128 which went against one of my design constraints of being able to detect 100Hz frequencies and I was unable to determine the cause. The next most important improvement would be to find a way to calibrate the FFT output for optimal

display. The way I calibrated the system was via trial and error through changing the conversion factor I used to save the data sampled into the array. This is commented in the code with much detail but I will explain it here also. The standard way to interpret the binary data would be in volts with the general equation: (range / 2^N). In my case I used a bipolar 5V range therefore my conversion factor would be (5.0/2048.0); however, I did not use this in the code because the output after FFT calculation was too low. After many trials I found that (60.0 * 5.0/2048.0) was a fairly good factor to show a very prominent peak but yet also keep most of the noise down. This conversion value and the graph height should certainly be modified with for different applications. The next improvement would be to increase the consistency at which the display is refreshed. As you can see in the results the average was 75.8 ms for a size of 64 samples however the actual data was about (70, 70, 70, 100, 70, 70,…) and this one glitch where it takes 30 ms longer seems to be contributing to the error in the detected frequency range.

Future Additions:

Given more time it would be nice to upgrade the graph to a full VGA display which could be possible by sending the computed FFT array to another board with a VGA controller through a socket. Another appropriate addition would be to fully implement the microphone, amplifier, and filter so the system could be used without the generator and would be more interactive and useful. It might also be beneficial to experiment with applying a smoothing/regression algorithm to the FFT data to remove the semi-erratic behavior of the graph to increase usability. Lastly, it might be nice to modify the code so the user can turn debugging options on and off with command line parameters instead of having to modify preprocessor #define statements and recompiling.