Goertzel Algorithm for an Embedded system using Code Composor Studio

George Burns
dept. Electrical and Electronic Engineering
University of Bristol
Bristol, United Kingdom
george.burns.2001@bristol.ac.uk

***Abstract*—During this report I will detail my design of a Geortzel Algorithm implementation in C specifically using Code Composor studio emulating a *C6000-TMS320C6678* DSP (Digital signal processor) [1]. I will also detail the use of the Texas instruments real time operating system (*TI-RTOS*) [2]. The results from this where both tasks working to specification and an audio file being generated that matches the given `.wav` file.**

I. Introduction

In this assignment I will be using Code Composer Studio to develop a DSP (Digital signal processor) I will have 2 specific tasks both using the Goertzel algorithm to develop a algorithm that can detect certain frequencies in a signal. This was all done in a TMS320C6678 [1] and will be implemented using Code Composer Studio with TIROTS [2] a real time operating system that is capable of running both sequential and parallel tasks.

For task one I had to develop a Goertzel algorithm that detects one frequency in a chosen signal. The program will have two parallel process running at the same time one to generates the DTMF (Dual tone multi frequency) signal and the other to detect if a specific frequency is present. The task to generate a signal will be using a sin wave in conjunction with the `TICK_PERIOD` and `tick` both variables that are used by TIROTS to determine the system run time. Thus generating the signal shown below with Equation 1. Thus by sampling that frequnecy at a given time we can develop a `sample` or $x(n)$

$$x(t) = 32768sin(2\pi t f_1) + 32768sin(2\pi t f_2) \tag{1}$$

Where t is the system time gained from `TICK_PERIOD` and `tick`, f_1 is `freq1` and f_2 is `freq2`.

For task two the task is to develop the algorithm to detect keys in a pre-generated DTMF signal. This pregenerated signal is read from a `.dat` file and then is read into a variable `buffer` that has been pre-allocated memory. This signal is split into 8 tones and thus programme has `Task_sleep(210)` functions to sleep for the appropriate time. During each tone the algorithm will sweep for frequencies. After the set time it will print the two highest ones. From this information it will deterimine the key that is present. Then it will recreate this signal and write it to a `.wav`.

II. Goertzel Algorithm

As mentioned in Section I the Goertzel Algorithm will be used to detect specific frequencies, it will do this by deploying a bank of filters as shown in Figure 1, this no doubt brings similarities to operations like fourier transform [4] and thus the digital fast fourier transform [4]. However the Goertzel Algorithm abuses the term $e^{-j2\pi k/N}$ in Equation 2

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi k/N} \tag{2}$$

Where $X(k)$ is value of the DFFT at that frequency, $j = \sqrt{-1}$, k is a constant, $x(n)$ is the sampled input and N is the total samples.

More specifically it abuses the phase factor of the DFT to reduce the complexity of the calculation and thus is a lot more efficient then the DFT and FFT. This is great for fast applications like the DTMF I have been tasked with.

To implement the Goertzel Algorithm we need Equation 3 and Equation 4.

$$Q_n = x(n) + 2cos(\frac{2\pi k}{N})Q_{n-1} - Q_{n-2} \tag{3}$$

$$|y_k(N)| = Q^2(N) + Q^2(N-1) - 2cos(\frac{2\pi k}{N})Q(N)Q(N-1) \tag{4}$$

Where the Q factors represent the value relevant to their place on the block diagram. The $cos(\frac{2\pi k}{N})$ factor is the effect of the block with that coefficient. Using this we can create the structure shown in Figure 1.

To calculate the constant k we an use equation 5

$$k = N\frac{f_{tone}}{f_s} \tag{5}$$

Where f_{tone} is the frequency of the tone and f_s is the sampling frequency.

$$c = 2cos(\frac{2\pi k}{N}) \tag{6}$$

Where c is the coefficient value.

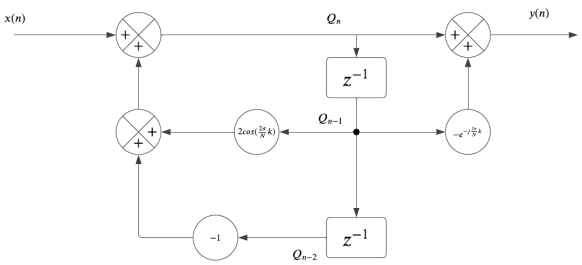

Fig. 1: Goertzel Algorithm system diagram

Frequency	k	Coefficient Decimal	Coefficient Float
697	18	1.703275	0x6D02
697	18	1.703275	0x6D02
770	20	1.635585	0x68B1
852	22	1.562297	0x63FC
941	24	1.482867	0x5EE7
1209	31	1.163138	0x4A70
13367	34	1.008835	0x4090
1477	38	0.790074	0x6521
1633	42	0.559454	0x479C

TABLE I: Coefficient values

Thus can calculate the coefficient Equation 6. With all this information we are able to calculate all the necessary coefficients Table I.

III. Setup of TIRTOS

The advantage of using TIRTOS is that it can use parallel computing and when it comes to DSP's the best option for our application is using parallel computing to get the most performance out of the DSP. Our processor is a *TMS320C6678* and uses eight cores [1] thus we have lots of cores to run parallel computing. According to Amdahl's law [5] with 95% parallel computing we get a speed up a factor of 6.

A. One frequency implementation

For the detecting of one frequency it was important to have both the detection of the frequency and the generation of the signal to run parallel so that both can run simultaneously. This way that the run time of the detection does not interfere with the generation.

The `clk_SWI_Generate_DTMF`/`clk_SWI_GTZ_0697Hz` functions both call the clock variable and thus run when the clock has a rising edge. Both `task_1` and `task_2` run synchronously. No other hardware of software interrupts where present in this task.

B. Multi frequency implementation

Unlike the single frequency detector the multi stage only has one task as the second task (Used for generating the `.wav` file) which needs to run from inside task 1. The data is read from a `.dat` and then once the signal is detected it will run `task2` to generate the `.wav` file.

IV. Implementation of the one frequency detector

To begin the implementation of the one frequency detector I used the Goertzel code supplied in [3] and made some adjustments to my liking. The first adjustment I made was the removal of all the input characteristics that are present in this code so that it would run with my program. The second change I made was to edit the coefficient of the frequency being detected with Table I, I changed the coefficient c so that it would detect frequencies of $697Hz$.

The next change I made was removing the shifting of the Goertzel Value. I did this as after inspection it was not very sensitive with these shifting values in place.

The next change I made was in the calculation of `gtz_out()` where I squared the input term to remove negative values. This also means that it will increase the value overall however this is a good trade off. I could also gain the modules of the value by square rooting the output however that would be heavy on computation and increase the time it took for a calculation to be done. I also divided by `N` to average the value.

Once this was completed I made a simple case statement for all the possible keypad entries, and then set `freq1` and `freq2` to generate the signal Equation 1. Where it is sampled at each `tick`.

Then when the user starts the program they are promoted to enter a DTMF key and then the program will first generate $x(n)$ in accordance with that keys's frequencies then it determines if a frequency of $697Hz$ is present.

Below is the output for the single frequency program Figure 2.

```
[TMS320C66x_0]
I am in main :
Please enter a DTMF key
1
I am in Task 1 for the first time, please wait:
The GTZ value for 697Hz is 270
I am leaving Task 1, please wait for a minute or so to get the next GTZ:
```

(a) Output for a input of 1

```
[TMS320C66x_0]
I am in main :
Please enter a DTMF key
5
I am in Task 1 for the first time, please wait:
The GTZ value for 697Hz is 0
I am leaving Task 1, please wait for a minute or so to get the next GTZ:
```

(b) Output for a input of 5

Fig. 2: Output from single frequency task

V. Implementation of the multi-freq sweep

My understanding of this section is that it is just Section IV but for all possible frequencies. So I began by setting the

```
[TMS320C66x_0] System Start
Calculating keys
Freq1 = 697, Freq2 = 1209, key = 1. With a time to calculate of 401
Freq1 = 770, Freq2 = 1633, key = B. With a time to calculate of 394
Freq1 = 697, Freq2 = 1336, key = 2. With a time to calculate of 399
Freq1 = 697, Freq2 = 1336, key = 2. With a time to calculate of 401
Freq1 = 852, Freq2 = 1209, key = 7. With a time to calculate of 405
Freq1 = 941, Freq2 = 1209, key = *. With a time to calculate of 405
Freq1 = 852, Freq2 = 1633, key = C. With a time to calculate of 404
Freq1 = 852, Freq2 = 1209, key = 7. With a time to calculate of 410
Final code of 1 B 2 2 7 * C 7
Detection finished
Generating audio
for 1
for B
for 2
for 2
for 7
for *
for C
for 7
Finished
```

Fig. 3: Output for multi frequency sweep

first sections implementation of the Goertzel algorithm into a for loop. This was accomplished by creating an array with 8 entries. Then iterating through the feedback section for each coefficient. Then `N` would increase, when `N == 206` I then iterated through the feedforward part a further 8 times and calculated each specific value.

Once I tested this with the original `.dat` file which came out as 12345678 and I assumed it was a calibration file, I also checked by doing an FFT in MATLAB to confirm. However on certain iterations it was not displaying the correct values but after tweaking the scaling of values helped fix this issue.

To ensure that only the correct frequencies where stored I made a `if` statement to compare `gtz_out[i]` against a threshold value. So in future if my code was put against nosier `.dat` files it would still work.

I then began constructing the part of the code that maps the keys with reference to the `gtz_out` array. I first started by searching for a `gtz_out` value that was greater then the threshold value indicated the paragraph before. Once this was completed I used a series of `if` loops to determine if the Goertzel value was second or first highest if it was then it was set to either `f1` or `f2`. Then after I sorted those values so the smallest frequency was stored in `f2` and the highest in `f1`.

Once this was one I made a case statement to determine the key dependent on what frequencies where present in the tone then printed the tone value, frequencies and time taken to complete to the terminal. Once this was done for all tones I finally printed the full code.

Below is a screen shot of my code outputting my given data Figure 3.

VI. Implementation to writing a .wav file

For the creation of the `.wav` file it is necessary to fill the variable `buffer` with the signal Equation 1 sampled at at a frequency $10kHz$ I changed this value from $1kHz$ to improve the output quality of the signal. Then for each tone fill the index in reference to that tone with the corresponding frequencies.

To get the corresponding frequencies from the key I made another case statement for each key, and then set the frequencies accordingly.

Once this was done my code generated a audio file that matched the `.wav` file provided.

VII. Possible improvements

I also believe that using TIRTOS (Texus instrements Real time operating system) [2] there is the possibility to have both the detect task and the generate the task to run parallel and thus constantly print the value of the Goertzel algorithm while also taking new values.

I believe I could drastically reduce the time it takes to find a frequency in the second task. Currently my code takes roughly 400 time units to find the certain frequency however if I added a while loop that checks to see if two values are greater then the threshold in the `clk_SWI_GTZ_All_Freq` function this time would be reduced dramatically, this of course would still be limited by the time it needs to read a tone however it would be nice to save memory and power.

References

[1] T. Instruments, *tms320c6678*.

[2] ——, *SYS/BIOS (TI-RTOS Kernel) User's Guide (Rev. V)*.

[3] N. Dahnoun, *Digital Signal Processing Implementation Using the TMS320C6000 DSP Platform*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[4] S. Tancock, "Digital processing systems: Choosing a processor."

[5] N. Dahnoun, "Chapter 17 - goertzel algorithm."