Implementing the Goertzel Algorithm in the C Programming Language for Processing DTMF Signals

Otis Lee   
*Electrical and Electronic Engineering*  
*University of Bristol*st20714@bristol.ac.uk

Mohamed Eldebawy   
*Electrical and Electronic Engineering*  
*University of Bristol*@bristol.ac.uk

Samuel Cheung   
*Electrical and Electronic Engineering*  
*University of Bristol*@bristol.ac.uk

Abstract—This project implements the Goertzel Algorithm for interpreting DTMF signals using Code Composer Studio (CCS) in the C programming language. The algorithm is implemented both using SYS/BIOS, Texas Instrument’s own developed operating system, and without using SYS/BIOS. The implementation with SYS/BIOS is then further optimized using several techniques for the TMS320 C6000 compiler, including C intrinsics, loop unrolling and compiler switches.

Keywords— Goertzel Algorithm, Intrinsics, SYS/BIOS, compiler switches,

NOMENCALTURE

|  |  |  |
| --- | --- | --- |
| f­tone | Filtering frequency | Hz |
| fsampling | Sampling frequency | Hz |
| N | Total number of samples/iterations (=205) | N.A |
| n | Sample Number | N.A |

# Introduction

In a Dual Tone Multi-Frequency (DTMF) signalling system, the digits 0-9 and characters A, B, C, D, \*, # are each represented by a unique signal. This signal is formed by the summation of two different frequency tones from a selection of 8 available frequencies as seen in Table I.1.

Therefore, a system which receives a DTMF signal must determine the signal’s two frequencies components to correctly interpret which character has been sent.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Freq [Hz]** | **1209** | **1336** | **1477** | **1633** |
| **697** | **1** | **2** | **3** | **A** |
| **770** | **4** | **5** | **6** | **B** |
| **852** | **7** | **8** | **9** | **C** |
| **941** | **\*** | **0** | **#** | **D** |

Table I.1 Combination of eight frequencies corresponding to a character

Whilst the Fast Fourier Transform (FFT) algorithm is the traditional technique for extracting the frequency components of a signal, it is inefficient when applied to a DTMF system since it unnecessarily checks for the presence of all frequencies. Since only 16 different DTMF signals can be produced, made up of eight specific, predetermined frequencies to be checked for, a far more computationally efficient algorithm for this application can be derived from the FFT. This is called the Goertzel Algorithm (Equations (1) and (2)).

This report presents and examines the successful application of the Goertzel Algorithm in C, designed for implementation in an embedded system receiving samples from a DTMF signal. The algorithm has been realised using fixed-point arithmetic which yields advantages (with respect to size, power, speed and memory usage) over a floating-point implementation with less complex code. For the purposes of testing the output of the algorithm, the DTMF samples have been generated by the code.

# Variable Types In C

## Fixed Point Arithmetic

Since the algorithm is designed to work with fixed-point arithmetic, and therefore fixed-point data types, only integer values of the sample can be stored whilst the value of the sample after the decimal point is discarded. Therefore, the samples of the input signal are scaled up by a magnitude of 32,768; the maximum range of possible values that a sample can take is now –32,768 to +32,767. This is equivalent to the full possible range of values that 16 bits can represent, therefore providing a sufficiently high-resolution representation of the input samples for fixed-point arithmetic. To achieve high memory efficiency the values are treated in Q15 format for fractional arithmetic to prevent multiplication overflow and the need for larger data types. Since Q15 limits values from –1 to 0.99999, the algorithm “coefficients” (discussed later - see Equation (3) andTable VI.1) are stored as half of their correct value to ensure they are within range. This is compensated for later in the algorithm’s implementation.

## Floating Point Arithmetic

Implementing the Goertzel Algorithm with floating-point arithmetic is considerably easier, since it does not require shifting numbers. Variables of type “float” facilitate non-integer decimal numbers. The simplicity in terms of coding, however, has its drawbacks. Operations on floating-point numbers are much more computationally expensive. For comparison, the algorithm is also realised using floating point arithmetic and the function “Timestamp\_get32()” [4], used to time both the fixed- and floating-point approaches, proves that the latter takes longer to compute. Therefore, when efficiency is of utmost importance, fixed-point numbers are desired. However, if the main concern is the coding time and complexity then the floating-point approach is superior.

# Generating Signals and Output of Program

The program is split into 2 working sections: “one\_freq” and “all\_freq”. The former section deals with one frequency alone, i.e. the user inputs their desired character, the first DTMF frequency component of that character is generated, and the program detects if a single, targeted frequency is present using the calculated Goertzel value. The latter section deals with 2 frequencies: the user will input a character as before, but now both DTMF frequency components of the character are produced and the Goertzel values for each of the 8 possible frequencies are calculated and stored in an array. The program attempts to find the highest two Goertzel values from the array to determine the respective character and prints all information onto the console.

If an incorrect character is entered, the frequency and magnitude of sample signals will be set to ‘0’ (more about sample signals in section IV). Therefore, the returned information will be redundant. After the processing time, the user can continue to input their desired character to be tested. One alternative method is to skip the evaluation of incorrect characters, however , a few problems were encountered during implementation, specifically, a boolean could not be returned to tell the program to stop executing the analysis function. This is because all functions are void, that means it produces no return. It would shorten the processing time and effort hugely if the algorithm wouldn’t run on an incorrect character. The printed figures include Goertzel values for all frequencies, the number of cycles elapsed using both fixed-point and floating-point method (more on both methods in section VI), and the time elapsed. Screenshots in Figure III.1andFigure III.2 illustrate the results of typing the character ‘1’ into the program.

# Generating Samples

Once the frequencies are chosen the program calls the “clk\_SWI\_Generate\_DTMF” function, see Figure IV.1, to sample over the sine wave frequencies, sum them and create a new discrete signal, where “tick” is the sample number, n, that is taken from the signal. With each iteration of the algorithm, “tick” is incremented to retrieve the next sample from the input signal. The function is called periodically utilizing the clock. The clock is covered in depth in the Section VII.

Since the algorithm is designed to work with fixed-point arithmetic, and therefore fixed-point data types, only integer values of the sample can be stored whilst the value of the sample after the decimal point is discarded. Therefore, the samples of the input signal are scaled up by a magnitude of 32,768; the maximum range of possible values that a sample can take is now –32,768 to +32,767. This is equivalent to the full possible range of values that 16 bits can represent, therefore providing a sufficiently high-resolution representation of the input samples for fixed-point arithmetic.

# Array of Coefficients

Performing the algorithm requires calculation of the 8 frequency-unique coefficients used to detect each of the 8 frequencies (required in Equation (1) and (2)). Their values are calculated using Equations (3), (4) (from [1]) and can be found in Table VI.1 (provided in [1]).

# Implementation of the Goertzel Algorithm

The algorithm can be divided into two sections: feedback and forward loop, as can be seen in the schematic, Figure VI.1.

## Feedback Loop with Q15 Format

The code for this part is shown in Figure VI.2.Static variables are initialized to ensure their values are retained and updated on each iteration: *N*; *Goertzel\_Value*; *delay 1, 2, 3*. Intermediate variables used for calculation are initialized normally: *prod 1, 2, 3* and the *input* sample. The counter, *N*, only varies between 0 and 206, therefore it has been declared

|  |  |  |  |
| --- | --- | --- | --- |
| frequency | k | Coefficient (decimal) | Coefficient (Q15) |
| 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 |
| 1336 | 34 | 1.008835 | 0x4090 |
| 1477 | 38 | 0.790074 | 0x3290 |
| 1633 | 42 | 0.559454 | 0x23CE |

Table VI.1 Coefficients for each frequency in floating point and Q15, fixed point format. Q15 values are halved.

as the 8-bit memory type “unsigned char” to increase memory efficiency.

The sample is cast to a short: *input*.

The 2nd term in Equation (1)(coefficient\*delayed value) is calculated and stored in *prod1*. The coefficient is multiplied twice and added to account for it being half of its correct value. The result is shifted by 15 to retain the Q15 format. This is cast to short and the value of *delay* (Qn in Equation (1)) is calculated.

*delay\_2* (Qn-2)and *delay\_1* (Qn-1)take the values of delay\_1 (Qn-1) and delay (Qn) respectively, ready for the next iteration. The counter, *N*, is incremented and the process repeats.

## Feedforward Loop, Q15 Format

The final stage of the algorithm, used to calculate the magnitude of the output Goertzel Value, is entered once N=206 (shown in Figure VI.3). The way the code was written means that the effective value of N is 205, i.e. the algorithm has technically *completed* 205 iterations. The robustness of the program depends on the sampling rate and the length of the block (*N*). With a fixed sampling rate, the correct N has to be chosen for the algorithm to yield the most accurate results. The minimum average absolute error is when N=205. The terms of Equation (2)and the output Goertzel Values are calculated in the “feedforward” loop. The output Goertzel Value is then stored in the correct position in the *gtz\_out* array (dependent on the corresponding frequency being checked for) and, after resetting *N,* the output *Goertzel\_Value* and the iterated *delay\_1* and *delay\_2* values back to zero, the algorithm is then repeated to continue testing for the presence of the specified frequency and updating the *gtz\_out* array with the output Goertzel Value.

The algorithm is continually performed whilst the “*task1\_dtmfDetect*” task is sleeping. When the task finishes sleeping, the values in the *gtz\_out* array are printed to the console before going back to sleeping. If the frequency being tested for is present in the input signal then the corresponding printed Goertzel Value is large, otherwise it is close to or equal to zero as seen in the example results in Figure III.2.

The floating-point format works in exactly the same way but without having to shift values to fit it into a memory size. The values are simply multiplied and added. To implement that approach all variables need to be declared as float. Figure VI.4shows the floating-point equivalent code.

# Storing, Interpretting And Outputting Goertzel Values

Sixteen separate functions, for each of the eight frequency-specific coefficients in the “coef” array using both fixed and floating point arithmetic, compute the Goertzel algorithm and calculate the Goertzel Values for each of the eight possible frequencies in parallel via the method shown in the code snippets. The eight Goertzel values are stored in the *gtz\_out* and *gtz\_out\_float* array, for the fixed- and floating- point implementations respectively, ready to be compared to determine the two frequencies present in the input DTMF signal.

The outputs of the algorithm are collected in a 2-by-4 array. The first row contains the frequency outputs of 697Hz, 770Hz, 852Hz, and 941Hz, whereas the second 1209Hz, 1336Hz, 1477Hz, and 1633Hz (array *freq\_mat*). Such grouping allows one to easily obtain the coordinates of the transmitted button, seeTable I.1, by finding the index of the highest Goertzel value in each row.

A “for loop” is used to find the index of the highest Goertzel value for both the floating-point and fixed-point implementations, as seen in Figure VII.1. The indices are then interpreted as the coordinates of the array containing the original characters represented by the DTMF signal (array *symbol\_mat*) ready to be printed.

# Implementation of the Clock Using Software Interrupts

The program utilises software interrupt to generate a clock, with its parameters specified in the file Sys\_BIOs\_config.cfg such as the tick period, which is set to 125µs. An Swi is created using *Clock\_create*() [3]. The function, as its arguments, takes a function that is to be called periodically, a timeout value which defines the time interval between the calls (number of ticks), and other parameters. An SWI has priority over most of the threads and at the same time is one of the fastest. Only a Hardware interrupt (Hwi) has a higher priority and is quicker, however it is not used in this code.

The function generating the discrete signal, *clk\_SWI\_Generate\_DTMF*, is called periodically by combining it with the clock mentioned above. The discrete signal takes as its independent variable the value of *tick.* The variable is obtained using the function *Clock\_getTics*() [3] which returns the number of clock cycles. As the signal is sampled at every clock tick, functions responsible for filtering specific frequencies are called in parallel to the sampling function, creating a filtering process explained in Section VI.

# Implementation of Task in Generation and Detection of Signals

The program performs the frequency/frequencies selection and detection process using a thread called Task. It has a lower priority than Swi but higher than Idle. Similarly to Swi, Task is set up in Sys\_BIOs\_config.cfg file. There are two tasks present *task0\_dtfmGen()* and *task1\_dtfmDetect()*, they operate at the same priority which means that they run in series. Task is the lowest priority thread present in the code. When an Swi is applied, a given Task stops to allow a function specified in the software interrupt to be completed. Once it has finished, the function within the Task is recontinued.

*Task()\_dtfmGen()* and *task1\_dtfmDetect()* run in series in a loop until another function is called by a software interrupt, in this case it is the sampling and filtering process, see Section VIII. Once the Swi function finishes, *Task0\_dtfmGen()* and *task1\_dtfmDetect()* are resumed. There is a time interval present between the generation and detection process (using *Task\_sleep()* [5]) to allow the software interrupts to call the sampling and filtering functions enough times to calculate the Goertzel values.

# Conclusion

The Goertzel algorithm has been successfully implemented in a C IDE for the use of interpreting DTMF signals. Both fixed-point and floating-point arithmetic-based methods have been successfully used and their relative performance compared. The next stage would be to test the algorithm code within a physical embedded system that’s receiving input DTMF signal samples. Both fixed-point and floating-point embedded systems can be realized, enabling a comparison between their physical performance if desired.

##### References

1. N. Dahnoun, “C for Embedded Systems” Lecture Slides, 2022.
2. N. Dahnoun, “Formal report: Goertzel Implementation”, 2022.
3. Texas Instruments, “module ti.sysbios.knlClock”, May 2018, <https://software-dl.ti.com/>
4. Texas Instruments, “TI-RTOS Kernel (SYS/BIOS), User’s Guide”, June 2020.
5. Texas Instruments, “module ti.sysbios.knl.Task”, May 2018, <https://software-dl.ti.com/>