Implementation, optimization and analytics of the Goertzel Algorithm in C using the TMS320 C6000 compiler

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

Mohamed Eldebawy   
*Electrical and Electronic Engineering*  
*University of Bristol*bs19616@bristol.ac.uk

Samuel Cheung   
*Electrical and Electronic Engineering*  
*University of Bristol*jm20101@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

# **Introduction: brief overview of the goertzel algorithm**

A Dual Tone Multi-Frequency (DTMF) signal system comprises of digits 0 up to 9 and characters A, B, C, D, \*, #. They each are made up of 2 frequencies, one high and one low. A DTMF signal is formed by superimposing two different frequency tones from a selection of 8 available frequencies as seen in Table I.1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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

The Fast Fourier Transform (FFT) algorithm is inefficient when applied to a DTMF system since it checks for the presence of all frequencies. The Goertzel algorithm is tailored to only detect the 8 different frequencies (16 different tone combinations), which makes it more computationally efficient They are based on equations below, and its process summarized in Figure 1:

For each desired frequency **k**, the feedback loop yields a product Q(n) at step **n.** The feedforward loop yields a Goertzel Coefficient yk(n) at step **n=206** and is stored in an 8-element array. From each array, 2 of the largest coefficients are extracted, which represents the 2 most dominant frequency components in the signal. This information can further determine the key pressed or the data being extracted, depending on the user’s usage.

This report aims to cover the following parts:

* Detection of one frequency component using the Goertzel Algorithm implemented in C with TI’s SYS/BIOS operating system, samples created internally using clock ticks.
* Detection of all 8 frequency components using the Goertzel Algorithm implemented in C with TI’s SYS/BIOS operating system, sourced from data file named ***data.bin***
* Detection of all 8 frequency components using the Goertzel Algorithm implemented in pure C language, sourced from user inputs.
* Optimization of task 2 using C intrinsics, compiler switches and other techniques

# **Brief overview of optimization in C and equivalent code in assembly**

## Debawy please fill in here

# **Task 1: Detect 1 frequency using the Goertzel Algorithm**

## Samuel Please fill in here

# **Task 2: Detecting all frequencies using *the* goertzel Algorithm**

This section aims to cover the design choice, process and goals of the code ***main\_gtz.c*** and ***util.c*** under ***gtz\_all\_freq***. The Goertzel algorithm consists of 3 main events:

* Tone Generation
* Goertzel Computation
* Frequency Detection

A diagram of a process

Description automatically generated

Figure 0: Top-Level Flow Diagram for general Goertzel Algorithm Implementation

In the case of this task, the tone generation and computation events are undertaken by ***main\_gtz.c*** and the detection by ***util.c***. SYS/BIOS allows these events to run in parallel and in a specific order. The roles of SYS/BIOS in the code are as follows:

* Creates a clock instance such that internal actions can be synchronized to clock ticks
  + For example: the function Clock\_getTicks() returns the number of elapsed clock cycles since the start of the program
* Creates interrupt service routines (ISRs) for tone generation
* Creates ISR for computing the Goertzel Coefficient
* Facilitates communication between ***main\_gtz.c*** and ***util.c***

In ***util.c***, our frequency detection algorithm includes a for loop that goes through the 8 samples that exist in the ***data.bin*** file. For each iteration, a while loop is conditioned by a ***flag***; if detection of all frequencies is finished, the ***flag*** variable is set to 0, which exits the while loop and increments **n** by 1, moving onto computing the next sample. This loop structure is illustrated below:

A diagram of a algorithm

Description automatically generated

Figure I: Flow Diagram for **util.c**

By integrating both ***util.c*** and ***main\_gtz.c***, we obatin the flow diagram below:

A diagram of a process flow

Description automatically generated

Figure I: Flow Diagram for **util.c** + **main.gtz.c**

To validate the code, a test case is implemeted, by sourcing the sample from preset values instead of from the ***data.bin*** file.

|  |  |
| --- | --- |
| Freq1 (Hz) | 852 |
| Freq2 (Hz) | 1477 |
| Mag1 | 32768 |
| Mag2 | 32768 |
| (3)  Where:   * Tick = Clock\_getTicks() * Period = 1/8000 | |

Table I.1 Values assigned to test case sample

The result of the implementation is as follows:

|  |  |
| --- | --- |
| Sourced using data.bin file | Test case |
| A white text with black text  Description automatically generated | A white text with black text  Description automatically generated |

|  |
| --- |
| Clock Cycles used: |

Table I.1 Console output when detection is finished

# **task 3: implementing task 2 without sys/bios, input sourced by user inputs instead of binary data file**

This section aims to cover the design choice, process and goals of the code ***main.c***. Functionally speaking, ***main.c*** performs the identical task of ***main\_gtz.c*** and ***util.c*** combined in the previous section. There are 2 main technical challenges:

1. Pure C code is utilized instead of using SYS/BIOS
2. Instead of sourcing sample from ***data.bin*** file, the sample is sourced from user input

To address the first technical challenge, the sample generation function ***clk\_SWI\_Generate\_DTMF()*** in ***One\_freq*** is studied, especially the ***Clock\_getTicks()*** function. This function returns the “time elapsed in clock ticks”. For example, if the SYS/BIOS clock runs at 1.0MHz and 1ms has massed, calling the function ***Clock\_getTicks()*** will return a UInt32 of 1000.

To get around not having SYS/BIOS, a for loop can be utilized to emulate the clock ticks as obtained from ***Clock\_getTicks()***. The sample is generated in a similar fashion as equation (3), looping the tick variable from n=0 to n=205.

Once all samples are generated, the computation algorithm executes. An interesting phenomenon observed is the relationship between the value of the computed goertzel coefficient and the convergence of the algorithm. The algorithm doesn’t converge for goertzel coefficient values roughly > 105 or < 10-3 . If the algorithm fails to converge, it will return a faulty coefficient. To guarantee convergence, we must consider the values of mag1, mag2 and the scaling factor when outputting the Goertzel\_Value variable into the magnitude[] array; we will call this value C.

|  |  |  |  |
| --- | --- | --- | --- |
| Mag1 | Mag2 | C | Result |
| 32768.0 | 32768.0 | 1010 | Convergence |
| 0.5 | 0.5 | 1 | Non-convergence |
| 1024.0 | 1024.0 | 105 | Non-convergence |

Table I.1 Effect of varying mag1, mag2, C values on results convergence

Therefore, the value combination (mag1,mag2,C) = (32768.0, 32768.0, 1010) is chosen.

Addressing the 2nd technical challenge, we simply perform the detection algorithm in the reverse direction. Instead of obtaining the character from the pad array using the dominant Goertzel coefficients, we generate the frequencies using user inputs. This can be done using a while loop conditioned by the user input, shown below in table 2:

|  |  |  |
| --- | --- | --- |
| Input character value | Example | Action |
| Numbers 0-9,  letters A-D,  \*,  # | 5 | Proceeds to Goertzel computation and detection |
| Any other characters | K | exists for loop early and produces an error message, asks for user inputs again |
| string = ‘quit’ | quit | Breaks loop, program ends |

Table I.1 All possible input combinations of **main.c**

The function of main.c can be summarized in the following flow diagram:

A diagram of a computer program

Description automatically generated

Figure I: Flow Diagram for **main.c**

After running

# Task 4: Optimizing task 2 using compiler switches, C intrinsics and other techniques

# Conclusion

We can draw several conclusions from the design process, code execution and results:

##### References

1. N. Dahnoun, “Advanced DSP and FPGA Implementation” Lecture Slides, 2024.
2. N. Dahnoun, “Formal report: Goertzel Implementation”, 2024.
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, “System Clock Manager in SYS/BIOS”, Dec 2009, https://software-dl.ti.com/dsps/dsps\_public\_sw/sdo\_sb/targetcontent/sysbios/6\_21\_01\_16/exports/docs/docs/cdoc/ti/sysbios/knl/Clock.html#get.Ticks
6. Texas Instruments, “module ti.sysbios.knl.Task”, May 2018, <https://software-dl.ti.com/>
7. Texas Instruments, “TMS320C6000 Compiler Optimization User’s Guide”, June 2020, <https://software-dl.ti.com/>