FACULTY OF MECHATRONICS,
INFORMATICS AND INTERDISCIPLINARY
STUDIES TUL

Master Thesis

Implementation of real-time digital signal processing algorithms using STM32F7xx series microcontrollers

Study programme: N0714A150003 Mechatronics
Author: **Muhammed Çagri Küçükalp**
Thesis Supervisors: Ing. Miroslav Holada, Ph.D.
Institute of Information Technology and Electronics

Liberec 2024

FACULTY OF MECHATRONICS,
INFORMATICS AND INTERDISCIPLINARY
STUDIES TUL

Master Thesis Assignment Form

Implementation of real-time digital signal processing algorithms using STM32F7xx series microcontrollers

Name and surname:	**Muhammed Çagri Küçükalp**
Identification number:	M21000195
Study programme:	N0714A150003 Mechatronics
Assigning department:	Institute of Information Technology and Electronics
Academic year:	2022/2023

Rules for Elaboration:

1. Familiarize yourself with the STM32F7xx microcontroller. Focus on hardware support for digital signal processing computations.
2. Design firmware that will process the digital signal from the ADC in real time and write the results to the DA converter or send them to a higher-level system via a communication line.
3. Functionalities to be performed by this firmware are digital filtering (FIR and IIR filters), spectrum calculation, and simple recognition algorithms (DTMF detection).
4. Create a Matlab application that will be used as a graphical user interface for this firmware. Its purpose will be to compute the parameters of data processing algorithms according to the user specification and send them to the STM board via USB (CDC Virtual COM port) interface. It will also display the data resulting from data processing performed by firmware.
5. Test the developed firmware and analyze its performance limits (maximum sampling and data processing speeds). Make sure that your firmware fully utilizes the capabilities of the processor used.

Scope of Graphic Work:	as needed by documentation
Scope of Report:	40-50 pages
Thesis Form:	printed/electronic
Thesis Language:	english

List of Specialised Literature:

[1] Chassaing R.: Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK, Wiley-IEEE Press, ISBN-13: 978-0470138663, 2008.

[2] Porat B.: A Course in Digital Signal Processing', John Wiley & Sons, 1997.

[3] Norris D.: Programming with Stm32 Getting Started with the Nucleo, McGraw-Hill Education, ISBN: 978-1-26-003132-4, 2018

Thesis Supervisors:	Ing. Miroslav Holada, Ph.D. Institute of Information Technology and Electronics
Date of Thesis Assignment:	January 26, 2023
Date of Thesis Submission:	May 15, 2024

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
Dean

doc. Dr. Ing. Jaroslav Hlava
study programme guarantor

Liberec January 26, 2023

Declaration

I hereby certify, I, myself, have written my master thesis as an original and primary work using the literature listed below and consulting it with my thesis supervisor and my thesis counsellor.

I acknowledge that my master thesis is fully governed by Act No. 121/2000 Coll., the Copyright Act, in particular Article 60 – School Work.

I acknowledge that the Technical University of Liberec does not infringe my copyrights by using my master thesis for internal purposes of the Technical University of Liberec.

I am aware of my obligation to inform the Technical University of Liberec on having used or granted license to use the results of my master thesis; in such a case the Technical University of Liberec may require reimbursement of the costs incurred for creating the result up to their actual amount.

At the same time, I honestly declare that the text of the printed version of my master thesis is identical with the text of the electronic version uploaded into the IS/STAG.

I acknowledge that the Technical University of Liberec will make my master thesis public in accordance with paragraph 47b of Act No. 111/1998 Coll., on Higher Education Institutions and on Amendment to Other Acts (the Higher Education Act), as amended.

I am aware of the consequences which may under the Higher Education Act result from a breach of this declaration.

May 14, 2024 Muhammed Çagri Küçükalp

ACKNOWLEDGEMENT

I would like to express my sincere gratitude and appreciation to the individuals and institutions who have played a significant role in completing my thesis. Their support and contributions have been invaluable throughout this journey.

I am grateful to the numerous authors, researchers, and scholars whose work I have referenced and consulted as part of my literature review. Their contributions have provided valuable insights and knowledge that have shaped the foundation of my thesis. I express my sincere appreciation to these individuals and the sources they have published.

Furthermore, I would like to thank **TUL – Technical University of Liberec, specifically the Faculty of Mechatronics, Informatics, and Interdisciplinary Studies Informatics**. I am grateful to the respected Dean, the Head of the Department, and all the staff for providing the necessary resources and knowledge. The academic environment and the learning opportunities everyone offers have been instrumental in my growth and development as a student.

I want to extend my heartfelt thanks to my parents, Recep Küçükalp and İlknur Küçükalp, for their unwavering love and encouragement. I also extend my gratitude to my sister, Elif Küçükalp, my grandmother, Nermin Yılmaz, and my friend, Ing. Elçin Tören, for their constant support and motivation. Their belief in my abilities has been the driving force behind my accomplishments.

Last but certainly not least, I am deeply indebted to my guiding professor and thesis supervisor, **Ing. Miroslav Holada, Ph.D.**, for their unwavering support, continuous guidance, and invaluable mentorship throughout my thesis journey. Their expertise, patience, and constant encouragement have been pivotal in shaping the outcome of my research. I am truly grateful to all the individuals and institutions mentioned above who have significantly completed my thesis. Their contributions, whether big or small, have impacted my academic and personal growth.

ABSTRACT

This thesis details the design and implementation of firmware tailored for the STM32F7 series microcontroller, focusing on efficiently processing digital signals acquired from an ADC (Analog-to-Digital Converter). The firmware's primary objective is to process these signals in real-time, either outputting them to a DA converter or transmitting them to a higher-level system via various communication methods, including USB (Universal Serial Bus) CDC (Communications Device Class) Virtual COM port. This work includes the development of digital filtering techniques using FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filters, execution of FFT (Fast Fourier Transform) for spectrum analysis, and implementation of simple recognition algorithms such as DTMF (Dual-Tone Multi-Frequency) detection.

A MATLAB-based graphical user interface is also developed to interact seamlessly with the STM32 microcontroller via a USB CDC Virtual COM port, enabling real-time configuration and visualization of signal processing parameters and results. The firmware architecture supports comprehensive testing and performance analysis, providing insights into the microcontroller's operational limits, efficiency, and scalability in handling real-world signal processing tasks. These analyses demonstrate the capability of the STM32F7 microcontroller to meet the demands of embedded DSP applications.

Keywords: Embedded DSP Firmware, STM32F7 Microcontroller, Real-Time Signal Processing, USB CDC Virtual COM port.

ABSTRAKT

Tato diplomová práce popisuje návrh a implementaci firmwaru přizpůsobeného pro mikrokontroléry řady STM32F7, se zaměřením na efektivní zpracování digitálních signálů získaných z ADC (převodníku analogového signálu na digitální). Hlavním cílem firmwaru je zpracování těchto signálů v reálném čase, buď jejich výstupem na DA převodník nebo jejich přenosem do vyššího systému prostřednictvím různých komunikačních metod, včetně USB (Univerzální Sériové Magistrály) CDC (Třída Zařízení Komunikace) Virtuálního COM portu. Tato práce zahrnuje vývoj technik digitálního filtrování pomocí FIR (konečné impulzní odezvy) a IIR (nekonečné impulzní odezvy) filtrů, provádění FFT (rychlé Fourierovy transformace) pro analýzu spektra a implementaci jednoduchých algoritmů rozpoznávání, jako je detekce DTMF (dvoutónové vícefrekvenční).

Také bylo vyvinuto grafické uživatelské rozhraní založené na MATLABu pro bezproblémovou interakci s mikrokontrolérem STM32 prostřednictvím USB CDC virtuálního COM portu, které umožňuje konfiguraci a vizualizaci parametrů zpracování signálu v reálném čase a výsledků. Architektura firmwaru podporuje komplexní testování a analýzu výkonu, která poskytuje přehled o operačních limitech, efektivitě a škálovatelnosti mikrokontroléru při zvládání úkolů zpracování signálů v reálném světě. Tyto analýzy demonstrují schopnost mikrokontroléru STM32F7 splnit požadavky vestavěných DSP aplikací.

Klíčová slova: Firmware vestavěného DSP, Mikrokontrolér STM32F7, Zpracování signálů v reálném čase, USB CDC virtuální COM port.

LIST OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODES

1. OBJECTIVE

This thesis is dedicated to designing and developing advanced firmware for real-time digital signal processing (DSP) using the STM32F767ZI microcontroller, known for its robust digital signal computation capabilities. The core goal is to engineer firmware that efficiently processes signals from an Analog to Digital Converter (ADC), transforming them into formats that can be transmitted to higher-level systems. This involves implementing sophisticated digital filtering techniques, including Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters, and exploring the integration of simple recognition algorithms such as DTFM (Dual-Tone Multi-Frequency) detection.

A critical component of this project is to establish an interactive interface between the STM32 microcontroller and a personal computer, utilizing MATLAB. This interface will leverage the USB CDC (Communications Device Class) Virtual COM port to provide user-friendly access for configuring DSP parameters, enhancing the system's adaptability and responsiveness to diverse operational conditions.

Comprehensive testing and evaluation of the firmware are essential for this research, mainly focusing on the STM32F767ZI microcontroller's configuration limits. This phase aims to optimize the firmware according to the microcontroller's specifications and assess performance under these constraints. This evaluation is crucial for providing insights into the system's efficacy in real-time applications.

The foundational STM code, central to this thesis, manages UART (Universal Asynchronous Receiver/Transmitter), interrupts, and processes MATLAB commands, which are vital for developing the firmware. It covers essential aspects of digital signal processing, such as digital filtering, gain control, and interactions with the SAI (Serial Audio Interface) for audio data management. Interaction with the SAI (Serial Audio Interface) for audio data management.

1.1 Motivation And Scope

1.1.1 Motivation

In an era of rapid technological advancements, DSP emerges as a pivotal field, influencing various sectors such as communications, control systems, multimedia, and consumer electronics. The essence of DSP lies in its ability to manipulate vast streams of real-time data, highlighting the need for robust and efficient microcontrollers. The STM32F7 series stands out among various options due to its formidable processing capabilities and specialized support for DSP operations.

This research is driven by the transformative impact of DSP in practical scenarios and the untapped potential of the STM32F767ZI microcontroller series. The primary aim is to exploit the full capabilities of this microcontroller in DSP applications. The envisioned outcome is a comprehensive firmware that processes digital signals in real-time and seamlessly relays processed data to more complex systems, enhancing functionality and performance.

The implications of this research extend beyond its immediate Scope, offering benefits to various groups:

1. **Researchers**: As a platform for testing theoretical DSP concepts or as a foundation for developing more advanced systems.

2. **Audio Engineers**: Providing valuable tools for sound manipulation and analysis, which are crucial in music production, broadcast engineering, and acoustics.

3. **Software Developers**: Facilitating the incorporation of DSP methodologies into diverse software applications.

4. **Educators in Related Fields**: Assisting instructors in illustrating key computational concepts through practical DSP applications.

Additionally, as envisioned by my supervisor, this program's application in an educational setting underscores its importance. It introduces an innovative teaching methodology where students can engage with DSP systems in real-time, effectively bridging the gap between theoretical understanding and practical application. This hands-on experience is invaluable for students and educators, enhancing comprehension of DSP and its relevance in various technological domains. This thesis transcends the boundaries of a mere academic endeavor. It is a gateway to innovation, education, and practical implementation in numerous fields. The development of this firmware and its accompanying tools are set to contribute significantly to DSP and microcontroller technology, paving new paths for research, development, and education.

1.1.2 Scope

The Scope of this thesis encompasses developing and implementing a real-time digital signal processing (DSP) system using the STM32F767ZI microcontroller series. This endeavor aims to explore and demonstrate the capabilities of this microcontroller in processing complex DSP tasks. The focus areas and objectives of this research are outlined as follows:

1. **Understanding the STM32F767ZI Microcontroller**: The initial phase involves an in-depth study of the STM32F767ZI microcontroller. This includes examining its architecture, processing capabilities, and specific features that support DSP operations. A comprehensive understanding of the hardware is crucial for optimizing the firmware design and implementation.
2. **Designing and Implementing DSP Firmware**: This research aims to design firmware capable of real-time processing digital signals from an Analog-to-Digital Converter (ADC). The firmware will also be able to transmit the processed data to a Digital-to-Analog Converter (DA) or a higher-level system through a communication interface. This includes implementing functionalities like digital filtering (using FIR and IIR filters), spectrum analysis, and basic signal recognition algorithms like dual-ton multi-frequency (DTMF) detection.
3. **Developing a MATLAB Interface**: A significant part of this project involves creating a MATLAB application to serve as a graphical user interface. This application will compute DSP algorithm parameters based on user specifications and facilitate the communication of these parameters to the STM32 board via a USB interface. Additionally, it will display data resulting from the DSP operations performed by the firmware.
4. **Performance Analysis and Optimization**: Testing the developed firmware's performance limits is integral to this research. This includes determining the maximum achievable sampling and data processing speeds and ensuring that the firmware leverages the full potential of the STM32 microcontroller's processing capabilities.
5. **Real-world Applications and Educational Use**: Identifying potential real-world applications for the developed system is crucial. Additionally, the project aims to demonstrate how this system can be utilized in an educational context, providing a practical tool for teaching DSP concepts.

This thesis, therefore, not only focuses on the technical aspects of DSP implementation but also considers its practical applications and educational value. The project is designed to contribute to the microcontroller-based DSP field and the broader technological education and application context.

2. LITERATURE REVIEW

As I embark on my thesis journey, a foundational step is the comprehensive review of existing literature and references. This process is crucial in shaping the research trajectory, providing validation and insight into the complexities of Digital Signal Processing (DSP) with a focus on MATLAB applications and STM32 microcontrollers. The subsequent literature review addresses pivotal questions and challenges inherent in the field, offering a structured approach to the research objectives.

Key Questions and Challenges:

- **DSP Algorithm Implementation:** What are the optimal algorithms for effective DSP, and how can they be efficiently implemented using MATLAB and STM32 microcontrollers?
- **MATLAB's Integration in DSP:** How does MATLAB facilitate DSP applications, particularly in algorithm development and data analysis?
- **Utilization of STM32 Microcontrollers:** What role do STM32 microcontrollers play in DSP, and what are their capabilities and limitations?
- **Real-Time Processing Challenges:** How can real-time signal processing be achieved, and what are its associated challenges?
- **Hardware and Software Synchronization:** How can the synchronization between hardware (STM32) and software (MATLAB) effectively manage for optimal DSP performance?

3. THEORETICAL BACKGROUND

3.1. Overview of STM32F7 Series Microcontroller

The STM32F7 series represents a robust family of microcontrollers developed by STMicroelectronics, a leader in the semiconductor industry. This series, distinguished by its high-performance ARM Cortex-M7 cores, offers an unparalleled balance of efficiency and power. Engineered to meet the demanding requirements of sophisticated digital signal processing (DSP) applications, the STM32F7 series is an exemplary choice for a wide range of embedded systems [1].

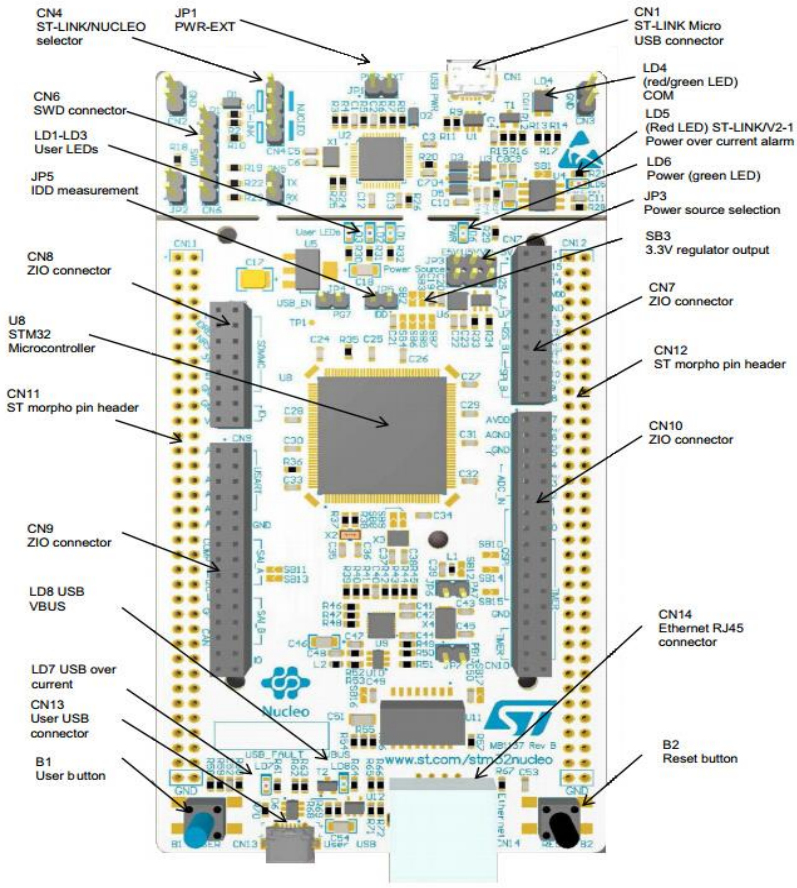

Figure 1. NUCLEO-144 Board with STM32 Microcontroller and interface connectors.[2]

Key Features and Architecture: The STM32F7 series is renowned for its ARM Cortex-M7 processor, which operates at speeds up to 216 MHz. This core is notable for its high-performance and low-power operation, enabled by its advanced architecture and adaptive real-time ART Accelerator™ [3]. The Cortex-M7 core features a floating-point unit (FPU), enhancing the microcontroller's capability to handle complex mathematical operations and DSP tasks, as depicted in Figure 1. [2].

Memory and Storage: The STM32F7 series has up to 2 MB of flash memory and 512 KB of SRAM, facilitating extensive code storage and high-speed data processing. This series also supports external memory interfaces, which enhances system design flexibility and scalability [4].

Connectivity and I/O Interfaces: The STM32F7 series excels in connectivity, incorporating diverse communication interfaces such as UART/USART, SPI, I2C, CAN, and USB OTG with full-speed and high-speed capabilities. The series is also well-equipped with extensive I/O ports and peripherals, including ADCs, DACs, timers, and GPIOs, addressing the varied needs of applications across different domains [4].

Energy Efficiency and Power Management: Energy efficiency is paramount in the STM32F7 series, featuring multiple power-saving modes, including Sleep, Stop, and Standby. These modes are designed to optimize power consumption without compromising system responsiveness, which is crucial for applications requiring prolonged operational periods [4].

Development and Tool Support: STMicroelectronics provides comprehensive development support for the STM32F7 series through tools like the STM32Cube software ecosystem. This includes the Hardware Abstraction Layer (HAL) libraries, middleware components, and numerous software examples. For this thesis, the STM32Cube's DSP library, optimized explicitly for the STM32F7 series, plays a vital role in enhancing the efficiency of digital signal processing. This library facilitates seamless integration from initial development stages to final deployment, offering a collection of optimized DSP functions and algorithms. These resources prove indispensable in streamlining the development of DSP applications, demonstrating practical effectiveness in DSP tasks [4].

3.2. Principles of Analog to Digital Conversion (ADC)

In digital signal processing, particularly in systems that integrate STM32 microcontrollers with peripheral modules like PMOD I2S2, the Analog Digital Conversion (ADC) process plays a pivotal role. This process involves converting analog signals, such as audio inputs, into their digital counterparts, effectively bridging the gap between analog and digital realms. The STM32F7 series demonstrates proficiency in handling this conversion, utilizing the I2S interface to ensure a high-quality data transfer from analog to digital format [5].

The ADC process within this system involves a well-coordinated interaction between the STM32F767 microcontroller and the PMOD module. The PMOD I2S2, featuring the Cirrus CS5343

AD and CS4344 DA converters, is critical in this context (Figure 2) [6] Audio signals are captured by the microcontroller through the I2S RX line and undergo digital processing. They are then output via the I2S TX line, thus completing their journey from the analog to the digital domain and back [7].

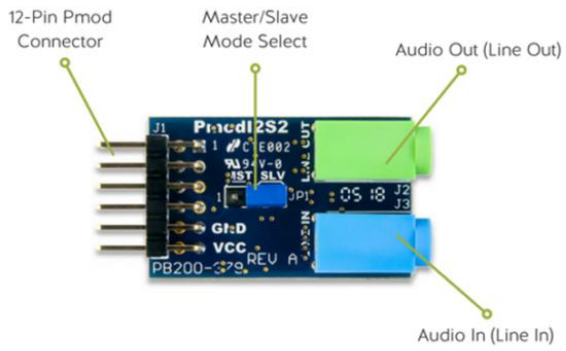

Figure 2. PMOD Audio Module [6]

A critical aspect of this process is understanding the electrical characteristics of the AD and DA converters. Recognizing their different full-scale voltage ranges is essential, as this affects the amplitude of the output signal relative to the input, particularly in a passthrough configuration. This gain, a byproduct of the voltage range differences, is quantified through precise calculations, highlighting the intricate nature of signal conversion in this system [8].

The ADC's capability to accurately capture input voltages, considering elements like voltage dividers on the PMOD board, is validated through comprehensive testing. These tests ensure adherence to datasheet specifications, confirming that the ADC avoids exceeding its clipping threshold and maintains signal integrity within its 24-bit dynamic range. MATLAB visualizations further support these tests, offering a graphical representation of the signal's amplitude range and showcasing the ADC's precision [9].

The full-scale analog voltage range of the AD and DA converters on the PMOD board differs, affecting the output amplitude in a passthrough configuration. The CS5343 AD converter has a full-scale input peak-to-peak range of 0.568 times Vcc, while the CS4344 DA converter has an output peak-to-peak range of 0.65 times Vcc. This variance results in a conversion gain. For instance, an AD converter sampling at 0.5Vcc will produce a digital value of approximately 4922289 ((0.5*(30.568))*(2^24-1)). When passed through the DA converter, this value is converted back into an analog signal, completing the digital-to-analog signal conversion cycle and illustrating the amplitude difference due to the converters' varying voltage ranges.

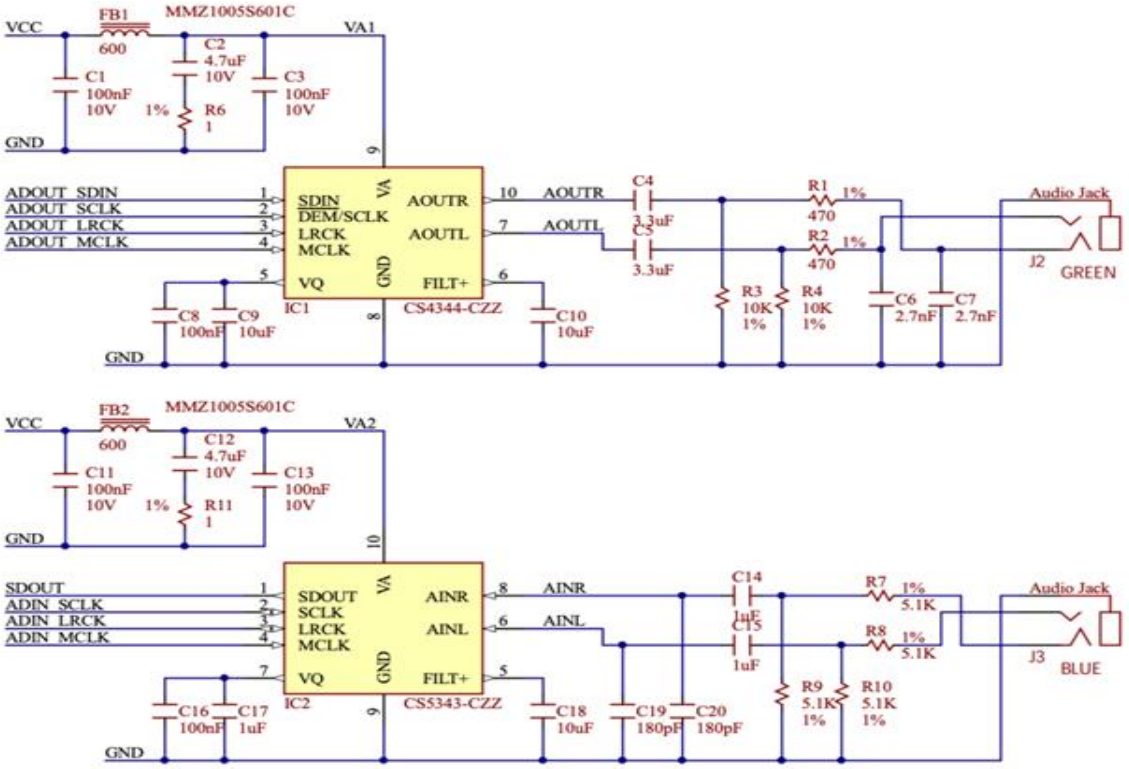

Figure 3. Schematic of Audio Signal Processing Circuit with ADC and DAC [10]

The schematic provided (Figure 3) [10] delineates the CS5343 analog-to-digital converter (ADC) interface connected to an STM32 microcontroller, a crucial component in the digital signal processing chain explored in this thesis. This ADC is instrumental in transducing the analog audio input from the blue jack into a digital representation, employing an I2S interface for high-fidelity data transfer. The schematic incorporates passive components that condition the audio signal, ensuring a clean, noise-free conversion. The precision of this ADC, with its 24-bit resolution, is central to capturing the nuanced acoustic information necessary for high-quality audio processing tasks performed by the STM32. Considering the voltage divider on the PMOD board, the A/D converter perceives the input voltage as halved. Tests with a precise 3.3V supply from the Analog Discovery 2 USB multitool showed that a 2V amplitude input signal (4V peak-to-peak) is the maximum before clipping occurs in the A/D converter. These findings align with the datasheet specifications for the A/D converter's full-scale input range, confirming the system’s ability to handle input signals up to 2V in amplitude without clipping. MATLAB plots of passthrough sample data for unclipped and clipped signals reveal the effective utilization of the 24-bit sample range, further substantiating the system’s adeptness in ADC [6,10].

3.3. Digital Filtering Techniques: FIR

In digital signal processing (DSP), Finite Impulse Response (FIR) filters are highly regarded for their stability and linear phase characteristics. Unlike Infinite Impulse Response (IIR) filters, FIR filters rely solely on present and past inputs, devoid of feedback mechanisms. This section delves into implementing FIR filters within DSP frameworks, focusing on STM32F7 series microcontrollers.

FIR filters are pivotal in DSP for isolating specific frequency bands and mitigating noise from signals. Their digital nature allows for greater flexibility and configurability compared to analog filters, facilitating adjustments in filter characteristics without needing physical component changes.

The shift towards digital filters, such as FIR, significantly reduces reliance on external hardware components like resistors and capacitors, cutting manufacturing costs and boosting reliability. Unaffected by environmental changes that impact analog filters, digital filters offer consistent performance, making them a more stable and reliable choice for signal processing tasks.

The digital filter equations are based on the following essential transfer function shown in the z domain [11]:

$$Y(z) = H(z)X(z), \tag{1}$$

Where Y(z) is the filter output, X(z) is the filter input, and H(z) is the transfer function of the filter. H(z) can be expanded as follows [11]:

$$Y(z) = \frac{b(1)+b(2)z^{1}+\cdots.+b(n+1)z^{n}}{a(1)+a(2)z^{-1}+\cdots.+a(n+1)z^{-n}} X(z) \tag{2}$$

A and b are coefficient sets, and z is a delay element. Differing fundamentally from its counterparts with feedback mechanisms, the Finite Impulse Response (FIR) filter operates without relying on feedback. This aspect can be understood by examining its transfer function, which, at its core, is derived similarly to other filters. A notable distinction in the FIR filter's formulation is the simplification of its coefficients, particularly with the 'a' coefficient being singular and assigned a value of one (a(1) = 1). This simplification is evident when the output equation, y(t), is formulated in the context of an FIR filter [11]:

$$y(t) = (z) = \frac{b(1)x(k)+b(2)x(k-1)+\cdots.+b(n+1)x(k-n)}{a(1)} \tag{3}$$

For the FIR algorithm, the current output is generated based only on the current and previous inputs. In effect, an FIR is a weighted sum operation. FIR filters have several advantages and drawbacks. One of the main advantages is that FIR filters are inherently stable. This characteristic makes

designing FIR filters easier than designing IIR filters. In addition, FIR filters can provide linear phase response, which may be necessary for some applications. Another essential advantage of FIR filters is that they are more resistant to quantization noise in their coefficients. Replacing a(1)=1 and C for the b constants, the equation for the FIR filter is as follows [11]:

$$y(n) = C0x(n) + C1x(n-1) + C2x(n-2) + C3x(n-3) + \ldots, \quad (4)$$

Y (n) is the most recent filter output, and x(n) is the most recent filter input. The filter does rely on previous inputs, as shown by the terms x(n-1), x(n-2), etc. The Cx constants determine the filter response and can be derived using many different algorithms, yielding different characteristics.
The first input, x(1), is multiplied by C0. The output y(1) is as follows [11]:

$$y(1) = C0x(1) \quad (5)$$

The x(1) input is saved for the next pass through the FIR algorithm. The second input, x(2), is multiplied by C0, and the previous input, x(1), is multiplied by C1. The output y(2) is as follows [11]:

$$y(2) = C0x(2) + C1x(1) \quad (6)$$

x(1) The and x(2) inputs are saved for the following input, x(3), and so on. The order of an FIR filter is equal to one less than the number of constants and indicates the degree of complexity and the number of input samples that need to be stored. The higher the order, the better the characteristics of the filter (sharper curve and flatter response in the non-attenuation region) [11,12].

3.3.1. STM32 Implementation of FIR Filters

A pivotal component of implementing Finite Impulse Response (FIR) filters in digital signal processing (DSP) within the STM32 microcontroller environment is the arm_fir_f32 function. This function, integral to the CMSIS (Cortex Microcontroller Software Interface Standard) DSP library, is specifically tailored for the ARM Cortex-M processor series. This section delineates the role and functionality of the arm_fir_f32 function as it is applied in the FIR filter implementation on an STM32F7 series microcontroller [13].

```
void PerformFIR() {

static float32_t firOutput[BLOCK_SIZE];
```

```
float32_t *inputSignal;
inputSignal = (selectedChannel == 0) ? sample_L : sample_R; // Channel selection
arm_fir_f32(&S_FIR, inputSignal, firOutput, BLOCK_SIZE);
SendData2MTLB(ID_FIR, (uint8_t*)firOutput, BLOCK_SIZE);
}
```

Source Code 1. STM FIR Filter Processing by CIMSIS-Library

The **arm_fir_f32** function is a processing function for the floating-point FIR filter. It is designed to apply an FIR filter to an input data block and produce a corresponding output data block. The function signature is as follows:

```
void arm_fir_f32(const arm_fir_instance_f32 *S,
const float32_t *pSrc,
float32_t *pDst,
uint32_t blockSize);
```

Source Code 2. arm_fir_f32 function by CIMSIS-Library

Parameters and Functionality[13]:

- **S**: This parameter points to an instance of the floating-point FIR filter structure. This structure contains information about the filter's order, coefficients, and state buffer.
- **pSrc**: This represents the pointer to the block of input data. In a typical DSP application, this could be a segment of a signal, such as an audio waveform or sensor data, that needs to be processed.
- **pDst**: This is the pointer to the block of output data. It stores the filtered signal after the application of the FIR filter.
- **blockSize**: This parameter specifies the number of samples to process. It determines the size of the input and output data blocks.

The firOutput is an array where the filtered signal will be stored and input-signal points to the current input data. The function applies the FIR filter defined by S_FIR to the input signal and stores the result in firOutput. In the provided source code, firOutput is an array storing the filtered signal. The input-signal variable points to the current input data, and the function applies the FIR filter defined by S_FIR to this input, storing the result in firOutput.

The arm_fir_f32 function is indispensable for DSP applications that demand fast and efficient

signal processing. Its capability to handle floating-point operations renders it particularly suitable for precision-critical applications, such as audio signal processing and other real-time DSP tasks. The use of floating-point arithmetic ensures a broad dynamic range and mitigates common issues associated with fixed-point processing, like quantization errors and overflow, thereby enhancing the fidelity and reliability of the DSP applications.

3.4. Digital Filtering Techniques: IIR

The IIR topology extends directly from this equation by moving the denominator of the expanded H(z) to the left side of the equation [14,15]:

$$(a(1)+a(2)z–1+…+a(n+1)z–n)Y(z) = (b(1)+b(2)z–1+…+b(n+1)z–n)X(z) \quad (7)$$

In the time domain, this equation appears as follows [14,15]:

$$a(1)y(k) + a(2)y(k –1)+…+a(n+1)y(k –n) = b(1)x(k) + b(2)x(k –1)+…+b(n+1)x(k –n) \quad (8)$$

where y(k) represents the current filter output, x(k) represents the current input, y(k-1) represents the previous output, x(k-1) represents the previous input, and so on. If this equation is solved for y(k) [14,15]:

$$y(k) = \frac{b(1)x(k)+b(2)x(k-1)+\ldots+b(n+1)x(k-n)-a(2)y(k-1)-\ldots-a(n+1)y(k-n)}{a(1)} \quad (9)$$

This equation shows that the IIR filter is a feedback system that generates the current output based on the current and previous inputs and outputs. The IIR structure has unique advantages and drawbacks. The main advantage of the IIR structure is that it provides a frequency response comparable to an FIR filter of a higher order. This results in fewer calculations necessary to implement the filter. IIR filters can suffer from instability because they rely on feedback. As a result, they are more challenging to design, and special care must be taken to prevent an unstable system. IIR filters may also have a non-linear phase response, making them inappropriate for some applications where a linear phase is necessary. Finally, because they rely on past outputs, they tend to be more sensitive to quantization noise, making them difficult to implement with 16-bit fixed point hardware. Generally, 32-bit hardware is necessary for an IIR filter implementation. Digital IIR filters reduce dependence on external components like resistors and capacitors, lowering production costs and enhancing reliability. Unlike analog filters, whose performance might be impacted by environmental changes, digital IIR filters maintain consistent functionality due to their algorithmic basis. IIR filters are integral in DSP due to their recursive nature, which employs current and past inputs and outputs. This characteristic enables IIR filters to replicate the behavior of analog filters

efficiently while retaining the advantages of digital processing. They are precious in scenarios where filter stability and computational efficiency are paramount [15].

3.4.1. STM32 Implementation of IIR Filters

In the STM32 platform, the implementation of IIR filters involves initializing and processing the filter structure and parameters through dedicated CMSIS library functions. The following conceptual overview provides insight into how IIR filters can be integrated into STM32 applications:

Initialization of the Filter Structure: This step involves setting up the IIR filter's parameters, including the order, coefficients, and state buffer. The initialization ensures the filter is correctly configured to process the incoming signal according to the desired specifications.

Processing the Input Signal: Once the filter is initialized, the input signal is processed through the filter. This process involves manipulating the input data based on the filter's characteristics, producing an output that reflects the desired frequency response.

Handling the Filtered Output: The IIR filter process output is typically used for further processing or analysis within the application. Depending on the application's specific requirements, this could include data visualization, transmission, or storage tasks.

The STM32 microcontroller environment utilizes functions like arm_biquad_cascade_df1_f32 for IIR filter implementation, part of the CMSIS DSP library designed for ARM Cortex-M processors. Here is a sample implementation of an IIR filter in STM32:

```
void PerformIIR() {
float32_t *inputSignal = (selectedChannel == 0) ? sample_L : sample_R;
float32_t iirOutput[BLOCK_SIZE];
arm_biquad_cascade_df1_init_f32(&S_IIR, IIR_ORDER, iirCoeffs, iirState);
arm_biquad_cascade_df1_f32(&S_IIR, inputSignal, iirOutput, BLOCK_SIZE);
SendData2MTLB(ID_IIR, (uint8_t*)iirOutput, BLOCK_SIZE);
}
```

Source Code 3. STM IIR Filter Processing by CIMSIS-Library

In this implementation, **arm_biquad_cascade_df1_f32** processes the input signal using the IIR filter defined by **S_IIR** and stores the result in **iirOutput**.

Functionality of arm_biquad_cascade_df1_f32

- **S**: This parameter refers to an instance of the floating-point IIR filter structure containing information about the filter's stages, coefficients, and state buffer.
- **pSrc**: Represents the pointer to the block of input data.

- **pDst**: Points to the output data block where the filtered signal is stored.
- **blockSize**: Specifies the number of samples to process per block.

The IIR filter's output is determined by its order and the coefficients defining its frequency response.

3.5 Fast Fourier Transform (FFT) in Signal Analysis

The Fourier Transform, a fundamental concept in signal processing, interprets a signal in the continuous time domain to discern its frequency composition. In practical applications, especially those involving Analog-to-Digital Converters (ADC), signals exist discretely, necessitating the Discrete Fourier Transform (DFT). The Fast Fourier Transform (FFT), an algorithmic advancement, mirrors the DFT's functionality but achieves the results with greater computational efficiency.

The operational essence of the FFT lies in its methodical breakdown of the input data array. It systematically halves the data, progressing recursively until it attains a pair-wise format. Commencing with these pairs, the FFT executes a 2-point transformation, employing these initial results to advance to a 4-point FFT. This process iterates, utilizing the outcomes of each step (2-point, 4-point, etc.) to progress to the subsequent 8-point FFT, continuing in this manner until the N-point FFT is accomplished [16]:

Contrasting the computational demands, the DFT typically requires N^2 complex calculations to output N data points. The FFT, however, streamlines this process significantly, requiring only (N/2) x log2(N) complex calculations. This optimized calculation requirement of the FFT becomes increasingly advantageous as the number of input points (N) escalates, solidifying its efficiency over the traditional DFT, particularly for large datasets [16]:

N (number of input samples)	8	256	1024
DFT (complex calculations)	64	65536	1048576
FFT (complex calculations)	12	1024	5120

Table 1. Comparison of Computational Complexity between DFT and FFT for Different Sample Sizes [16]

The FFT allows for frequency analysis in a system and is an essential tool for any digital signal processing (DSP) system. Traditionally implemented on dedicated DSP hardware, FFT functionality is also available in DSP-enabled Microcontroller Units (MCUs). These advanced MCUs blend the specialized capabilities of DSPs with the adaptability of general-purpose programmable microcontrollers, allowing embedded systems to perform FFT operations efficiently. This integration provides a versatile platform for a wide range of applications, leveraging the power of FFT in more

flexible and integrated environments [16,17]:

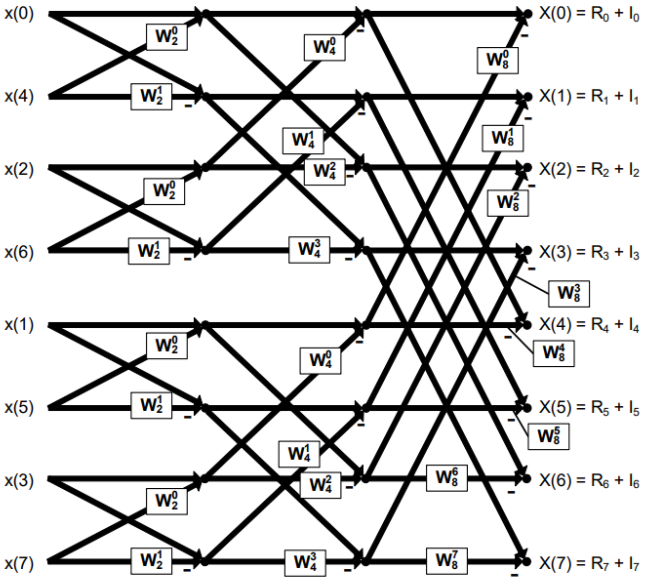

Figure 4. Fast Fourier Transform (FFT) Computational Graph.[18]

In each phase of the FFT process, the count of complex data points remains consistent, illustrating the intensive computational nature of this algorithm. For instance, whether it is a 2-point stage or a 4-point stage, the algorithm handles 32 data points in each. This characteristic of the FFT underscores its computational intensity, particularly as the value of N, representing the number of data points, increases. Moreover, the precision of calculations in the initial stages of the FFT is crucial, as any inaccuracies or errors tend to amplify as the process progresses through subsequent stages. Therefore, ensuring high accuracy in the early calculations is paramount for the optimal performance of the FFT algorithm. This aspect highlights the need for meticulous implementation and execution of the FFT, especially in applications involving large datasets [18]:

3.5.1. STM32 Implementation of FFT

In the digital signal processing domain (DSP) within the STM32 microcontroller framework, the Fast Fourier Transform (FFT) is pivotal in analyzing signal frequency components. Implementing FFT on STM32 microcontrollers, mainly using the CMSIS DSP software library, exemplifies leveraging advanced computational techniques in embedded systems. The following provides an insight into how FFT is implemented in the STM32 environment, as demonstrated by the specific code utilized in the application.

```
void PerformFFT() {
// Create the Hamming window
```

```
float32_t window[FFT_LENGTH];
createHammingWindow(window, FFT_LENGTH);
// Apply the window to the input signal
float32_t windowedSignal[FFT_LENGTH];
float32_t *input signal;
// Determine the input signal to process based on the selected channel
if (selectedChannel == 0) {
inputSignal = sample_L; // Left channel
} else {
inputSignal = sample_R; // Right channel
}
// Windowing the signal to be processed
for(int i = 0; i < FFT_LENGTH; i++) {
windowedSignal[i] = (float32_t)inputSignal[i] * window[i];
}
// Initialize the FFT instance
arm_rfft_fast_init_f32(&S, FFT_LENGTH);
// Perform the FFT operation on the windowed signal
arm_rfft_fast_f32(&S, windowedSignal, fftOutput, 0);
// Calculate the magnitude of the FFT
arm_cmplx_mag_f32(fftOutput, fftMagnitude, FFT_LENGTH / 2);
// Send the FFT magnitude data back to MATLAB
SendData2MTLB(ID_FFT, (uint8_t*)fftMagnitude, FFT_LENGTH / 2);
}
```

Source Code 4. STM FFT Processing by CIMSIS-Library

Code Breakdown and Explanation:

1. **Window Creation and Application**:
    - The **PerformFFT()** function initiates by creating a Hamming window, a method used to reduce spectral leakage in FFT analysis. The window is generated by **createHammingWindow(window, FFT_LENGTH)**; where **FFT_LENGTH** defines the size of the data set.
    - The generated window is then applied to the input signal. This process involves multiplying each data point in the input signal with the corresponding window

function value to minimize edge effects in the FFT analysis.

2. **Signal Processing and FFT Execution**:
    - The function determines the input signal (**sample_L** for the left channel or **sample_R** for the right channel) based on the selected channel (**selectedChannel**).
    - Windowed signal data is prepared by applying the Hamming window to the input signal.
    - The **arm_rfft_fast_init_f32(&S, FFT_LENGTH);** call initializes the FFT process, configuring the FFT length and setting up internal structures necessary for the computation.
    - The actual FFT computation on the windowed signal is performed by **arm_rfft_fast_f32(&S, windowedSignal, fftOutput, 0);**, where the third parameter indicates that a regular FFT (not inverse FFT) is to be computed.
3. **Magnitude Calculation and Data Transmission**:
    - Post-FFT, the magnitude of each frequency component is calculated using **arm_cmplx_mag_f32(fftOutput, fftMagnitude, FFT_LENGTH / 2);**. This step converts the complex FFT output into a real-valued magnitude spectrum.
    - Finally, the FFT magnitude data is sent back to MATLAB for further analysis or visualization, facilitated by **SendData2MTLB(ID_FFT, (uint8_t*)fftMagnitude, FFT_LENGTH / 2);**.

Understanding CMSIS DSP Library Functions:

- **arm_rfft_fast_init_f32()**: This function initializes the FFT algorithm instance, setting up internal buffers and variables required for FFT computation. It considers the length of the FFT to be processed, thereby optimizing subsequent computations.

Parameters:

- **S (in/out)**: A pointer to the **arm_rfft_fast_instance_f32** structure will hold the FFT process configuration.
- **fftLen (in)**: The length of the real sequence for which the FFT is computed. This parameter determines the size of the FFT computation and should be one of the supported lengths (32, 64, 128, 256, 512, 1024, 2048, 4096).

- **arm_rfft_fast_f32()**: The core function for performing the FFT. It takes the initialized FFT instance the input data buffer, and computes the FFT, placing the output in the specified buffer. The **ifftFlag** parameter determines whether a forward or inverse FFT is performed.

Flag to specify the operation type. A value of 0 indicates RFFT, and 1 indicates RIFFT.

Parameters:

- **S (in)**: A pointer to the initialized **arm_rfft_fast_instance_f32** structure containing the configuration for the FFT.
- **p (in)**: Pointer to the signal data input buffer.
- **pOut (out)**: Pointer to the output buffer where the FFT result (IFFT, if specified) will be stored.
- **ifftFlag (in)**: Flag to specify the operation type. A value of 0 indicates RFFT, and 1 indicates RIFFT.

- **arm_cmplx_mag_f32()**: After the FFT is performed, this function calculates the magnitude of the complex FFT output, which is essential for analyzing the frequency spectrum of the input signal.

Parameters:

- **pSrc (in)**: Pointer to the input vector containing complex numbers (real and imaginary parts interleaved).
- **pDst (out)**: Pointer to the output vector where the magnitudes of the complex numbers will be stored.
- **numSamples (in)**: Number of complex samples in the input vector.

3.6. Real-Time Processing of DTMF Signals

In embedded systems, where the interest often lies in identifying specific frequencies within an input signal, the Goertzel Algorithm emerges as an instrumental tool. This algorithm excels in scenarios where the frequencies of interest are predetermined, offering a streamlined and focused approach to frequency detection [19,20].

At its core, the Goertzel Algorithm is designed to detect the presence of a singular frequency component within a signal. Its operational foundation can be likened to a two-pole IIR filter. However, its theoretical underpinnings are closely tied to the principles of a single-bin output from the Discrete Fourier Transform (DFT). This relationship with DFT allows the Goertzel Algorithm to isolate and analyze a specified frequency within a signal efficiently [17].

The Mathematical Formulation of the Goertzel Algorithm

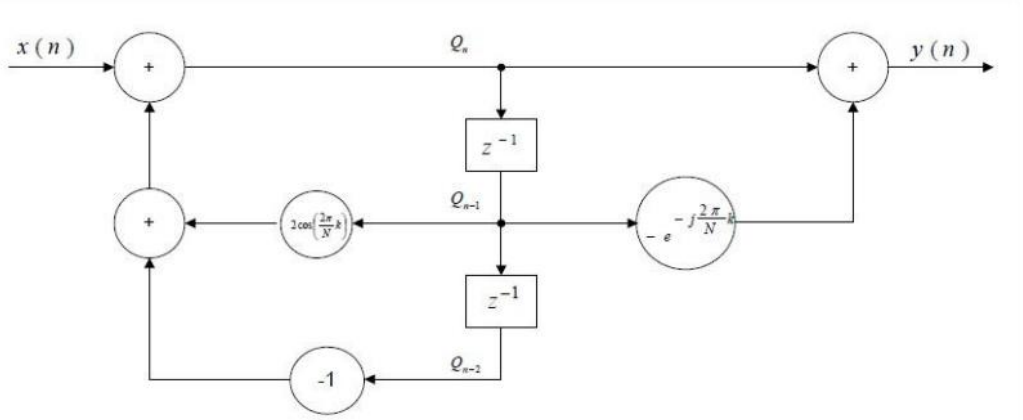

Figure 5. Block Diagram of the Goertzel Algorithm [21]

1. **Calculation of Qn**: $Qn = x(n) + 2\cos\left(\frac{2\pi k}{N}\right) . Q_{n-1} - Q_{n-2}$

   Where:

   - $x(n)$ represents the current input signal at time n.
   - Qn is the output of the algorithm at time n.
   - k is a constant determined by the targeted frequency.
   - N is the total number of samples.

2. **Magnitude Calculation**: $|y_k(N)| = Q^2(N) + Q^2(N - 1) - 2\cos\left(\frac{2\pi k}{N}\right) . Q(N) \cdot Q(N-1)$

   This equation calculates the magnitude of the frequency component at the target frequency, k, after processing N samples.

In the above equation, if N is set to 205[1], then the value of k needs to be determined. The value of this constant k also determines the tone we are trying to detect and is given by [18]:

$$k = N * ftone / fs \quad (10)$$

Where: ftone = frequency of the tone

fs = sampling frequency.

Now we can calculate the value of the coefficient and obtain Table 2. [18]

$$2\cos(2\pi k/N) \quad (11)$$

A distinctive feature of the Goertzel Algorithm is its output validity, which is contingent on processing a complete set of input samples. The output becomes meaningful and accurate only after the algorithm processes N input samples, where N denotes the total count of inputs utilized [21].

.

This characteristic underscores the algorithm's efficacy in the real-time processing of a predetermined number of required samples.

Frequency	K	Coefficient
697	18	1.703275
770	20	1.635585
852	22	1.562297
941	24	1.482867
1209	31	1.163138
1336	34	1.008835
1477	38	0.790074
1633	42	0.559454

Table 2. DTMF Tone Coefficients [22]

The Goertzel Algorithm's focused frequency detection makes it particularly suitable for embedded systems that require real-time analysis of specific frequency components. Its ability to isolate individual frequencies from a composite signal allows for efficient processing in applications such as DTMF decoding, signal monitoring, and targeted frequency analysis in various industrial and communication systems.

For instance, integrating the Goertzel Algorithm into STM32-based systems leverages the microcontroller's DSP capabilities to perform precise and real-time frequency analysis. This integration is critical in applications that demand high accuracy and efficiency in frequency detection, such as in telecommunication systems and signal processing modules.

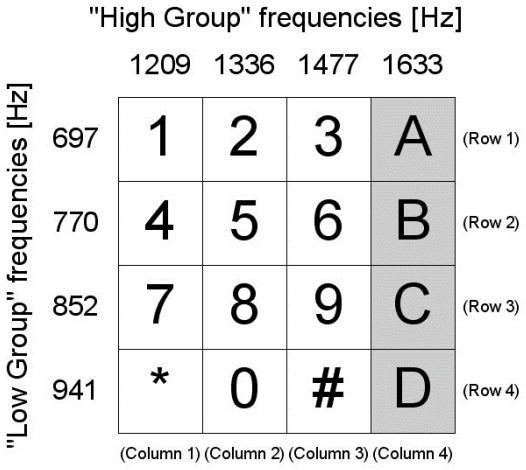

Figure 6. DTMF Frequency Table for Keypad Tones.[23]

DTMF tone generation is an easy problem that can be solved by stepping through constant SINE

tables and adding the tones together. For example, the "5" tone combines the Row 2 tone of 770 Hz and the Column 2 tone of 1336 Hz, as shown.

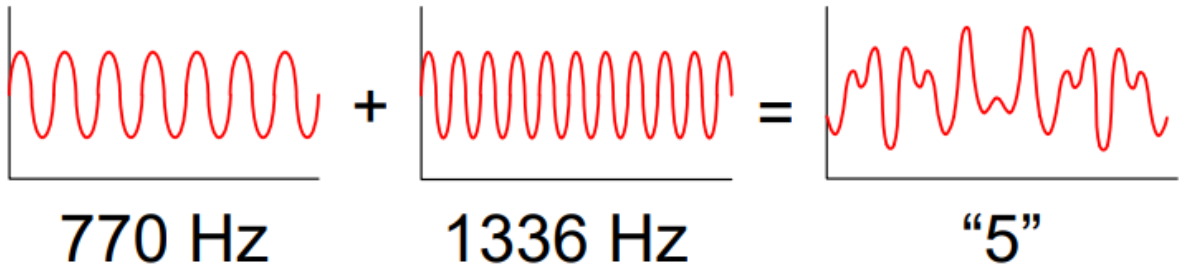

Figure 7. Generation of the DTMF Tone for the Digit "5" Using 770 Hz and 1336 Hz Frequencies.[18]

Detecting DTMF signals necessitates a system's capability to discern both a row and a column tone, distinguishing between actual DTMF tones and voice. The Goertzel Algorithm is a suitable method for DTMF signal recognition due to its computational efficiency and non-reliance on historical input data. This algorithm is adept at isolating specific frequencies, facilitating the decoding of DTMF signals with promptness and accuracy.

These equations are tailored explicitly to detect DTMF signals. They should be included in the signal processing logic of the firmware, especially in systems utilizing STM32 microcontrollers, to handle real-time audio signal processing efficiently [24].

3.6.1. STM32 Implementation of DTMF

In signal processing within embedded systems, the STM32 microcontroller series is a potent platform for implementing real-time Dual-Tone Multi-Frequency (DTMF) detection algorithms. The STM32's processing power and internal architecture, complemented by its digital signal processing (DSP) capabilities, make it an ideal candidate for executing computationally intensive tasks, such as DTMF detection using the Goertzel Algorithm.

```
void detect DTMF(uint32_t blockSize) {
// Define a pointer to the input signal array
float32_t *input signal;
// Select the appropriate input signal based on the selected channel
inputSignal = (selectedChannel == 0) ? sample_L : sample_R;
// Compute the Goertzel coefficients for the DTMF frequencies
for (int i = 0; i < DTMF_NUM_FREQS; i++) {
float32_t k = (0.5f + ((blockSize * dtmfFreqs[i]) / Fs));
```

```
goertzelCoeff[i] = 2 * arm_cos_f32((2 * M_PI * k) / blockSize);
}
// Loop over each DTMF frequency to compute its power
for (int i = 0; i < DTMF_NUM_FREQS; i++) {
// Initialize variables for the Goertzel algorithm
float32_t prev1 = 0.0f, prev2 = 0.0f, total power = 0.0f;
// Apply the Goertzel algorithm to the signal block
for (int j = 0; j < blockSize; j++) {
float32_t y = inputSignal[j] + goertzelCoeff[i] * prev1 - prev2;
prev2 = prev1;
prev1 = y;
}
// Calculate the total power of this frequency
totalPower = prev1 * prev1 + prev2 * prev2 - goertzelCoeff[i] * prev1 * prev2;
// Compare the power to a predefined threshold
if (totalPower > THRESHOLD) {
// DTMF frequency detected, perform the necessary action
}
// Store the power values for further processing or transmission
dtmfPower[i] = totalPower;
}
// Use the DTMF power values for transmission to MATLAB or another system
SendData2MTLB(ID_DTMF_DETECTED, (uint8_t*)dtmfPower, DTMF_NUM_FREQS);
}
```

Source Code 5. STM DTMF Processing by CIMSIS-Library

The Goertzel Algorithm, a DSP technique, is particularly effective in detecting specific frequency components within a signal. It is commonly employed in DTMF detection due to its efficiency and lower computational overhead than the Fast Fourier Transform (FFT). The STM32 series, with its comprehensive suite of peripherals and hardware accelerators, allows for the seamless integration of such algorithms to process and analyze audio signals in real-time.

The code snippet presented for detecting the DTMF function illustrates a practical approach to DTMF detection on an STM32 microcontroller. This function is crafted to compute the power of predefined DTMF frequencies within a signal block and identify the presence of DTMF tones based on a threshold comparison.

The function delineates several vital steps:

1. Initialization begins by defining a pointer to an array holding the input signal. This pointer is assigned to the appropriate channel data (**sample_L** or **sample_R**) based on user selection.
2. Coefficient Calculation: Utilizing a loop, the function calculates the Goertzel coefficients for each frequency of interest. The coefficient calculation involves an iterative computation of the cosine function, **arm_cos_f32**, which is efficiently executed thanks to the STM32's floating-point unit (FPU).
3. Power Computation: The function then enters another loop to apply the Goertzel Algorithm to the signal block. It initializes the variables **prev1** and **prev2** to store the intermediate values required for the Goertzel calculation. Each input signal sample within this loop contributes to computing the power for the current DTMF frequency.
4. Threshold Comparison: After computing the power for a frequency, the function compares it against a predefined threshold. If the power exceeds this threshold, it suggests detecting a DTMF tone, triggering the appropriate response within the system.
5. Data Storage and Transmission: The computed power values are stored for further analysis or communication with other systems, such as MATLAB, for visualization or additional processing.

4. METHODOLOGY AND SYSTEM IMPLEMENTATION

4.1. System Architecture and Design

The architecture delineated in this thesis is specifically engineered for digital signal processing (DSP) applications, utilizing STM32 microcontrollers combined with a MATLAB interface to ensure optimal performance, flexibility, and real-time processing capabilities. This system architecture includes essential components such as the STM32 microcontroller and peripheral devices like Analog-to-Digital Converters (ADCs). These are augmented by a MATLAB-based graphical user interface (GUI) that enhances control and data visualization, as detailed in Figure 8.

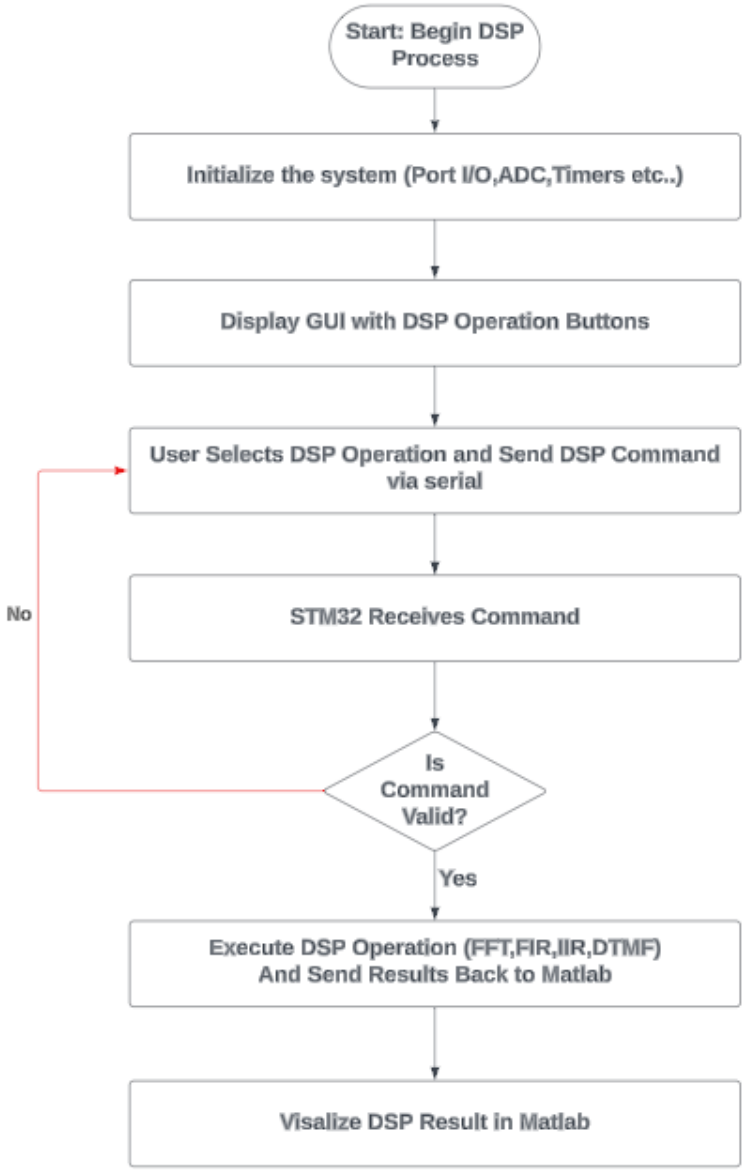

Figure 8. Workflow diagram of Matlab and STM32 DSP Operations

Central to the system is the STM32 microcontroller, particularly chosen for its superior performance traits within the STM32F7 series. This series is strategically selected for its exceptional high-performance characteristics, crucial for meeting real-time DSP applications' rigorous demands. Integrating a Floating-Point Unit (FPU) within the STM32F7 series significantly augments the efficiency of executing complex DSP algorithms. The STM32F7 series, equipped with the Cortex-M7 processor, is renowned for its robust processing capabilities at a core clock frequency of 216 MHz. This processor's speed and advanced architectural features make it exceptionally well-suited for demanding DSP tasks. The ADCs play a pivotal role in the system, converting analog signals into a digital format for subsequent processing. These converters' meticulous configuration and proficient utilization are paramount to maintaining signal integrity and achieving the desired real-time performance.

In embedded audio processing, the system's architecture is meticulously orchestrated to align with the Integrated Just-In-Time Sound (I-JITS) protocol, ensuring high-fidelity sound reproduction. The configuration employs the STM32 microcontroller that operates at a core clock frequency of 216 MHz, independent of the I2S interface's Master Clock (MCLK). The MCLK serves as the cornerstone for audio signal synchronization, typically derived from an external crystal oscillator or a phase-locked loop (PLL) circuit to meet the precise requirements of the analog-to-digital converters (ADCs) within the audio codec.

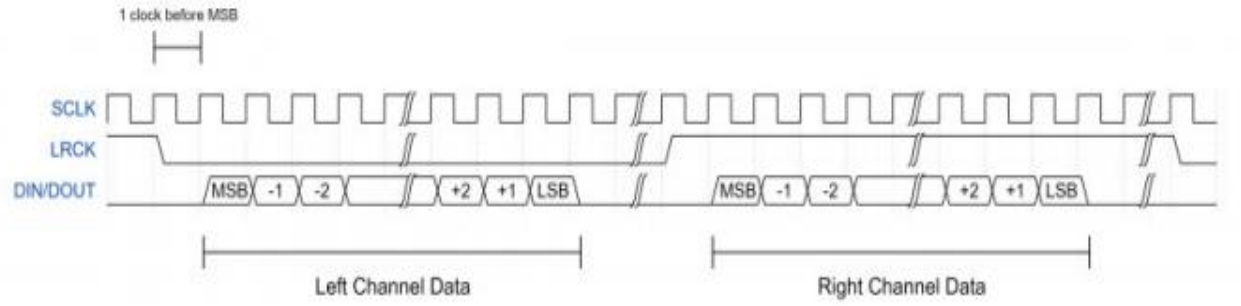

Figure 9. I2S Data Transmission Format Showing Left and Right Channel Bit Alignment.[14]

The MCLK is set distinctly from the microcontroller's operating frequency, tailored to facilitate a sampling rate of 48 kHz, a standard in professional audio applications. This rate is achieved by calibrating the MCLK to a suitable frequency that, when divided down through the I2S clock management system, provides the exact bit clock (SCLK) and word select (LRCK) rates necessary for seamless stereo audio streaming. A 2.8224 MHz MCLK in this implementation would be inappropriate, as it aligns with the 44.1 kHz sampling rate commonly used for CD-quality audio.

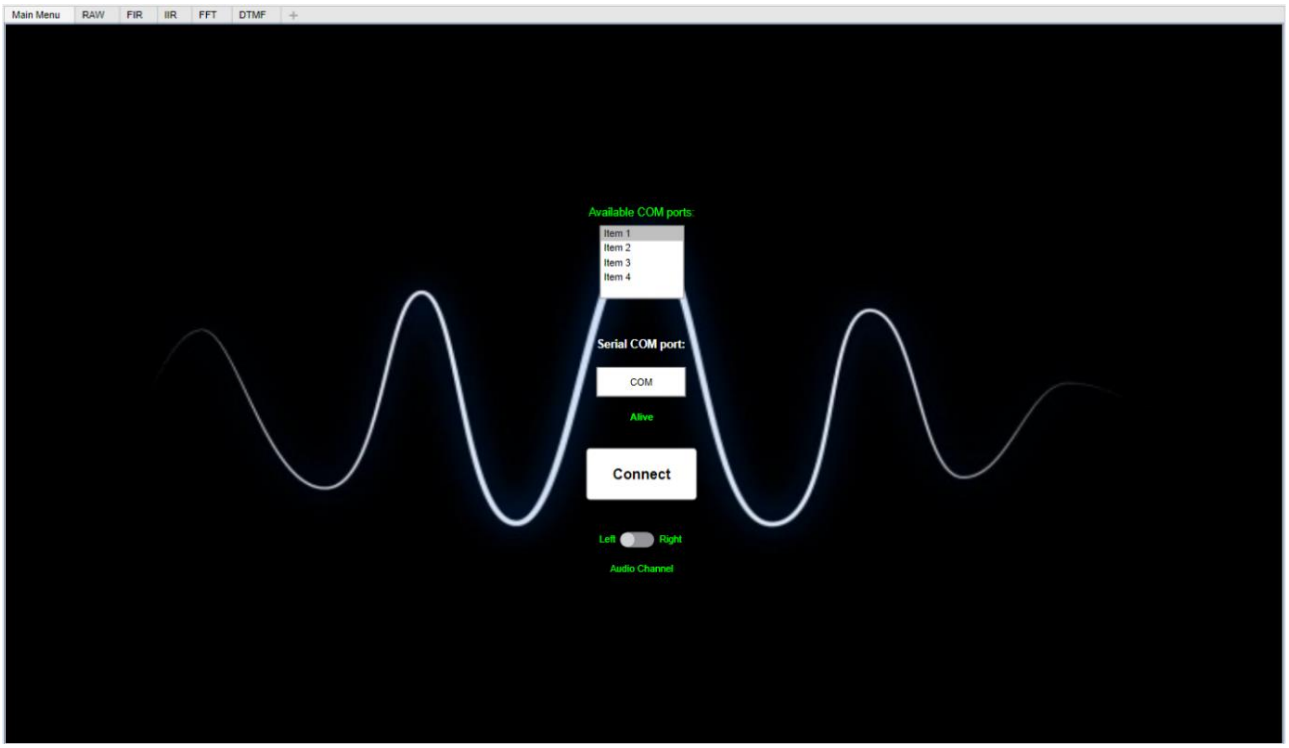

Figure 10. Graphical User Interface for DSP Application with Serial COM Port Connection.

Figure 10 illustrates the MATLAB-based GUI, which enables the user's ability to connect to the device by selecting from available COM ports, toggle between audio channels, and monitor the results of DSP processes in a visual format. This interface empowers the user to interact with the system dynamically, offering an enhanced user experience in real-time system management.

The MATLAB interface is further extended to include dedicated tabs for Finite Impulse Response (FIR), Infinite Impulse Response (IIR), Dual-Tone Multi-Frequency (DTMF) signaling, and Fast Fourier Transform (FFT) analysis, each facilitating specialized signal processing functions. The FIR tab, for instance, provides tools for generating and applying filter coefficients, visualizing magnitude and phase responses, and sending filter configurations to the STM32 microcontroller. Similarly, the IIR segment allows intricate filter design and real-time response observation. DTMF and FFT tabs offer user-friendly interfaces for tone generation, detection, and spectral analysis, contributing to a comprehensive DSP toolkit. The GUI's modular design ensures a seamless workflow, enabling users to navigate between different DSP operations efficiently, enhancing the interactive experience, and promoting practical exploration of DSP concepts.

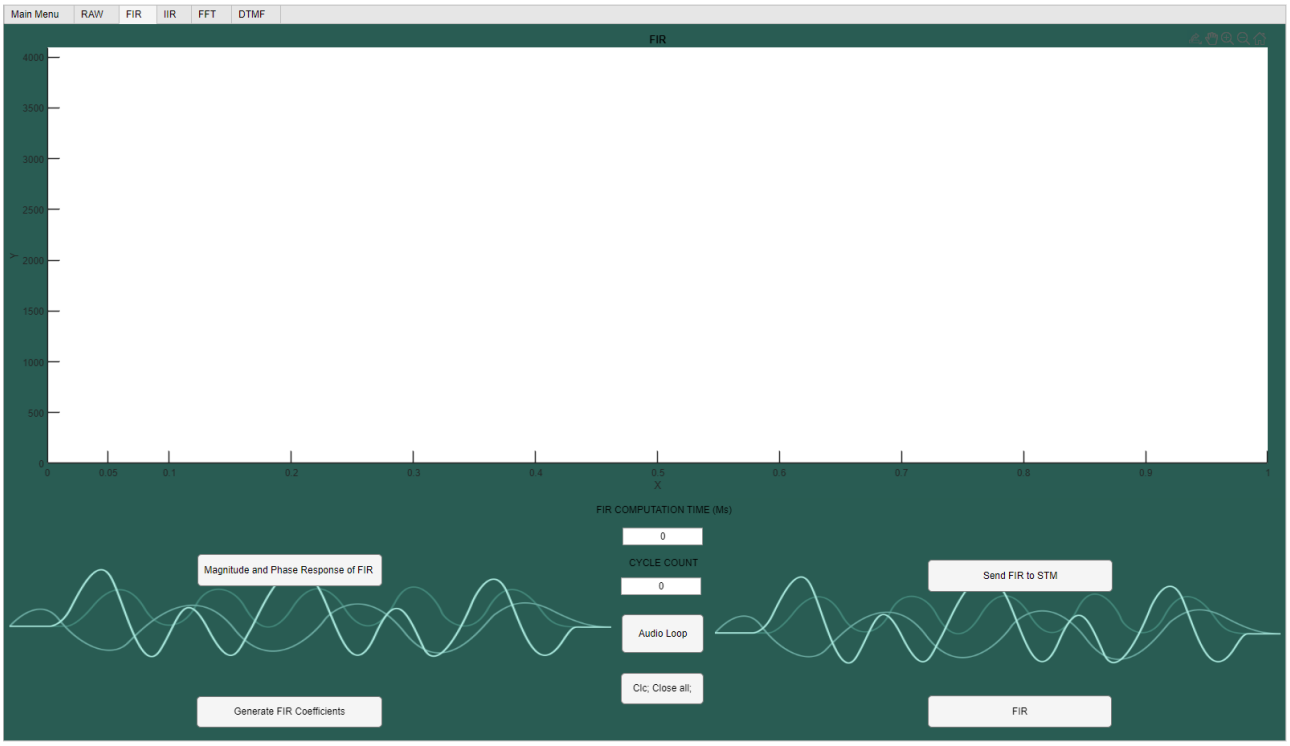

Figure 11. Graphical User Interface for DSP Application

Figure 11 depicts the GUI's additional segments, showcasing the user's ability to interact with various DSP functionalities. These include generating filter coefficients, transmitting filter data to the hardware, and real-time audio loopback testing. This level of interaction exemplifies the GUI's role as an essential facilitator in applying complex DSP algorithms within an educational or research context.

4.2. Firmware and Software Development

In embedded systems, integrating firmware development for the STM32 microcontroller with a MATLAB Graphical User Interface (GUI) establishes a cohesive platform for executing real-time digital signal processing (DSP). This section explores the complexities of developing firmware that equips the STM32F7 series with sophisticated DSP functionalities, including digital filtering, spectral analysis, and signal recognition.

The firmware capitalizes on the robust processing power of the STM32F7 series to efficiently perform a range of DSP functions. A notable innovation in the firmware design is the protocol for identifying incoming data types. This is facilitated by an identifier (ID), where IDs greater than 215,215 denote floating-point numbers, and lower values represent 32-bit unsigned integers. This identification strategy optimizes handling varied data types, aligning them with the specific computational demands of DSP operations.

A standardized block size for data packets has been adopted to improve system performance.

This measure addresses previous challenges associated with variable data sizes, which have led to performance degradation and occasional system failures. Standardizing data packet size has enhanced system stability and mitigated negative impacts on processing performance during increases in sampling rates.

Concurrently, a MATLAB GUI has been meticulously designed to manipulate and configure the DSP algorithms executed on the microcontroller. Via the USB CDC virtual COM port, the GUI facilitates dynamic adjustments to DSP parameters that directly influence the microcontroller's real-time processing activities. The MATLAB script's **readDataSTM32** function exemplifies this interaction, efficiently sorting and classifying data based on the ID. Additionally, the **isReady** utility verifies data availability within the serial buffer, ensuring consistent and dependable data acquisition.

To further augment system performance, the firmware development incorporated the use of the CMSIS DSP library. This strategic choice was influenced by empirical evidence demonstrating its advantages in efficiency. Previous endeavors without this library had experienced noticeable performance declines, particularly under conditions involving extensive buffer sizes, leading to significant computational delays. The CMSIS DSP library, specifically optimized for the ARM Cortex processor series, provides a suite of highly optimized signal processing functions that markedly enhance DSP operations on the STM32 microcontroller. The adoption of this specialized DSP library has proven essential for achieving the requisite performance levels in real-time signal processing tasks.

4.3. Hardware Setup and Firmware Integration

In this thesis's development, the clock system's configuration in STM32 microcontrollers is meticulously outlined, underscoring its critical role in ensuring optimal microcontroller operations across various digital signal processing (DSP) tasks. This setup is paramount for achieving accurate signal sampling, processing, and analysis, and it is pivotal in applications involving audio signal modulation, data conversion, and real-time signal analysis.

The STM32's clock system configuration involves defining multiple clock sources and adjusting various multiplexers and dividers to tailor the operational frequencies for different peripherals. This process begins by selecting an appropriate external high-speed clock (HSE) source, set at 8 MHz, serving as the reference for the Phase-Locked Loop (PLL), which multiplies the HSE to achieve the higher frequencies required by the system

SAI A	
Synchronization Inputs	Asynchronous
Basic Parameters	
Audio Mode	Master Transmit
Output Mode	Stereo
Companding Mode	No companding mode
SAI SD Line Output Mode	Driven
Protocol Parameters	
Protocol	I2S Standard
Data Size	24 Bits
Number of Slots (only Even Values)	2
Clock Parameters	
Master Clock Divider	Enabled
Audio Frequency	48 KHz
Real Audio Frequency	48.0 KHz
Error between Selected	0.0 %
Advanced Parameters	
Fifo Threshold	Empty
Output Drive	Disabled
SAI B	
Synchronization Inputs	Asynchronous
Basic Parameters	
Audio Mode	Master Receive
Output Mode	Stereo
Companding Mode	No companding mode
Protocol Parameters	
Protocol	I2S Standard
Data Size	24 Bits
Number of Slots (only Even Values)	2
Clock Parameters	
Master Clock Divider	Enabled
Audio Frequency	48 KHz
Real Audio Frequency	48.0 KHz
Error between Selected	0.0 %
Advanced Parameters	
Fifo Threshold	Empty
Output Drive	Disabled

Figure 12. SAI Configuration on STM

Accurate configuration of the Phase-Locked Loop (PLL) is essential as it determines the core operating frequency of the microcontroller. Specific multipliers (N), prescalers (P), and dividers (Q and R) are carefully selected to produce the required internal clock speeds. The output of the PLL, referred to as PLLCLK, drives the SYSCLK at 216 MHz, providing the necessary operational speed for high-performance tasks.

This meticulous clock configuration for the I2S peripheral, which handles audio data, is crucial. The I2S_CKIN, derived from the PLLI2S, is adjusted to closely match the standard audio rate of 48 kHz. Given the precision of modern digital systems and thorough calibration, the audio frequency precision targets the standard 48 kHz, ensuring optimal audio processing without any deviations.

The accurately configured clock frequencies are essential for data acquisition and processing in real-time DSP applications. For example, the correct setup of I2S_CKIN ensures that the audio data is

sampled and processed at the intended rate, which is crucial for maintaining audio integrity and quality. The absence of frequency variation confirms the system's capability to handle precise audio processing tasks, meeting the stringent performance and accuracy requirements in digital audio applications. Moreover, integrating a meticulously configured hardware setup and specialized firmware development is vital in effectively deploying embedded systems for real-time digital signal processing. This section explores the nuanced processes involved in setting up the hardware and integrating firmware that supports the execution of DSP algorithms, mainly focusing on enhancing the functionality of Analog-to-Digital Converters (ADCs) and optimizing real-time processing tasks such as Dual-Tone Multi-Frequency (DTMF) detection. The hardware configuration is vital in preparing the microcontroller environment for precise operation. An essential aspect of this setup is configuring the timers, which serve as the heartbeat for tasks requiring strict timing, such as sampling signals for DSP.

1. **Prescaler (PSC - 16-bit value)**: The prescaler value is pivotal in scaling the input clock frequency (f_clk) to a more manageable timer clock frequency. **Timer Clock Frequency = f_clk / (PSC + 1)** utilizes the prescaler value to divide the clock frequency. The subtraction by one accounts for the zero-based counting nature of the hardware timers. For instance, with an f_clk of 21.6 MHz and a PSC of 21600, the resultant timer clock frequency would be 1 Hz, derived from 21.6 MHz / (21600 + 1).
2. **Counter Mode**: This setting dictates the count sequence of the timer. An 'Up' configuration signifies an ascending count from 0, instrumental in standard timing operations.
3. **Counter Period (AutoReload Register - ARR)**: The counter period is crucial in defining the upper limit of the timer count. Upon reaching this predefined value, the timer can trigger a designated event or reset, creating a consistent time base for various operations. An ARR setting of 10000-1 implies a counting sequence up to 10000 before initiating the subsequent action.
4. **Auto-reload preload**: This functionality allows a buffer register for the ARR, with 'Disable' indicating a direct application of the ARR value, facilitating immediate changes to the timer's behavior without needing a buffer phase.
5. **Trigger Output (TRGO) Parameters**: These parameters determine the specific event that prompts the timer to emit a trigger signal. The 'Reset (UG bit from TIMx_EGR)' configuration suggests the trigger is linked to the update generation event, typically utilized for counter resets or register updates.

Calculating the Timer Overflow Time: The timer overflow time represents the duration for a full count cycle, from 0 to the ARR value. The formula for calculating this time interval is **(PSC + 1) × (ARR + 1) / f_clk**. Applying the given values, with an f_clk of 21.6 MHz, we get **(21600 + 1) × (10000 + 1) / 21.6 × 10^6**. This calculation is essential for synchronizing the embedded system's operations with the required precision for DSP tasks.

$$\text{Duration} = \frac{(TimerPrescaler+1)(TimerPeriod+1)}{TimerClock} \tag{12}$$

4.4. System Interface and Data Analysis

This segment elucidates the interface intricacies between MATLAB and STM32 microcontroller, which underpins the robust digital signal processing (DSP) framework delineated herein. The intercommunication is facilitated by a high-throughput serial communication channel, operating at an expedited baud rate of 2*115200. This rate is meticulously selected to accommodate the high-bandwidth requirements essential for real-time DSP while maintaining the integrity and fidelity of the data exchange.

4.4.1. Serial Communication Protocol and Data Format

The strategic implementation of the communication protocol is a testament to the system's efficiency. **SendData2MTLB** within the STM32 firmware stands as a paradigm of this, adeptly packaging data alongside its corresponding identifier before dispatching it over the CDC protocol via a micro-USB connection. The data format is judiciously determined by the iD, which predicates the subsequent processing routine, typically utilizing single floating-point precision to accommodate the rigorous demands of DSP computations.

The STM32 firmware's **DataReceive_MTLB_Callback** function further exemplifies meticulous data handling, where incoming data packets from MATLAB are classified and directed toward their designated DSP operations. This includes FFT computations, FIR and IIR filtering, DTMF detection, and other signal processing tasks, each delineated by a unique ID and processed accordingly.

4.4.2. MATLAB Data Handling and Real-Time Analysis

The readDataSTM32 function epitomizes the data parsing intelligence of the system in Matlab. It exhibits a protocol that waits for a sufficient data threshold before initiating the read operation, thus ensuring that complete data packets are received. This function's adeptness at handling varying data types and sizes is demonstrated by its conditional logic, which flexibly adapts its reading method depending on the size of the received iD.

The GUI in Matlab serves as the port for real-time control and feedback. It allows dynamic visualization of time-domain and frequency-domain analyses and provides immediate information on the effects of user-set parameters on DSP processes executed by the STM microcontroller. This interaction is vital for time-sensitive applications such as audio signal processing, where user input must be translated into instantly noticeable results.

4.4.3. Visualization and Performance Metrics

MATLAB's prowess in data visualization is leveraged to its full potential, furnishing users with sophisticated tools to interpret the results of DSP applications graphically. For instance, FFT data is plotted to depict the signal's spectral content, while DTMF magnitudes are represented through a lollipop graph, elucidating the nuances of the signal decoding process.

Including the isReady helper function in MATLAB's code underscores the system's intelligent data readiness check, precluding the processing of incomplete data sets, thus preserving the veracity of the subsequent analysis.

Key Interfacing Strategies:

1. **Serial Communication Protocol**: The MATLAB GUI employs a serial communication protocol to send configuration commands to the STM32 microcontroller. These commands include filter coefficients, sampling rates, and operation modes for various DSP functionalities such as FIR, IIR, DTMF, and FFT processing.
2. **Real-time Control and Feedback**: The GUI allows users to observe the real-time effects of parameter adjustments on the DSP processes. This feature is critical for applications requiring immediate response, such as audio signal processing and adaptive filtering.
3. **Data Reception and Parsing**: MATLAB scripts are adept at receiving and parsing data sent from the STM32 microcontroller. The processed data, typically digital audio signals or spectral analysis results, is decoded and prepared for visualization.

4.4.2. Data Visualization and Analysis

The MATLAB environment offers powerful data visualization and analysis tools for interpreting DSP application results.

Visualization Techniques:

1. **Time-Domain Analysis**: The GUI presents time-domain representations of signals, showcasing variations in amplitude over time. This visualization is vital for assessing the performance of filters and observing the effects of signal processing algorithms on audio signals.
2. **Frequency-Domain Analysis**: FFT results and spectral content of processed signals are displayed in the frequency domain. This analysis is crucial for identifying dominant frequency components and evaluating the efficacy of frequency-based algorithms like DTMF decoding.
3. **Filter Response Curves**: The MATLAB GUI graphically represents filter response curves, including magnitude and phase responses. This feature aids in designing and verifying digital filters, ensuring they meet the specified criteria.

The functions **SendData2MTLB**, **DataReceive_MTLB_Callback**, and **readDataSTM32** play crucial roles in facilitating data transfer and processing between the STM32 microcontroller and MATLAB in a digital signal processing (DSP) system, as mentioned below. Here is a detailed explanation of what each function does and how they contribute to data transfer and processing:

1. SendData2MTLB (STM32 Firmware)

Purpose: To send data from the STM32 microcontroller to MATLAB.

Code:

```
int SendData2MTLB(uint16_t iD, uint8_t * xData, uint16_t nData_in_values)
{
// Memory Check: Ensure buffer size accommodates the sent data.
if((sizeof(buf_UART_TX)-4)<(nData_in_values*4)) return -2;
// Setting the status to indicate data is being sent.
s2m_Status=1;
// Data Packaging: Assigning the unique identifier and the number of data points.
((uint16_t *) buf_UART_TX)[0] = iD;
((uint16_t *) buf_UART_TX)[1] = nData_in_values;
// Data Copying: Copying the data into the buffer if the data size is non-zero.
if(nData_in_values>0) memcpy(buf_UART_TX+1, xData, nData_in_values*4);
// Transmission: Sending the data using CDC protocol over a micro-USB connection.
```

```
CDC_Transmit_FS((uint8_t*) buf_UART_TX, nData_in_values*4 + 4);
// Return 0 to indicate successful execution.
return 0;
}
```

Source Code 6. STM Data transmit function from STM to Matlab

Process:

- **Data Packaging**: The function starts by packaging the data with a unique identifier (**ID**) and the number of data points (**nData_in_values**). This identifier is crucial as it helps MATLAB understand the data type received and how to process it.
- **Memory Check**: It checks if the buffer size (**buf_UART_TX**) can accommodate the data to be sent. If the buffer is too small, the function returns an error.
- **Data Copying**: If the data size is appropriate, the function copies the data into the buffer.
- **Transmission**: It uses the CDC (Communication Device Class) protocol over a micro-USB connection to transmit the data to MATLAB.

2. DataReceive_MTLB_Callback (STM32 Firmware)

Purpose: To handle and process data received from MATLAB.

Code:

```
void DataReceive_MTLB_Callback(uint16_t iD, uint32_t * xData, uint16_t nData_in_values)
{
// Switch statement to handle different data types based on the iD.
switch(iD)
{
case ID_CCommand_SetFIR_Left: // ID for setting FIR filter parameters for LEFT channel
// Initialize the FIR filter for the left channel with the received parameters.
arm_fir_init_f32(&S_L_FIR, nData_in_values, (float *) xData, pState, BlockSize);
break;
}}
```

Source Code 7. Matlab Data receive callback function from STM to Matlab

Process:

- **Data Classification**: Upon receiving data, the function first classifies it based on the **ID** value. Each **iD** corresponds to a specific DSP operation, such as FFT calculations, FIR/IIR filtering, or DTMF detection.

- **Data Routing**: The function then routes the data to the appropriate processing routine. For example, data meant for FIR filtering is directed to the FIR filter processing function.
- **Action Execution**: Depending on the **iD**, the function executes different actions. It might initiate a DSP computation, adjust filter parameters, or perform other signal-processing tasks.

3. readDataSTM32 (MATLAB Script)

Purpose: To read and process data sent from the STM32 microcontroller.

Code:

```
function [iD, nData, xData] = readDataSTM32(s)
try
iD = 0; nData = 0; xData = 0;
% Check if at least 4 bytes (2 for iD and 2 for nData) are available
if (s.NumBytesAvailable > 3)
iD = read(s, 1, "uint16");      % Read the ID
nData = read(s, 1, "uint16");  % Read the number of data points
if nData == 0
return; % No data to read, exit the function
end
% Read the data based on the iD
if iD > 2^15
if isReady(s, nData)
xData = read(s, nData, "single"); % Read as single if iD is large
end
else
if isReady(s, nData)
xData = read(s, nData, "uint32"); % Read as uint32 for smaller iD
end
end
end
catch ME
disp(ME.message) % Display any caught errors
end
end
```

Source Code 8. Matlab Read Data function

Process:

- **Data Availability Check**: Initially, it checks if enough data is available to read. This ensures that MATLAB does not attempt to read incomplete data packets.
- **Reading Data**: The function reads the **iD** and **nData** values first. The **iD** determines the type and format of the data to be read.
- **Conditional Data Reading**: Depending on the value of **iD**, the function decides whether to read the data as single precision floating-point (**"single"**) or unsigned 32-bit integers (**"uint32"**). This flexibility allows the function to handle different types of data appropriately.
- **Error Handling**: The function includes error handling to catch and display any issues during the data reading process.

Data Transfer Overview

- **Data Sending (STM32 to MATLAB)**: The **SendData2MTLB** function in STM32 prepares and sends data to MATLAB. This data could be raw sensor readings, processed signals, or other relevant DSP data.
- **Data Reception and Processing (STM32)**: When MATLAB sends commands or data back to STM32, the **DataReceive_MTLB_Callback** function receives this data, identifies its purpose based on the **iD**, and routes it for appropriate processing or action.
- **Data Reading and Handling (MATLAB)**: MATLAB, through the **readDataSTM32** function, reads the data sent by STM32, interprets it based on the **iD**, and processes or visualizes it as required.

5. RESULTS AND DISCUSSION

5.1. Performance of Digital Filters

In evaluating the performance of digital filters within the designed digital signal processing (DSP) system, a critical aspect is computational efficiency, which is precisely the time required to process a data block. This time, fundamentally related to the filter's ability to operate in real-time applications, depends on several key parameters, as outlined below.

1. Sampling Rate and BLOCKSIZE Impact:

Sampling Rate (Fs):

- The sampling rate, denoted as Fs, is essential in digital signal processing, defining the frequency at which analog signals are converted to digital form. Its selection is crucial for determining the temporal resolution of each sample.
- An increase in Fs leads to a shorter duration per sample, influencing the time required to process a given BLOCKSIZE since the processing time for one data block is inversely proportional to the sampling rate. This relationship is captured by the formula

BLOCKSIZE:

- BLOCKSIZE refers to the number of samples processed in a single batch. It is a fundamental parameter in the computational framework of digital signal processing.
- A larger BLOCKSIZE means more samples are processed in each operation, directly affecting the computational load and the processing duration.
- The impact of BLOCKSIZE on processing time can be expressed as Processing Time = BLOCKSIZE / Fs. This formula encapsulates the relationship between the number of samples in a block and the sampling rate, offering a method to estimate the time required to process a single block of data.

2. Processor Performance and Algorithm Efficiency:

- **Microcontroller Attributes:** The STM32 microcontroller, characterized by its CPU_MHZ and CYCLES_PER_SECOND and its architectural nuances, notably the incorporation of Cortex-M7 features conducive to DSP tasks, plays a pivotal role in algorithmic execution efficiency. This efficiency is particularly pertinent in the context of digital signal-processing algorithms.

- **DSP Algorithm Complexity:** The processing time is significantly influenced by the specific DSP algorithms implemented, such as FFT (Fast Fourier Transform), FIR (Finite Impulse Response), IIR (Infinite Impulse Response), and DTMF (Dual-Tone Multi-Frequency) decoding. These algorithms, exemplified by **arm_fir_f32**, **arm_biquad_cascade_df1_f32**, and **arm_rfft_fast_f32**, impart varying computational demands.

3. Hardware Constraints and Capabilities:

- The STM32 microcontroller's hardware features, encompassing memory and peripheral capabilities, are integral to its efficacy in DSP tasks. This includes configurations for the SAI (Serial Audio Interface) for handling audio data and UART for serial communication, which are pivotal in determining the microcontroller's DSP performance.

4. Code Execution Profiling:

- Profiling the execution of the DSP code on the STM32 is indispensable for an accurate assessment of processing time. This involves analyzing functions like **ProcessSamples**, **HAL_SAI_RxCpltCallback**, and **HAL_SAI_TxHalfCpltCallback** to quantify the time required for processing each sample and aggregating it across the BLOCKSIZE.

5. Real-Time Signal Processing Dynamics:

- Functions such as **PerformFFT**, **PerformFIR**, and **detectDTMF** underscore the system's real-time processing prowess. The integration of DMA (Direct Memory Access) and interrupts like **HAL_SAI_RxCpltCallback** and **HAL_SAI_TxHalfCpltCallback** fortifies the system's ability to handle audio signals efficiently in real-time scenarios.

5.1.1. Efficacy of Infinite Impulse Response (FIR) Filters

The efficacy of FIR filters within the DSP framework is illustrated through time-domain output graphs, which reveal the filter's capability to maintain the integrity of the signal's amplitude response while handling real-time processing demands effectively. As depicted in Figure 13, the FIR filter output demonstrates well-defined peaks that align with the expected periodic features of the input signal, highlighting the filter's robustness in maintaining signal integrity without noticeable overshoot or ringing effects. This stable and consistent response across time underscores the filter's reliability and its alignment with design specifications, ensuring that the critical parts of the signal are preserved without loss.

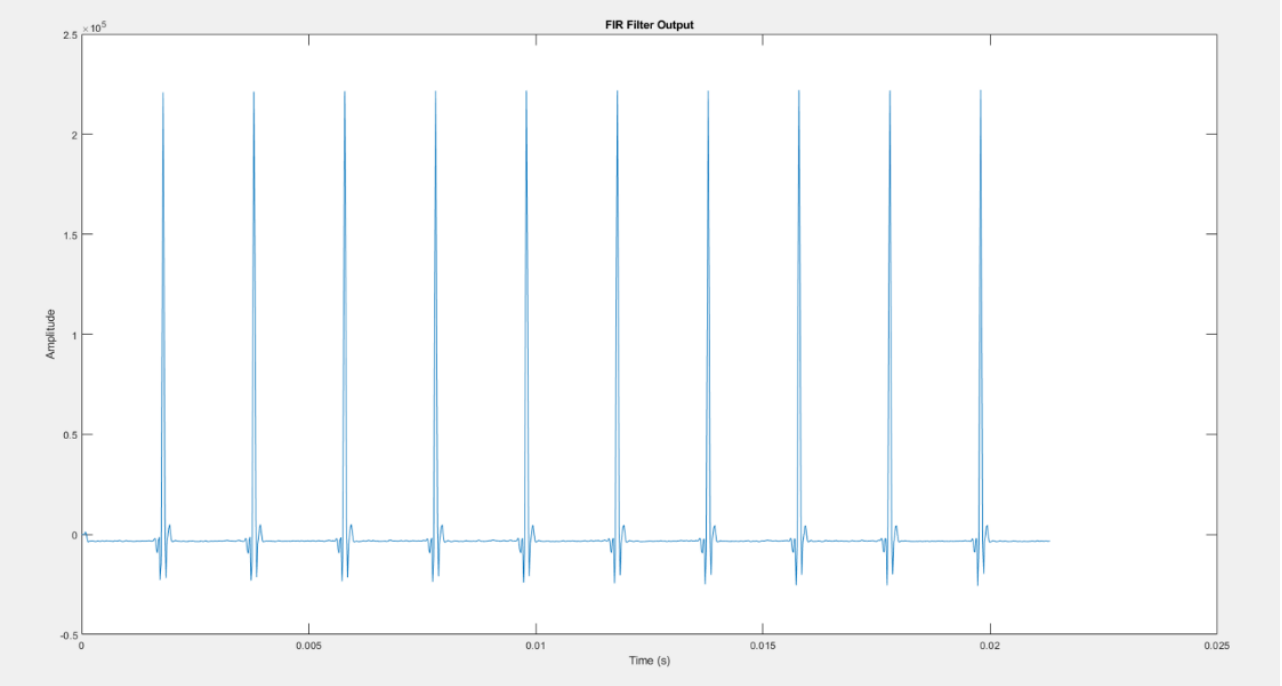

Figure 13.FIR Filter Output Graph

Temporal Characteristics and Amplitude Response The FIR filter's output, as depicted in Figure 13, exhibits well-defined peaks consistent with the input signal's expected periodic features. The precise and sharp transitions between the peaks and troughs underscore the filter's robustness in maintaining signal integrity without noticeable overshoot or ringing effects. Notably, the amplitude levels across these peaks are uniform, indicating a stable and consistent response over time.

Filter Stability and Signal Fidelity The stability of the filter is essential for reliable performance and is reflected in the steady baseline of the output signal, free from any unexpected variations. This demonstrates the filter's reliability during continuous operation. The preservation of the signal's original amplitude after processing indicates that the filter coefficients were selected correctly, ensuring that the critical parts of the signal are maintained without loss.

Filter Design and Application The filter design was tailored to cater to specific application needs, which, based on the observed output, included suppressing unwanted noise or interference while preserving the integrity of the signal's primary frequency components. The performance captured in the graph corroborates the filter's suitability for applications where signal clarity and precision are paramount, such as in digital communication systems or audio processing. A user-friendly function, GenerateFIRCoefficientsButton, embodies the interactive nature of the system. This function prompts the user to select a filter type from a menu of standard filter designs, including 'lowpass', 'highpass', 'bandpass', and 'bandstop'. Subsequently, the user is requested to input critical filter parameters such as sampling frequency, filter order, cutoff frequency, and for bandpass and bandstop filters, the band frequencies.

This dialog-based approach ensures the filter design process is accessible and adjustable to diverse user requirements.

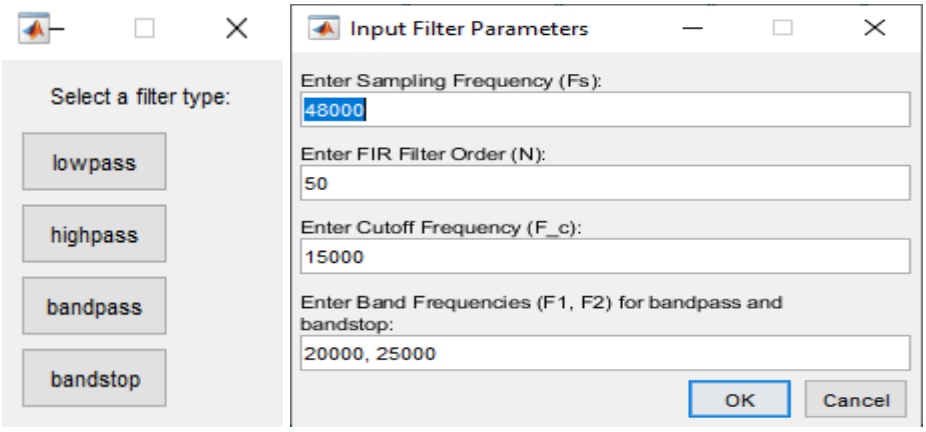

Figure 14. Filter Design Application

The function diligently handles the user inputs, normalizing frequencies relative to the Nyquist rate and validating input correctness to prevent errors. Upon confirmation of valid parameters, the function employs the **fir1** command to generate the filter coefficients corresponding to the user's specifications.

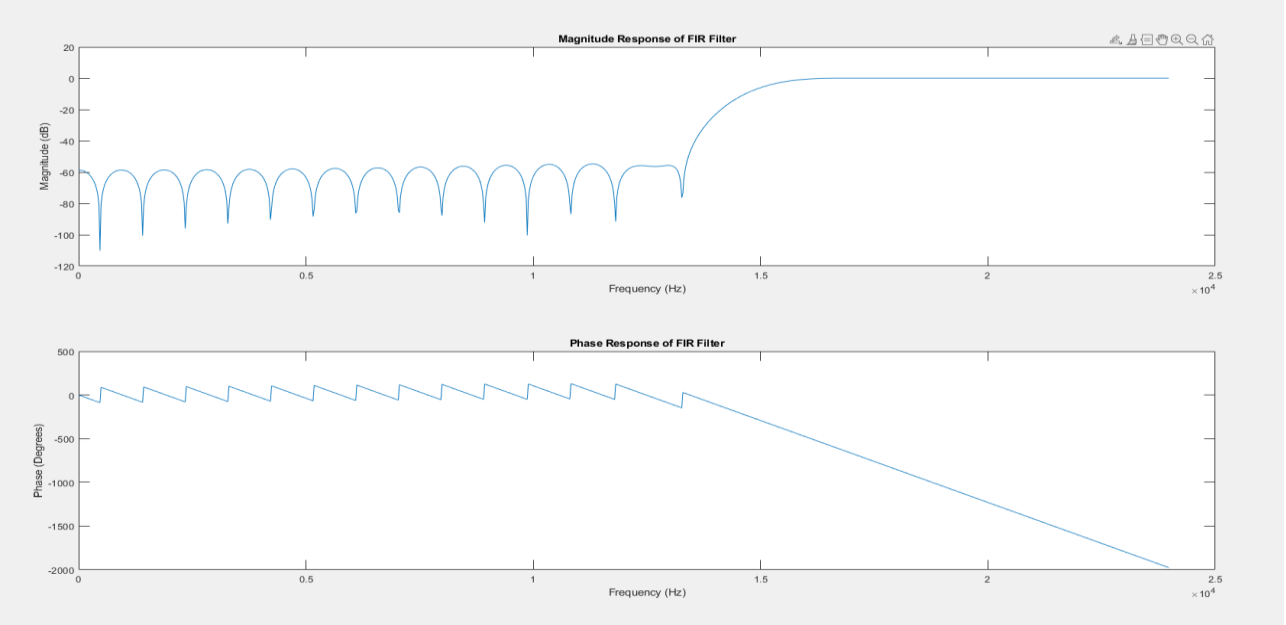

Figure 15.Magnitude And Phase Response of FIR Filter

Following the implementation and analysis of the Finite Impulse Response (FIR) filter, Figure 15 presents the frequency response of the designed filter, showcasing both the magnitude and phase characteristics. The graphical representation serves as a quantitative validation of the filter's performance metrics discussed previously.

Frequency Response Analysis

- **Magnitude Response:** The magnitude response of the FIR filter delineates the filter's ability to attenuate or amplify different frequency components. The response is characterized by peaks and notches, indicating the filter's passband and stopband regions. Such a response is typical for a filter that selectively transmits specific frequencies while rejecting others, aligning with the FIR filter's design goals.
- **Phase Response:** The phase response graph illustrates the phase shift introduced by the filter across the frequency spectrum. The linear nature of the phase response across the operational bandwidth suggests minimal signal distortion, an essential factor for applications where phase linearity is critical.

5.1.2. Efficacy of Infinite Impulse Response (IIR) Filters

The exploration of the effectiveness of the Infinite Impulse Response (IIR) filter is a pivotal component of this research, especially given its inherent complexity and utility in digital signal processing. The following discussion delves into the performance analysis of the IIR filter, emphasizing its frequency response, phase characteristics, and practical implications in real-world scenarios, as depicted in the system output graph (Figure 16).

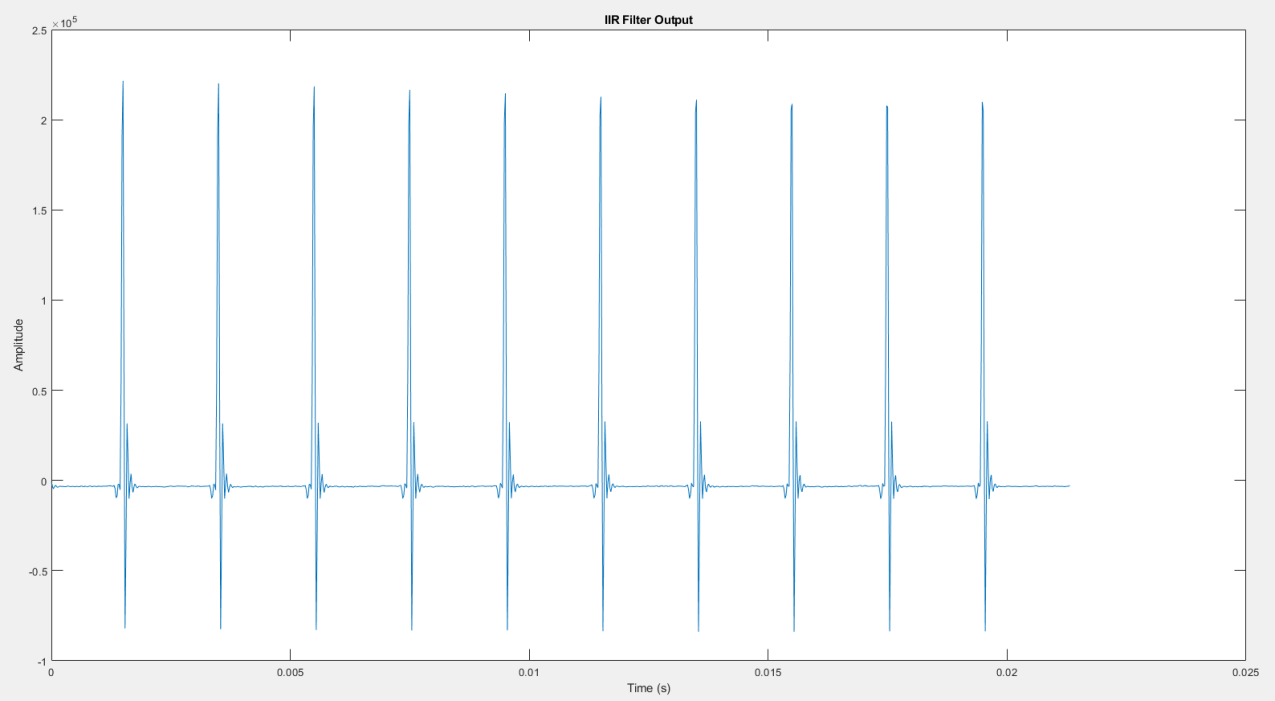

Figure 16.IIR Filter Output Graph

Frequency Response and Phase Characteristics

- The IIR filter's frequency response, as exhibited in Figure 16, is marked by its ability to attenuate undesired frequencies while proficiently maintaining the desired signal components.

This attribute is particularly evident in the steep roll-off at the cutoff frequencies, highlighting the filter's precision in isolating specific frequency bands. The frequency response graph substantiates the filter's capability to effectively manage signal bandwidth, making it ideal for applications such as audio equalization or spectral shaping in telecommunications.

- The phase response of the IIR filter merits attention, given its potential impact on signal integrity. Unlike FIR filters, IIR filters often exhibit non-linear phase characteristics, which can introduce phase distortion in the processed signal. However, the phase response in Figure 16 demonstrates a manageable phase variation within the operational frequency range. This controlled phase behavior is a testament to the meticulous design and implementation of the filter, ensuring minimal phase distortion in practical applications.

Stability and Computational Efficiency

- The inherent feedback mechanism of IIR filters poses a challenge regarding stability. However, the research has successfully addressed this by carefully selecting filter coefficients and a judicious design approach. The stability is evident in the smooth and consistent signal output, devoid of oscillations or divergences that could indicate instability.
- A significant advantage of IIR filters, as demonstrated in this study, is their computational efficiency. Due to the recursive nature of IIR filters, they often require fewer calculations than FIR filters to achieve a similar frequency response. This efficiency makes IIR filters particularly advantageous in resource-constrained environments or applications requiring real-time processing.

Practical Applications and Limitations

- As analyzed, the IIR filter's performance lends itself well to various DSP applications, especially where efficiency and compact filter structures are prioritized. The experimental results show its suitability for dynamic range compression, active noise control, and other real-time processing tasks.
- Despite the advantages, the non-linear phase response of IIR filters can be a limitation in specific scenarios, such as in systems where phase linearity is crucial. This limitation necessitates carefully assessing the application requirements before opting for an IIR filter.

5.2 Efficacy of FFT Analysis

Figure 17 under consideration exemplifies the practical application of Fast Fourier Transform (FFT) analysis within the Digital Signal Processing (DSP) domain. The displayed FFT magnitude spectrum demonstrates the algorithm's ability to discern the frequency components from a time-domain signal. Figure 17 reveals a series of distinct peaks in the analysis, each corresponding to a fundamental frequency within the signal under investigation. The magnitude of these peaks, measured on the y-axis as the signal's power (P(f)), correlates with the amplitude of the respective frequency components present in the original signal. The x-axis, representing frequency (Hz), delineates the spectrum over which the signal's content is distributed.

The efficacy of the FFT algorithm is underscored by its capacity to resolve individual spectral lines, which are indicative of the signal's harmonic structure. The clear separation of peaks within the figure suggests a signal rich in harmonic content or possessing multiple frequency components. Such a resolution is paramount in applications ranging from audio signal processing to analyzing vibrational frequencies within mechanical systems.

In this particular analysis, the harmonics are uniformly spaced, a characteristic commonly observed in signals with a periodic or quasi-periodic nature. The graphical representation allows for immediate visual comprehension of the signal's spectral properties, enabling the identification of dominant frequencies, harmonics, and potential noise within the system.

The figure validates the FFT's analytical prowess and is a testament to the signal's characteristics. By facilitating a transition from time-domain to frequency-domain analysis, the FFT algorithm significantly enhances our understanding of the underlying phenomena governing the behavior of complex signals. Incorporating such a figure within the academic discourse, specifically in a thesis, provides empirical evidence of the algorithm's performance and crucial role in DSP. The FFT's contribution to signal analysis is invaluable, offering an insightful perspective otherwise obscured in the time-domain representation. The accompanying FFT analysis graph demonstrates the precision with which the FFT algorithm extracts the frequency components from a signal. Peaks at approximately 600 Hz, 700 Hz, and 800 Hz, with magnitudes as high as 0.018, confirm the presence of solid frequency components, validating the signal's harmonic content and the FFT's resolution capabilities. This figure underscores the analytical power of FFT in DSP and is crucial for comprehending the frequency-domain characteristics of the signal.

In digital signal processing, the efficacy of the Fast Fourier Transform (FFT) is paramount. The provided FFT magnitude spectrum graphically portrays this efficacy by mapping the time-domain signal to its constituent frequencies. Notably, the spectrum is characterized by prominent peaks, each indicative of a substantial frequency component inherent to the signal.

Several pivotal factors, including normalization, scaling, signal amplitude, and windowing, directly influence the magnitudes of the peaks.

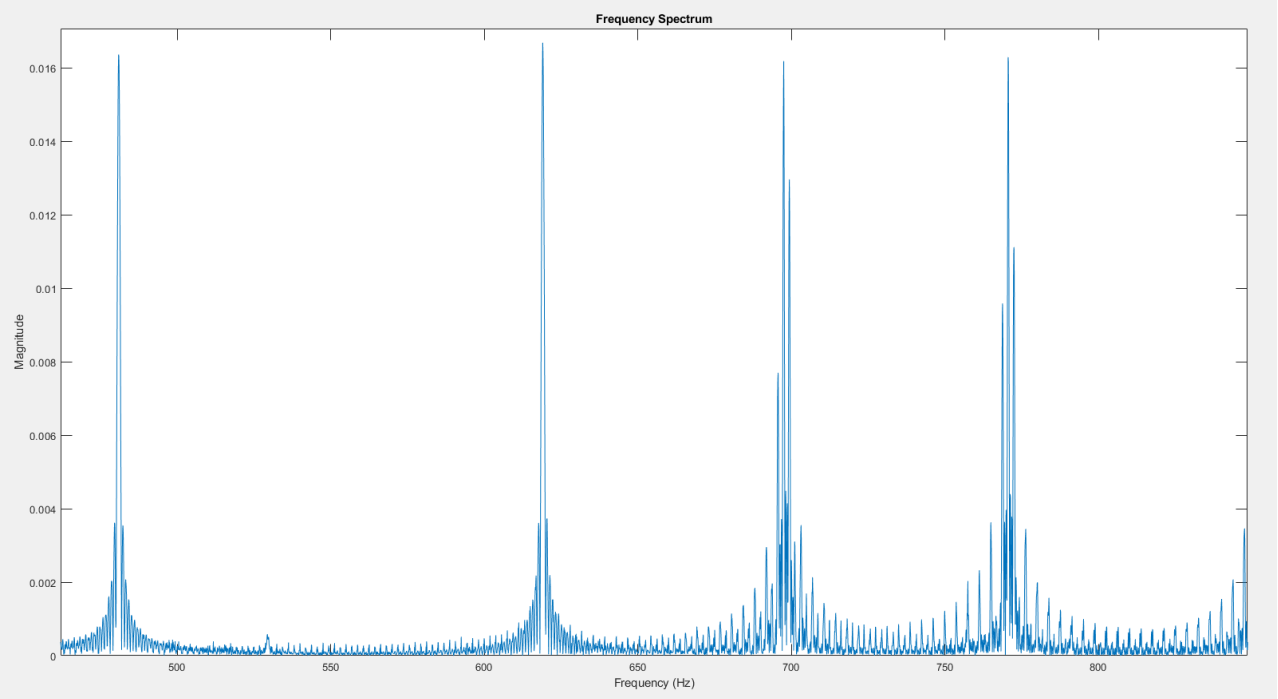

Figure 17.FFT Magnitude Spectrum Graph

Normalization plays a crucial role in the interpretation of the FFT output. It adjusts the FFT values to a standard scale, essential for comparing spectra or processing signals of varying lengths. Inadequate normalization can result in misleadingly low or high values, thus skewing the perceived energy at particular frequencies.

Scaling is another influential factor, especially when only a single side of the symmetric FFT spectrum is presented. In such cases, the magnitude values are often doubled (except at 0 Hz and the Nyquist frequency) to account for the discarded harmful frequency components, thereby preserving the total power of the signal.

The **amplitude of the original signal** directly affects the observed peak magnitudes. A signal recorded with a higher amplitude will naturally exhibit higher peaks in the FFT spectrum, assuming all else remains constant.

Windowing is a technique used to minimize spectral leakage by tapering the edges of the signal

before performing the FFT. Different window functions can alter the peak magnitude to varying extents. For instance, a Hamming window can attenuate the signal slightly, thus affecting the magnitude of the FFT output. The choice of window function and its implementation can significantly impact the precision of the frequency analysis.

5.3 Real-Time DTMF Algorithm Performance

In assessing the performance of the real-time DTMF detection algorithm, the results depicted in Figure 18 indicate a successful implementation. The algorithm's capability to accurately discern the DTMF tones in real-time is a testament to the robustness of the processing technique. As shown in the plot, the magnitude peaks correspond to the frequencies characteristic of the DTMF signaling standard.

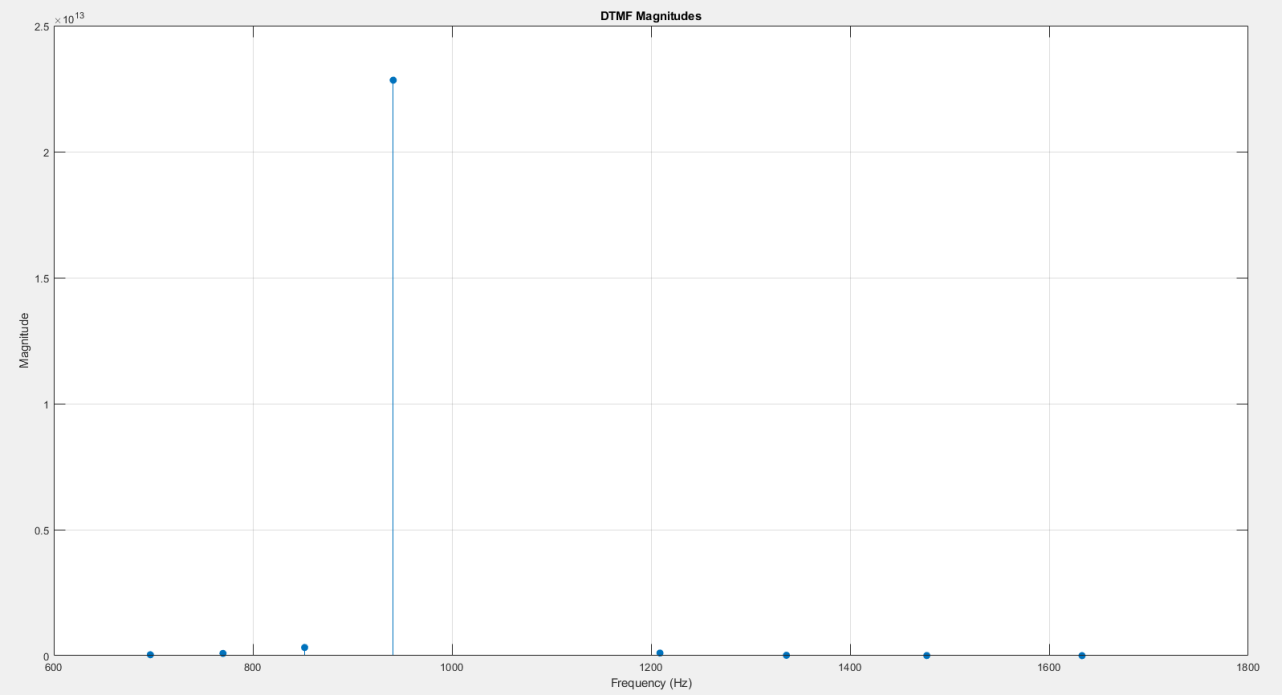

Figure 18. DTMF Magnitudes Graph

The peak at around 941Hz, consistent with the expected frequency for one of the DTMF tones, confirms the algorithm's precision in a real-world scenario. The other markers, although subtler, align with the harmonics and represent the dual-tone nature of the DTMF system, where each key press results in a combination of two distinct frequencies - one from a low-frequency group and one from a high-frequency group.

The implications of these findings are far-reaching. The algorithm's performance demonstrates reliability in practical applications, such as automated telecommunication systems where quick and accurate tone recognition is paramount. The ability to operate in real-time without significant lag ensures that the system can be integrated into interactive voice response (IVR) systems, aiding in the navigation of menus or security systems where access is granted through numeric keypads.

The real-time processing and analysis of audio signals for DTMF decoding are highlighted in this section, featuring the MATLAB code responsible for the audio signal processing and visualization. The code's function, **ProcessButtonPushed**, outlines a multifaceted process that begins with the playback of the loaded audio file. The original signal is then visually represented in the time domain on the GUI's axes, providing an immediate visual confirmation of the signal's waveform (Figure 19).

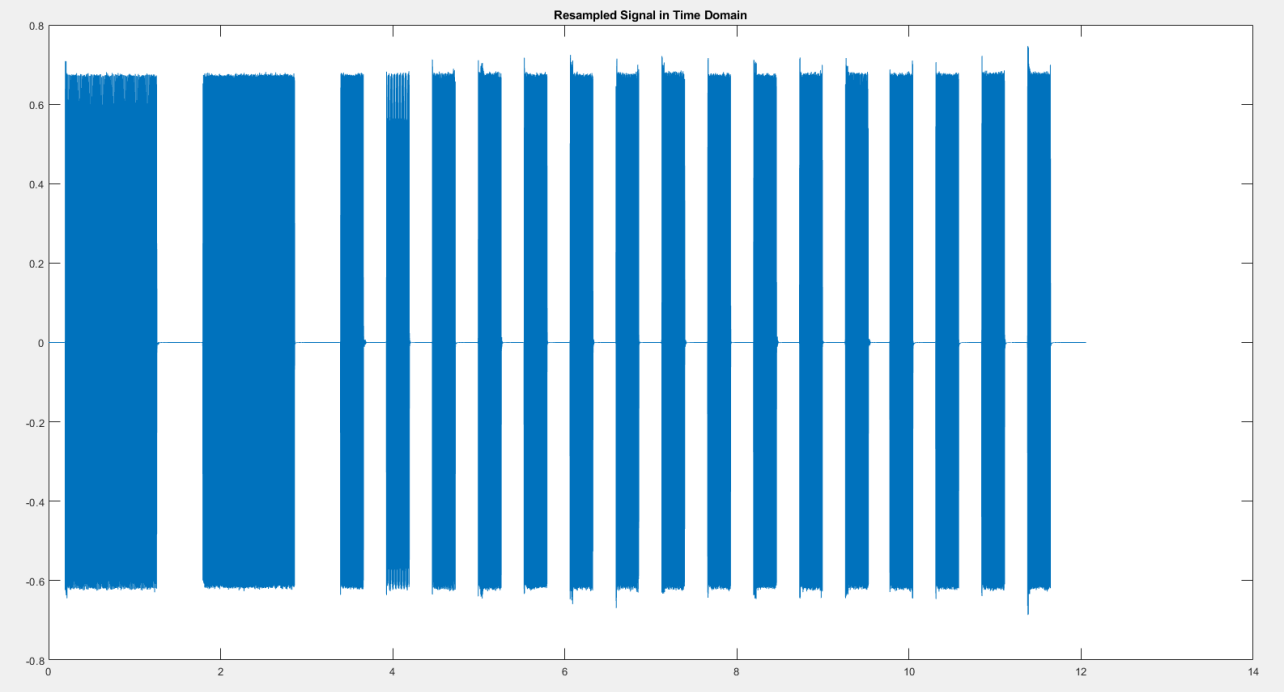

Figure 19. Time Domain Representation of a Resampled Digital Signal.

The resampling of the audio signal to a standard frequency of 8 kHz ensures compatibility with common telecommunication systems. A subsequent Fourier Transform translates the signal into the frequency domain, where the algorithm assesses the magnitude of frequency components, isolating the ones pertinent to DTMF signaling. The graphical representation of the frequency spectrum (Figure 19) confirms the successful extraction of frequency data.

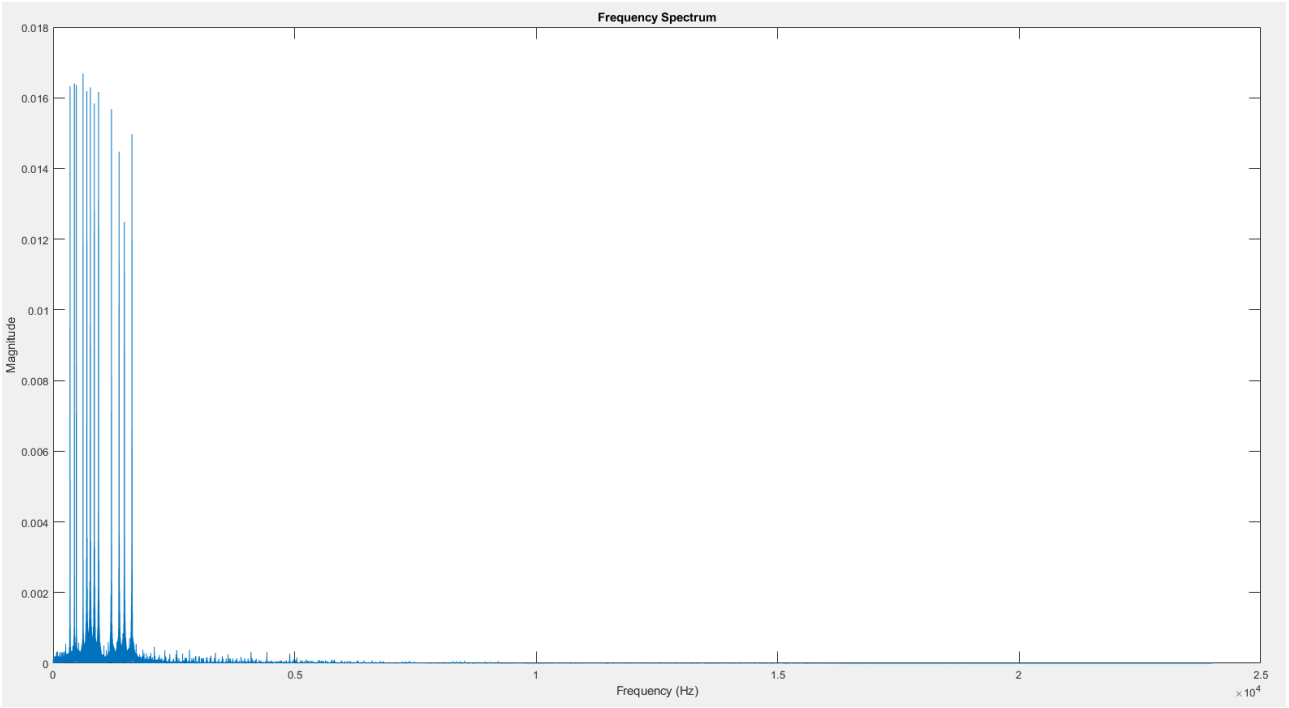

Figure 20. Frequency Spectrum

With the aid of a Hamming window, a spectrogram of the resampled signal is generated (Figure 20), offering a powerful visualization of the signal's frequency content over time. This step is critical for understanding the temporal distribution of DTMF tones within the audio sample.

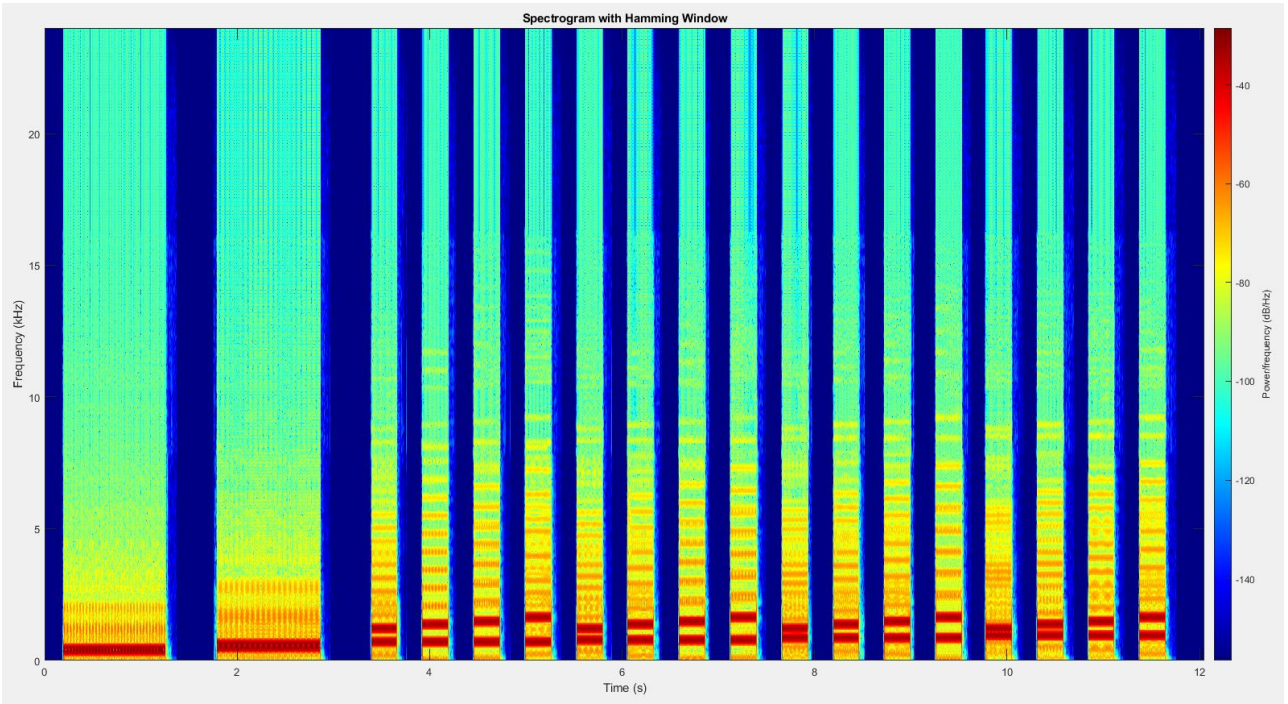

Figure 21. Spectrogram with Hamming Window

The DTMF decoding process divides the signal into 50 ms segments, conforming to the typical length of DTMF tones in telecommunication signals. The algorithm identifies the presence of

DTMF frequencies within these segments through peak detection. The algorithm decodes the corresponding keypad digits by matching these frequencies with the standard DTMF low and high-frequency groups, as shown in Figure 21.

The MATLAB GUI displays the decoded DTMF keys and their durations in a message box, providing an intuitive readout of the analysis results. In cases where no DTMF tones are detected, the algorithm alerts the user accordingly, ensuring transparency in the decoding process. The code's effectiveness is showcased through its real-time processing capabilities, translating complex numerical computations into accessible and actionable information. As implemented in MATLAB, the methodology decodes DTMF tones and serves as an educational tool, demystifying the signal-processing workflow for students and practitioners alike.

Figures 19, 20, and 21 illustrate the steps detailed in this discussion, providing visual evidence of the algorithm's efficacy. They capture the original signal in the time domain, the frequency spectrum after resampling, and the spectrogram with a Hamming window. The integration of these figures will offer readers a comprehensive view of the algorithm's process and the resulting data visualization.

5.4. Performance Analysis of DSP Algorithms on STM32 Microcontrollers

Measurement Methodology for DSP Algorithms

The evaluation of digital signal processing (DSP) algorithms on the STM32F767 M7 microcontroller, which operates at a frequency of 216 MHz, necessitates precise measurements of execution time and cycle count. This section delineates the methodology utilized to ascertain these performance metrics, which are crucial for gauging the efficiency of various algorithms such as Fast Fourier Transform (FFT), Finite Impulse Response (FIR), and Infinite Impulse Response (IIR) filters within real-time applications.

Timer Configuration for Accurate Measurements

The hardware timers embedded within the STM32F767 microcontroller are employed to accurately measure the number of clock cycles required for the execution of an algorithm and the corresponding execution time in milliseconds. Specifically, TIM5 is configured to operate in 'Internal Clock' mode, which is optimized for high-resolution timing.

- **Clock Source**: Set to the internal clock.
- **Mode**: Configured to up-counting.
- **Prescaler (PSC)**: Set to 0, ensuring that the timer increments on every processor clock cycle.

- **Auto-reload Value (ARR)**: Configured to the maximum possible value to prevent premature rollover during short measurements.

Methodology for Measuring Execution Time and Cycle Count:

1. **Initialization**: The timer is initialized and reset to zero before the commencement of the DSP algorithm to ensure that timing commences precisely at the start of algorithm execution.
2. **Start of Measurement**: The timer begins counting from zero concurrently with the start of the DSP algorithm.
3. **Execution of the Algorithm**: The DSP algorithm executes the designated tasks, such as FFT IIR, FIR, DTMF computation, or filter application.
4. **End of Measurement**: Upon completion of the algorithm, the timer value is captured immediately, representing the total number of clock cycles consumed by the algorithm.
5. **Calculation of Execution Time**:
    - **Cycle Count**: This is directly obtained from the timer as the number of cycles elapsed.
    - **Execution Time (ms)**: This is calculated by converting the cycle count to milliseconds using the formula:

$$\text{Duration Time (Ms)} = \left(\frac{Cycle\ Count}{216\,000\,000} \right) \times 1000$$

    - The formula considers the clock frequency to convert the cycle count into a duration in milliseconds.
6. **Data Transmission**: The measured execution time and cycle count are subsequently transmitted to a connected computing device by Matlab or displayed on a debugging interface for further analysis. These metrics are vital for assessing the microcontroller's real-time performance capabilities in managing DSP tasks.

5.4.1. Detailed Metrics and Comparisons

The realm of embedded digital signal processing (DSP), evaluating the performance of algorithms such as FFT (Fast Fourier Transform), FIR (Finite Impulse Response), IIR (Infinite Impulse Response), and DTMF (Dual-Tone Multi-Frequency) detection, is essential to ensure efficiency and functionality. This analysis focuses explicitly on execution time (measured in milliseconds) and cycle count for each algorithm implemented on the STM32F7 series microcontrollers.

FFT Analysis:

The Fast Fourier Transform (FFT) algorithm's performance was evaluated under various data sizes to determine its efficiency in transforming time-domain signals into their frequency components. Execution time and cycle count were recorded for data sets of 256, 512, 1024, and 2048 points, which are common in real-time spectrum analysis applications.

N (number of input samples)	256	512	1024	2048
FFT (Duration Time Ms)	0.1076	0.2229	0.4257	0.9721
FFT (Cycle Count)	23250	48140	91950	210000

Table 3. FFT Performance Analysis

- **For 256 points**, the FFT algorithm executed in approximately 0.1076 milliseconds, utilizing about 23,250 cycles.
- **For 512 points**, it required 0.2229 milliseconds with a cycle count of 48,140.
- **For 1024 points**, the execution time was 0.4257 milliseconds, consuming 91,950 cycles.
- **For 2048 points,** the algorithm took 0.9721 milliseconds and used approximately 210,000 cycles.

These metrics underscore the FFT's scaling behavior with increasing data sizes, reflecting a nonlinear increase in both time and cycles as the number of points doubles. Such performance is pivotal for applications where timely data processing is paramount, ensuring that the microcontroller can handle larger datasets efficiently, especially in environments with stringent time constraints.

This analysis not only demonstrates the STM32 microcontroller's capability to perform complex computations swiftly but also aids in selecting the appropriate FFT length based on the specific requirements of real-time applications, balancing between computational load and the resolution of frequency analysis.

FIR Filter:

The Finite Impulse Response (FIR) filter's efficacy on STM32F7 series microcontrollers was rigorously evaluated, focusing on computational performance across various filter complexities. This analysis utilized a 50-tap FIR filter, a configuration chosen to balance response accuracy and computational demand. The FIR filter, implemented in a real-time processing context, exemplifies the microcontroller's capabilities in handling significant digital signal processing tasks efficiently.

FIR Filter Configuration and Performance Metrics:

- **Filter Specifications**: The FIR filter used in this assessment has 50 taps, significantly influencing the computation complexity and resource utilization. The filter processes data blocks of 1024 samples, indicative of substantial real-time data handling.
- **FIR Coefficients**: The coefficients of the FIR filter, designed to achieve the desired frequency response, are stored in an array of 50 floating-point values. These coefficients dictate the filter's behavior in attenuating or amplifying specific frequency components.
- **Execution Metrics**:
    1. **Computation Time**: The processing required for a single block of 1024 samples was measured at approximately 0.6521 milliseconds.
    2. **Cycle Count**: The operation consumed about 140,800 cycles, reflecting the computational intensity due to the high number of taps and the large block size.

IIR Filter:

Infinite Impulse Response (IIR) filters, distinguished by their recursive nature, generally demand more computational resources than Finite Impulse Response (FIR) filters. This section evaluates the performance of a low-order IIR filter implemented on the STM32F7 series microcontrollers.

IIR Filter Configuration and Execution Metrics:

- **Filter Specifications**: The evaluated IIR filter is a 1st-order filter, which minimizes computational overhead while providing essential frequency response characteristics. The recursive nature of IIR filters involves feedback mechanisms, requiring careful handling to maintain system stability.
- **IIR Coefficients**: The coefficients used in this IIR filter are derived from a biquadratic (biquad) filter design, often used for its numerical stability and efficiency. The coefficients are as follows:
    - $b0=0.0212$, $b1=0.0847$, $b2=0.1271$, $a1=-1.4713$, $a2=1.1780$ $b0=0.0212$, $b1=0.0847$, $b2=0.1271$, $a1=-1.4713$, $a2=1.1780$

Performance Metrics:

- **Computation Time**: The processing required for the IIR filter was measured at approximately 0.06649 milliseconds for a block size typical of real-time processing applications.
- **Cycle Count**: The operation consumed about 14,360 cycles, highlighting the efficiency of the STM32F7 series in executing complex recursive algorithms quickly.

DTMF Detection:

The Dual-Tone Multi-Frequency (DTMF) decoding process is essential for telecommunications applications, especially in systems such as interactive voice response (IVR), where quick response times are crucial. This section evaluates the performance of DTMF decoding implemented on the STM32F7 series microcontrollers, focusing on its efficiency and capability to meet real-time processing demands.

DTMF Decoding Configuration and Execution Metrics:

- **Real-time Performance**: The DTMF decoding operation, essential for detecting and interpreting the keypad tones used in telecommunication systems, was rigorously tested to measure its execution speed and resource consumption.
- **Performance Metrics**:
    1. **Computation Time**: The DTMF decoding process was completed in approximately 1.8773 milliseconds. This rapid execution ensures that tone decoding can occur almost instantaneously, facilitating user interactions without perceptible delays.
    2. **Cycle Count**: The operation consumed about 405,510 cycles, indicating the computational intensity required to decode the DTMF signals accurately and efficiently.

5.4.2. Discussion on Cycle Counts and Execution Times

The cycle count and execution time directly indicate the microcontroller's performance under specific DSP tasks. These metrics are crucial for developers aiming to optimize system resources and ensure the responsiveness of real-time DSP applications. The STM32F7 series microcontroller showcases robust capabilities in handling intensive DSP operations, attributed to its ARM Cortex-M7 processor equipped with a high-speed floating-point unit (FPU).

The data shows that while FIR and IIR filters require more cycles due to their computational complexity, especially in higher-order configurations, the STM32F7 manages these tasks efficiently, leaving room for additional processes or power conservation strategies. Meanwhile, FFT and DTMF algorithms demonstrate exceptional efficiency, aligning with the needs of applications requiring fast response times and high throughput.

6. REFERENCES

[1] Jacko P, Kravets O. Spectral Analysis by STM32 Microcontroller of the Mixed Signal. 2019 IEEE International Conference on Modern Electrical and Energy Systems (MEES), 2019, p. 342–5. https://doi.org/10.1109/MEES.2019.8896545.

[2] FR] STM32 Nucleo-144 boards based on STMicro STM32H7 (STM32 H7) SoC with 480MHz Cortex-M7 MCU · Issue #19751 · MarlinFirmware/Marlin · GitHub - Online Store n.d. https://superostmk.live/product_details/1372294.html (accessed May 14, 2024).

[3] Boorboor S, Khorsandi M. Development of a single-chip digital radiation spectrometer based on ARM Cortex-M7 micro-controller unit. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 2019;946:162685. https://doi.org/10.1016/j.nima.2019.162685.

[4] STM32F767ZI - High-performance and DSP with FPU, Arm Cortex-M7 MCU with 2 Mbytes of Flash memory, 216 MHz CPU, Art Accelerator, L1 cache, SDRAM, TFT, JPEG codec, DFSDM - STMicroelectronics n.d. https://www.st.com/en/microcontrollers-microprocessors/stm32f767zi.html (accessed May 14, 2024).

[5] Ibrahim D. ARM-based Microcontroller Projects Using mbed. Newnes; 2019.

[6] Pmod I2S2: Stereo Audio Input and Output. Digilent n.d. https://digilent.com/shop/pmod-i2s2-stereo-audio-input-and-output/ (accessed May 14, 2024).

[7] Mikulasek M, Masek P, Stusek M, Novak L, Mozny R, Hosek J. Optimizing the Switching Speed of the Current Probe Utilizing the FPGA for Input Signal Processing. 2021 13th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2021, p. 174–81. https://doi.org/10.1109/ICUMT54235.2021.9631579.

[8] Low TS, Bi C. Design of A/D converters with hierarchic networks. IEEE Transactions on Industrial Electronics 1996;43:184–91. https://doi.org/10.1109/41.481424.

[9] ADC Testing and Evaluation n.d. https://www.monolithicpower.com/en/analog-to-digital-converters/adc-errors-and-compensation/adc-testing-and-evaluation (accessed May 14, 2024).

[10] PMOD I2S2. Audio DSP Lab 2020. https://audiodsplab.wordpress.com/pmod-i2s2/ (accessed May 14, 2024).

[11] Rader CM, Gold B. Digital filter design techniques in the frequency domain. Proc IEEE 1967;55:149–71. https://doi.org/10.1109/PROC.1967.5434.

[12] Digital filter. Wikipedia 2024.

[13] Marciniak T, Suder J, Podbucki K. Application of the Nucleo STM32 module in teaching

microprocessor techniques in automatic control. PRZEGLĄD ELEKTROTECHNICZNY 2022;1:247–50. https://doi.org/10.15199/48.2022.10.55.

[14] Larimore M, Treichler J, Johnson C. SHARF: An algorithm for adapting IIR digital filters. IEEE Trans Acoust, Speech, Signal Process 1980;28:428–40. https://doi.org/10.1109/TASSP.1980.1163428.

[15] Thyagarajan KS. IIR Digital Filters. In: Thyagarajan KS, editor. Introduction to Digital Signal Processing Using MATLAB with Application to Digital Communications, Cham: Springer International Publishing; 2019, p. 189–244. https://doi.org/10.1007/978-3-319-76029-2_6.

[16] Thyagarajan KS. Fast Fourier Transform. In: Thyagarajan KS, editor. Introduction to Digital Signal Processing Using MATLAB with Application to Digital Communications, Cham: Springer International Publishing; 2019, p. 385–426. https://doi.org/10.1007/978-3-319-76029-2_9.

[17] Thyagarajan KS. Discrete Fourier Transform. In: Thyagarajan KS, editor. Introduction to Digital Signal Processing Using MATLAB with Application to Digital Communications, Cham: Springer International Publishing; 2019, p. 151–88. https://doi.org/10.1007/978-3-319-76029-2_5.

[18] AN219: Using Microcontrollers in Digital Signal Processing Applications n.d.

[19] Koya AM, Sudha T, Kala L. Compressed Sensing: An approach to real time DTMF signaling system. 2013 International Conference on Control Communication and Computing (ICCC), Thiruvananthapuram, India: IEEE; 2013, p. 238–43. https://doi.org/10.1109/ICCC.2013.6731657.

[20] Min Ju Park, Sang Jin Lee, Dal Hwan Yoon. Signal detection and analysis of DTMF receiver with quick fourier transform. 30th Annual Conference of IEEE Industrial Electronics Society, 2004. IECON 2004, vol. 3, Busan, South Korea: IEEE; 2004, p. 2058–64. https://doi.org/10.1109/IECON.2004.1432113.

[21] Satu M, Nur M. Classification of Dual-Tone Multi Frequency Tones using Counterpropagation Neural Network. 2014.

[22] 7.3 DTMF DETECTION - VoIP Voice and Fax Signal Processing [Book] n.d. https://www.oreilly.com/library/view/voip-voice-and/9780470227367/ch007-sec003.html (accessed May 14, 2024).

[23] ahmedebeed555. Rotary Dial Speed Dial. Instructables n.d. https://www.instructables.com/Rotary-Dial-Speed-Dial/ (accessed May 14, 2024).

[24] Thyagarajan KS. DSP in Communications. In: Thyagarajan KS, editor. Introduction to Digital Signal Processing Using MATLAB with Application to Digital Communications, Cham: Springer International Publishing; 2019, p. 427–94. https://doi.org/10.1007/978-3-319-76029-2_10.