5-2023

Online Fourier Analysis of Time-Varying Signals in a Real-Time Embedded Environment

Ty Freeman
tkf1604@rit.edu

Online Fourier Analysis of Time-Varying Signals in a Real-Time Embedded Environment

by

Ty Freeman

Graduate Paper

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Approved by:

_____

Mr. Mark A. Indovina, Senior Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

_____

Dr. Ferat Sahin, Professor
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York

May, 2023

Dedication

I dedicate this work to my closest engineering friends: Ben Bellantoni, Sophie Buckwalter, Alissa Mann, and Cameron Villone, who helped keep me honest and on track. In addition, I must thank my mother, father, brother, and sister, who always knew I would make it here, and my partner Carly Strohl for keeping me as sane as possible. Finally, I would be remiss if I did not especially thank Mark Indovina, Jason Hoople, and the other professors who have supported and challenged me in earning this degree over the last five years.

Ty Freeman

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Ty Freeman

May, 2023

Acknowledgements

I would like to thank my advisor Professor Mark A. Indovina for his support, guidance, feedback, and encouragement from the very start of my academic career as an electrical engineer to the completion of this research project.

Ty Freeman

Abstract

A software-based, constant-flow implementation of the radix-2 Cooley-Tukey fast Fourier transform (FFT) algorithm is presented in this paper. The program is built within a real-time embedded environment running FreeRTOS. The system is used for the online frequency analysis of time-varying, one-dimensional signals of an arbitrary length. The system's design is validated through testing to produce results accurately on signals within specific boundaries. The proposed system has a maximum transform size of 256 samples due to the memory limitations on the development board. The hardware in use is an STM32L476 Nucleo development board which simulates the lightweight, low power, and limited resource design of IoT (internet of things) processors in the modern world. Memory is the primary constraint of the program, as the number of samples dictates the functional bandwidth of signals that can be analyzed. The system performs several validation tests that prove its effectiveness within the discovered bounds and verify the possibility of project expansion for a more robust implementation in future iterations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems are utilized in almost every technical industry and many facets of everyday life around the world today. The rapid development and adoption of the Internet of Things (IoT) has surrounded us with more wirelessly communicating systems than ever before; objects like toothbrushes, mirrors, and fridges are now capable of recording/analyzing user statistics and wirelessly communicating this data to other IoT devices. Light bulbs can now receive over-the-air updates or be synchronized with other light bulbs in the same room wirelessly; their colors and schedules are infinitely reprogrammable, and some even sport the ability to act as Bluetooth speakers. There are very few limitations to the amount and type of information that can be remotely accessed, recorded, or manipulated as technology has evolved to be smaller, cheaper, and more precise.

The Fast Fourier Transform (FFT) has been central in this wireless communication and technology development. Some form of the algorithm can be found in almost every embedded system requiring frequency analysis. The importance of the FFT comes from the savings it creates in complex computations and time when transforming a signal into the frequency domain. For example, a standard Discrete Fourier Transform can perform a single transform

of length N in N^2 computations while the FFT performs the same transform in $Nlog(N)$ computations, so the savings increase with N. These savings, and the many uses of the Fourier Transform, have made the FFT one of the most important algorithms of the last century, if not ever. For example, mathematicians can solve complex partial differential equations in record times; images can be compressed to save memory space in a digital computer; and videos are compressed, transmitted, received, and uncompressed fast enough to stream from a phone or TV seamlessly, all using the FFT.

This paper presents a software implementation of the FFT algorithm in a real-time environment that simultaneously records samples and processes them for a continuous transformation of the input signal. The microcontroller is a NUCLEO-STML476 running FreeRTOS, an open-source operating system. The algorithm is a one-dimensional radix-2 implementation of the original Cooley-Tukey Algorithm [1]. This program can calculate FFTs of large sample sizes in smaller time windows by sampling smaller sections of a continuously filling buffer. In performing this windowed transform over long continuous signals, the frequency content can be exported along with a timescale based on the number of transforms performed and the known sampling frequency. Other output forms could include raw data with real and imaginary components or the signal's power spectral density.

1.1 Research Goals

The goal of this research is the development of a real-time embedded sliding window FFT algorithm capable of continuous signal analysis over signals multiple times the length of the transform buffer. The research focused on the mathematical basis of the FFT algorithm and techniques for performing windowed, short time Fourier Transforms (STFT). Research was also performed to manipulate the algorithm and techniques used to conserve memory space

and computation time on the microcontroller.

1.2 Contributions

The significant contributions to the projected are listed below.:

1. Development of a software based radix-2 FFT.
2. The implementation of the FFT into a real-time embedded system.
3. Development of variable sample sliding window FFT in real time system.
4. The tracking of operating statistics during signal analysis.
5. Comparable results to proven implementations from MATLAB.

1.3 Organization

The structure of the thesis is as follows:

- Chapter 2: Provides background information with references to the FFT algorithm and its use in signal analysis as well as a brief on real-time operating systems.
- Chapter 3: Explains the proposed program architecture and implementation.
- Chapter 4: Experimental results and procedures.
- Chapter 5: Project conclusions and directions for future work

Chapter 2

Bibliographical Research

The following chapter is a literature survey to provide background information on the major topics covered within this paper. The first few sections will discuss the history of the FFT, a derivation of the Cooley-Tukey FFT algorithm, and a brief of other implementations of the FFT. Following sections will discuss real-time operating systems and the differences between certain open-source versions as well as some of the research currently taking place in the field.

2.1 A brief history of the FFT

The history of the FFT started long before it was first published by James Cooley (1926-2016) and John Tukey (1915-2000) in 1965. Estimates made by Heideman et al. [2] in their investigation of the origins of the FFT place the original discovery of the algorithm in 1805, which is two years before even Jean Baptiste Joseph Fourier first published his theories on heat propagation [3]. Carl Friedrich Gauss (1777-1855) is credited with this initial discovery by building on and generalizing previous research on trigonometric series by mathematicians Alexis-Claude Clairaut (1713-1765) and Joseph Louis Lagrange (1736-1813). The former

mathematician, Clairaut, published a formula for a cosine-only series which is held to be the earliest discrete Fourier transform (DFT) representation. Lagrange then went on to define a sine-only series in a similar vane for interpolating the orbits and analyzing orbit mechanics of celestial bodies based on finite and periodic observations. Gauss' work extended beyond definitively odd or even trigonometric functions, generalizing instead to periodic functions in the form

$$f(x) = \sum_{k=0}^{m} a_k \cos(2\pi kx) + \sum_{k=0}^{m} b_k \sin(2\pi kx) \tag{2.1}$$

where $m = (N-1)/2$ if the sample size, N, is odd and $m = N/2$ is even. By dividing N into two subgroups such that $N = n_1 n_2$, Gauss first calculates the coefficients a_k and b_k for n_2 sets of n_1 samples and then again calculating the coefficients for n_1 sets of length n_2 which come from the n_2 sets of coefficients. Comparing the results of these coefficient calculations to the intermediate steps of the Cooley-Tukey algorithm, one can see that they are equivalent with even the same $N\log(N)$ computation complexity. However, Newtonian physics became a much more popular method for celestial observations, and thus Gauss left his algorithm unpublished in a series of treatises on interpolation. The work was posthumously published in [4], but his use of neo-Latin and strange notations such as the symbol \pi for N made the work hard to understand without involved translations.

In [1], Cooley and Tukey only refer to the work of Good in [5] during the development of their elegant algorithm. After the publication of [1], Rudnick [6] described a similar computer program with the same complexity as Cooley and Tukey's version based on the work of Danielson and Lanczos [7]. Cooley, Tukey, and Peter D. Welch began an investigation into the history of the FFT algorithm [8], where they determined that the work seen in [5] is not equivalent to the FFT algorithm that they proposed; it has since been classified as one of the

prime factor algorithms (PFA's) which is a different method for calculating the DFT of a signal. Another example of a PFA can be seen in work from Thomas [9], and so it is sometimes referred to as the Good-Thomas FFT. This transform only works with factors of N that are mutually prime, so it is less generalized than the algorithm presented in [1]. Cooley et al. [8] did not trace the FFT back to Gauss as this connection was made later by H. Goldstine [10] and then later verified in [2].

2.2 FFT Derivation

For the development of the FFT algorithm it will be easiest to begin with a modified DFT of size $N = 4$ to extract certain patterns which can be used to describe a generalized algorithm. The radix-2 Cooley-Tukey algorithm, which serves as the basis for the FFT presented in this paper, can be performed for any sample size of $N = 2^m$ or, by zero padding the sample array, any arbitrary sample size.

2.2.1 DFT Derivation for $N = 4$

As the FFT is only an alternative method for calculating the DFT of a signal it would be beneficial to first define the standard DFT as the starting point in the derivation of the FFT algorithm,

$$X(n) = \sum_{k=0}^{N-1} x(k)W^{nk}, \qquad n = 0, 1, \cdots, N-1 \tag{2.2}$$

$$e^{\mathrm{j}\theta} = \cos(\theta) + \mathrm{j}\sin(\theta) \tag{2.3}$$

where W represents the N^{th} complex root of unity: $e^{\mathrm{j}2\pi/N}$. This complex term is also

known as the twiddle factor as coined in [1]. From Euler's formula (2.3), both sinusoidal components found in Gauss' original DFT equation (2.1) are accounted for with the single complex exponential. This standard DFT definition requires N^2 complex operations where an operation, as defined in [1], is a complex multiplication followed by a complex addition. By assuming $N = 4$, both k and n can be represented by two-bit binary numbers as suggested in [11]:

$$k = (k_1, k_0) = 00, 01, 10, 11$$
$$n = (n_1, n_0) = 00, 01, 10, 11$$

where k_1, k_0, n_1, n_0 can only be either 0 or 1. A mathematical representation of the real values of k and n can be defined as

$$k = 2k_1 + k_0 \quad n = 2n_1 + n_0 \tag{2.4}$$

Applying these assumptions and Eq. (2.4) to Eq. (2.2) it can be rewritten as a double summation over the terms of k in the form of

$$X(n_1, n_0) = \sum_{k_1=0}^{1} \sum_{k_0=0}^{1} x_0(k_1, k_0) W^{(2n_1+n_0)(2k_1+k_0)} \tag{2.5}$$

Where x_0 is the sampled signal and k represents the sample index. By the product rule of exponents where $a^{x+y} = a^x a^y$ the new twiddle factor in Eq. (2.5) can be simplified for terms $k_1$and k_0 to

$$W^{(2n_1+n_0)(2k_1+k_0)} = W^{(2n_1+n_0)(2k_1)}W^{(2n_1+n_0)(k_0)} = W^{(4n_1k_1)}W^{(2k_1n_0)}W^{(2n_1+n_0)(k_0)}$$

Notice that the highest powered term in equation above can be simplified to 1 since

$$[W^4]^{n_1k_1} = [e^{j2\pi 4/4}]^{n_1k_1} = 1^{n_1k_1} = 1$$

This reduction of the complex twiddle factor then forms the following equation:

$$X(n_1,n_0) = \sum_{k_1=0}^{1}\left[\sum_{k_0=0}^{1} x_0(k_1,k_0)W^{(2k_1n_0)}\right]W^{(2n_1+n_0)(k_0)} \tag{2.6}$$

and

$$x_1(n_0,k_0) = \sum_{k_0=0}^{1} x_0(k_1,k_0)W^{(2k_1n_0)} \tag{2.7}$$

Focusing only on Eq. (2.7), the innermost summation of Eq. (2.6), it should be noted that this portion is only dependent on terms n_0 and k_0. The following equations are created by enumerating Eq. (2.7). These summations are found in Tbl. 2.1 in a base ten representation. Notice that the first term in the summation will always be multiplied by unity sincek_1 is zero. In contrast, the second term in the summation is multiplied by some principal root of unity W^m where $m = 2k_1n_0$. Summation enumerations will be represented in base ten truth tables for readability, making it easier to visualize the underlying patterns that generalize the equation.

$$
\begin{aligned}
x_1(0,0) &= x_0(0,0)W^0 + x_0(1,0)W^0 \\
x_1(0,1) &= x_0(0,1)W^0 + x_0(1,1)W^0 \\
x_1(1,0) &= x_0(0,0)W^0 + x_0(1,0)W^2 \\
x_1(1,1) &= x_0(0,1)W^0 + x_0(1,1)W^2
\end{aligned} \tag{2.8}
$$

Table 2.1: Enumeration of Eq. (2.8) in Base 10 Truth Table Form

	$k_1 = 0$		$k_1 = 1$	
$x_1(n_0,k_0)$	$x_0(k_1,k_0)$	m	$x_0(k_1,k_0)$	m
0	0	0	2	0
1	1	0	3	0
2	0	0	2	2
3	1	0	3	2

$$
x_2(n_0,n_1) = \sum_{k_0=0}^{1} x_1(n_0,k_0)W^{(2n_1+n_0)(k_0)} \tag{2.9}
$$

This outer summation is now dependent on the values of $n_0$and $n_1$though the indexing of $x_2$is bit reversed which will become important later for accurately ordering the FFT output. For now, the enumeration of Eq. (2.9) is shown in Tbl. 2.2.

Table 2.2: Enumeration of Eq. (2.9) in Base 10 Truth Table Representation

	$k_0 = 0$		$k_0 = 1$	
$x_2(n_0, n_1)$	$x_1(n_0, k_0)$	m	$x_1(n_0, k_0)$	m
0	0	0	1	0
1	0	0	1	2
2	2	0	3	1
3	2	0	3	3

Both summations are now accounted for, and a calculation of the DFT is theoretically complete, though the bit reversal mentioned earlier means that $x_2(n) \neq X(n)$. So instead, a final step must be undertaken to reorder the calculated values to present the FFT output in the correct order accurately. Luckily the reordering only requires the bit reversal of the indices, as seen below.

$$X(0,0) = X(0) = x_2(0) = x_2(0,0)$$
$$X(0,1) = X(1) = x_2(2) = x_2(1,0)$$
$$X(1,0) = X(2) = x_2(1) = x_2(0,1)$$
$$X(1,1) = X(3) = x_2(3) = x_2(1,1)$$

Both summations required two complex multiplications and a complex addition for a single value of *x(n)*, compounding to N operations for a single frequency domain value *X(n)*. Therefore, calculating the total DFT of an N-sized signal would require N^2 operations. A few patterns, however, could be seen, which will lead to the impressive computation savings of the

FFT.

2.2.2 Pattern Extraction

2.2.2.1 Relating to the values of x_t

There is a repeating pattern seen in Tbl.2.1 where the x_0 indices can be divided into odd and even pairings. Several patterns are gleaned from this division into separate sets. In Tbl. 2.1 the even indices of $x_1$correspond to the even indices of x_0 and these even indices of x_0 remain the same between $x_1(0)$ and $x_1(2)$ the only difference between the summations of these two x_1 values is actually the twiddle factor scaling the secondary x_0 term. The same holds for the odd set of x_1 indices. Rearranging Tbl. 2.1 it can be seen that every $\frac{N}{2}$ indices repeat each other. The locations that share their input values are dual-node pairs [11]. In Tbl. 2.3 the twiddle factors scaling the primary x_0 term have been removed as they remain constant.

Table 2.3: Rearranged Tbl. 2.1 Without Primary m Values

	$k_1 = 0$	$k_1 = 1$	
$x_1(n_0, k_0)$	$x_0(k_1, k_0)$	$x_0(k_1, k_0)$	m
0	0	2	0
2	0	2	2
1	1	3	0
3	1	3	2

The index values of $x_0$within a given summation also have a constant difference of $\frac{N}{2}$ for both the odd and even sets.

Similar patterns can also be seen in Tbl. 2.2. The values of x_1, are repeated at different indices of x_2 though the difference factor which was $\frac{N}{2}$ in the expansion of the first summations has now reduced to $\frac{N}{4}$. These dual-node pairs are now adjacent to each other within Tbl. 2.2 and the difference of the $x_1$indices in individual summations holds the same difference factor as the distance between the dual-node pairs.

2.2.2.2 Relating to the Roots of Unity

Since it has been shown that half of each summation reuses the same x values one could start to see the redundancies of the general DFT calculation. Due to the differing complex scaling factor, W amongst these almost redundant operations though there is nothing yet suggesting any sort of short cuts that can be taken. However, thanks to the symmetrical nature of the roots of unity it can be proven that there exists a relationship such that

$$W^m = -W^{(\frac{N}{2})m} \tag{2.10}$$

. This can be proven for the case of $m = 2$ from Tbl. 2.1 and Eq. (2.8):

$$W^2 = e^{\mathrm{j}2\pi(2)/4} = e^{j\pi} = -1 = -W^0$$

Taking advantage of this relationship finally removes the DFT calculations of many of its redundancies. Observe that the twiddle factor scaling the secondary x term of the latter of the dual node pair always takes the value m such that $(\frac{N}{2})m_1 = m_2$ therefore, Eq. (2.10) applies to each dual-node pair. Tbl. 2.4 and Tbl. 2.5 replace all of the m_2 values using Eq. (2.10) and accounts for the new sign of W. From these tables, it can be seen that the dual-node pairs can now share complex scaling factor W as well as long as the change in sign is accounted for.

Table 2.4: Revision of Tbl. 2.3 using Eq. 2.10

	$k_1 = 0$	$k_1 = 1$		
$x_1(n_0, k_0)$	$x_0(k_1, k_0)$	$x_0(k_1, k_0)$	Sign of W	m
0	0	2	+	0
2	0	2	-	0
1	1	3	+	0
3	1	3	-	0

Table 2.5: Revision of Tbl. 2.2 using Eq. 2.10

	$k_0 = 0$	$k_0 = 1$		
$x_2(n_0, n_1)$	$x_1(n_0, k_0)$	$x_1(n_0, k_0)$	Sign of W	m
0	0	1	+	0
1	0	1	-	0
2	2	3	+	1
3	2	3	-	1

2.2.2.3 Putting the Patterns Together

The dual-node pairs have been defined, their distances are known, and the changes across summations can be accounted for. Using Eq. (2.10), the dual node pairs now share almost all the same values except that the latter term has a negated secondary x term. Using these

two patterns, calculating two x terms of the following summation requires a single complex multiplication and two complex additions. A general form of the calculation can be made by defining γ as the current summation or stage indexed from 0:

$$x_{t+1}(n) = x_t(n) + x_t(n+a)W^m \tag{2.11}$$

$$x_{t+1}(n+a) = x_t(n) - x_t(n+a)W^m \tag{2.12}$$

where $a = \frac{N}{(2^\gamma)}$. Notice that Eqns. (2.11) and (2.12) only differ in a single sign and thus require no extraneous computations other than a second addition to account for the sign. This set of equations encompass the elegance of the FFT algorithm and account for the computational savings that have made it so important.

2.2.3 FFT for $N = 8$

Starting again at Eq. (2.2), the values of n and k will be assigned to three bit binary representations as

$$k = (k_2, k_1, k_0) = 000, 001, 010, 011, 100, 101, 110, 111$$

$$n = (n_2, n_1, n_0) = 000, 001, 010, 011, 100, 101, 110, 111$$

In this binary representation the values of k and n can be redefined as

$$k = 4k_2 + 2k_1 + k_0 \quad n = 4n_2 + 2n_1 + n_0 \tag{2.13}$$

Using Eq. (2.13) the expansion of W^{nk}looks like

$$W^{(4n_2+2n_1+n_0)4k_2}W^{(4n_2+2n_1+n_0)2k_1}W^{(4n_2+2n_1+n_0)k_0}$$

$$\left[W^{16n_2k_2}W^{8n_1k_2}W^{8n_2k_1}\right]W^{4n_0k_2}W^{4n_1k_1}W^{2n_0k_1}W^{(4n_2+2n_1+n_0)k_0} \tag{2.14}$$

The bracketed terms of Eq. (2.14) are all equivalent to one and can be dropped. Accounting for these changes, Eq. (2.2) can be rewritten in the form

$$X(n_2,n_1,n_0) = \sum_{k_0=0}^{1}\sum_{k_2=0}^{1}\sum_{k_2=0}^{1} x_0(k_2,k_1,k_0)W^{4n_0k_2}W^{4n_1k_1}W^{2n_0k_1}W^{(4n_2+2n_1+n_0)k_0} \tag{2.15}$$

Notice there are now three summations, or intermediate steps, to compute the full DFT. Because this is a radix-2 algorithm, the number of intermediate steps required to perform the DFT is equal to $\gamma = \log_2(N)$. The steps can be separated and seen below.

$$x_1(n_0,k_1,k_0) = \sum_{k_2=0}^{1} x_0(k_2,k_1,k_0)W^{4n_0k_2} \tag{2.16}$$

$$x_2(n_0,n_1,k_0) = \sum_{k_1=0}^{1} x_1(n_0,k_1,k_0)W^{4n_1k_1+2n_0k_1} \tag{2.17}$$

$$x_3(n_0,n_1,n_2) = \sum_{k_0=0}^{1} x_1(n_0,n_1,k_0)W^{(4n_2+2n_1+n_0)k_0} \tag{2.18}$$

A signal flow diagram in Fig. 2.2 represents the calculation of the $N = 8$ case. These signal flow diagrams graphically display the FFT process and are known as butterfly diagrams for how the signals intersect. A base, $N = 2$ example of a butterfly diagram is seen in Fig. 2.1. Where the arrows meet at the node of the following stages, a summation occurs between the two signals. A scaling W term can be seen below the signals before the summations. In

Fig. 2.1, this case has only a single stage but notice that both of the x_1 terms are dependent on the same x_0 terms making them a dual-node pair. The other patterns discussed in the previous section can be obtained even for this simple case. The distance between the dual-node pair is equal to $\frac{N}{2}$ indices, one in this particular case, and the scaling terms have the same power, but the term is negated for the $x_1(1)$ term.

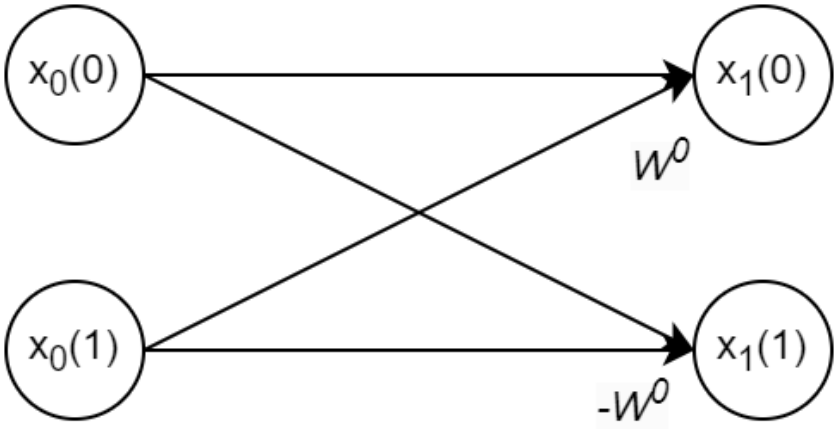

Figure 2.1: $N = 2$ Butterfly Diagram

In Fig. 2.2 the spacing between the dual-node pairs changes as defined in the section above. The distance, defined as a begins in stage 0 as $\frac{N}{2} = 4$, $\frac{N}{4} = 2$ in stage 1, and then in the final stage both members of the pair are adjacent to each other only $\frac{N}{8} = 1$ index apart. It is also worth noting that the second and third stages, $\gamma = 1$ and $\gamma = 2$ respectively, in Fig. 2.2 encompass the entirety of the $N = 4$ case. Following terms $x_1(0) - x_1(3)$ through to the end of $\gamma = 2$ will net the same calculations with the same dual-node pairs as the $N = 4$ the only difference being the reordering required at the end of the process for getting the correct output.

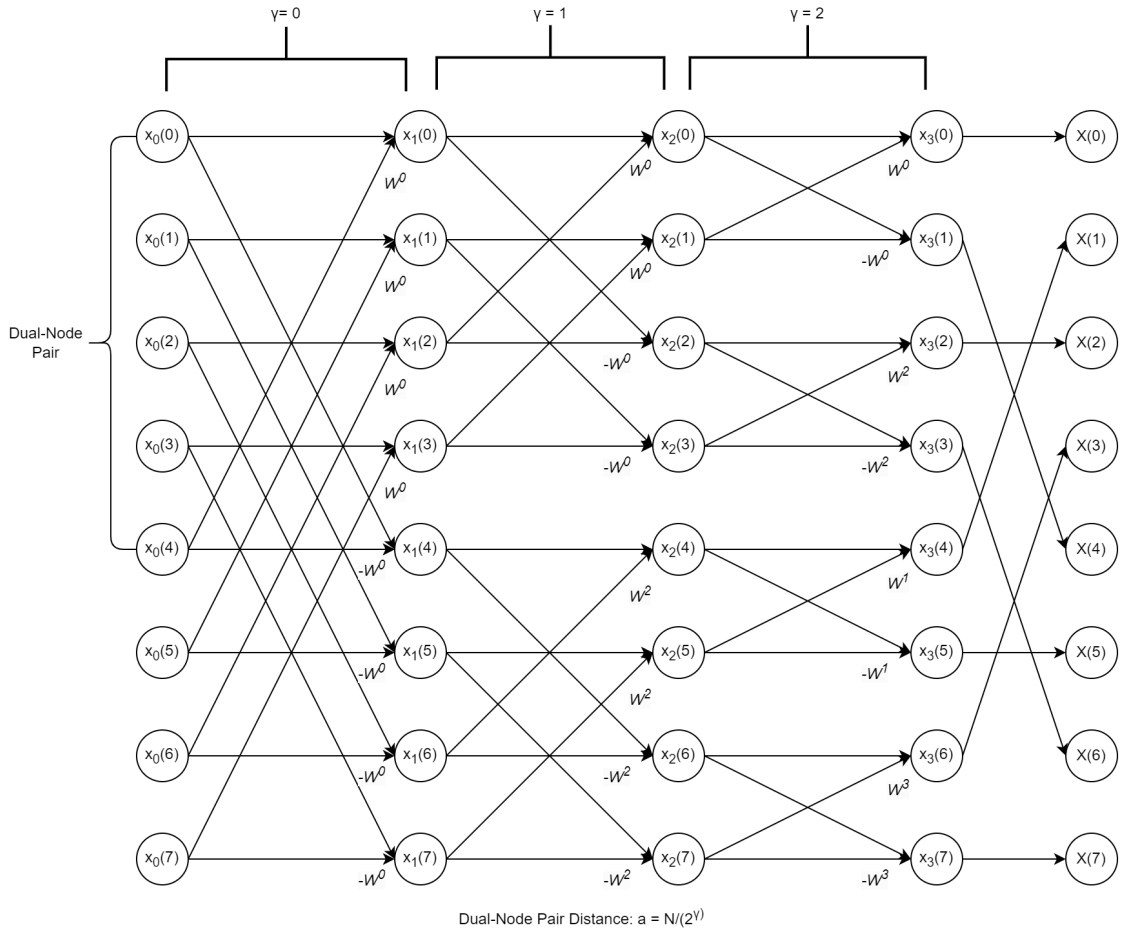

Figure 2.2: $N = 8$ Butterfly Diagram

2.3 Alternate FFT Algorithms

Today's most popular FFT algorithm is the radix-2 Cooley-Tukey algorithm [12–14] explored in the above section. Though, even Cooley and Tukey tried different radices to find the most useful for modern computations in [1]. The efficiencies calculated by Cooley and Tukey in

[1] can be found in Tbl. 2.6. Radix-3 is the most efficient of all radices tested by Cooley and Tukey. However, the radix-2 is popular because the values can be split into pairs without any losses, and it offers advantages when using binary arithmetic, as most modern hardware relies on. It is stated in [1] that highly composite values of N would produce the most savings when performing the algorithm. However, the advantages and simplicity of the radix-2 have allowed it to be the most prolific.

Duhamel and Hollmann introduced a split-radix FFT algorithm [15] which utilizes a radix-2 index mapping for the even-indexed terms and a radix-4 mapping for the odd-indexed terms. This does not decrease the number of multiplications necessary to carry out the complete transform but does decrease the number of additions. Multiple implementations of the split-radix approach have since touted improved complexity over the original and the Cooley-Tukey algorithm [16–19]. Fixed radix implementations of the FFT have also been a heavy research focus in finding the most savings using the largest factors of highly composite sample sizes [20]. Headway in this field has allowed for algorithms with lower arithmetic complexities than the small-radix ones without sacrificing structural complexity, which can create a throughput bottleneck.

Table 2.6: Relative Efficiencies of Varying Radix FFT's

r	$\frac{r}{log_2 r}$
2	2
3	1.88
4	2
5	2.15
6	2.31
7	2.49
8	2.67
9	2.82
10	3.01

Additionally, the Winograd-type FFTs [21] do not take a divide-and-conquer approach like the Cooley-Tukey algorithm. Instead, the Winograd FFT is similar to the Good-Thomas implementation based on prime factors of the sample size. Utilizing recursive multi-dimensional convolution techniques, the Winograd FFT can attain much fewer multiplication operations, which comes at the cost of more additions. As a result, the Winograd and Good-Thomas implementations are often combined to break a large transform into smaller sizes, where the Winograd FFT has more advantages [22].

2.4 Continuous Flow FFTs

The FFT is great for analyzing stationary signals whose frequency content does not vary much over time. However, most applications of the FFT require on-line or continuous monitoring

of some incoming signal which is non-stationary. The FFT can still be utilized for such applications but must be slightly adjusted. The Short Time Fourier Transform (STFT) applies a rectangular time window to the incoming signal so that, within each window, the signal appears to be stationary. This window can then "slide" down the signal taking consecutive FFTs over this shorter time window, producing individual spectrum's all over the same frequency range [23, 24]. The time associated with a calculated spectrum corresponds to the time defined as $t = \frac{N}{2} * T$, where T is the inverse of the sampling frequency. A three-dimensional plot of the spectrum amplitudes, frequency range, and timescale can be used to analyze how the frequency content of a signal changes over time and is known as a spectrogram. Spectrograms can also be displayed in two dimensions using a spatial heat map format. The x-axis represents time, the y-axis represents frequency, and the color value represents the magnitudes/amplitudes of the signal's frequency content.

When deciding the length of the desired time window, it is essential to remember that time and frequency resolution have an inverse relationship. A longer time window (i.e., more samples) gives a higher frequency resolution. However, it may be more challenging to differentiate the frequency content of a rapidly changing signal. In contrast, a smaller window can have the opposite effect holding that the sampling frequency remains constant. The window size selection ultimately depends on the spectrogram's application and the input signal.

2.4.1 Windowing Continuous Flow FFT's

As discussed above, the frequency analysis of continuous, non-stationary signals entails taking the FFT of a windowed signal portion. The DFT of a signal assumes that the input signal is periodic, and an integer number of these periods are transformed in each sample window. Unfortunately, chunking a continuous signal usually produces partial cycles of the incoming signal, especially if the signal is non-stationary. These aperiodic portions create discontinuities in the

measured signal, translating into frequencies that usually span across multiple frequency bins or between frequency bins [25]. The energy of these frequencies then becomes shared across bins causing inaccurate results and a spectrum that looks “smeared”; this is known as spectral leakage. One of the most prevalent ways to combat spectral leakage is using windowing functions with varying frequency domain characteristics.

The STFT naturally applies a rectangular window to the signal; in the frequency domain, the rectangular window appears as a sinc function where the energy of a frequency component at the center of a bin at the result of the FFT fits inside of the main lobe and the energy of adjacent frequencies or those falsely created by discontinuities are spread into the adjacent side lobes. The plots in Fig. 2.3 show a rectangular window’s time and frequency domain characteristics. Notice the relatively narrow main lobe and low side lobe attenuation across the right plot.

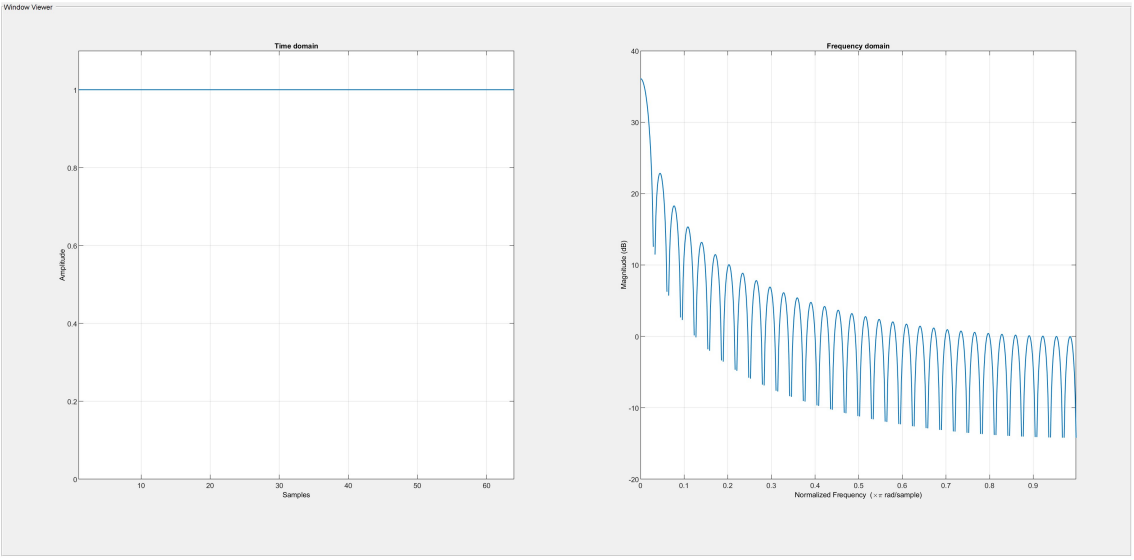

Figure 2.3: Rectangular Filter in Time and Frequency Domain

The window utilized in this paper is a Hanning window which is defined as:

$$h(n) = 0.5 * (1 - cos(2\pi \frac{n}{N})), \quad n = 0, ..., N-1 \tag{2.19}$$

The resulting window and its frequency response can be seen in Fig. 2.4; however, this is far from the only window in use today. All windows aim to eliminate discontinuities in sample chunks by attenuating either end of the sampled signal to zero. The window choice largely depends on the type of signal being monitored and the application of spectral information. The main lobe width of the windows frequency response defines the spectral resolution of each bin, so distinguishing between two frequency components close together would require a window with a narrow main lobe; however, if there is a strong interfering frequency near the desired frequencies, a low maximum lobe level, and quick side lobe roll-off will attenuate this interference. The Hanning window is very commonly used as its frequency characteristics lie in the middle of the road. It is often recommended to begin analysis using the Hanning window and changing to a more suitable window if needed [26].

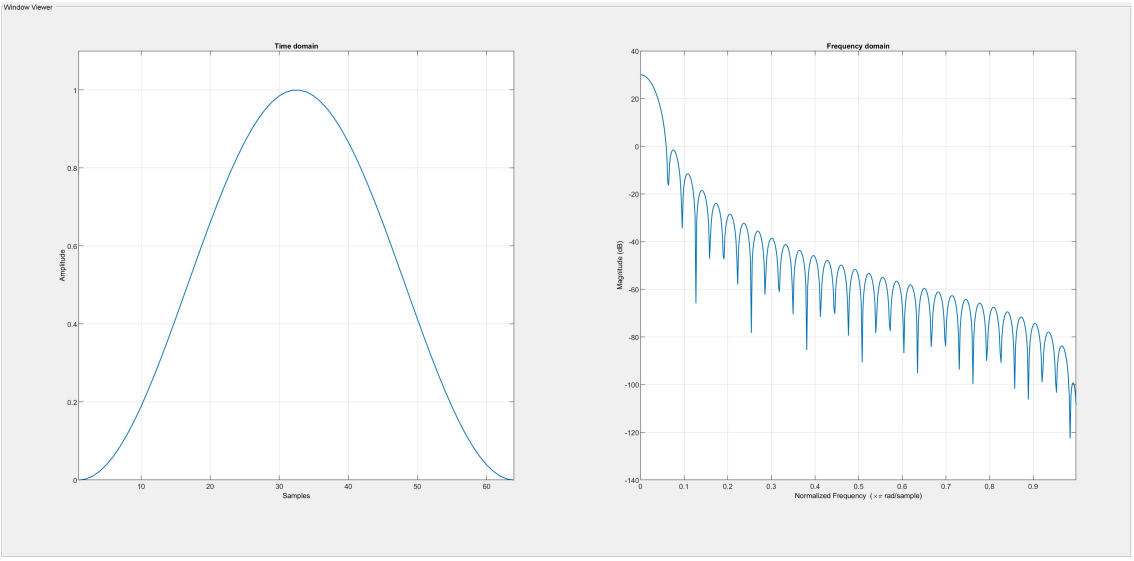

Figure 2.4: Hanning Window in Time Domain and Frequency Domain

2.5 Real-Time Operating Systems

This section aims to discuss very generally the idea of real-time operating systems (RTOS) in their current uses and in relating to on-line analysis in specific. A real-time operating system uses a scheduling kernel to complete tasks in a timely manner. These tasks are able to effectively run in parallel with each other, known as multi-threading, and certain priorities can be defined between tasks to guarantee the timeliness of the most important functions within a real-time environment.

2.5.1 Modern RTOS

In computing, it is given that the resulting output of a system should be accurate according to the theory behind the development of the said system. Bare-metal embedded systems are

designed to produce the correct response consistently, but besides the on-board timers, there is no consideration for when the result is seen. This is okay and even preferred for some applications, as the absence of a full OS makes for a small and less expensive program. As embedded processors have matured, however, using real-time operating systems can now guarantee not only the system output's accuracy but the output's timing as well [27].

Other than the scheduler, RTOS provides several useful features for developers. For example, mutexes and semaphores allow developers to lock away a resource while a task within the program is using it. These key/lock systems provide more control over the system's resources, especially since multiple threads run simultaneously within a real-time system. Queues also allow for precise data movement and handling across tasks and a way for tasks to block themselves while waiting to receive data or access a shared resource.

2.5.2 Problems with an RTOS

As mentioned above, bare-metal environments offer certain advantages, especially when dealing with devices with limited memory capacity. One of the most significant bottlenecks of an embedded system is the memory requirement, as usually, everything remains on board. External memory is sometimes added to alleviate some of the burdens and free up space on the controller itself [27]. However, the more precise the timing required, the larger and more complex the scheduling algorithms will be, and ultimately, the more memory the RTOS will require in the system. The ultimate balancing act of cost and functionality depends on system application; functions pertaining to the health and well-being of individuals, such as running a pacemaker, will prioritize the system's timeliness over the cost of memory.

Of the many RTOS implementations available, the main difference between them is the amount of jitter, which is the variability associated with accepting and completing a real-time task. The amount of jitter separates RTOSs into two categories: soft and hard. A soft RTOS

will have more jitter but can still usually meet a required deadline, while hard RTOSs can deterministically meet their required deadlines [#open_source_RTOS]. IEEE released a set of standards for RTOS in small-scale embedded systems [28] with typical IoT edge devices in mind. [28] Defines a kernel with a minimal footprint for single-chip MCUs and systems with limited memory while retaining the functionality and purpose of real-time OSs. Every task in an RTOS system is provided its own stack partition at a developer-defined depth. It must be large enough to encompass all task variables and the system state saved during context switching. When a kernel switches tasks, it saves the whole system state and dumps it into the stack. When the scheduler returns to this task, it must unload the saved system state before resuming functionality. Thus the more significant the allocated stack, the longer it may take to dump and reapply the system state, ultimately affecting the system's operation as a whole. The complex nature of the kernel/scheduler algorithms also limits the amount of modeling that developers can accomplish and hinders the debugging process, which could increase development time in some cases. Research has been conducted to try and develop modeling techniques for RTOSs though it is not widely used [29].

2.5.3 Use of an RTOS

The effectiveness of an RTOS has prompted much research into the issues stated above. Across the vast number of available RTOS implementations, almost always one version or another can accomplish the necessary functions of an embedded program while fitting within the problem constraints. Extensive analysis of RTOS timing has shown that even softer RTOS implementations remain useful in completing necessary functions within a timely manner while maintaining a small enough footprint not to hinder the performance of small microcontrollers or processors [30–32]. Many uses for RTOS include actively monitoring systems of all shapes and sizes. On-line monitoring, as it is known, performs its checks in real-time while the moni-

tored device is running instead of collecting data for analysis afterward (off-line analysis). The immediate benefits of using an RTOS while actively monitoring a device are easy to see, especially in detecting faults as they occur. If a fault is detected, it must be dealt with immediately before propagation through the rest of the system occurs. A controller using an RTOS should be able to make all necessary measurements and checks before the monitored process completes and perform any fault handling necessary as long as the system is designed correctly. A significant factor for an on-line monitoring system is the amount of data that needs processing. The increase in memory requirements could cause a loss of real-time features or even a system failure in the event of a stack overflow.

Chapter 3

Program Architectures

The principal continuous flow program comprises four real-time (RT) tasks that handle data acquisition, manipulation, analysis, and communication. The following sections in this chapter will cover the program's architecture in detail, followed by a bare-metal test program developed parallel to the continuous flow program. This chapter will begin with a high-level overview of the data path before delving into an explanation of each task and its functionality. The FFT implemented in this program is discussed in detail at the end of this chapter, before the discussion of the test program, as it is separate from any of the individual tasks.

3.1 Data Path Overview

The program begins its initialization in the main.c file I.1. On startup, all peripherals are initialized before creating three RT queues and three of the four tasks. The queues act as data mailboxes between the tasks. The task responsible for reading and writing to the UART, called the *read_Task*, is created first as it also acts as a controller for the rest of the program; both the transform task (*txm_Task*), which windows the captured data and begins the transform and the

analysis task (*ansys_Task*) are created immediately afterward. These three tasks remain idle until the user inputs a start command. Since this command comes through the UART interface, the UART task functions as the data path's start just after the initialization. Upon observing the go command, the UART task spins up a thread to control the ADC peripheral. This task starts the ADC and associated timer before filling the raw data buffer. Once the buffer is full, it is sent through the buffer queue (*buf_mbx*), and the package ready flag is set to signal the transform task that the data is ready for a transformation.

The transform task calls the FFT function and returns a double-sided DFT of the data sent from the ADC task; this results buffer, along with the operation stats, are packaged into a struct and delivered through the stats queue (*stats_mbx*). The analysis task begins by pulling data from the stats queue and preparing it for output by performing mathematical operations. When the data is packaged and sent through the results queue (*res_mbx*), it has been converted to a single-sided buffer of size $N/2 + 1$ full of amplitude values in units of Vrms. Finally, the data path ends in the UART task, which converts the data from floating point values to packets of 10-byte strings formatted for transmission over UART. The conversion allows for efficient transmission and the use of the Direct Memory Access (DMA) controller so the CPU can remain attentive to the continuous flow of data until the program ends.

Two different conditions can cause the program to end. First, the user can set a limit before signaling for the program to begin. This limit is set in units of data packets processed through the program and subsequently passed to the terminal or in terms of seconds if the user sets the designated flag. If an infinitely long signal is fed to the device, the program will run until the specified limit is reached, at which point it will cut the data output and print the stats collected during runtime. If the signal stops and the buffers being processed contain zeros or pure noise for three consecutive seconds, the program will automatically end and print the runtime stats. This end condition can trigger even if a user's specified limit has yet to be reached. A graphical

representation of the program can be seen in Fig. 3.1.

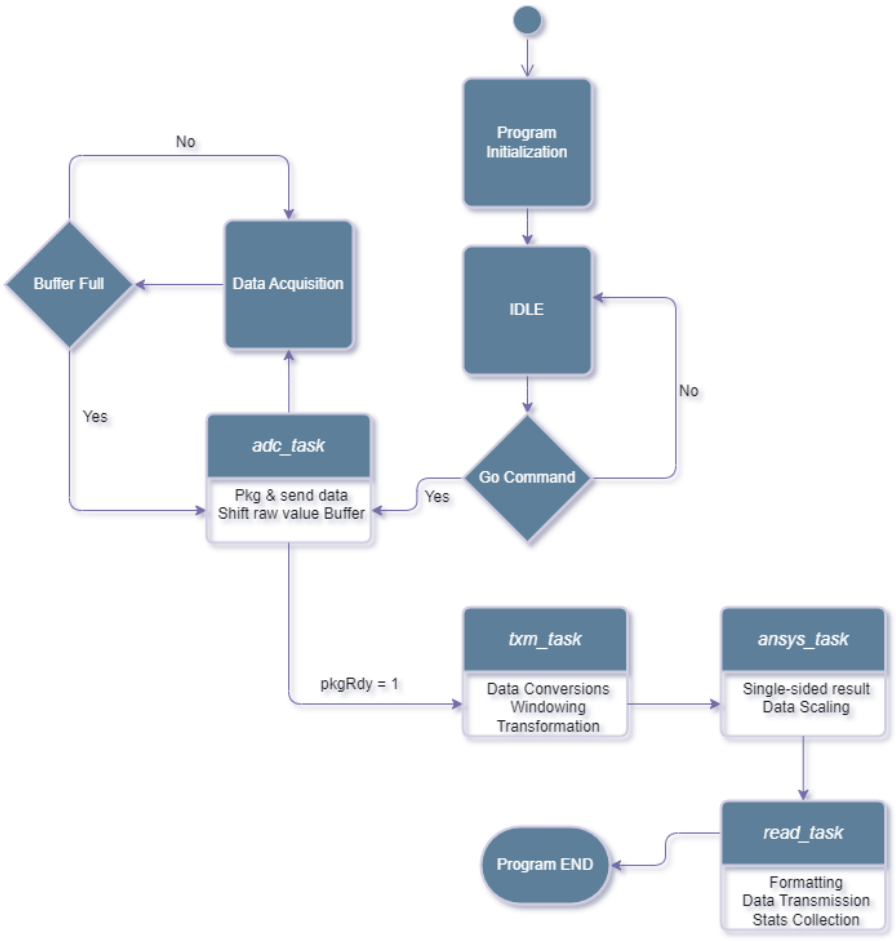

Figure 3.1: Program Flowchart

3.2 *read_Task*

The *read_Task* serves as a program manager through its terminal operation. In order to protect the UART peripheral without blocking any other tasks during the runtime of a program, only

read_Task can read from or write to it. Therefore, it can be split into three parts designated by the active control flags. A signal flow diagram of the full *read_Task* can be seen in Fig. 3.2.

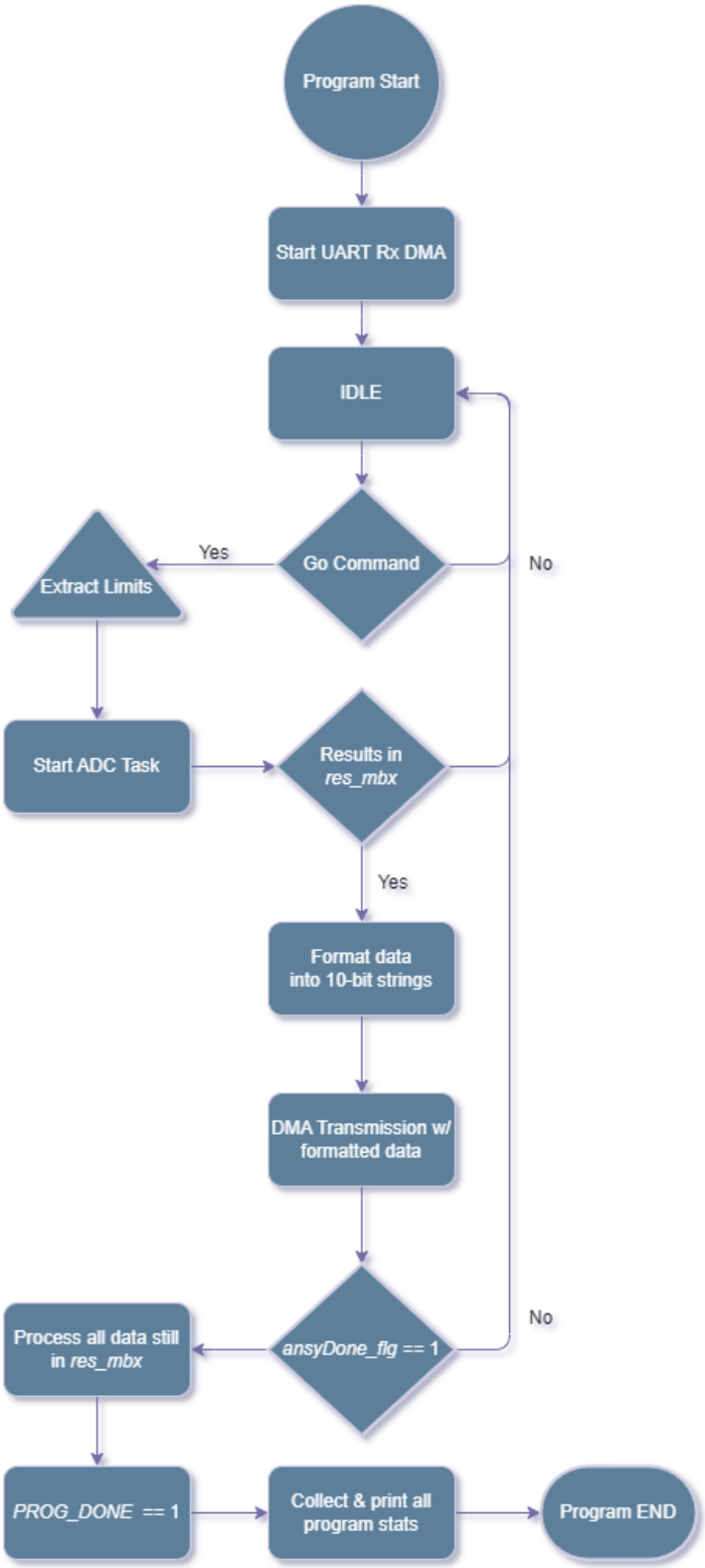

Figure 3.2: *read_Task* Signal Flow Diagram

```
108
109 void read_Task(void * pvParameters)
110 {
111
112     TickType_t lastWake = 0;
113     TickType_t Period = pdMS_TO_TICKS(5);
114
115     float fft_res[SMP_2+1];
116     char TxBuf[(SMP_2+1)*STR_SZ];
117
118     int avg_MC = 0;
119     int avg_AC = 0;
120
121     HAL_UART_Transmit(&huart2, caret, sizeof(caret), 2); // Print starting CMD caret
122
123     while(1)
124     {
125         HAL_UARTEx_ReceiveToIdle_DMA(&huart2, &rxbuf, 1); // Begin DMA
126         __HAL_DMA_DISABLE_IT(&hdma_usart2_rx, DMA_IT_HT);
127
128
129         if(cr_flg)
130         {
131             cr_flg = 0; // Clear carriage return flag
132             ttl_pkgs = atoi(nbuf); // Extract user set limit
133             memset(nbuf,0,sizeof(nbuf)); // Clear number buffer
134             if(timLim_flg) ttl_pkgs = (int) floor(ttl_pkgs/tm2full); // Calculate number of packages needed
135             HAL_UART_Transmit(&huart2, cr, sizeof(cr), 2);
136             xTaskCreate(adc_Task, "adc", 1024,(void *) &ttl_pkgs, PriorityNormal, &adc);
137         }
138
```

Figure 3.3: Start of *read_Task*

3.2.1 Top of Program

The top part of the program, seen in Fig. 3.3, begins before the *while(1)* forever loop with the initialization of a data reception buffer *fft_res* and a UART transmission buffer *TxBuf*. The initialization phase ends by sending a caret through the UART, signaling the user that the program is ready to receive a start command. Within the infinite loop, the UART receive line is started in DMA mode and then the task idles until the *cr_flg* is set in the UART callback function seen in Fig. 3.6. A further exploration of the UART Callback is done in section 3.2.4.

After the read_Task sees the *cr_flg* set, it will clear it and then read the number buffer *(nbuf)* as seen on line 132 in Fig. 3.3. The buffer is then cleared, and the limit is calculated if specified in seconds. The *read_Task* then creates an instance of the task responsible for data

acquisition, the *adc_Task*.

3.2.2 End of Data Path

The *read_Task* sits idle again after beginning the adc_Task instance. At this point, the *bufRdy_flg* is set to true as a part of the program initialization but will remain idle until the data manipulation is completed within the analysis task and a set of results is found within the results mailbox. The *bufRdy_flg* signals when the UART Tx line is being used or free. It is signaled within the UART Tx callback function (Fig. 3.7), which is discussed in Section 3.2.5. The xQueueReceive function seen within the *if-statement* at line 139 in Fig. 3.4 returns false if no objects are waiting in the *res_mbx* queue.

```
138
139         if(xQueueReceive(res_mbx, fft_res, 0) && bufRdy_flg) // Results ready and Tx buffer clear
140         {
141
142             uart_pkgs++;
143
144             for(int i=0; i<SMP_2+1; i++)
145             {
146                 /* Maintain standard 10 bytes of data after sprintf */
147                 if(fft_res[i] > 9 && fft_res[i] < 100)
148                     sprintf(TxBuf+i*STR_SZ, "%.5f,", fft_res[i]);
149
150                  else if(fft_res[i] > 99)
151                     sprintf(TxBuf+i*STR_SZ, "%.4f,", fft_res[i]);
152
153                  else
154                     sprintf(TxBuf+i*STR_SZ, "%.6f,", fft_res[i]);
155
156             }
157
158                 HAL_UART_Transmit_DMA(&huart2, (unsigned *) TxBuf, sizeof(TxBuf)); // Start DMA
159                 bufRdy_flg = 0; // Tx buffer being used by new data
160
161         }
162
```

Figure 3.4: End of *read_Task*

The cleared flags signal that the mailbox should be empty, and the UART Tx line is now in use with the new data. The *for loop* formats the incoming floating point data into a 10-

byte string. The *if-else* ladder checks the magnitude and appropriately assigns decimal point precision to keep the string 10 bytes long with the trailing comma and null terminator. Finally, the formatted strings are assigned a spot in the *TxBuf* buffer and transmitted over DMA as seen in Fig.3.7. The *bufRdy_flg* is reset after each DMA transmission is complete, so this process repeats every time both flags are set unless the program end conditions have been met.

3.2.3 End of Program

If the *PROG_END* flag has been set within the UART Tx callback function (Section 3.2.5) then all data has been transmitted and the *bufRdy_flg* should not be set. The program will fall into this final *if-statement*. The average number of complex multiplications and additions per transform is then calculated on lines 167 and 168 in Fig. 3.5. The total number of data transmission or reception failures are tallied on lines 170 and 171, respectively. Then all stats collected during the program's runtime are printed to the console.

```
162
163
164         if(PROG_END) // All data printed and ADC done
165         {
166             PROG_END = 0; // Clr flag
167             avg_MC = ceil((float) Total_mult/pkg_cnt); // Calculate average number of multiplications per transform
168             avg_AC = ceil((float) Total_add/pkg_cnt); // Calculate average number of additions per transform
169
170             Tx_fails = trans_tx_fail + ansy_tx_fail + adc_tx_fail;
171             Rx_fails = trans_rx_fail + ansy_rx_fail;
172
173             /* Print Stats Message Block */
174             printf("\n\n\r***** STATS *****\n\n\r");
175             printf("Average Mult Ops per transform:\t%d\n\r", avg_MC);
176             printf("Average Add Ops transform:\t%d\n\n\r", avg_AC);
177
178             printf("Number of Rx Failures: %d\n\r", Rx_fails);
179             printf("Number of Tx Failures: %d\n\n\r", Tx_fails);
180             printf("Packages seen by Task:\n\r");
181             printf("\t\tADC\t%d\n\r", pkg_cnt);
182             printf("\t\tTXM\t%d\n\r", txm_pkgs);
183             printf("\t\tANSY\t%d\n\r", ansys_pkgs);
184             printf("\t\tUART\t%d\n\r", uart_pkgs);
185             printf("\t\tSENT\t%d\n\r", pkgs_sent);
186         }
187
```

Figure 3.5: End of Program Statistics Printing

```
60
61 /*Callback for UART receiver. Every 1 character triggers this callback function which does light processing of value*/
62 void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t size)
63 {
64     static int index = 0;
65
66     HAL_UART_Transmit(&huart2, &rxbuf, 1, 2); //Echo character
67
68     if(!cr_flg)
69     {
70         if((rxbuf == '\n' || rxbuf == '\r') && rd_flg) //If enter key has been hit
71         {
72             cr_flg = 1; // Set the flag which is checked in the reader task below
73             rd_flg = 0; // Clear trigger flag
74             index = 0;
75             HAL_UART_Transmit(&huart2, cr, sizeof(cr), 2); // Echo character
76         }
77         else if((rxbuf == 'g' || rxbuf == 'G') && !rd_flg) // g or G for Go
78         {
79             rd_flg = 1; // Ready for read task
80         }
81         else if(rxbuf > 47 && rxbuf < 58 && index < 5)
82         {
83             nbuf[index] = rxbuf; // Collect up to 5 numbers in this buffer
84             index++; // Increment buffer index
85         }
86         else if(rxbuf == 's' || rxbuf == 'S') // s or S for seconds
87         {
88             timLim_flg = 1; // User specified time limit
89         }
90         else if(rxbuf != ' ') // If not acceptable letter
91         {
92             HAL_UART_Transmit(&huart2, bkspc, sizeof(bkspc), 2); // Auto backspace
93         }
94     }
95 }
```

Figure 3.6: UART Rx Callback

3.2.4 UART Rx Callback

The UART Rx DMA (Fig. 3.6) is set only to receive a single character at a time. This character is analyzed in an *if-else* ladder to set the corresponding flags. The user can enter capital or lowercase letters s and g, and up to 5 numbers feed directly into their own buffer. This buffer is read as the user-set program limit. If an s is entered before or after the number, the program will interpret this number in units of seconds; otherwise, the limit is set as processed data packets. Entering a g sets a preliminary starting flag, and pressing enter will set the *cr_flg* to start the program.

3.2.5 UART Tx Callback

The UART Tx callback is much simpler than the Rx callback function as it only tracks how many packages have been transmitted through the *pkgs_sent* counter and then checks ending conditions. For example, if the analysis task has signaled it has completed its functionality through the *ansyDone_flg* and all data has been removed from the results mailbox, the UART transmission is turned off, and then the end of the program is signaled with the *PROG_END* flag.

```
 96
 97 void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) // UART Tx DMA transfer complete callback
 98 {
 99     pkgs_sent++; // Keep track of packages sent
100     if(ansyDone_flg && uxQueueMessagesWaitingFromISR(res_mbx) == 0) // Need adc done and all data sent
101     {
102         PROG_END = 1; // Signal program end
103         HAL_UART_AbortTransmit(huart); // Turn off continuous printing
104     }
105     else
106         bufRdy_flg = 1; // Signal buffer ready for new data
107 }
```

Figure 3.7: UART Tx Callback Function

3.3 *adc_Task*

Like the *uart_Task*, the *adc_Task* divides neatly into two separate sections. A primary functionality block handles the data collected by the ADC, and then a second section begins the ending phase of the whole program. The initialization portion of the task immediately starts the ADC in interrupt mode and the associated timer, configured with a 1 MHz clock in up-counting PWM mode. The generated PWM pulse is tied to the ADC external trigger source and precisely controls the program's sampling frequency through the macro FS I.6. Both these peripherals are started to begin collecting data. An LED on the development board is toggled,

and the program limits are extracted from the parameters passed to the task during its creation. A signal flow diagram of the full *adc_Task* can be seen in Fig. 3.8.

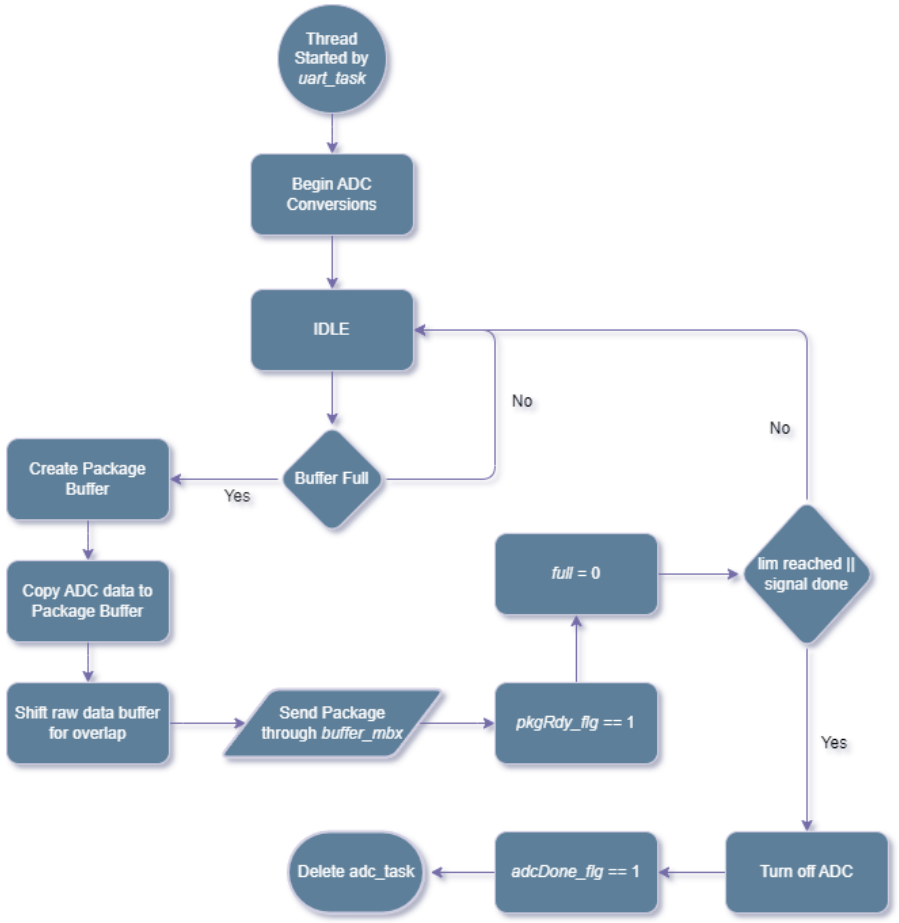

Figure 3.8: Signal Flowchart of *adc_Task*

3.3.1 ADC Main Functionality

```
58 void adc_Task(void * pvParameters)
59 {
60     TickType_t lastwake = 0;
61
62     HAL_ADC_Start_IT(&hadc1); // Start ADC
63     HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1); // Start ADC trigger timer
64
65     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, 1); // Turn on LED to signal ADC start
66
67
68     uint16_t *package = NULL;
69     pkg_cnt = 0; // Reset for repeat trials
70     quiet_cnt = num_emptyBuf; // Reset quiet cnt number at start of ADC task
71
72     /* User Set Limit Signals */
73     _Bool lim_set = 0;
74     int *pkg_lim = (int *) pvParameters;
75     if(*pkg_lim > 0) // Pull pkg limit from parameter
76     {
77         lim_set = 1;
78     }
79
80     while(1)
81     {
82         if(full)
83         {
84             package = (uint16_t *)malloc(SAMPLES*sizeof(uint16_t)); // Create package buffer
85             memcpy(package, RWM, SAMPLES*sizeof(uint16_t)); // Move values from capture to package buffer
86             memmove(RWM, RWM+SMP_2, SMP_2*sizeof(uint16_t)); // Shift capture buffer values down for window overlap
87
88
89             if(xQueueSend(buffer_mbx, package, 1)) // Send package to transform task for processing
90             {
91                 pkg_cnt++; // Count number of packages processed
92                 pkgRdy_flg = 1; // Data package in mailbox ready for transform
93             }
94
95             else adc_tx_fail += 1; // If failed to post, keep track of failure
96
97             if(lim_set && pkg_cnt == *pkg_lim) lim_flg = 1; // Signal time to stop
98
99             full = 0; // Reset full flg
100
101            free(package); // Release package space
```

Figure 3.9: *adc_Task* Main Functionality Block

The task remains idle until the *full* flag is set within the ADC conversion callback function (Fig. 3.11) when the input buffer has been filled. The task then creates a new array named *package* and copies the newly acquired data to it. The second half of the raw data buffer, *RWM,* is then shifted down to occupy the first half to create a window overlap of 50%. The indexing of this overlap is handled in the callback function and discussed in Section 3.3.3. *SMP_2* is a macro defined in the global.h file, I.6, as half of the number of samples within a window. The package of data is loaded into the raw data queue, *buf_mbx*, the number of packages that have been transmitted is updated, and then the *pkgRdy_flg* is set to signal the transform task to

begin its operation. Since the data has been sent and the raw data buffer has been shifted, it is no longer considered full, so the flag is cleared. Finally, the dynamically allocated package buffer is freed (Fig. 3.3.3).

3.3.2 Ending Conditions

Suppose a limit has been set, which is decided during task initialization (Fig. 3.3.3). In that case, the central portion of the task will continuously check this limit every time a new package has been sent through; if the package count matches that of the limit, then the limit reached flag, *lim_flg* is set so that the task can enter its ending conditions. Another signal can trigger this secondary state but is set within the transform task and will be discussed in Section3.4.

This section begins with the setting of the *adcDone_flg*, which provides the "time to end" signal to the rest of the program. Then, if the program is ending because the signal stopped, those empty buffers are removed from the total package count so as not to skew the final operation calculations seen in Fig. 3.5. Next, several of the control flags are cleared, and then the task deletes itself to free memory for the rest of the program to operate on the final data buffer traveling through the system.

```
102
103             if(lim_flg || sigDone_flg) // Two possible ending conditions
104             {
105                 /* Turn off Timer and ADC before deleting the adc task */
106                 HAL_TIM_PWM_Stop(&htim3, TIM_CHANNEL_1);
107                 HAL_ADC_Stop_IT(&hadc1);
108 
109                 adcDone_flg = 1; // Signal program that adc is done
110                 if(sigDone_flg) pkg_cnt -= num_emptyBuf; // Remove empty buffers from total count
111 
112                 sigDone_flg = 0; // Clear signal flag
113                 lim_flg = 0; // Clear limit flag
114 
115                 vTaskDelete(NULL);
116             }
117         }
```

Figure 3.10: *adc_Task* Wrap-up Portion

3.3.3 ADC Conversion Callback

This is another very simple callback which mostly keeps track of the current index for the raw data buffer. The index is represented by the integer variable *idx*. It is simply incremented after every pass, until it reaches the maximum buffer index, $SAMPLES - 1$. When the buffer is full it sets the flag and then resets the index variable to the halfway point of the buffer represented as *SMP_2*. This handles the overlap indexing discussed in Section 3.3.1 so this function should never write the lower half of the buffer after the initial pass.

```
44
45 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) // ADC callback function
46 {
47     RWM[idx] = HAL_ADC_GetValue(&hadc1); // Acquire ADC value
48     HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5); // Toggle LED
49     if(idx == SAMPLES-1) // Check index value. Restart?
50     {
51         full = 1;
52         idx = SMP_2; // Only saving half of samples for window overlapping
53     }
54     else
55         idx++;
56 }
```

Figure 3.11: ADC Conversion Callback Function

3.4 *txm_Task*

Unlike the previous two tasks, the transform task, *txm_task*, only has a single section besides its initialization. The initialization (Fig. 3.12) is responsible for calculating the number of empty packages that must be seen before deciding whether the signal has ended. Lines 57-59 in Fig. 3.12 show these calculations with *quiet_cnt* representing the counter used in the transform task to track the number of empty packages seen. The *num_emptyBuf* variable is

used by *adc_Task* to remove the empty buffer counts from the count of total packages. The initialization also creates a struct named *res_fft* at line 55. The struct type definition is found in the global.h file I.6 and can be seen in Fig. 3.13. This struct contains and transports data and information/statistics collected as it progresses through the transform and analysis tasks.

```
48 void txm_Task(void * pvParameters)
49 {
50     TickType_t lastWake = 0;
51     TickType_t Period = pdMS_TO_TICKS(20);
52
53     uint16_t *rec = NULL;
54
55     stats_t res_fft;
56
57     tm2full = (float) SAMPLES/FS; // Time it takes to fill one buffer worth of samples
58     quiet_cnt = (int) floor(3/tm2full); // 3 seconds of silence signals end of incoming signal
59     num_emptyBuf = quiet_cnt;
60
61     while(1)
62     {
```

Figure 3.12: Transform Task Initialization Phase

```
52
53 typedef struct stats{
54     unsigned int pkg_num;
55     unsigned long mult_cnt;
56     unsigned long add_cnt;
57     int zCnt;
58     float complex res_buf[SAMPLES];
59 }stats_t;
```

Figure 3.13: *stats_t* Typedef Definition

The main functionality, however, can be seen in Fig. 3.14, starting with extracting the new package's associated number and creating a receiving buffer in the correct data type. Data from the ADC is a 12-bit unsigned integer, but the buffer meant to hold data within the stats struct is of type complex float. Therefore, some manipulation is required to convert the data

types. The *rec* buffer is a temporary intermediate container created for this conversion process. The data moves from the queue to the new buffer, and the *pkgRdy_flg* is cleared. Then, the data is converted in the following for loop. Each iteration calculates a new Hanning window coefficient to properly window the current data sample. The following line, 78, shows the conversion. The current data value is multiplied by the Hanning window coefficient and a conversion factor defined as *TOREAL*. This macro converts ADC digital values from 0-4096 to a voltage equivalent between 0V and 3.21V. Adding $0 * I$ typecasts the whole value as a float complex to fit inside the *res_buf* buffer within the *res_fft* struct properly.

After converting all data values, the receiving buffer is no longer necessary and freed before the FFT function is called. Since the FFT algorithm is calculated in place, the struct is passed by reference into the function. The *zCnt* element of the *res_fft* struct counts the number of empty values seen after FFT calculation. If this number exceeds 95% of the total samples, the whole package is considered empty, and the *quiet_cnt* is decremented before checking if it has reached 0. If so, the *sigDone_flg* is set, and the end of the program will begin once the *adc_Task* observes it. If the buffer is not empty, the quiet counter is reset, so the condition remains 3 seconds worth of consecutively empty packages. The task then sends the entire struct through the statistics mailbox, *stats_mbx*, for the analysis task. Once the *adcDone_flg* has been seen by the transform task and all data has been pulled from *buf_mbx*, the *txmDone_flg* is set, and the task suspends itself to preserve resources in the rest of the program and stops task functionality since no more data is coming through. A signal flow diagram of the full *read_Task* can be seen in Fig. 3.15.

```
63          if(pkgRdy_flg)
64          {
65              res_fft.pkg_num = pkg_cnt; // Find current package number
66              rec = (uint16_t *)malloc(SAMPLES*sizeof(uint16_t)); // Create reception buffer
67
68              if(xQueueReceive(buffer_mbx, rec, 0) != pdTRUE) trans_rx_fail++; // Receive data buffer from mailbox
69
70              else
71              {
72                  txm_pkgs++;
73                  pkgRdy_flg = 0; // Reset pkgRdy flag
74
75                  for(int i = 0; i < SAMPLES; i++)
76                  {
77                      float hann = 0.5-0.5*cos(2*PI*i/SAMPLES); // Calculate Hanning window coefficient
78                      res_fft.res_buf[i] = (float)rec[i]*hann*TOREAL + 0*I; // Apply Hanning window and convert to complex number
79                  }
80
81                  free(rec); // Free temporary transfer buffer
82                  FFT(&res_fft); // Run FFT
83
84                  float perEmpty = (float) res_fft.zCnt/SAMPLES;
85                  if(perEmpty >= 0.95) // If 95% of buffer is zero, consider it empty
86                  {
87                      quiet_cnt--; // Keep track of empty captures
88
89                      if(quiet_cnt == 0)
90                      {
91                          sigDone_flg = 1; // Signal finished, adc no longer needed
92                      }
93                  }
94                  else // Reset quiet count
95                      quiet_cnt = num_emptyBuf; // Reset quiet_cnt
96
97                  if(xQueueSend(stats_mbx, &res_fft, 0)) txm_pkgs++;
98
99                  else trans_tx_fail++;
100
101                 if(adcDone_flg && uxQueueMessagesWaiting(buffer_mbx) == 0)
102                 {
103                     txmDone_flg = 1;
104                     vTaskSuspend(NULL); // Suspend if ADC done and no more data to process
105                 }
106             }
```

Figure 3.14: Transform Task Main Functionality

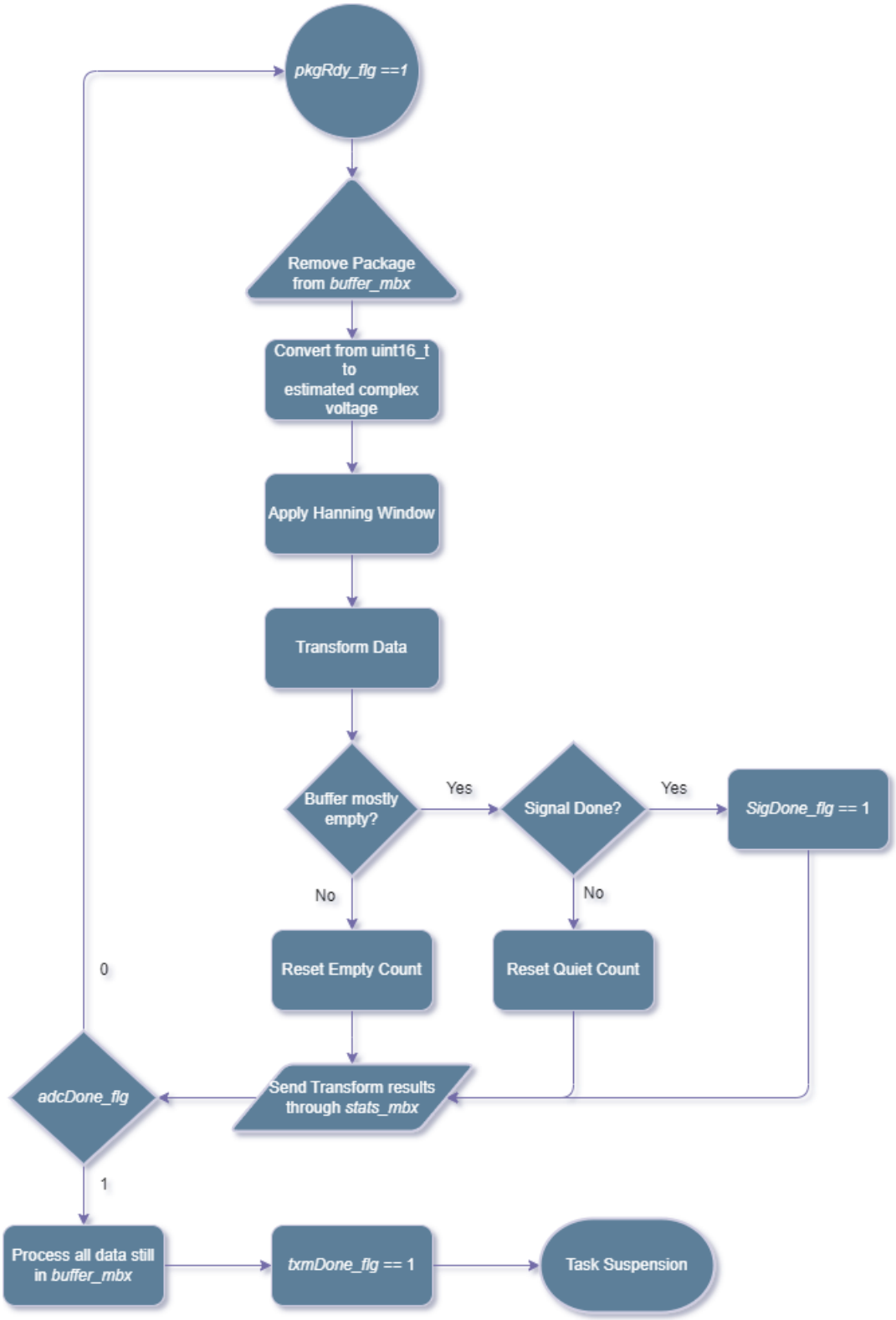

Figure 3.15: Signal Flowchart for *txm_task*

3.5 *ansys_Task*

The final data manipulation stage along the data path is the analysis task, named *ansys_Task* in the program. There is very little in the way of initialization, as it instead does all the math once data has been sent through the *res_mbx*. The flag is cleared before a new temporary buffer is created for storing the manipulated data. Part of the manipulation sees the second half of the buffer being dropped since this system was built with real-valued signals in mind. For real-valued signals, this second half contains the same information as the first half, and as such, a conversion to a single-sided result preserves all the calculations at half the size. This is why the *temp_buf* is only half plus one sample large, the plus one accounting for the DC component bucket at index zero within the results buffer. A complete flowchart of the *ansys_Task* can be found in Fig. 3.16.

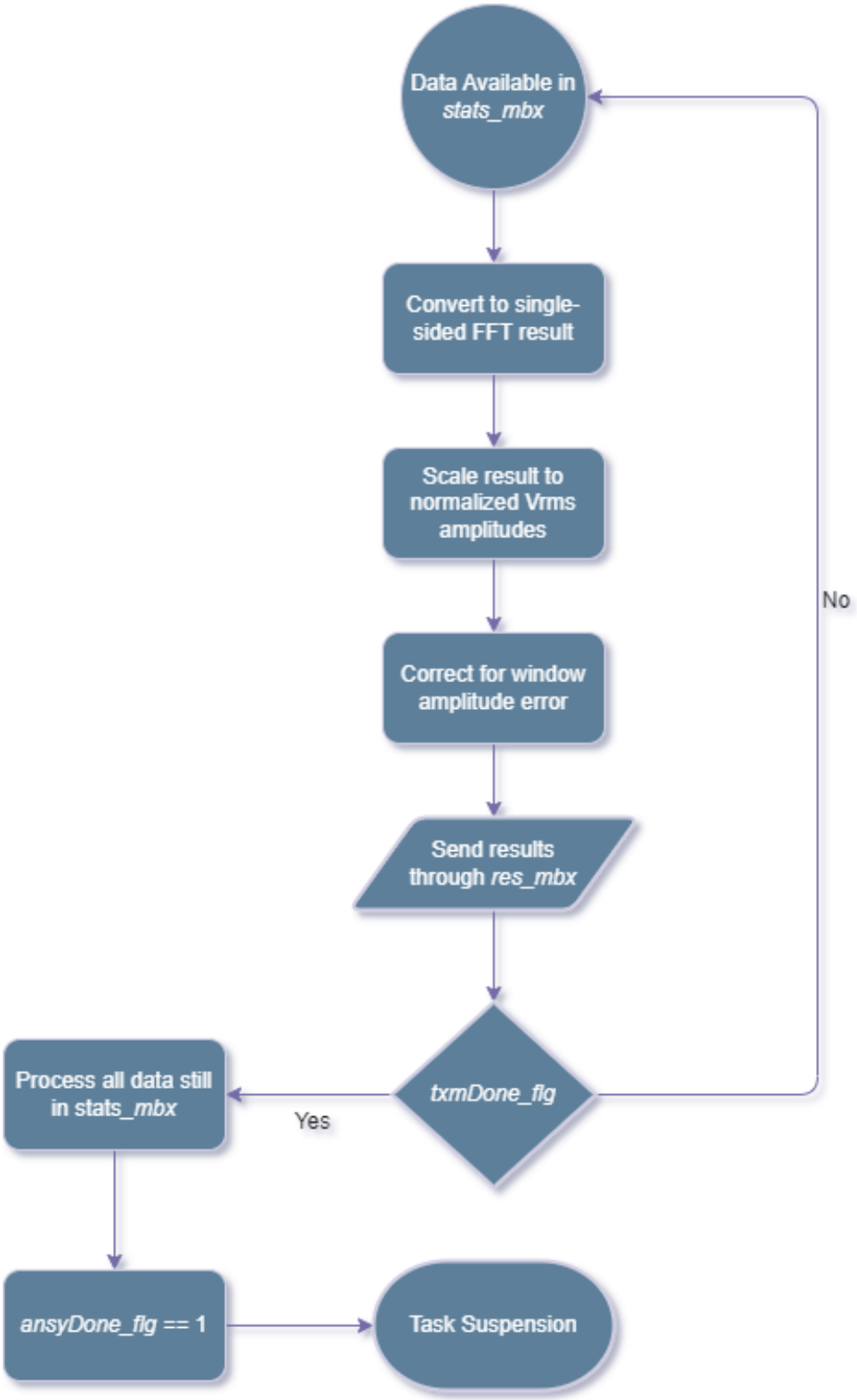

Figure 3.16: *ansys_Task* Signal Flow Diagram

The *fft_res* variable, which receives data from the stats mailbox, is a struct of the typedef *stats_t* defined in Fig. 3.13. This is the receptacle for data from the stats mailbox queue at line 54 within Fig. 3.17. The *for loop* immediately following this converts the data to a single-sided amplitude spectrum in units of rms (root-mean squared) volts. Conversion is carried out using the following equations:

For DC component:

$$A_{Vrms}(i) = \frac{magnitude(FFT(i))}{N}, \quad i = 0; \tag{3.1}$$

For non-DC components:

$$A_{Vrms}(i) = \sqrt{2} * \frac{magnitude(FFT(i))}{N}, \quad i = 1,2,3,...,\frac{N}{2}+1; \tag{3.2}$$

These equations are suggested in [26], and the power spectrum of this result can be easily calculated by squaring each element in the resulting array. Doing so, in the program, produced values that were too small to fit within the 10-byte character string, and as such, only the amplitude spectrum is computed. The mag function in lines 63 and 67 is defined in the same *analysis.c* file (I.5) and can be seen in Fig. 3.18. After the completion of this *for-loop*, the data is finished being processed and sent into the results mailbox queue, *res_mbx*. With the data being sent away, the temporary buffer is freed. Once the *ansys_Task* sees the *txmDone_flg* is set and it processed all data that was passed to it, the analysis task signals it has finished with the *ansyDone_flg* and suspends itself to limit the context switching occurring during the final print statements.

```
void ansys_Task(void * pvParameters)
{
    TickType_t lastWake = 0;
    TickType_t Period = pdMS_TO_TICKS(10);
    stats_t fft_res;

    float *temp_buf = NULL;

    while(1)
    {
        if(xQueueReceive(stats_mbx, &fft_res, Period)) // If analysis has been triggered
        {
            temp_buf = malloc((SMP_2+1)*sizeof(float)); // N/2+1 buffer for calculations

            ansys_pkgs++;
            for(int i = 0; i < SMP_2+1; i++) // Convert to single sided
            {
                if(i==0)
                {
                    fft_res.res_buf[i] = mag(fft_res.res_buf[i])/SAMPLES; // Normalize magnitude of DC component
                }
                else
                {
                    fft_res.res_buf[i] = sqrt(2)*mag(fft_res.res_buf[i])/SAMPLES; // Convert to Amplitude rms value
                }

                temp_buf[i] = creal(fft_res.res_buf[i])*wErr; // Correct windowed amplitude and transfer to smaller buffer
            }

            if(xQueueSend(res_mbx, temp_buf, 0)) ansys_pkgs++;// Send smaller buffer for printing

            else ansy_tx_fail++;

            free(temp_buf);

            /* For averages at end of the program */
            Total_mult += fft_res.mult_cnt;
            Total_add  += fft_res.add_cnt;

            if(txmDone_flg && uxQueueMessagesWaiting(stats_mbx) == 0)
            {
                ansyDone_flg = 1;
                vTaskSuspend(NULL);
            }
```

Figure 3.17: Analysis Task Function

```
float mag(float complex N)
{
    float r2 = creal(N)*creal(N);
    float i2 = cimag(N)*cimag(N);

    return sqrt(r2 + i2);
}
```

Figure 3.18: User Defined Function for Calculating Magnitude of Complex Numbers

3.6 FFT Implementation

A transformation begins in Fig. 3.19 with nested *for-loops* that perform the FFT's butterfly operations. Each iteration of the outer loop represents a stage of the transformation, with the number of stages defined as $log_2(N)$. The inner loop performs the math within each stage, iterating through each sample within the buffer. The *if-statement* in the inner loop accounts for the dual-node pairs so that the second term, $x_m(k+a)$, is not operated on twice. This distance, a, is updated at the end of every stage; in the implementation, it is bit-shifted right which functionally divides the value in half without performing an actual division operation. The *bit_reverse()* function found in Fig. 3.19 is a helper function that is discussed in Section 3.6.1, but as its name implies, this function returns the bit-reversed value of the integer passed to it. In line 45, this function takes in the index of the sample currently being operated on to calculate p, the power of the twiddle factor. This algorithm for calculating the power of the twiddle factor is recommended by Brigham [11, pg. 140]. The variable twexp is defined at the top of the fft.c file (I.7) as $2\pi IN$ the constant part of every twiddle factor exponent. This is then multiplied by the calculated p-value and passed to the complex exponential C function of the complex library. Notice that variable x is not altered, but the twiddle factor scales *x1* before being used in Eqns. (2.11) and (2.12). The number of operations are then counted in their respective data variables.

```
37      for(int j = 1; j <= stages; j++)
38      {
39          for(int k = 0; k < SAMPLES; k++)
40          {
41              if(!(k & a)) // Remove redundant computations
42              {
43                  m = bit_reverse(stages, k >> (stages - j)); // Calculate twiddle power
44                  x = results->res_buf[k];
45                  xp = cexp(twexp*m)*results->res_buf[k+a];
46
47                  results->res_buf[k] = x + xp;
48                  results->res_buf[k + a] = x - xp;
49                  results->mult_cnt += 1; // One complex multiplication
50                  results->add_cnt += 2; // Two complex additions
51              }
52          }
53
54          a >>= 1; // Change Dual-Node distance
55      }
```

Figure 3.19: Butterfly Loops

The *for-loop* in Fig. (3.20) shows the index reordering algorithm. The loop iterates over every sample within the buffer, in each iteration the current buffer index is bit reversed and then the value at the current index and the bit reversed index are swapped. The *if-statement* keeps the already bit reversed values from being double counted.

```
56
57      /* In Place Array Re-Indexing */
58      for(int i = 0; i < SAMPLES; i++)
59      {
60          int p = bit_reverse(stages, i);
61
62          if(i < p) // Only swap elements once
63          {
64              float complex n = results->res_buf[i];
65              results->res_buf[i] = results->res_buf[p];
66              results->res_buf[p] = n;
67          }
68
69          if(creal(results->res_buf[i]) == 0)
70              results->zCnt++;
71      }
72  }
```

Figure 3.20: Results Buffer Reordering

3.6.1 Helper Functions

Two FFT helper functions are also defined within the fft.c file, the first of which is the bit reversal algorithm (Fig. 3.21) which takes two parameters: *sz*, is the number of bits that the second parameter, *index* must be reversed within. The *sz* in this program is always the number of bits needed to represent the size of the transform, saved as stages in the FFT function 3.19. A *for-loop* iterates through each bit and compares it with the *index* value; if a match is found, then the temporary variable *p* is filled with a high bit and left-shifted *i* positions. Since *i* begins at zero, but the bit checking begins at *sz*, *i* will reflect the inverse bit position of the match found in the *if-statement*.

```
80 int bit_reverse(int sz, int index) // Bit reversal algorithm
81 {
82     int p = 0;
83 
84     for(int i = 0; i <= sz; i++)
85     {
86         if(index & (1 << (sz - i)))
87         {
88             p |= 1 << (i - 1);
89         }
90     }
91 
92     return p;
93 }
```

Figure 3.21: Bit Reversal Algorithm

The second helper function is an implementation of the $log_2()$ function using no multiplication or division operations. The function, seen in Fig. (3.22), takes in a single integer value which is shifted right over every iteration of a *while-loop*. With each iteration, a counter is incremented keeping track of how many bits are in the value. Once all bits have been shifted out, the loop ends and the counter value is decremented to account for the extra increment after the last bit is shifted out.

```
100 int log_2(unsigned int N)
101 {
102     int pow = 0;
103
104     while(N)
105     {
106         N >>= 1;
107         pow++;
108     }
109
110     return pow - 1;
111 }
```

Figure 3.22: log_2 Function

3.7 Global Header

Many of the important program signals are defined or shared through a header file named *global.h* I-52. This contains definitions of various macros used throughout the program as well as the control flags that are shared across all the files within the program. The priority enum defines the task priorities used when creating a new task and then finally there is the definition of the *stats_t* typedef.

3.8 C Test Program

The bare-metal designation of the test program means that no operating system is running on the board and, consequently, no real-time features. This environment borrows much of the same code implemented for the continuous flow program but runs only a single conversion

before it ends. The lack of an operating system provides much more memory in this program. As a result, the number of samples per DFT calculation can increase from 256, the maximum for the continuous flow program, to 4096. This dramatically increases frequency resolution and thus increases the practical sampling frequency maximum achievable in this test environment. The small profile of the test program fits within just the *main.c* file (I.1) where a signal is created or read from the ADC and then windowed in the same manner as seen at the start of the transform task described in Section 3.4. Fig. 3.23 illustrates the signal creation portion. In the current configuration, the ADC reads a signal before converting it from the ADC integer representation to the real-valued voltage value and applying the same Hanning window as seen in the continuous flow program. Since this program only computes a single conversion on a signal with non-varying frequency content, the window is unnecessary to prevent spectral leakage. However, it is still applied to verify an accurate implementation compared to the built-in Hanning window MATLAB function. The commented-out lines from 156-158 represent other tests that produce an arbitrary signal using C's math library. Using a synthetic signal this way would require the commenting of lines 149 and 150 since the ADC will not be needed and additionally the commenting of line 159 since the RWM buffer will remain empty if the ADC is not used.

3.8.1 FFT Results Manipulation

The following section of the test program combines the analysis task and the end of data path section of the read_Task discussed in Section 3.2.2. The for loop beginning at line 172 in Fig. 3.24 converts the double-sided complex FFT result into a single-sided real-valued buffer of length $\frac{N}{2}+1$. After that, starting at line 182, the *if-else* ladder is pulled directly from

```
145     while (1)
146     {
147         if(full)
148         {
149             HAL_ADC_Stop_IT(&hadc1);
150             float *temp_buf = NULL;
151             char pBuf[(SMP_2+1)*STR_SZ];
152 
153             for(int i = 0; i < SAMPLES; i++)
154             {
155                 float hann = 0.5-0.5*cos(2*PI*i/SAMPLES); // Calculate Hanning window coefficient
156 //              fft_res.res_buf[i] = sin(2*PI*200*i/FS); // Single-Tone 200 Hz
157 //              fft_res.res_buf[i] = (sin(2*PI*1000*i/FS)+1) + 0.5*(cos(2*PI*200*i/FS)+1); // Dual-Tone 200 Hz and 1 kHz
158 //              fft_res.res_buf[i] = ((sin(2*PI*fc*i/FS)) + (sin(2*PI*fg*i/FS)) + (sin(2*PI*fe*i/FS))); // C Chord triad
159                 fft_res.res_buf[i] = TOREAL*(RWM[i]); // ADC Implementation
160                 fft_res.res_buf[i] *= hann; // Apply Hanning window and convert to complex number
161             }
162 
163 
164             classic_FFT(&fft_res);
165 
166             full = 0;
167 
```

Figure 3.23: Test Program Signal Creation

the *read_Task* and similarly formats the results into 10-byte strings for transmission out of the UART Tx line using the DMA controller. The callbacks for the UART Tx and ADC peripherals are the same as shown in Figs. 3.7 and 3.11, respectively, so they will not be rehashed in this chapter. The only difference is that the UART Tx callback does not have to check finishing conditions as this is a single-pass program.

```
168             temp_buf = malloc((SMP_2+1)*sizeof(float));
169 
170             for(int i = 0; i < (SMP_2+1); i++)
171             {
172                 if(i==0)
173                 {
174                     fft_res.res_buf[i] = mag(fft_res.res_buf[i])/SAMPLES; // Normalize magnitude of DC component
175                 }
176                 else
177                 {
178                     fft_res.res_buf[i] = sqrt(2)*mag(fft_res.res_buf[i])/SAMPLES; // Convert to Amplitude rms value
179                 }
180                 temp_buf[i] = creal(fft_res.res_buf[i])*2; // Correct windowed amplitude and transfer to smaller buffer
181 
182                 if(temp_buf[i] > 9 && temp_buf[i] < 100)
183                     sprintf(pBuf+i*STR_SZ, "%.5f,", temp_buf[i]);
184 
185                 else if(temp_buf[i] > 99)
186                     sprintf(pBuf+i*STR_SZ, "%.4f,", temp_buf[i]);
187 
188                 else
189                     sprintf(pBuf+i*STR_SZ, "%.6f,", temp_buf[i]);
190 
191                 HAL_UART_Transmit_DMA(&huart2, pBuf, sizeof(pBuf));
192             }
```

Figure 3.24: Test Program Results Manipulation

The program's end is similarly pulled directly from the *read_Task* without any of the data flow statistics. The *done* flag is set in the UART callback after the data is transmitted, and then the program statistics are printed to the terminal. These are not averages as seen before. Instead they are the raw statistics collected during the FFT calculation. The time per transform is calculated in microseconds units as the timer has a tick frequency of 80 MHz.

```
197
198         if(done)
199         {
200             HAL_UART_Abort(&huart2);
201             printf("\n\n\r***** STATS *****\n\r");;
202             printf("Time per transform:\t%f uS\n\r", (float) fft_res.time/80);
203             printf("Mult Ops per transform:\t%d\n\r", fft_res.mult_cnt);
204             printf("Add Ops per transform:\t%d\n\n\r", fft_res.add_cnt);
205             exit(1);
206         }
```

Figure 3.25: Test Program Ending

Chapter 4

Experimental Results

The following chapter will discuss the validation of the proposed program. It begins with a description of the MATLAB validation environments and how they are used to compare with the results of the test and main programs. Results of the test program's validation tests will be presented after this explanation and then the chapter will conclude with a presentation of the main program's validation results.

4.1 MATLAB Validation Environment

Each test is validated using a MATLAB script designed to simulate the conditions defined within the program, such as the sampling frequency and number of samples per DFT. Built-in MATLAB functions are used in the script to see that the proposed implementation achieves the same results. The environment setup is seen in Fig. 4.1; this setup is meant to mimic the system settings found in the *global.h* file I.6. These settings must be manually adjusted to match those found within global.h and the t and f arrays represent the time and frequency scales for the calculated signals, which are shared amongst the MATLAB and C results. The

fc, fg, and fe variables are the frequencies of musical notes C_4, G_5, and E_5, respectively, in the $A_4 = 440$ Hz tuning. This triad forms the standard C chord used in one of the validation tests discussed in Section 4.2.3.

```
5   %% Environment Variables
6   N = 256;
7   sz = N/2+1;
8   fs = 5000;
9   fc = 261.63;
10  fg = 783.99;
11  fe = 659.25;
12  dt = 1/fs;
13  t = 0:dt:(N-1)*dt;
14  i = 0:N-1;
15  f = linspace(0,fs/2,N/2+1);
16  h = hann(N)';
17
18  %% Signal Generation
19  % x = sin(2*pi*200*t)+1;
20  % x = (sin(2*pi*fc*t) + sin(2*pi*fg*t) + sin(2*pi*fe*t)); % Create C Chord Sinusoid
21  x = (sin(2*pi*1000*t)+1) + 0.5*(cos(2*pi*200*t)+1);
22
23  xUnwin = x;
24  x = h.*x; % Apply Hanning Window
```

Figure 4.1: MATLAB Environment Variables

Notice in Fig. 4.1 the built-in *hann()* function for generating a 256-point Hanning window which scales the calculated test signal. A function for loading the C Program data follows in Fig. 4.2, which can extract only the data portion of the logged terminal output and none of the statistics directly into a MATLAB-compatible datatype. After the program-under-test (PUT) data is imported to the validation environment, the test signal is generated in MATLAB, with x undergoing the same operations found in the PUT. Lines 28-30 of Fig. 4.3 encompass this data manipulation starting with the built-in *fft()* MATLAB function of the signal before taking the signal's magnitude and normalizing it against the number of samples, *N*. The second half of the results are then dropped, and the non-DC components are converted to amplitude

values of $Vrms$. Following the math portion of the script is when all of the relevant data is plotted in separate figures. Four different plots are generated with every run of the validation environment. The first shows the test signal without the applied window in the time domain, and the second is the then windowed signal, again in the time domain. The third and fourth, however, share a figure broken into two subplots. The top subplot displays the MATLAB simulation results, and the bottom plot shows the PUT results..

```
59
60  function C = load_data(CDataPath, N)
61
62      %% Import Data From C Program
63      opts = delimitedTextImportOptions("NumVariables", N);
64
65      % Specify range and delimiter
66      opts.DataLines = [3, 3];
67      opts.Delimiter = ",";
68      [vartypes{1, 1:N}] = deal('double');
69      opts.VariableTypes = vartypes;
70
71      % Specify file level properties
72      opts.ExtraColumnsRule = "ignore";
73      opts.EmptyLineRule = "read";
74      opts.ConsecutiveDelimitersRule = "join";
75
76      % Import the data
77      C = readmatrix(CDataPath, opts);
78
79      % Clear temporary variables
80      clear opts
81  end
```

Figure 4.2: MATLAB Generated Data Importing

```
25
26      T = load_data("Data_Files\SR_Synth\SR_Dual-Tone.csv", sz);
27
28      X = abs(fft(x,N))/N; % Normalized Magnitude of fft output
29      X = X(1:N/2+1); % Single-sided conversion
30      X(2:end-1) = sqrt(2).*X(2:end-1); % Amplitude Vrms for non-DC components
31
32      %% Results plotting
33      figure
34      plot(t,xUnwin);
35      title("UnWindowed Signal in Time Domain")
36      xlabel("Time (s)")
37      ylabel("Amplitude (V)")
38
39      figure
40      plot(t,x);
41      title("Windowed Signal in Time Domain")
42      xlabel("Time (s)")
43      ylabel("Amplitude (V)")
44
45      figure
46      subplot(2,1,1)
47      plot(f,X);
48      title("MATLAB FFT Results")
49      xlim([-1 fs/2])
50      xlabel("Frequency (Hz)")
51      ylabel("Amplitude (Vrms)")
52
53      subplot(2,1,2)
54      plot(f, T);
55      title("CTP FFT Results")
56      xlim([-1 fs/2])
57      xlabel("Frequency (Hz)")
58      ylabel("Amplitude (Vrms)")
59
```

Figure 4.3: MATLAB Math and Data Plotting

4.2 Test Program Validation

Six validation tests for the test program are split between two sets of three test signals. The first time each signal is tested, it is generated onboard at the program's start using the *sin()* math function in the C math library. Theoretically, these signals are "perfect" in that there is no additive noise on them, their frequencies are precisely as defined in the equation, and there are no quantization errors from the DAC of the signal generator or the ADC of the microcontroller. The same signal equations are used in the MATLAB validation environment. Any errors in the algorithm will show since the inputs to the test program and validation environment should be identical.

The second set of validation tests uses two of the same signals and a third that simulates more real-world signal analysis. This second set of input signals is generated by an *Analog Discovery 2*, a portable USB-powered instrumentation system with an onboard dual-channel DAC with a 14-bit resolution. These signals were read in via one of the STM32 ADCs, and results from these signals are affected by noise and quantization errors that are not seen in the MATLAB results as they remain the same simulated noise-free signals from the previous set of validation tests. Therefore, every test for this program has two sample sets: those generated with 256-point DFTs and another generated with 4096-point DFTs. The 256-point DFT sample sets will simulate the kind of frequency resolution that will be expected from the main program.

4.2.1 Single Frequency Signal

4.2.1.1 Calculated Signal Validation Tests

The first validation test for the test program has a single-frequency sinusoidal input signal with a frequency of 200 Hz. For this first test, the sampling frequency is 500 Hz, and the sample size

is 256 samples with frequency bins about 2 Hz wide at the result. Fig. 4.4 shows the output at the test program's terminal. The total transform timing and operations statistics are below the data output. This terminal output is automatically logged into a .csv file and imported into the MATLAB test environment, as shown in Fig. 4.2. The resulting plots of this first 200 Hz test signal can be seen in Figs. 4.5 and 4.6 with Fig. 4.6 showing a direct comparison between the MATLAB and C test environments. Notice that both datasets result in an identical plot save for rounding errors on the order of less than a single micro-volt rms; both plots peak at the same frequency value, less than 1 Hz difference from the known frequency of the input signal.

Figure 4.4: Terminal Output for 200 Hz Single Frequency Test Validation

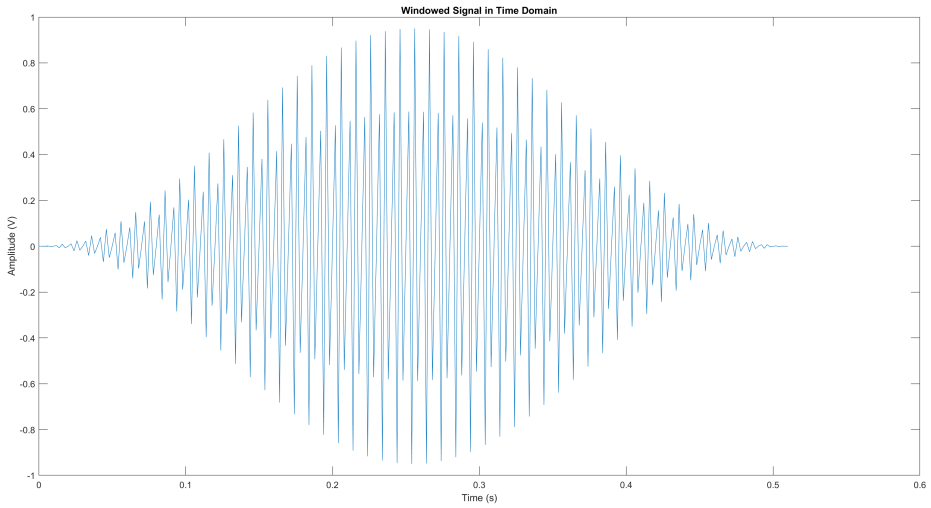

Figure 4.5: 200 Hz Windowed Test Signal

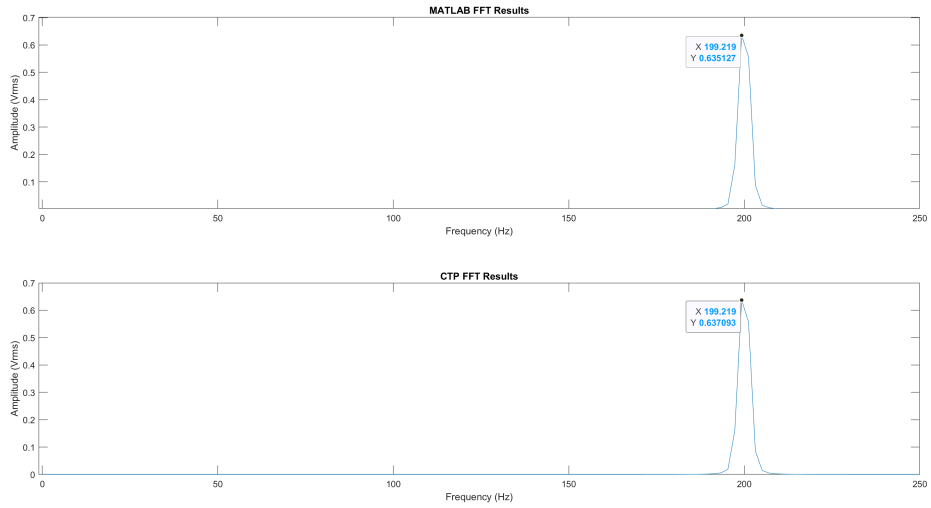

Figure 4.6: MATLAB and Test Program 200 Hz Results

Another test was run with the same test signal, trying to improve the accuracy of the resulting frequency spike. The number of samples is raised to the program max of 4096 while the sampling frequency remains 500 Hz. The results of this test can be seen in Fig. 4.7. The frequency spike is only minimally closer to the correct frequency at 199.951 Hz vs. the previous 199.219 Hz. However, for the added computation costs, thanks to the added sample size, it does not seem necessary for this low sampling frequency.

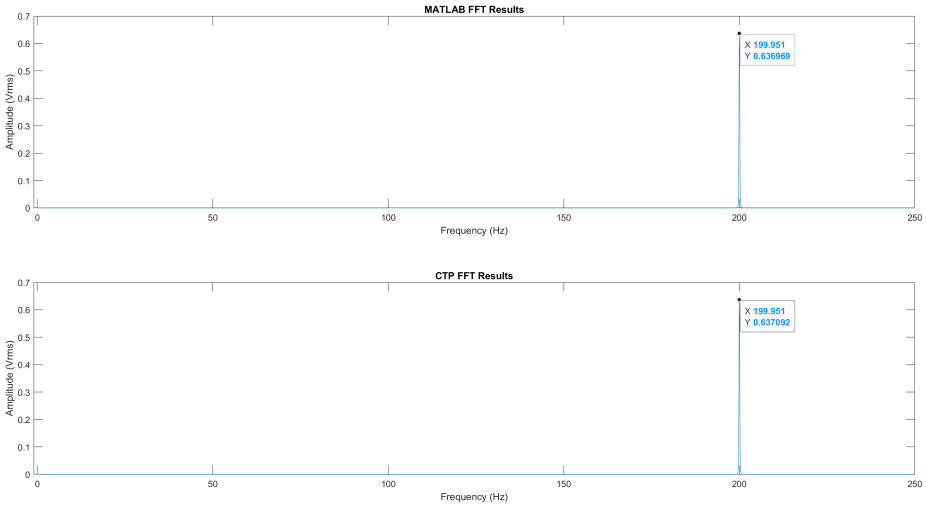

Figure 4.7: MATLAB and Test Program 200 Hz Results with 4096 Data Points

4.2.1.2 ADC Signal Validation Tests

Only a single channel of the signal generator was necessary for generating the 200 Hz signal, and the wire from the generator was connected directly to the input pin of the ADC. The results (Fig. 4.8) show a slightly more significant difference between the two results than was seen with the pure mathematical signals. The amplitudes vary by a few tenths of a volt rms, but

minimal signal distortion or noise is seen anywhere else in the test program signal, which is promising for future real-signaled tests.

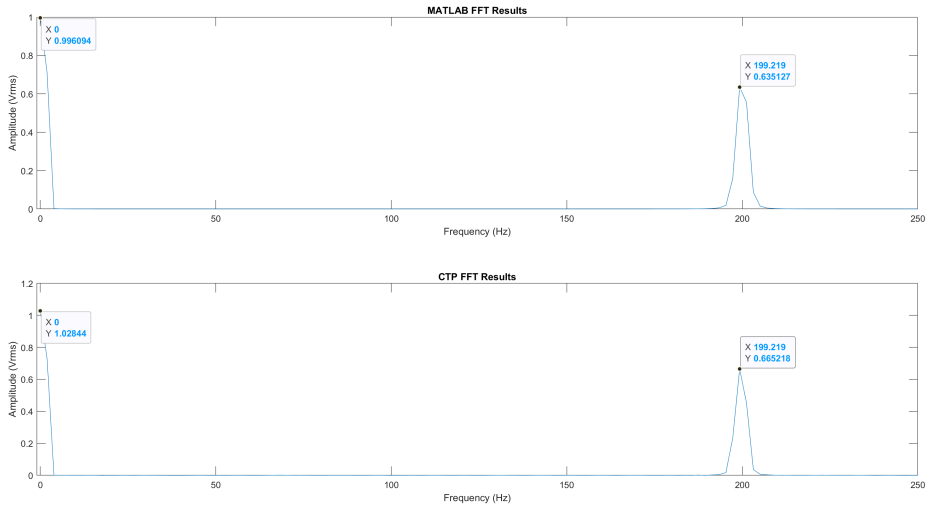

Figure 4.8: MATLAB and Test Program 200 Hz Real Signal

When bumping the number of samples up to the max, similar results to the mathematical signal can be seen in that the added samples do not make much of a difference in the accuracy of the frequency detected (Fig. 4.9). It is interesting that at 256 samples the calculated frequency of the ADC and mathematical signals match exactly but at 4096 samples the real-valued frequency does not match that of the pure signal test. This could reflect the imperfections of the signal generator as the signal is not guaranteed to be an exact 200 Hz as the pure signal is.

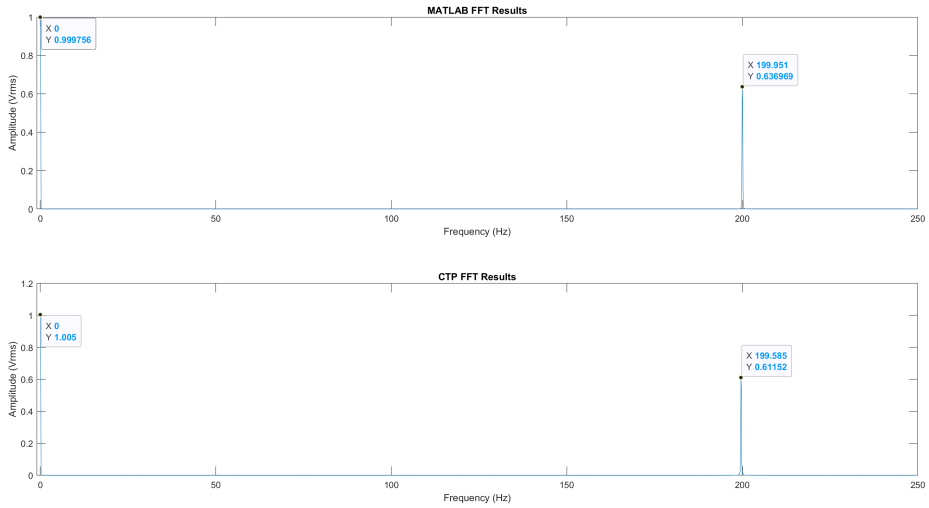

Figure 4.9: MATLAB and Test Program 200 Hz Real Signal, N = 4096

4.2.2 Dual-Tone Signal

The second test signal is a combination of two sinusoidal signals of different frequencies, amplitudes, DC components, and phases. The waveform can be represented as the following equation:

$$x = (sin(2\pi * 1000) + 1) + 0.5 * (cos(2\pi * 200) + 1) \tag{4.1}$$

The higher frequency content of the signal required an increased sampling frequency for the whole validation environment. In the following tests, the sampling frequency has been set to 5 kHz to remain above the Nyquist rate. In addition, the number of samples for this first iteration has been reduced to the original 256 samples, so each element of the output array is roughly 20 Hz wide; for a signal such as this with a significant frequency separation, the lower resolution

should not interfere with the results as long as there is not much noise in the signal.

4.2.2.1 Calculated Signal Validation Tests

Eq. (4.1) is implemented in both the MATLAB environment and in the program. Fig. 4.10 displays the dual-tone signal after application of the Hann window.

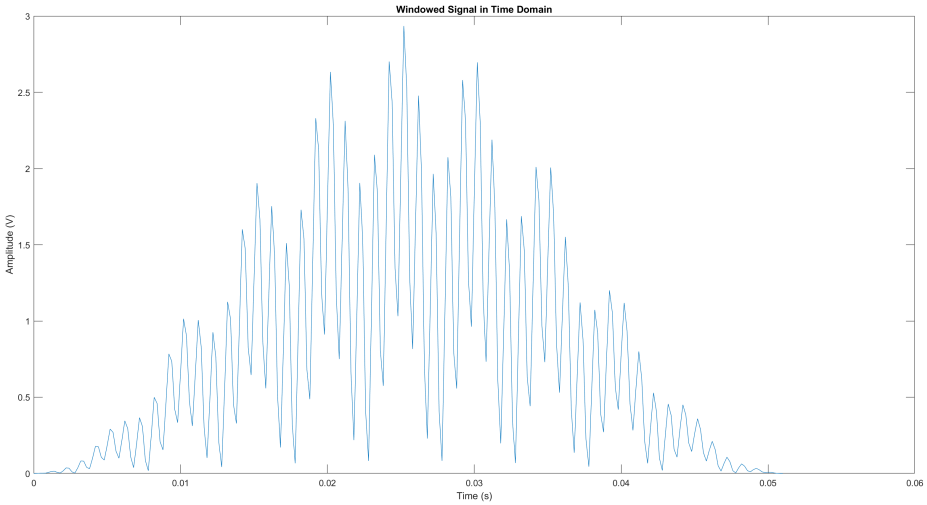

Figure 4.10: 200 Hz Windowed Sine Waveform

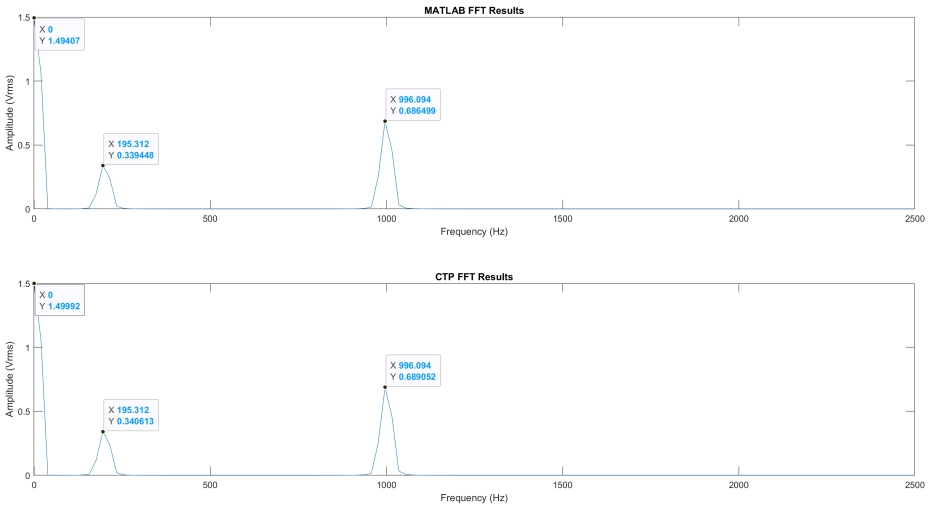

Figure 4.11: MATLAB and Test Program Dual-Tone Results

Comparing the subplots of Fig. 4.11, the test program and MATLAB results are incredibly close, only slightly varying in amplitude. A shift of only 4 Hz in the detected frequency spikes for the given frequency resolution is still reliably accurate for most applications. However, at 4096 samples, the results have an almost negligible amount of amplitude variance, and the frequency spikes' accuracy is much closer to the known signal components (Fig. 4.12). The increased accuracy makes sense due to the much higher frequency resolution with frequency bins slightly over 1 Hz wide.

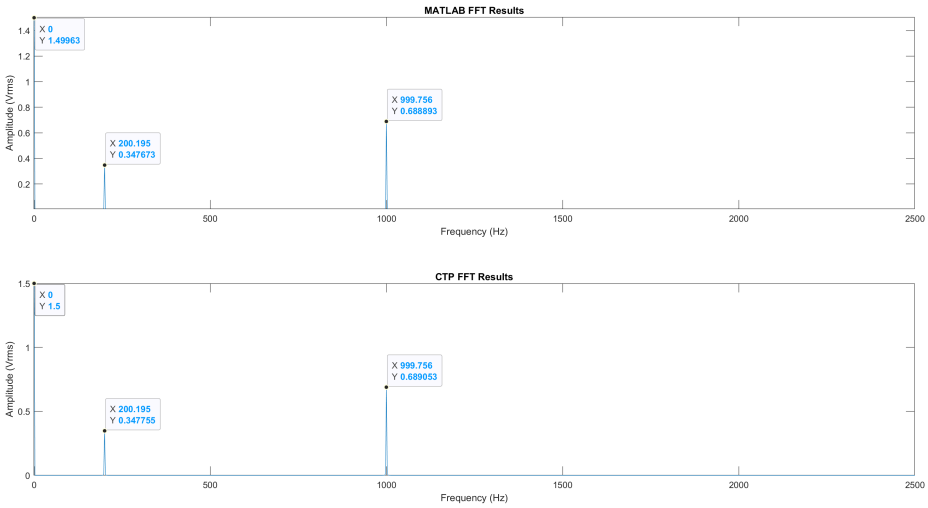

Figure 4.12: MATLAB and Test Program Dual-Tone Results, N = 4096

4.2.2.2 ADC Signal Validation Tests

Creating this signal with the signal generator required both channels with one of the frequency components on each. These signals were mixed in a breadboard, and a third wire carried the mixed signal to the ADC input pin. Introducing the breadboard into the system is another source of error, noise, and reflections that could distort the incoming signal. Thankfully, when looking at Figs. 4.13 and 4.14 it does not appear to be much noise, but there do appear to be more frequency spikes in the real-valued signal than there are frequency components in the signal.

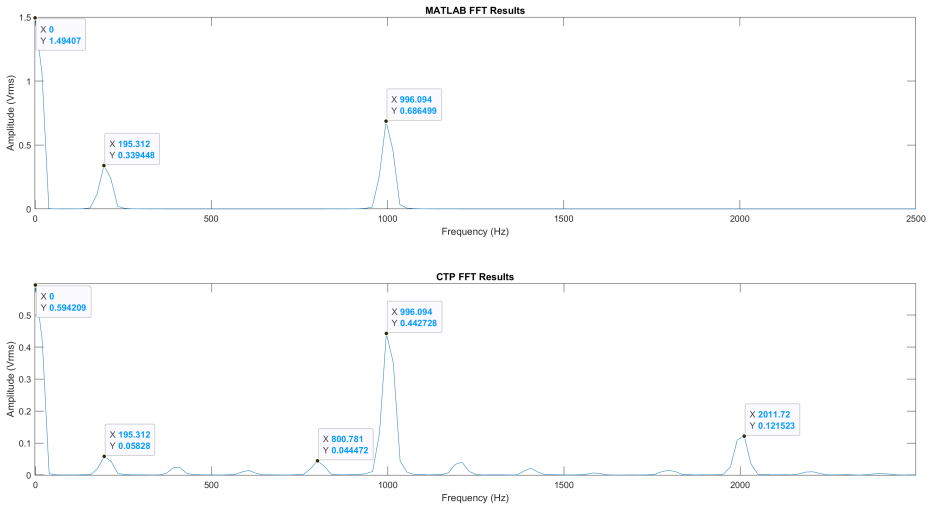

Figure 4.13: MATLAB and Test Program Dual-Tone Real Signal Results

Since these spikes appear at relatively the same frequencies, it is not a coincidence or random noise distortion. These spikes also did not appear when run with the pure signal in Fig. 4.11 or 4.12. The most significant spikes still appear at 1 kHz and 200 Hz, the target frequencies. The fact that the extraneous spikes appear at integer multiples of the low-frequency component of the signal around the high-frequency component illustrates that these may be harmonics introduced from how the input signal is mixed before entering the ADC.

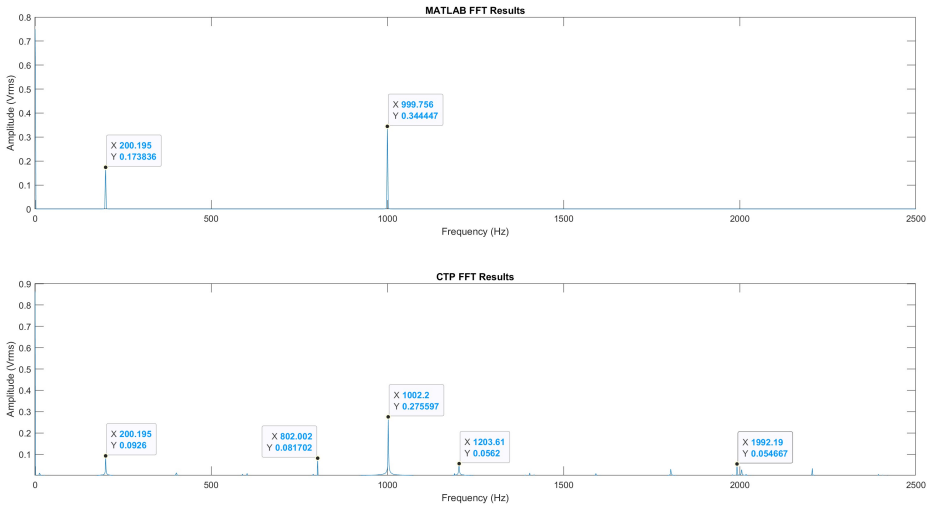

Figure 4.14: MATLAB and Test Program Dual-Tone Real Signal Results, N = 4096

For further inspection, the mixed signal was fed back into an oscilloscope to see how accurate the signal generator's reproduction of Eq. (4.1) is. Comparing Figures 4.15 and 4.16 it is clear that the signal seen by the ADC and the MATLAB environment are not the same though they carry the same frequency components. The signal appears distorted as it only reaches about half the amplitude created in MATLAB. This distortion is most likely the cause of the harmonics seen in the test program results.

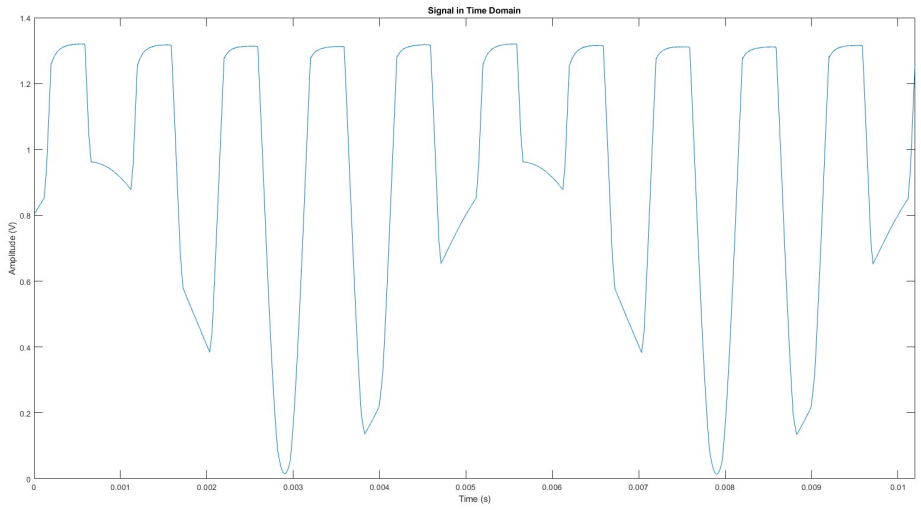

Figure 4.15: Real Dual-Tone Signal from Signal Generator

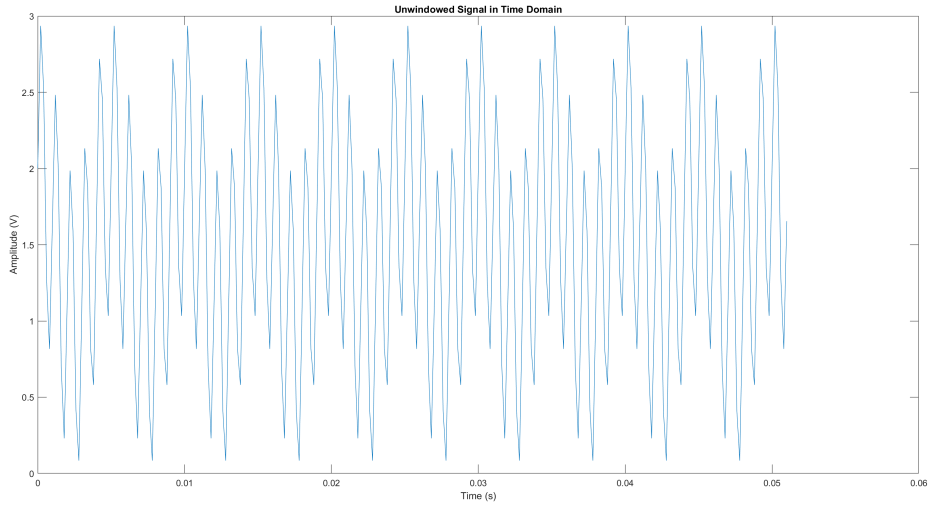

Figure 4.16: MATLAB Generated Dual-Tone Signal

4.2.3 Multi-tonal Musical Signals

The final set of tests differ in input signals as the three-tone signal used in the pure mathematical signal tests could not be recreated with a two-channel signal generator. The signal, instead read by the ADC, is a wav file of a chord in the key of C. The signal generator can playback waveforms from audio files through individual channels. These tests even negated the need for the breadboard and passive signal mixing that caused harmonics in the dual-tone validation tests. A windowed version of the pure mathematical signal used in the below tests can be seen in Fig. 4.17.

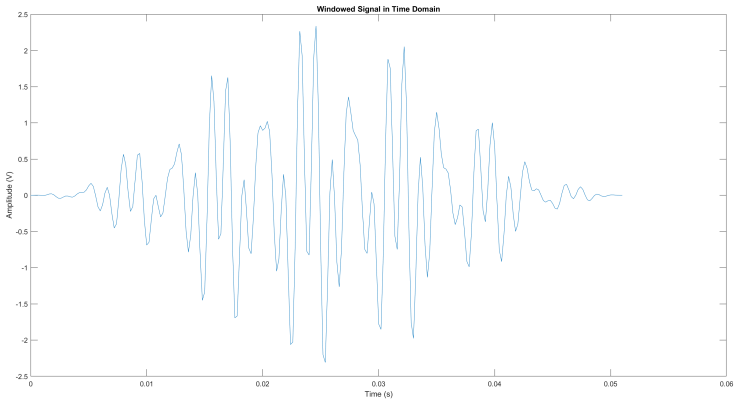

Figure 4.17: C Chord Triad Test Signal

4.2.3.1 Calculated Signal Validation Tests

The three-tone signal used for these tests consists of the frequencies for the notes C_4, E_5, and G_5 or 261.63, 659.25, and 783.99 Hz, respectively, which together form a standard C Major triad. The frequencies of the E and G notes are relatively close together when compared to the previous test signals used in this paper. Therefore, these signals can blend at low enough

frequency resolutions, and some information will be lost. An example of this can be seen in Fig. 4.20, where only 256 samples are used in the DFT calculation, but the sampling frequency is 44.1 kHz which is a standard audio sampling frequency. This ratio creates frequency bins that are 172 Hz wide. This is why the signal's frequency components get cannibalized within almost a single lobe.

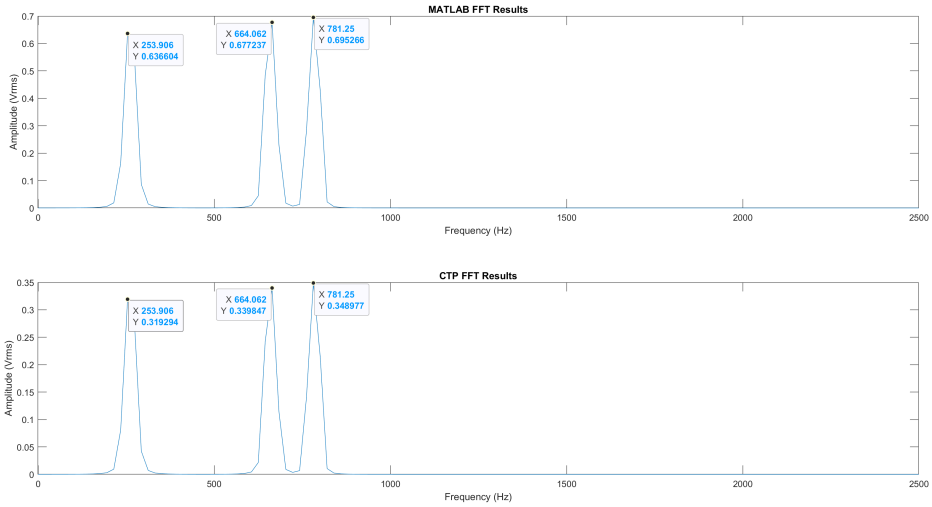

Figure 4.18: MATLAB and Test Program Triad Results

The total 44.1k sampling frequency is not necessary for this signal because of the relatively low-frequency content, but the simulation of real-world signal analysis begins in these validation tests. The results in Fig. 4.18. Of course, the results shown in Fig. 4.19 speak volumes about the importance of sample counts. The frequency spikes are close to representing perfect impulse spikes, so the ability to differentiate very close frequency components within an input signal is very high. The frequency positions are also highly accurate, only missing the exact frequencies by less than a full hertz.

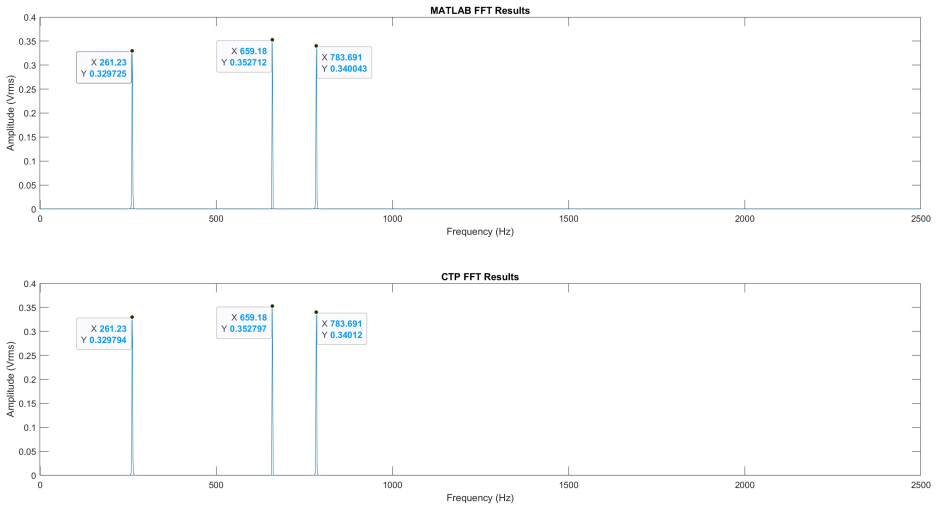

Figure 4.19: MATLAB and Test Program Triad Results, N = 4096

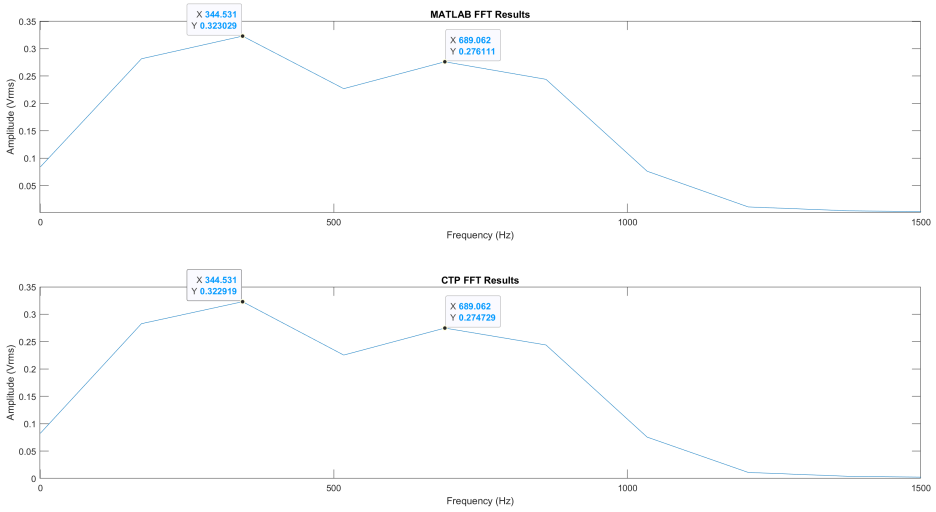

Figure 4.20: FFT of Triad Signal, N = 256, Fs = 44.1k

The total 44.1k sampling frequency is not necessary for this signal because of the relatively low-frequency content, but the simulation of real-world signal analysis begins in these validation tests. The results in Fig. 4.20 are not promising for the main program. If it is to be used to sample high-resolution audio signals, much of the frequency content will be too distorted to identify correctly. Using 4096 samples instead of only 256 did provide much better results. The frequency resolution is only 10 Hz, and the plot in Fig. 4.21 is much more comparable to the results found in Fig. 4.18.

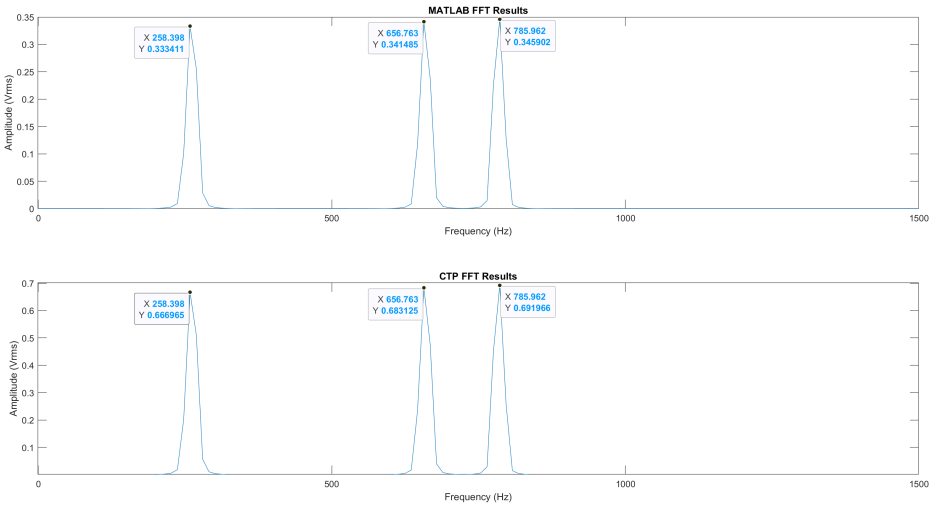

Figure 4.21: FFT of Triad Signal, N = 4096, Fs = 44.1k

4.2.3.2 ADC Signal Validation Tests

As mentioned above, the triad signal used for the test in Section 4.2.3.1 could not be recreated with the dual channels provided by the waveform. Instead, a wav file containing a minute of audio is played directly from channel 1 of the signal generator into the ADC input pin.

The segment of the signal seen by the ADC can be imported into MATLAB to compare the results between both programs. The signal's frequency content is only known as multiple tones forming a chord in the key of C, but the exact frequencies of these tones are unknown. Because the audio file is sampled at 44.1 kHz, using a lower sampling frequency to calculate the DFT resulted in an inaccurate spectrum. Therefore, using 256 points in the DFT ended in similar results to those seen in Fig. 4.20. The plots shown in Fig. 4.22 support this. Interestingly, a secondary peak is detected in the test program though it is hard to say what exactly the test program might be detected with as much information is lost.

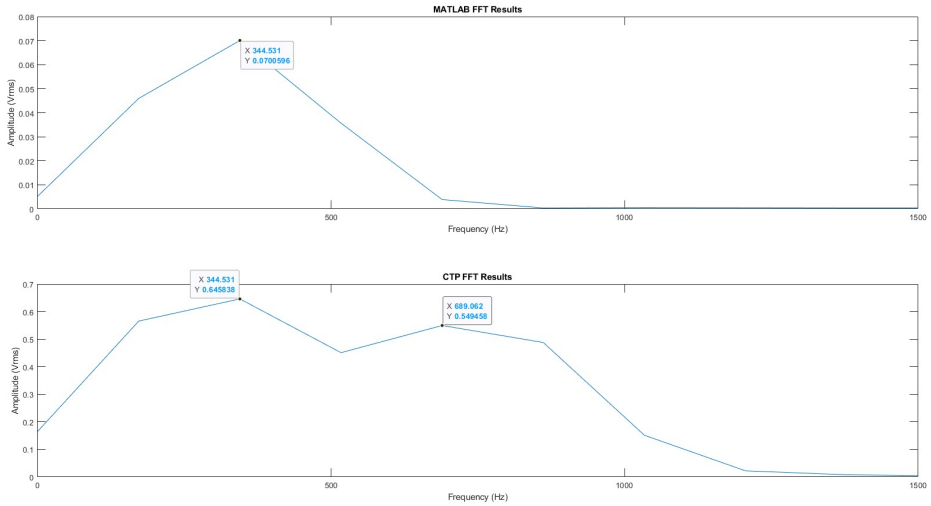

Figure 4.22: Frequency Content of Audio File, N = 256

Using 4096 samples provided some results that may help deduce the signal's frequency components. The program results contain many more points than the MATLAB signal. These points could be harmonics, as seen in earlier tests introduced from the signal generator's and ADC's imperfections. All of the components found in the top plot of Fig. 4.23 can be found

as spikes in the results from the test program though they may not be as accurate as is the case with the smaller peak at 322 Hz from the MATLAB results. The spike at 333 Hz in the test program results corresponds to this frequency though it seems to have shifted, possibly due to side lobe components introduced by the Hann window. There is also a DC component in the real-valued signal, which does not appear in the MATLAB environment; this will cause the rest of the frequency peaks in the spectrum to have lower amplitudes since this energy is shared across another component.

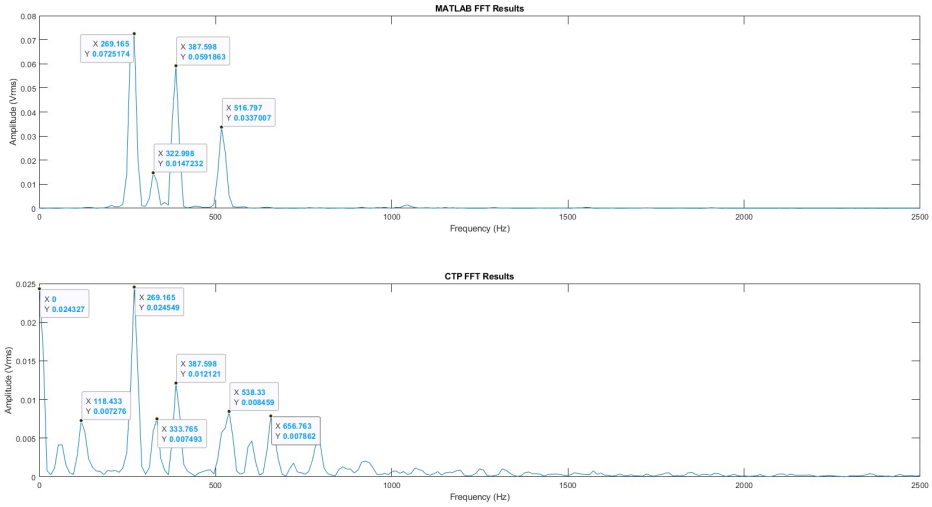

Figure 4.23: Frequency Content of Vocal Audio File

Fig. 4.24 shows the spectrum plot of the entire waveform of 470400 points. This super high-resolution FFT presents the frequency content of the entire signal instead of a small time window of the signal. Observing the spectrum shows many smaller spikes on either side of the major frequency components, which must come from slight variations within a note's frequency over time. It can be seen, however, that four main notes compose the chord. These

spikes correspond to the notes of C_4, E_4, G_4, and C_5, so they form a C major chord using the same three notes as the signal in the triad chord just different octaves of these notes. The E and G notes in this chord are lower frequency than those used in the triad chord of the previous tests, and the second C_5 note included at the top of the chord is a whole eight steps and, therefore, almost 300 Hz above the C_4 note at the bottom of the chord.

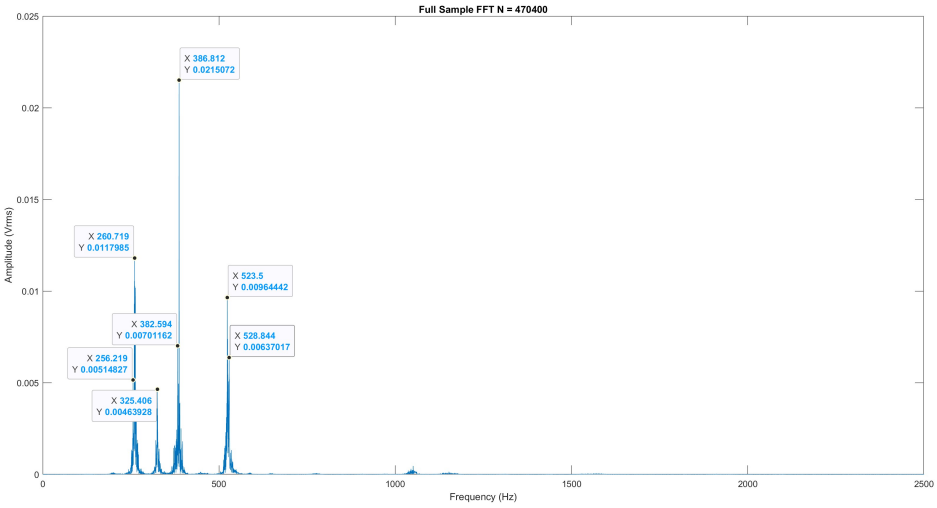

Figure 4.24: Frequency Content of Audio File Using Full File

4.3 Main Program Validation

The results of the previous section are meant to inform what might be seen in the tests of the main program, especially the final multi-tone signal validation tests that were completed for the test program. Because of the memory constraints of the main program, only a max of 256 samples are possible for each time window. However, before the main program produces any

spectrograms, the system timing and data path must be verified to ensure that all collected data is successfully processed and printed to the terminal for collection. Dropped data packets between tasks would be a significant issue as whole spectra could be lost during the run time of a program. After the validation of the data path, the continuous flow program will be tested using several frequency sweeps, which should produce linear results in a spectrogram.

4.3.1 Data Path Validation

In order to first check that all packets make it through the program, several counters are added throughout the program. Each task receives a new counter which is incremented every time a package arrives or leaves the task through one of the queues. The *adc_Task* does not need another counter as it already keeps the master count of how many samples windows have been captured by the ADC, and the task only sends data out to the other tasks. Both the transform and analysis tasks should increment their counters twice since they both send and receive packages through the queues, and the *uart_Task* will also only increment its counter once since it only receives data. However, there is another counter found in the *uart.c* file I.2, which is incremented in the UART Tx Callback Function to track how many packages have been printed to the terminal as the final leg of the data path. This test calls for running the program with ever-increasing package limits.

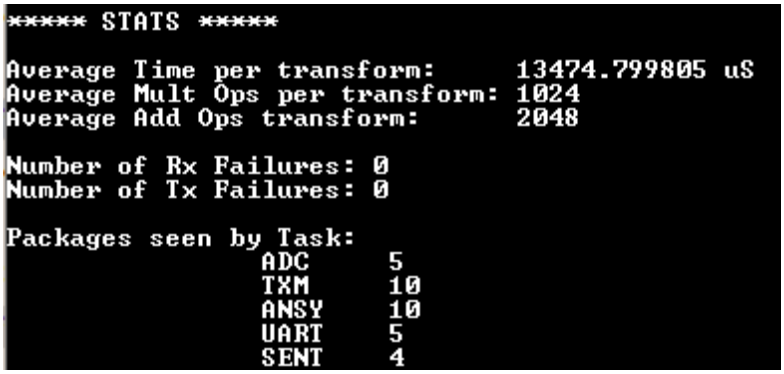

Figure 4.25: Data Path Validation Test 1: Limit = 5

Using only 5 packages it can be seen in Fig. 4.25 that there are no failures in delivering the data between tasks. The analysis and transform tasks send and receive five packages, but the program's bottleneck is noticeable even from this limited number of packages. There were only a total of four packages sent to the terminal, which is a consistent pattern across all data path validation tests (Figs. 4.26 and 4.27). At 35 packages, only a single package is still missing from the output. However, with 150 packages sent through the program at the highest test, two were missing at the output. Therefore, this issue scales as the requested number of packages increases. The number of samples per package was reduced to 128, and the results show that reducing the number of samples can effectively reduce bottlenecks made by printing to the terminal.

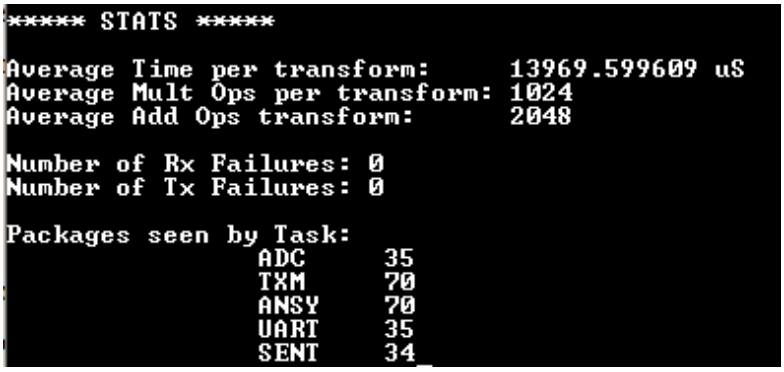

Figure 4.26: Data Path Validation Test 2: Limit = 35

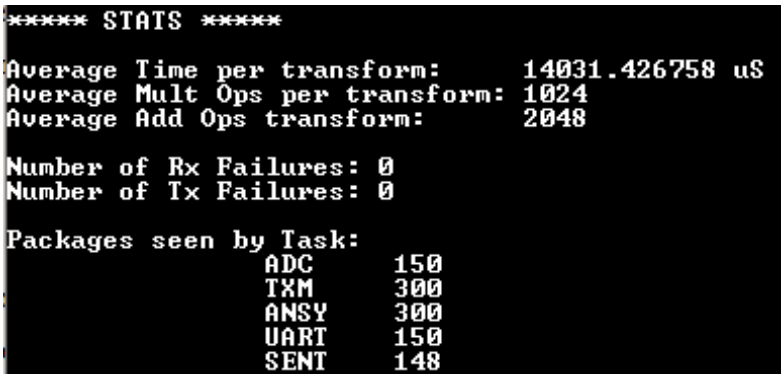

Figure 4.27: Data Path Validation Test 3: Limit = 150

Observing the same tests performed for $N = 256$ Fig 4.28 shows that all requested packages can be reliably transmitted over UART by reducing the sample count. Though, by looking at Figs. 4.29 and 4.30 , the printing is done so quickly that some spectra are printed twice to the terminal screen. The UART task does not see these extras because the packages sent counter is incremented in the UART Tx Callback function. Because the UART DMA is running in

circular mode, it will automatically begin re-sending the data it just finished sending until another request or update is triggered. An attempt to operate the UART Tx line in Normal DMA mode was made but substantially slowed the program so much that only two packages were sent over UART of the five requested.

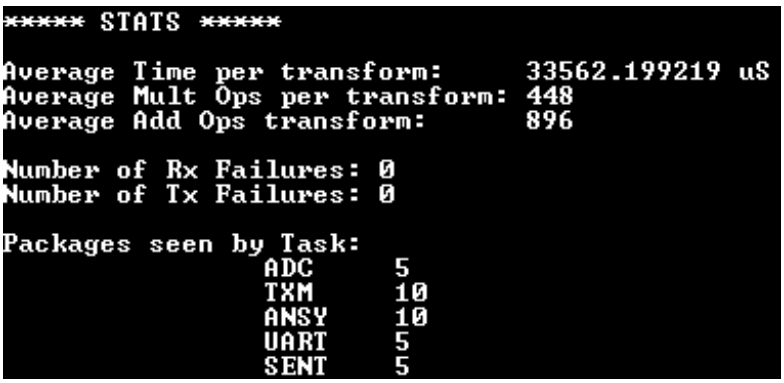

Figure 4.28: Data Path Validation Test 1: Limit = 5, N = 128

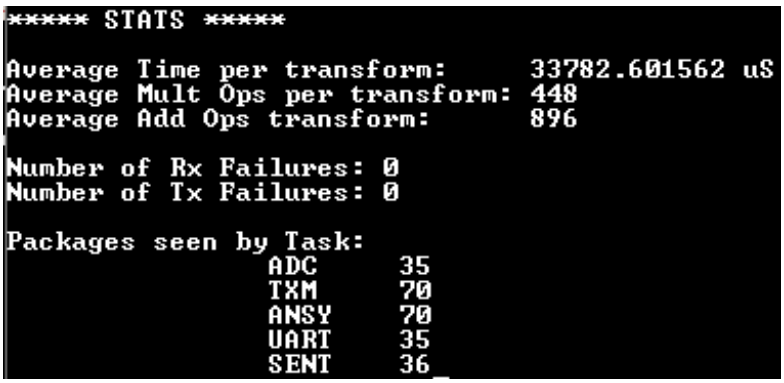

Figure 4.29: Data Path Validation Test 1: Limit = 35, N = 128

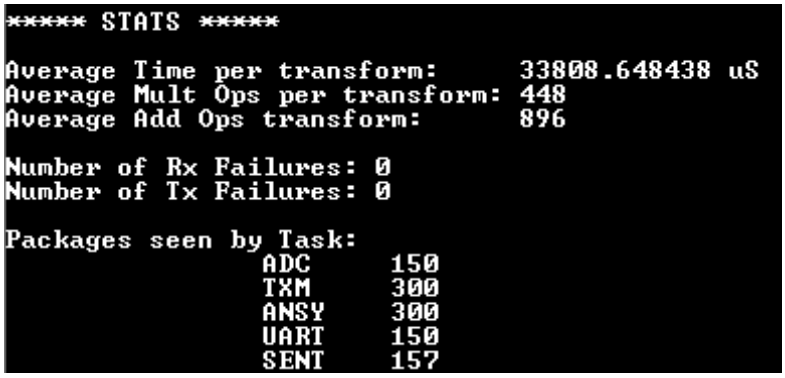

Figure 4.30: Data Path Validation Test 1: Limit = 150, N = 128

4.3.2 Terminal Printing Validation

Knowing that all the packages can make it through the system, the following tests ensure that each data point within a package is correctly sent without corruption or overlap. All math sections of the code are commented out, and instead of a raw ADC value, the RWM buffer is filled with the current index value. These index values are transferred through the program and printed at the output. Only two tests are performed for the index checking, a request for four packages and a request for 12 packages. The resulting prints are plotted in Excel and form a saw-tooth waveform. It can be seen that there are no inconsistencies in the index values that were printed. No data was overwritten or corrupted though this test does not account for doubled packages since the index values constantly repeat. These tests are run with the max 256 sample buffers as they had the issue with missing data packages. The results for each test are seen in Figs. 4.31 and 4.32.

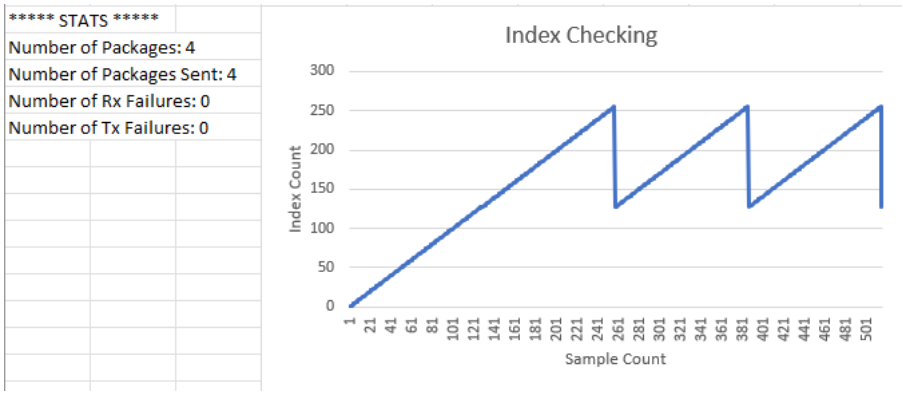

Figure 4.31: Terminal Printing Results, Limit = 4

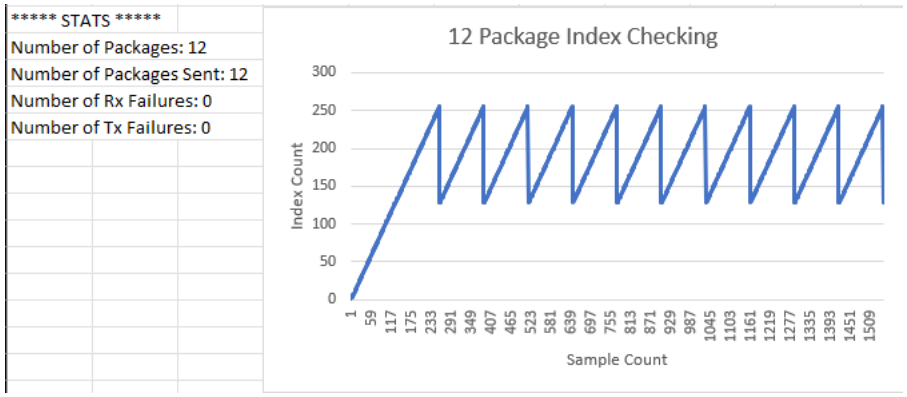

Figure 4.32: Terminal Printing Results, Limit = 4

4.3.3 Time-Varying Signal Analysis

Two different signals are under test in the following section; both signals contain linearly varying frequencies, with the first starting at 50 Hz and climbing to a peak of 2 kHz before starting over, and then the second is a low-frequency signal which begins at 1 Hz and reaches a

max of 200 Hz. Several factors are changed throughout the tests presented below, most notably the speed at which these frequency changes occur.

The first test sees the program configured with windows of 256 samples and a sampling frequency of 5 kHz to remain above Nyquist. The program is asked to process 35 windows of the signal as it reads through the ADC. The data path results of this request can be seen in Fig. 4.33; notice that only a single package is dropped somewhere between the analysis and the read_Task. However, the resulting spectrogram can be seen in Fig. 4.34, which has three axes: time, frequency, and Power Spectral Density (PSD) in $Vrms^2/Hz$. The plot has been reversed to better show the resulting linearly increasing frequency pattern. Otherwise, it is hidden behind the wall representing a constant DC component on the signal. The patterns seen in the spectrogram are interesting as the frequency spectrums are moving, but there are more sweeps than should be possible given the timescale, and they are not in the correct direction. The same signal read through the ADC is also exported to MATLAB, and the resulting spectrogram computed by MATLAB can be seen in Fig. 4.35, which does not have a DC component since one was not defined in the signal generator.

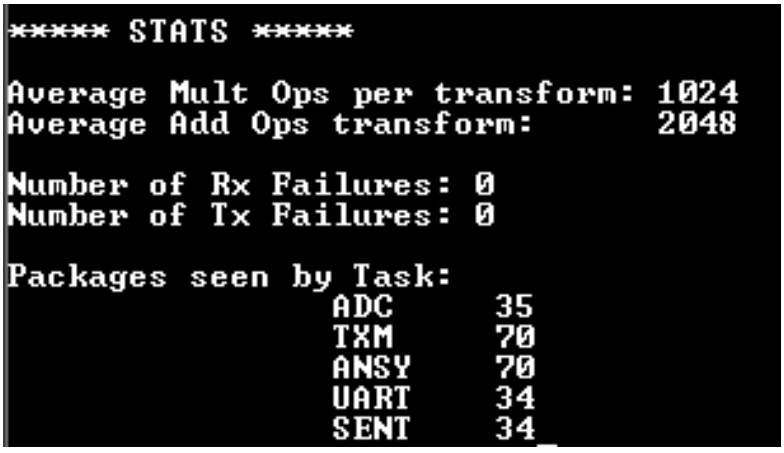

Figure 4.33: Program Statistics for Test 1

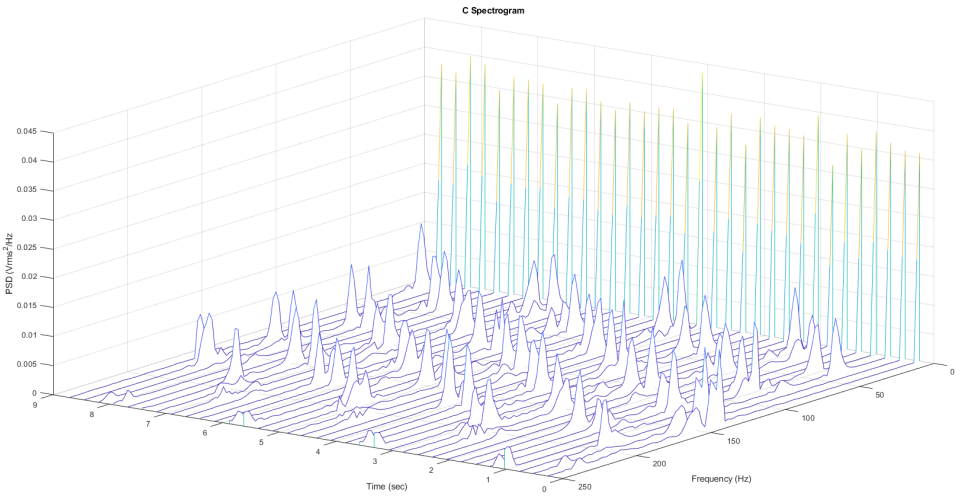

Figure 4.34: C Program Spectrogram of 50Hz-2kHz Sweep in 500 ms

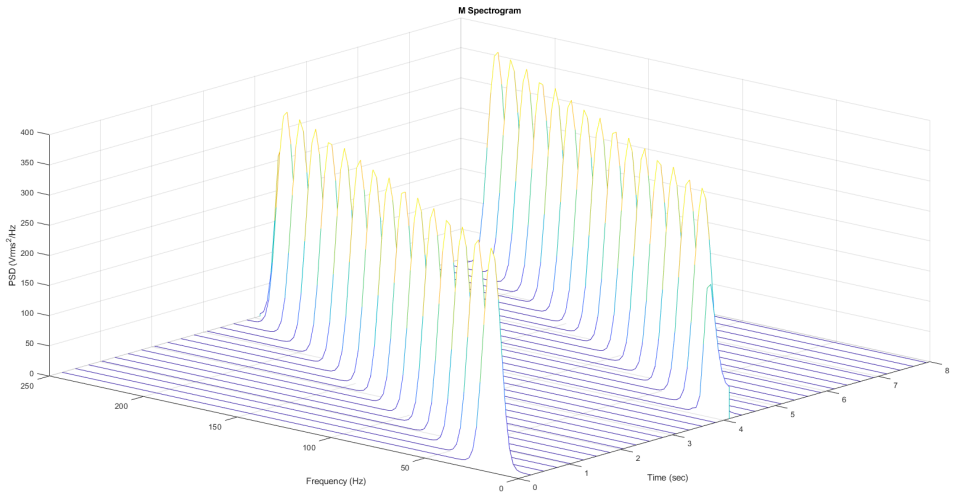

Figure 4.35: MATLAB Spectrogram of 50Hz-2kHz Sweep in 500 ms

This begs a fascinating question about where the DC component of the C Program results originates. If there is no specified DC offset to the signal, as seen by the MATLAB results,

then it is most likely a symptom of the ADC converting all values into unsigned integers. This would mean that any signal that crosses the x-axis will not be accurately converted and produce inaccurate results. This may also explain the extra sweeps in Fig. 4.34. As can be seen in Fig. 4.35 there should only be two resulting sweeps in a second long timescale, each running from 50 Hz to 2 kHz.

The signal generator is reconfigured to produce the same sweep but now with an amplitude of 500 mV with a 500 mV offset to investigate the effect of the DC offset. The signal is also stretched so that a full sweep across the signal's frequency range takes five seconds. The spectrogram of this signal from the C Program can be seen in Fig. 4.36, where the analysis began part-way through a sweep. It can be seen that the signal frequency climbs to around 2 kHz, stopping somewhere closer to 2050 Hz, before starting again at 50 Hz, and the frequency increases linearly until the time window ends. The mesh around all frequency peaks is almost flat meaning there was very little noise on the signal, and the program has created an accurate-looking spectrogram. In a weird twist, Fig. 4.37 shows MATLAB's attempt at processing this signal, and there are no discernable frequency peaks or patterns.

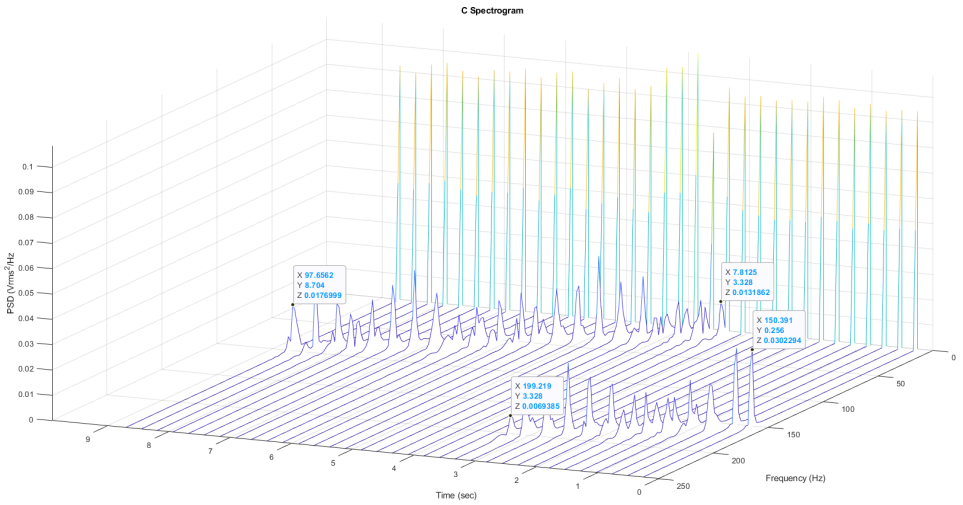

Figure 4.36: C Program Spectrogram of 50Hz-2kHz Sweep in 5s with 500 mV offset

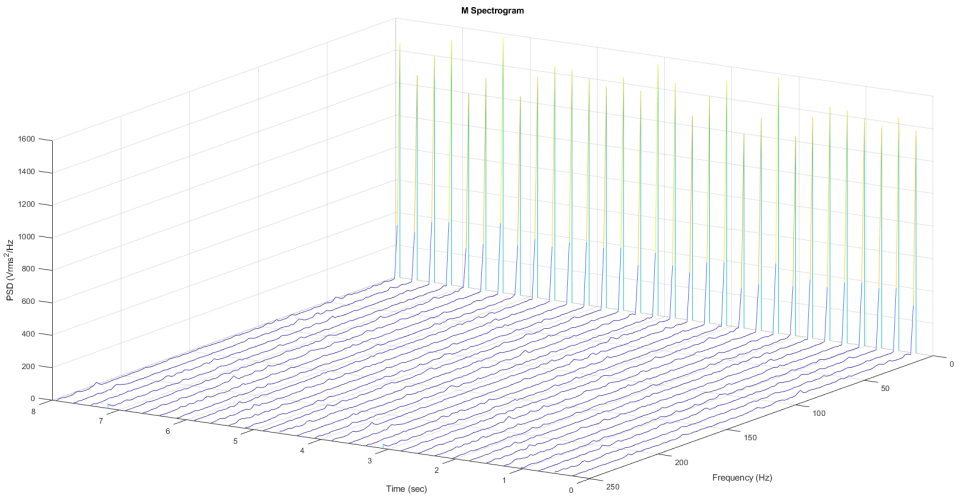

Figure 4.37: MATLAB Spectrogram of 50Hz-2kHz Sweep in 5s with 500 mV offset

Since including a DC offset improved the accuracy of the program, the next set of tests using the smaller frequency range also uses the same 500 mV offset and amplitude. The

lower frequency range allows the program sampling frequency to be reduced to 500 Hz, so each calculated spectrum has frequency bins of roughly 2 Hz in width. Unfortunately, the lower sampling frequency also meant data acquisition took longer, and the program had to be adjusted. Fig. 4.38 shows that the circular DMA implementation printed over double the number of user-requested packages. Adjusting the callback so that the DMA process is stopped each time the function is called, halting the excess printing without causing any data path errors (Fig. 4.39). This method is kept for the remainder of the tests.

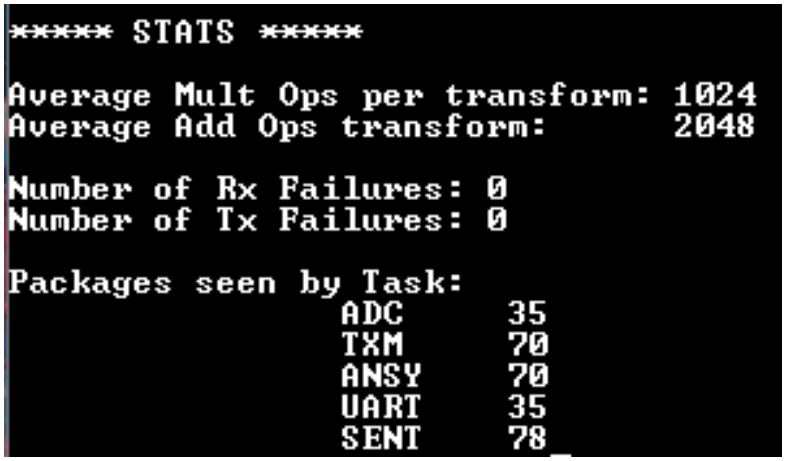

Figure 4.38: Program Statistics for Test 3 Before Adjustment

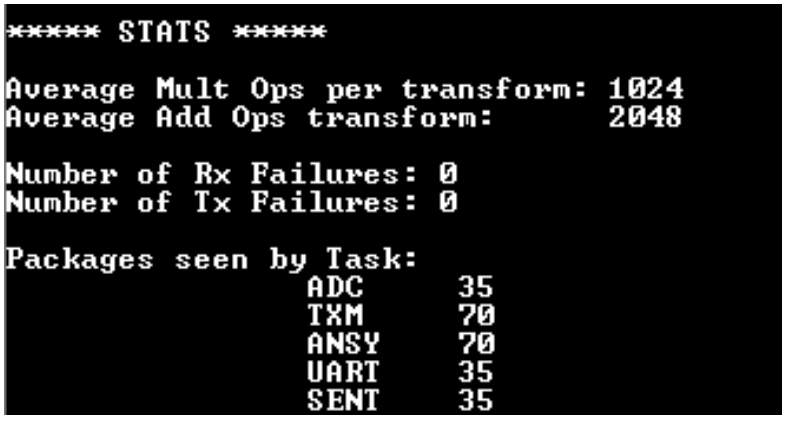

Figure 4.39: Program Statistics for Test 3 After Adjustment

The spectrogram created by the C Program is seen in Fig. 4.40 with results that do not have any clear frequency peaks or frequency change patterns. Similar to the plot in Fig. 4.34 when testing the last signal varying in 500 ms. The lack of accuracy may be due to the speed at which the signal varies. There is a very noticeable difference when compared to the MATLAB-produced results of the same signal (Fig. 4.41) . However, even the MATLAB-produced spectrogram does not cover the entirety of the frequency range, with the sweep seeming to restart after climbing to only around 50 Hz.

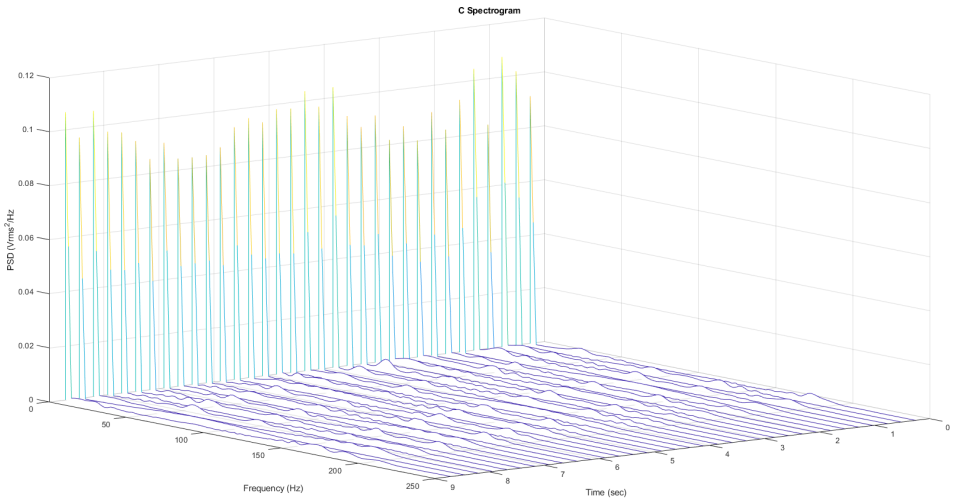

Figure 4.40: C Program Spectrogram of 1Hz-200Hz Sweep with 500 mV offset

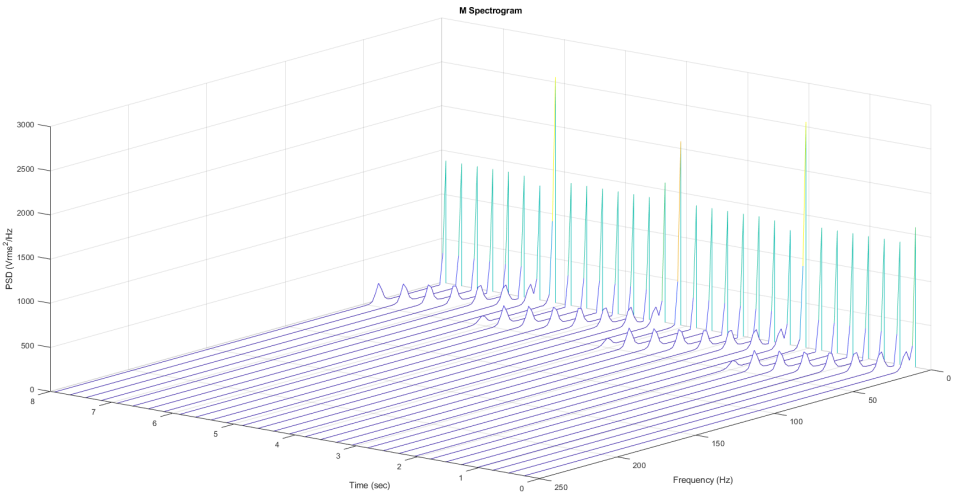

Figure 4.41: MATLAB Spectrogram of 1Hz-200Hz Sweep with 500 mV offset

As with the wider band signal from the first tests, this signal was elongated so a full sweep across the frequency range would take 5 seconds. The output plots seen from this change seem to point to the speed of the frequency changes being the culprit in the inaccurate results in Figs. 4.34 and 4.40. A clean spectrogram can be seen in Fig. 4.42 starting near 1 Hz and ending just shy of 200 Hz before starting over. All spectrums have a single peak which vary with the x and y-axis. A very similar plot is seen in Fig. 4.43 so both programs seem to be affected by the frequency change speeds.

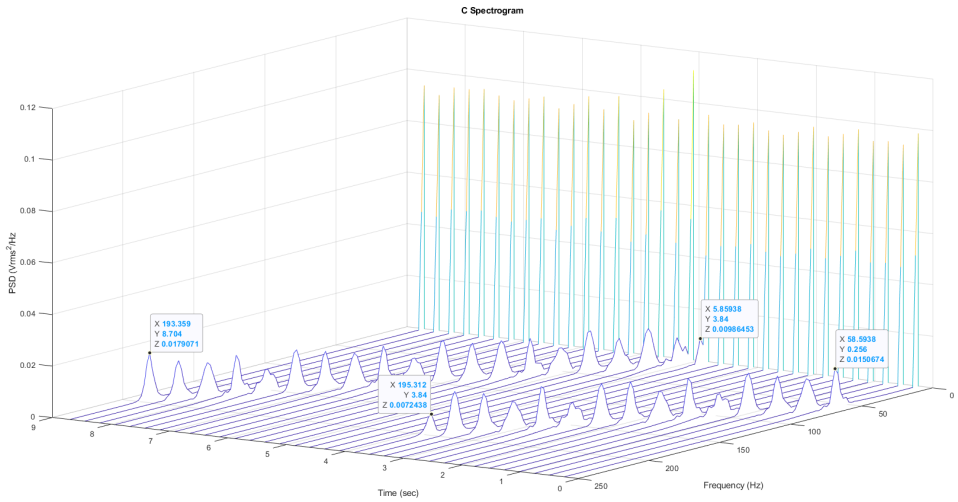

Figure 4.42: C Program Spectrogram of 1Hz-200Hz Sweep in 5s with 500 mV offset

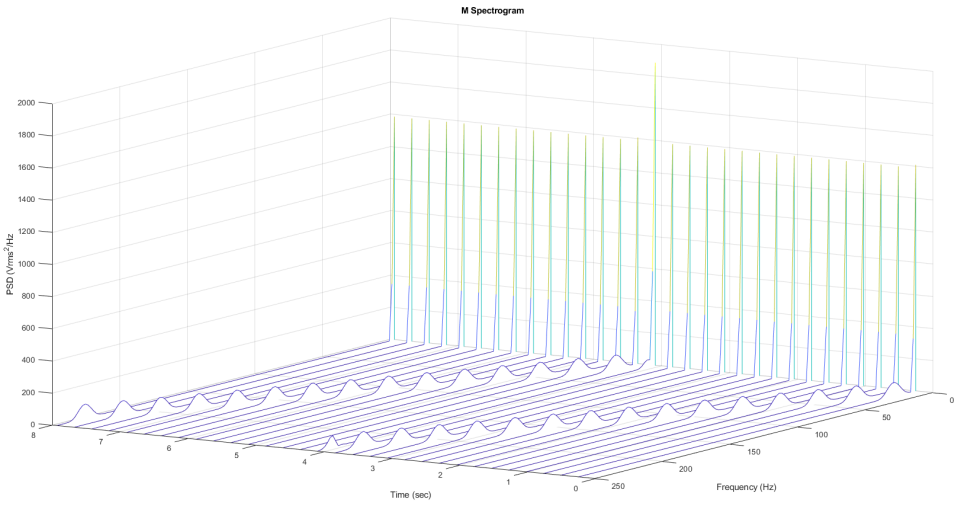

Figure 4.43: MATLAB Spectrogram of 1Hz-200Hz Sweep with 500 mV offset

Chapter 5

Conclusion

From the experimental results in Chapter 4, both sets of programs have been validated to work within certain conditions. The FFT algorithm implementation is accurate when compared to MATLAB's built-in functions, and real-valued signals, as well as purely mathematical signals, are capable of being analyzed. Without the RTOS, the FFT program can operate with 4096 samples in a transform. This allows for functional transformations at sampling frequencies up to at least 44.1 kHz, though signals composed of tightly clustered frequencies may start to see some degradation in the resulting spectrum.

Unfortunately, using an RTOS on such a small system creates a distinct lack of memory availability. This limitation caps the type of signals that the system can analyze effectively, with only 256 samples maximum per transform. As a result, analyzing any audio signals results in very low-quality spectrum's with almost no separation of frequency content. Through experimentation, it was seen that signals which vary too quickly do not produce accurate results, which may come down to the configured sampling rate. Further testing will need to be done to establish the bounds of frequency change speed. However, it has been proven that this system can do online analysis for slow-moving signals.

Having only a single core also to store, manipulate, analyze, and print values causes a bottleneck at the end of the data path since the maximum baud rate of the system is 115200 bits/second. However, if this system were embedded somewhere where UART communication in the data pipeline was unnecessary, the results could be more quickly sent over another communication interface such as SPI. This would remove the significant throughput bottleneck seen at the end of the program and could also free up memory space since the arrays necessary for the float-to-string conversions would not be necessary. Additionally, a dual-core system could split the work and have the printing process offloaded to operate in the background so all packages can be better guaranteed to be sent through the terminal. An additional core would allow for more sophisticated analysis and processing techniques with the added benefit of more memory. The additional memory would significantly increase the capabilities of the system and the types of signals that could be accurately analyzed.

5.1 Project Conclusion

The FFT is one of the most important algorithms ever discovered. Its applications are far-reaching, while its implementation is relatively straightforward. In this paper, an online signal analysis system was built utilizing the original radix-2 Cooley-Tukey algorithm and a Real-Time embedded environment. This system was shown to work within specific boundary conditions through parallel verification with a MATLAB testing environment. Data is transferred through a pipeline-like path with only a single throughput bottleneck at the very end when the results must be communicated to the testing environment.

This project has provided a deep dive into the history and theory behind Fourier Analysis and the FFT algorithm while providing a chance to apply it directly and find its uses and drawbacks. The memory requirements needed for complex computations do not lend themselves to

embedded systems which are often limited in memory. Including a real-time operating system in the project allowed for the further development of critical embedded skills and improved the reliability of the system's data path.

Through testing of the currently proposed system, even with its minimal memory allowance, it can accurately analyze time-varying signals at up to 5 kHz. Furthermore, the analysis results can be cleanly exported with a resolution of 1 μV and easily imported into any data analysis software in .csv format. Furthermore, non-continuous analysis using the same algorithms is proven effective up to 44.1 kHz, meaning that this system has the potential to analyze sound signals with alterations.

5.2 Future Work

Future work will improve the Fourier analysis of the system through different FFT implementations and eventually increase the system's effective bandwidth to support audio sampling frequencies. This work will begin with characterizing signals that the system can accurately analyze. By clearly defining the bounds of the system, future iterations could offer more targeted improvements in enhancing and expanding its capabilities. The system would greatly benefit from being implemented into a system that utilizes another communication method, even though a UART is convenient for testing. If a UART continues to be necessary, perhaps moving to a dual-core system would significantly improve its use since software running on a processor core is directly involved in feeding data to the UART. The board being used is one of the cheapest on offer by STM, with only 96 kB of RAM available to the program. At 256 samples, the program utilizes 90% of this memory space. Moving to a larger MCU could significantly increase the system's capabilities. Even just 512 kB of RAM is a 5-fold increase in memory size and would allow for many more samples per transform to increase the adequate

bandwidth.

The system should also be refactored to be more easily configurable by the user without reflashing the system to change any program features. The benefit of using a software-based FFT system is the ability to make quick and easy modifications. However, the current code-base only allows for a little user configuration. Attempts to improve the FFT implementation should also be made by incorporating other algorithms or experimenting with new techniques to reduce operational complexity. Allowing for a multi-dimensional FFT could also expand this project from just a signal analyzer to also being a partial differential equation solver.

References

[1] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex Fourier series," 1965. [Online]. Available: https://www.semanticscholar.org/paper/0e6beb95b5150ce99b108acdefabf70ccd3fee30

[2] M. Heideman, D. Johnson, and C. Burrus, "Gauss and the history of the fast fourier transform," *IEEE ASSP Magazine*, vol. 1, no. 4, pp. 14–21, 1984.

[3] A. DomÃnguez, "Highlights in the History of the Fourier Transform [Retrospectroscope]," *IEEE Pulse*, vol. 7, no. 1, pp. 53–61, 2016.

[4] C. F. Gauss, "Nachlass: Theoria interpolationis methodo nova tractata," *Carl Friedrich Gauss Werke*, vol. 3, pp. 265–327, 1866.

[5] I. J. Good, "The interaction algorithm and practical Fourier analysis: an addendum," *Journal of the Royal Statistical Society. Series B*, vol. 22, pp. 372–375, 1960.

[6] P. Rudnick, "Note on the calculation of Fourier series," *Mathematics of Computation*, vol. 20, no. 95, pp. 429–430, 1966.

[7] G. C. Danielson and C. Lanczos, "Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids," *Journal of the Franklin Institute*, vol. 233, no. 5, pp. 435–452, 1942.

[8] J. Cooley, P. Lewis, and P. Welch, "Historical notes on the fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2, pp. 76–79, 1967.

[9] L. H. Thomas, "Using a computer to solve problems in physics," *Applications of digital computers*, pp. 44–45, 1963.

[10] H. H. Goldstine, "A History of Numerical Analysis from the 16th through the 19th Century." 1976, pp. 249–253.

[11] E. Brigham, *The fast Fourier transform and its applications*, G. Szyferblatt, Ed. Prentice Hall, feb 1988, vol. 26, no. 06. [Online]. Available: https://www.semanticscholar.org/paper/70b187ac64e899219660684f25f1df1c78497ab8

[12] C. Yang, Y.-z. Xie, L. Chen, H. Chen, and Y. Deng, "Design of a configurable fixed-point FFT processor," in *IET International Radar Conference 2015*, 2015, pp. 1–4.

[13] R. Rathore and N. Kaur, "Comparison Study of DIT and DIF Radix-2 FFT Algorithm," *International Journal of Computer Applications*, vol. 150, pp. 25–28, 2016.

[14] P. ERGUL, H. F. UGURDAG, and D. DAVUTOGLU, "HC-FFT: highly configurable and efficient FFT implementation on FPGA," vol. 29, pp. 3150–3164, 2021.

[15] P. Duhamel and H. Hollmann, "'Split radix' FFT algorithm," 1984. [Online]. Available: https://www.semanticscholar.org/paper/3e410714fe55078a5072aaa952e2cfe16b0a0b45

[16] S. V. Bhavaraju, M. V. Munot, L. P. Patil, and S. Prabhukumar, "Implementation of Split-radix Fast Fourier Transform : A Survey," 2015.

[17] S. Bouguezel, M. O. Ahmad, and M. N. S. Swamy, "A General Class of Split-Radix FFT Algorithms for the Computation of the DFT of Length-2^m," *IEEE Transactions on Signal Processing*, vol. 55, no. 8, pp. 4127–4138, 2007.

[18] S. G. Johnson and M. Frigo, "A Modified Split-Radix FFT With Fewer Arithmetic Operations," *IEEE Transactions on Signal Processing*, vol. 55, no. 1, pp. 111–119, 2007.

[19] R. Stasinski, "Fast Discrete Fourier Transform algorithms requiring less than 0(NlogN) multiplications," Mar. 2023.

[20] S. Bouguezel, M. O. Ahmad, and M. N. S. Swamy, "An Alternate Approach for Developing Higher Radix FFT Algorithms," in *APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems*, 2006, pp. 227–230.

[21] S. Winograd, "On computing the Discrete Fourier Transform." *Proceedings of the National Academy of Sciences of the United States of America*, vol. 73, no. 4, pp. 1005–6, Apr. 1976.

[22] D. Kolba and T. Parks, "A prime factor FFT algorithm using high-speed convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, no. 4, pp. 281–294, 1977.

[23] S. Peng, J. Yang, J. Li, J. Deng, X. Li, J. Jin, and T. Wang, "The on-Line Monitoring of Time-Varying Amplitude and Frequency Characteristic of Sub-Synchronous Oscillation Based on Sliding Window FFT," in *2018 China International Conference on Electricity Distribution (CICED)*, 2018, pp. 1625–1630.

[24] T. Hiyama, N. Suzuki, and T. Funakoshi, "On-line identification of power system oscillation modes by using real time FFT," in *2000 IEEE Power Engineering Society Winter Meeting. Conference Proceedings (Cat. No.00CH37077)*, vol. 2, 2000, pp. 1521–1526 vol.2.

[25] F. Xu, "Algorithm to Remove Spectral Leakage, Close-in Noise, and Its Application to

Converter Test," in *2006 IEEE Instrumentation and Measurement Technology Conference Proceedings*, 2006, pp. 1038–1042.

[26] A. F. Harvey and M. Cerna, "The Fundamentals of FFT-Based Signal Analysis and Measurement in LabVIEW and LabWindows," 1993.

[27] A. Kaliszan and P. Zwierzykowski, "Application of Real Time Operating System in the Internet of Things," in *2016 10th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, 2016, pp. 1–6.

[28] "IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems," *IEEE Std 2050-2018*, pp. 1–333, 2018.

[29] Katcher, Kettler, and Strosnider, "Modeling DSP operating systems for multimedia applications," in *1994 Proceedings Real-Time Systems Symposium*, 1994, pp. 287–291.

[30] M. Balasubramanian and N. S. Usha, "Studies on open source real time operating systems: For vehicle suspension control," in *2017 International Conference on Information Communication and Embedded Systems (ICICES)*, 2017, pp. 1–3.

[31] P. Kumar and K. Sharma, "A novel task scheduling algorithm for real time systems," in *2013 International Conference on Communication and Signal Processing*, 2013, pp. 995–998.

[32] K. V. Prashanth, P. S. Akram, and T. A. Reddy, "Real-time issues in embedded system design," in *2015 International Conference on Signal Processing and Communication Engineering Systems*, 2015, pp. 167–171.

Appendix I

Source Code

I.1 Main

```
1  /* USER CODE BEGIN Header */
2  /**
3    ******************************************************************************

4    * @file           : main.c
5    * @brief          : Task, queue, peripheral initialization
         and starting of RTOS scheduler
6    ******************************************************************************

7    * @attention
8    *
9    * Copyright (c) 2022 STMicroelectronics.
10   * All rights reserved.
```

```
11    *
12    * This software is licensed under terms that can be found
         in the LICENSE file
13    * in the root directory of this software component.
14    * If no LICENSE file comes with this software, it is
         provided AS-IS.
15    *
16    ******************************************************************************


17    */
18  /* USER CODE END Header */
19  /* Includes
      ------------------------------------------------------------------
      */
20  #include "main.h"
21
22  /* Private includes
      ----------------------------------------------------------
      */
23  /* USER CODE BEGIN Includes */
24
25  // System Includes
26  #include "FreeRTOS.h"
27  #include "timers.h"
28  #include "queue.h"
```

```
29 #include "semphr.h"
30 #include "event_groups.h"
31 #include "task.h"
32
33 // General Includes
34 #include <stdlib.h>
35 #include <stdio.h>
36 #include <complex.h>
37 #include "math.h"
38
39 // File Specific Includes
40 #include "global.h"
41 #include "transform.h"
42 #include "analysis.h"
43 #include "uart.h"
44 #include "adc.h"
45 /* USER CODE END Includes */
46
47 /* Private typedef
      -----------------------------------------------------------
      */
48 /* USER CODE BEGIN PTD */
49
50 /* USER CODE END PTD */
51
```

```
52 /* Private define
      ------------------------------------------------------------
      */
53 /* USER CODE BEGIN PD */
54 #define TIM_PERIOD 1e6/FS
55 /* USER CODE END PD */
56
57 /* Private macro
      ------------------------------------------------------------
      */
58 /* USER CODE BEGIN PM */
59
60 /* USER CODE END PM */
61
62 /* Private variables
      ---------------------------------------------------------
      */
63 ADC_HandleTypeDef hadc1;
64
65 TIM_HandleTypeDef htim3;
66 TIM_HandleTypeDef htim6;
67
68 UART_HandleTypeDef huart2;
69 DMA_HandleTypeDef hdma_usart2_rx;
70 DMA_HandleTypeDef hdma_usart2_tx;
```

```
71 /* USER CODE BEGIN PV */
72 TaskHandle_t rdr;
73 TaskHandle_t adc;
74 TaskHandle_t tf;
75 TaskHandle_t ansys;
76
77 QueueHandle_t stats_mbx;
78 QueueHandle_t buffer_mbx;
79 QueueHandle_t res_mbx;
80
81 /* USER CODE END PV */
82
83 /* Private function prototypes
      -----------------------------------------------*/
84 void SystemClock_Config(void);
85 static void MX_GPIO_Init(void);
86 static void MX_DMA_Init(void);
87 static void MX_USART2_UART_Init(void);
88 static void MX_ADC1_Init(void);
89 static void MX_TIM6_Init(void);
90 static void MX_TIM3_Init(void);
91
92 /* USER CODE BEGIN PFP */
93
94 /* USER CODE END PFP */
```

```
95
96  /* Private user code
       ---------------------------------------------------------
       */
97  /* USER CODE BEGIN 0 */
98
99  /* USER CODE END 0 */
100
101 /**
102   * @brief  The application entry point.
103   * @retval int
104   */
105 int main(void)
106 {
107   /* USER CODE BEGIN 1 */
108
109
110   /* USER CODE END 1 */
111
112   /* MCU Configuration
        ---------------------------------------------------------
        */
113
114   /* Reset of all peripherals, Initializes the Flash
        interface and the Systick. */
```

```
115   HAL_Init();
116
117   /* USER CODE BEGIN Init */
118
119   /* USER CODE END Init */
120
121   /* Configure the system clock */
122   SystemClock_Config();
123
124   /* USER CODE BEGIN SysInit */
125
126   /* USER CODE END SysInit */
127
128   /* Initialize all configured peripherals */
129   MX_GPIO_Init();
130   MX_DMA_Init();
131   MX_USART2_UART_Init();
132   MX_ADC1_Init();
133   MX_TIM6_Init();
134   MX_TIM3_Init();
135   /* USER CODE BEGIN 2 */
136   HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED); //
        Calibrate ADC on startup
137   /* USER CODE END 2 */
138
```

```
139   /* USER CODE BEGIN RTOS_MUTEX */
140   /* add mutexes, ... */
141   /* USER CODE END RTOS_MUTEX */
142
143   /* USER CODE BEGIN RTOS_SEMAPHORES */
144   /* add semaphores, ... */
145   /* USER CODE END RTOS_SEMAPHORES */
146
147   /* USER CODE BEGIN RTOS_TIMERS */
148   /* start timers, add new ones, ... */
149   /* USER CODE END RTOS_TIMERS */
150
151   /* USER CODE BEGIN RTOS_QUEUES */
152   stats_mbx = xQueueCreate(5, sizeof(stats_t));
153   if(stats_mbx == NULL)
154     {
155       printf("Could not build stats mailbox\n\r");
156       exit(1);
157     }
158
159   buffer_mbx = xQueueCreate(2, SAMPLES*sizeof(uint16_t));
160   if(buffer_mbx == NULL)
161     {
162       printf("Could not build buffer mailbox\n\r");
163       exit(1);
```

```
164       }
165
166     res_mbx = xQueueCreate(20, (SMP_2+1)*sizeof(float));
167     if(res_mbx == NULL)
168     {
169       printf("Could not build results mailbox\n\r");
170       exit(1);
171     }
172     /* USER CODE END RTOS_QUEUES */
173
174     /* Create the thread(s) */
175     /* USER CODE BEGIN RTOS_THREADS */
176     xTaskCreate(read_Task, "rdr", 1024, NULL, PriorityNormal,
          &rdr);
177     xTaskCreate(txm_Task, "tf", 10000, NULL, PriorityHigh, &tf
          );
178     xTaskCreate(ansys_Task, "ansys", 2048, NULL,
          PriorityNormal, &ansys);
179
180     /* USER CODE END RTOS_THREADS */
181
182     /* Start scheduler */
183     /* Infinite loop */
184     /* USER CODE BEGIN WHILE */
185
```

```
186   vTaskStartScheduler();
187
188   while (1)
189   {
190     /* USER CODE END WHILE */
191
192     /* USER CODE BEGIN 3 */
193   }
194   /* USER CODE END 3 */
195 }
196
197 /**
198   * @brief System Clock Configuration
199   * @retval None
200   */
201 void SystemClock_Config(void)
202 {
203   RCC_OscInitTypeDef RCC_OscInitStruct = {0};
204   RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
205
206   /** Configure the main internal regulator output voltage
207   */
208   if (HAL_PWREx_ControlVoltageScaling(
        PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
209   {
```

```
210       Error_Handler();
211     }
212
213     /** Initializes the RCC Oscillators according to the
            specified parameters
214     * in the RCC_OscInitTypeDef structure.
215     */
216     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
217     RCC_OscInitStruct.HSIState = RCC_HSI_ON;
218     RCC_OscInitStruct.HSICalibrationValue =
            RCC_HSICALIBRATION_DEFAULT;
219     RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
220     RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
221     RCC_OscInitStruct.PLL.PLLM = 1;
222     RCC_OscInitStruct.PLL.PLLN = 10;
223     RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
224     RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
225     RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
226     if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
227     {
228       Error_Handler();
229     }
230
231     /** Initializes the CPU, AHB and APB buses clocks
232     */
```

```
233   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|
        RCC_CLOCKTYPE_SYSCLK
234                                 |RCC_CLOCKTYPE_PCLK1|
                                      RCC_CLOCKTYPE_PCLK2;
235   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
236   RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
237   RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
238   RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
239
240   if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct,
        FLASH_LATENCY_4) != HAL_OK)
241   {
242     Error_Handler();
243   }
244 }
245
246 /**
247   * @brief ADC1 Initialization Function
248   * @param None
249   * @retval None
250   */
251 static void MX_ADC1_Init(void)
252 {
253
254   /* USER CODE BEGIN ADC1_Init 0 */
```

```
255
256   /* USER CODE END ADC1_Init 0 */
257
258   ADC_MultiModeTypeDef multimode = {0};
259   ADC_ChannelConfTypeDef sConfig = {0};
260
261   /* USER CODE BEGIN ADC1_Init 1 */
262
263   /* USER CODE END ADC1_Init 1 */
264
265   /** Common config
266   */
267   hadc1.Instance = ADC1;
268   hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
269   hadc1.Init.Resolution = ADC_RESOLUTION_12B;
270   hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
271   hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
272   hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
273   hadc1.Init.LowPowerAutoWait = DISABLE;
274   hadc1.Init.ContinuousConvMode = DISABLE;
275   hadc1.Init.NbrOfConversion = 1;
276   hadc1.Init.DiscontinuousConvMode = DISABLE;
277   hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIG_T3_TRGO;
278   hadc1.Init.ExternalTrigConvEdge =
        ADC_EXTERNALTRIGCONVEDGE_RISING;
```

```
279   hadc1.Init.DMAContinuousRequests = DISABLE;
280   hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
281   hadc1.Init.OversamplingMode = DISABLE;
282   if (HAL_ADC_Init(&hadc1) != HAL_OK)
283   {
284     Error_Handler();
285   }
286
287   /** Configure the ADC multi-mode
288   */
289   multimode.Mode = ADC_MODE_INDEPENDENT;
290   if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode)
        != HAL_OK)
291   {
292     Error_Handler();
293   }
294
295   /** Configure Regular Channel
296   */
297   sConfig.Channel = ADC_CHANNEL_1;
298   sConfig.Rank = ADC_REGULAR_RANK_1;
299   sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
300   sConfig.SingleDiff = ADC_SINGLE_ENDED;
301   sConfig.OffsetNumber = ADC_OFFSET_NONE;
302   sConfig.Offset = 0;
```

```
303   if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
304   {
305     Error_Handler();
306   }
307   /* USER CODE BEGIN ADC1_Init 2 */
308
309   /* USER CODE END ADC1_Init 2 */
310
311 }
312
313 /**
314   * @brief TIM3 Initialization Function
315   * @param None
316   * @retval None
317   */
318 static void MX_TIM3_Init(void)
319 {
320
321   /* USER CODE BEGIN TIM3_Init 0 */
322
323   /* USER CODE END TIM3_Init 0 */
324
325   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
326   TIM_MasterConfigTypeDef sMasterConfig = {0};
327   TIM_OC_InitTypeDef sConfigOC = {0};
```

```
328
329   /* USER CODE BEGIN TIM3_Init 1 */
330
331   /* USER CODE END TIM3_Init 1 */
332   htim3.Instance = TIM3;
333   htim3.Init.Prescaler = 80 - 1;
334   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
335   htim3.Init.Period = (int) floor(TIM_PERIOD);
336   htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
337   htim3.Init.AutoReloadPreload =
         TIM_AUTORELOAD_PRELOAD_DISABLE;
338   if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
339   {
340     Error_Handler();
341   }
342   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
343   if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig)
          != HAL_OK)
344   {
345     Error_Handler();
346   }
347   if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
348   {
349     Error_Handler();
350   }
```

```
351   sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
352   sMasterConfig.MasterSlaveMode =
        TIM_MASTERSLAVEMODE_DISABLE;
353   if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &
        sMasterConfig) != HAL_OK)
354   {
355     Error_Handler();
356   }
357   sConfigOC.OCMode = TIM_OCMODE_PWM1;
358   sConfigOC.Pulse = 0;
359   sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
360   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
361   if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC,
        TIM_CHANNEL_1) != HAL_OK)
362   {
363     Error_Handler();
364   }
365   /* USER CODE BEGIN TIM3_Init 2 */
366
367   /* USER CODE END TIM3_Init 2 */
368
369 }
370
371 /**
372   * @brief TIM6 Initialization Function
```

```
373   * @param None
374   * @retval None
375   */
376 static void MX_TIM6_Init(void)
377 {
378
379   /* USER CODE BEGIN TIM6_Init 0 */
380
381   /* USER CODE END TIM6_Init 0 */
382
383   TIM_MasterConfigTypeDef sMasterConfig = {0};
384
385   /* USER CODE BEGIN TIM6_Init 1 */
386   // Used for timing transformations at uSecond precision
387   /* USER CODE END TIM6_Init 1 */
388   htim6.Instance = TIM6;
389   htim6.Init.Prescaler = 80 - 1;
390   htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
391   htim6.Init.Period = 65535;
392   htim6.Init.AutoReloadPreload =
        TIM_AUTORELOAD_PRELOAD_DISABLE;
393   if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
394   {
395     Error_Handler();
396   }
```

```
397     sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
398     sMasterConfig.MasterSlaveMode =
          TIM_MASTERSLAVEMODE_DISABLE;
399     if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &
          sMasterConfig) != HAL_OK)
400     {
401       Error_Handler();
402     }
403     /* USER CODE BEGIN TIM6_Init 2 */
404
405     /* USER CODE END TIM6_Init 2 */
406
407 }
408
409 /**
410     * @brief USART2 Initialization Function
411     * @param None
412     * @retval None
413     */
414 static void MX_USART2_UART_Init(void)
415 {
416
417     /* USER CODE BEGIN USART2_Init 0 */
418
419     /* USER CODE END USART2_Init 0 */
```

```
420
421   /* USER CODE BEGIN USART2_Init 1 */
422
423   /* USER CODE END USART2_Init 1 */
424   huart2.Instance = USART2;
425   huart2.Init.BaudRate = 115200;
426   huart2.Init.WordLength = UART_WORDLENGTH_8B;
427   huart2.Init.StopBits = UART_STOPBITS_1;
428   huart2.Init.Parity = UART_PARITY_NONE;
429   huart2.Init.Mode = UART_MODE_TX_RX;
430   huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
431   huart2.Init.OverSampling = UART_OVERSAMPLING_16;
432   huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
433   huart2.AdvancedInit.AdvFeatureInit =
         UART_ADVFEATURE_NO_INIT;
434   if (HAL_UART_Init(&huart2) != HAL_OK)
435   {
436     Error_Handler();
437   }
438   /* USER CODE BEGIN USART2_Init 2 */
439
440   /* USER CODE END USART2_Init 2 */
441
442 }
443
```

```
444 /**
445   * Enable DMA controller clock
446   */
447 static void MX_DMA_Init(void)
448 {
449
450   /* DMA controller clock enable */
451   __HAL_RCC_DMA1_CLK_ENABLE();
452
453   /* DMA interrupt init */
454   /* DMA1_Channel6_IRQn interrupt configuration */
455   HAL_NVIC_SetPriority(DMA1_Channel6_IRQn, 5, 0);
456   HAL_NVIC_EnableIRQ(DMA1_Channel6_IRQn);
457   /* DMA1_Channel7_IRQn interrupt configuration */
458   HAL_NVIC_SetPriority(DMA1_Channel7_IRQn, 5, 0);
459   HAL_NVIC_EnableIRQ(DMA1_Channel7_IRQn);
460
461 }
462
463 /**
464   * @brief GPIO Initialization Function
465   * @param None
466   * @retval None
467   */
468 static void MX_GPIO_Init(void)
```

```
469 {
470   GPIO_InitTypeDef GPIO_InitStruct = {0};
471 /* USER CODE BEGIN MX_GPIO_Init_1 */
472 /* USER CODE END MX_GPIO_Init_1 */
473
474   /* GPIO Ports Clock Enable */
475   __HAL_RCC_GPIOC_CLK_ENABLE();
476   __HAL_RCC_GPIOH_CLK_ENABLE();
477   __HAL_RCC_GPIOA_CLK_ENABLE();
478   __HAL_RCC_GPIOB_CLK_ENABLE();
479
480   /*Configure GPIO pin Output Level */
481   HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
482
483   /*Configure GPIO pin : B1_Pin */
484   GPIO_InitStruct.Pin = B1_Pin;
485   GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
486   GPIO_InitStruct.Pull = GPIO_NOPULL;
487   HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
488
489   /*Configure GPIO pin : LD2_Pin */
490   GPIO_InitStruct.Pin = LD2_Pin;
491   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
492   GPIO_InitStruct.Pull = GPIO_NOPULL;
493   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
```

```
494   HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
495
496 /* USER CODE BEGIN MX_GPIO_Init_2 */
497 /* USER CODE END MX_GPIO_Init_2 */
498 }
499
500 /* USER CODE BEGIN 4 */
501
502 /* USER CODE END 4 */
503 /**
504   * @brief  Period elapsed callback in non blocking mode
505   * @note   This function is called  when TIM1 interrupt
          took place, inside
506   * HAL_TIM_IRQHandler(). It makes a direct call to
          HAL_IncTick() to increment
507   * a global variable "uwTick" used as application time base
          .
508   * @param  htim : TIM handle
509   * @retval None
510   */
511 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
512 {
513   /* USER CODE BEGIN Callback 0 */
514
515   /* USER CODE END Callback 0 */
```

```
516   if (htim->Instance == TIM1) {
517     HAL_IncTick();
518   }
519   /* USER CODE BEGIN Callback 1 */
520
521   /* USER CODE END Callback 1 */
522 }
523
524 /**
525   * @brief  This function is executed in case of error
          occurrence.
526   * @retval None
527   */
528 void Error_Handler(void)
529 {
530   /* USER CODE BEGIN Error_Handler_Debug */
531   /* User can add his own implementation to report the HAL
          error return state */
532   __disable_irq();
533   while (1)
534   {
535   }
536   /* USER CODE END Error_Handler_Debug */
537 }
538
```

```
539 #ifdef  USE_FULL_ASSERT
540 /**
541   * @brief  Reports the name of the source file and the
        source line number
542   *         where the assert_param error has occurred.
543   * @param  file: pointer to the source file name
544   * @param  line: assert_param error line source number
545   * @retval None
546   */
547 void assert_failed(uint8_t *file, uint32_t line)
548 {
549   /* USER CODE BEGIN 6 */
550   /* User can add his own implementation to report the file
        name and line number,
551      ex: printf("Wrong parameters value: file %s on line %d\
          r\n", file, line) */
552   /* USER CODE END 6 */
553 }
554 #endif /* USE_FULL_ASSERT */
```

I.2 UART

```
1  /*
2   * uart.c
3   *
4   *  Created on: Nov 10, 2022
5   *      Author: Ty Freeman
6   */
7  #include "FreeRTOS.h"
8  #include "task.h"
9  #include "queue.h"
10 #include "semphr.h"
11 #include "event_groups.h"
12 #include "stm32l4xx_hal.h"
13
14 #include <stdlib.h>
15 #include <stdio.h>
16 #include "string.h"
17 #include "math.h"
18
19 #include "global.h"
20 #include "uart.h"
21 #include "adc.h"
22
23 #define STR_SZ 10
```

```
24
25 extern TaskHandle_t adc;
26 extern TaskHandle_t rdr;
27 extern TaskHandle_t tf;
28 extern TaskHandle_t ansys;
29 extern QueueHandle_t res_mbx;
30 extern UART_HandleTypeDef huart2;
31 extern DMA_HandleTypeDef hdma_usart2_rx;
32 extern ADC_HandleTypeDef hadc1;
33
34 extern unsigned long int Total_mult;
35 extern unsigned long int Total_add;
36 extern unsigned int pkg_cnt;
37 extern float tm2full;
38
39 uint8_t rxbuf = '\0';
40
41 unsigned char caret[] = "\n\r> ";
42 unsigned char cr[] = "\n\r";
43 unsigned char bkspc[] = "\b\0";
44 unsigned char clr = '\0';
45
46 _Bool cr_flg = 0;
47 _Bool rd_flg = 0;
48 _Bool timLim_flg = 0;
```

```
49 _Bool bufRdy_flg = 1;
50 _Bool PROG_END = 0;
51 extern _Bool ansyDone_flg;
52
53 int Tx_fails = 0;
54 int Rx_fails = 0;
55 char nbuf[5];
56
57 int ttl_pkgs = 0;
58 int pkgs_sent = 0;
59 int uart_pkgs = 0;
60
61 /*Callback for UART receiver. Every 1 character triggers
      this callback function which does light processing of
      value*/
62 void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart,
      uint16_t size)
63 {
64   static int index = 0;
65
66   HAL_UART_Transmit(&huart2, &rxbuf, 1, 2); //Echo character
67
68   if(!cr_flg)
69   {
70     if((rxbuf == '\n' || rxbuf == '\r') && rd_flg) //If
```

```
        enter key has been hit
71      {
72        cr_flg = 1; // Set the flag which is checked in the
            reader task below
73        rd_flg = 0; // Clear trigger flag
74        index = 0;
75        HAL_UART_Transmit(&huart2, cr, sizeof(cr), 2); // Echo
            character
76      }
77      else if((rxbuf == 'g' || rxbuf == 'G') && !rd_flg) // g
          or G for Go
78      {
79        rd_flg = 1; // Ready for read task
80      }
81      else if(rxbuf > 47 && rxbuf < 58 && index < 5)
82      {
83        nbuf[index] = rxbuf; // Collect up to 5 numbers in
            this buffer
84        index++; // Increment buffer index
85      }
86      else if(rxbuf == 's' || rxbuf == 'S') // s or S for
          seconds
87      {
88        timLim_flg = 1; // User specified time limit
89      }
```

```
90      else if(rxbuf != ' ') // If not acceptable letter
91      {
92        HAL_UART_Transmit(&huart2, bkspc, sizeof(bkspc), 2);
             // Auto backspace
93      }
94    }
95  }
96
97  void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) //
       UART Tx DMA transfer complete callback
98  {
99    pkgs_sent++; // Keep track of packages sent
100   if(ansyDone_flg && uxQueueMessagesWaitingFromISR(res_mbx)
         == 0) // Need adc done and all data sent
101   {
102     PROG_END = 1; // Signal program end
103     HAL_UART_AbortTransmit(huart); // Turn off continuous
          printing
104   }
105   else
106     bufRdy_flg = 1; // Signal buffer ready for new data
107 }
108
109 void read_Task(void * pvParameters)
110 {
```

```
111
112   TickType_t lastWake = 0;
113   TickType_t Period = pdMS_TO_TICKS(5);
114
115   float fft_res[SMP_2+1];
116   char TxBuf[(SMP_2+1)*STR_SZ];
117
118   int avg_MC = 0;
119   int avg_AC = 0;
120
121   HAL_UART_Transmit(&huart2, caret, sizeof(caret), 2); //
        Print starting CMD caret
122
123   while(1)
124   {
125     HAL_UARTEx_ReceiveToIdle_DMA(&huart2, &rxbuf, 1); //
          Begin DMA
126     __HAL_DMA_DISABLE_IT(&hdma_usart2_rx, DMA_IT_HT);
127
128
129     if(cr_flg)
130     {
131       cr_flg = 0; // Clear carriage return flag
132       ttl_pkgs = atoi(nbuf); // Extract user set limit
133       memset(nbuf,0,sizeof(nbuf)); // Clear number buffer
```

```
134         if(timLim_flg) ttl_pkgs = (int) floor(ttl_pkgs/tm2full
               ); // Calculate number of packages needed
135         HAL_UART_Transmit(&huart2, cr, sizeof(cr), 2);
136         xTaskCreate(adc_Task, "adc", 1024,(void *) &ttl_pkgs,
               PriorityNormal, &adc);
137       }
138
139       if(xQueueReceive(res_mbx, fft_res, 0) && bufRdy_flg) //
             Results ready and Tx buffer clear
140       {
141
142         uart_pkgs++;
143
144         for(int i=0; i<SMP_2+1; i++)
145         {
146           /* Maintain standard 10 bytes of data after sprintf
                 */
147           if(fft_res[i] > 9 && fft_res[i] < 100)
148             sprintf(TxBuf+i*STR_SZ, "%.5f,", fft_res[i]);
149
150            else if(fft_res[i] > 99)
151             sprintf(TxBuf+i*STR_SZ, "%.4f,", fft_res[i]);
152
153            else
154             sprintf(TxBuf+i*STR_SZ, "%.6f,", fft_res[i]);
```

```
155
156         }
157
158           HAL_UART_Transmit_DMA(&huart2, (unsigned *) TxBuf,
                sizeof(TxBuf)); // Start DMA
159           bufRdy_flg = 0; // Tx buffer being used by new data
160
161       }
162
163
164       if(PROG_END) // All data printed and ADC done
165       {
166         PROG_END = 0; // Clr flag
167         avg_MC = ceil((float) Total_mult/pkg_cnt); //
              Calculate average number of multiplications per
              transform
168         avg_AC = ceil((float) Total_add/pkg_cnt); // Calculate
               average number of additions per transform
169
170         Tx_fails = trans_tx_fail + ansy_tx_fail + adc_tx_fail;
171         Rx_fails = trans_rx_fail + ansy_rx_fail;
172
173         /* Print Stats Message Block */
174         printf("\n\n\r***** STATS *****\n\n\r");
175         printf("Average Mult Ops per transform:\t%d\n\r",
```

```
            avg_MC);
176         printf("Average Add Ops transform:\t%d\n\n\r", avg_AC)
            ;
177
178         printf("Number of Rx Failures: %d\n\r", Rx_fails);
179         printf("Number of Tx Failures: %d\n\n\r", Tx_fails);
180         printf("Packages seen by Task:\n\r");
181         printf("\t\tADC\t%d\n\r", pkg_cnt);
182         printf("\t\tTXM\t%d\n\r", txm_pkgs);
183         printf("\t\tANSY\t%d\n\r", ansys_pkgs);
184         printf("\t\tUART\t%d\n\r", uart_pkgs);
185         printf("\t\tSENT\t%d\n\r", pkgs_sent);
186       }
187
188
189       lastWake = xTaskGetTickCount();
190       vTaskDelayUntil(&lastWake, Period);
191     }
192 }
```

I.3 ADC

```c
/*
 * adc.c
 *
 *  Created on: Nov 21, 2022
 *      Author: Ty Freeman
 */

/* System Includes */
#include "FreeRTOS.h"
#include "queue.h"
#include "task.h"
#include "stm32l4xx_hal.h"

/* Lib Includes */
#include <stdio.h>
#include <stdlib.h>
#include "math.h"
#include "string.h"

/* User-Created Includes */
#include "global.h"
#include "adc.h"

```

```
24 extern ADC_HandleTypeDef hadc1;
25 extern TIM_HandleTypeDef htim3;
26 extern DMA_HandleTypeDef hdma_adc1;
27 extern QueueHandle_t buffer_mbx;
28 extern int num_emptyBuf;
29 extern unsigned int quiet_cnt;
30
31 uint16_t RWM[SAMPLES]; // Capture buffer
32
33 /* Control Flags */
34 _Bool full = 0;
35 _Bool pkgRdy_flg = 0;
36 _Bool lim_flg = 0;
37 _Bool adcDone_flg = 0;
38
39 /* Counters */
40 unsigned int pkg_cnt = 0;
41 int idx = 0;
42 int adc_tx_fail = 0;
43
44
45 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) //
      ADC callback function
46 {
47     RWM[idx] = HAL_ADC_GetValue(&hadc1); // Acquire ADC
```

```
        value
48      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5); // Toggle LED
49      if(idx == SAMPLES-1) // Check index value. Restart?
50      {
51      full = 1;
52        idx = SMP_2; // Only saving half of samples for window
              overlapping
53      }
54      else
55        idx++;
56  }
57
58  void adc_Task(void * pvParameters)
59  {
60    TickType_t lastwake = 0;
61
62    HAL_ADC_Start_IT(&hadc1); // Start ADC
63    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1); // Start ADC
        trigger timer
64
65    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, 1); // Turn on LED to
        signal ADC start
66
67
68    uint16_t *package = NULL;
```

```
69    pkg_cnt = 0; // Reset for repeat trials
70    quiet_cnt = num_emptyBuf; // Reset quiet cnt number at
         start of ADC task
71
72    /* User Set Limit Signals */
73    _Bool lim_set = 0;
74    int *pkg_lim = (int *) pvParameters;
75    if(*pkg_lim > 0) // Pull pkg limit from parameter
76    {
77      lim_set = 1;
78    }
79
80    while(1)
81    {
82      if(full)
83      {
84        package = (uint16_t *)malloc(SAMPLES*sizeof(uint16_t))
             ; // Create package buffer
85        memcpy(package, RWM, SAMPLES*sizeof(uint16_t)); //
             Move values from capture to package buffer
86        memmove(RWM, RWM+SMP_2, SMP_2*sizeof(uint16_t)); //
             Shift capture buffer values down for window overlap
87
88
89        if(xQueueSend(buffer_mbx, package, 1)) // Send package
```

```
            to transform task for processing
90          {
91            pkg_cnt++; // Count number of packages processed
92            pkgRdy_flg = 1; // Data package in mailbox ready for
                  transform
93          }
94
95          else adc_tx_fail += 1; // If failed to post, keep
               track of failure
96
97          if(lim_set && pkg_cnt == *pkg_lim) lim_flg = 1; //
               Signal time to stop
98
99          full = 0; // Reset full flg
100
101         free(package); // Release package space
102
103         if(lim_flg || sigDone_flg) // Two possible ending
               conditions
104         {
105           /* Turn off Timer and ADC before deleting the adc
                 task */
106           HAL_TIM_PWM_Stop(&htim3, TIM_CHANNEL_1);
107           HAL_ADC_Stop_IT(&hadc1);
108
```

```
109             adcDone_flg = 1; // Signal program that adc is done
110             if(sigDone_flg) pkg_cnt -= num_emptyBuf; // Remove
                    empty buffers from total count
111
112             sigDone_flg = 0; // Clear signal flag
113             lim_flg = 0; // Clear limit flag
114
115             vTaskDelete(NULL);
116           }
117       }
118
119
120       lastwake = xTaskGetTickCount();
121       vTaskDelayUntil(&lastwake, pdMS_TO_TICKS(30)); // Task
              suspension for 30 ms
122     }
123 }
```

I.4 Transform

```c
/*
 * transform.c
 *
 *  Created on: Feb 20, 2023
 *      Author: Ty Freeman
 */


/* System Includes */
#include "FreeRTOS.h"
#include "queue.h"
#include "task.h"
#include "stm32l4xx_hal.h"

/* Lib Includes */
#include <stdio.h>
#include <stdlib.h>
#include <transform.h>
#include "string.h"
#include "math.h"

/* User-Created Includes */
#include "global.h"
```

```
24 #include "fft.h"
25
26 #define TOREAL 3.3/4096
27 #define TODIG  4096/3.3
28
29 extern QueueHandle_t buffer_mbx;
30 extern QueueHandle_t stats_mbx;
31
32 /* Data Transmission Failure Flags */
33 int trans_tx_fail = 0;
34 int trans_rx_fail = 0;
35
36 /* Program Control Flags */
37 _Bool ansy_flg = 0;
38 _Bool sigDone_flg = 0;
39 _Bool txmDone_flg = 0;
40
41 /* Program Ending Variables */
42 float tm2full;
43 unsigned int quiet_cnt;
44 int num_emptyBuf;
45
46 int txm_pkgs = 0;
47
48 void txm_Task(void * pvParameters)
```

```
49 {
50   TickType_t lastWake = 0;
51   TickType_t Period = pdMS_TO_TICKS(20);
52
53   uint16_t *rec = NULL;
54
55   stats_t res_fft;
56
57   tm2full = (float) SAMPLES/FS; // Time it takes to fill one
         buffer worth of samples
58   quiet_cnt = (int) floor(3/tm2full); // 3 seconds of
        silence signals end of incoming signal
59   num_emptyBuf = quiet_cnt;
60
61   while(1)
62   {
63     if(pkgRdy_flg)
64     {
65       res_fft.pkg_num = pkg_cnt; // Find current package
            number
66       rec = (uint16_t *)malloc(SAMPLES*sizeof(uint16_t)); //
             Create reception buffer
67
68       if(xQueueReceive(buffer_mbx, rec, 0) != pdTRUE)
            trans_rx_fail++; // Receive data buffer from mailbox
```

```
69
70          else
71          {
72            txm_pkgs++;
73            pkgRdy_flg = 0; // Reset pkgRdy flag
74
75            for(int i = 0; i < SAMPLES; i++)
76            {
77              float hann = 0.5-0.5*cos(2*PI*i/SAMPLES); //
                    Calculate Hanning window coefficient
78              res_fft.res_buf[i] = (float)rec[i]*hann*TOREAL +
                    0*I; // Apply Hanning window and convert to
                    complex number
79            }
80
81            free(rec); // Free temporary transfer buffer
82            FFT(&res_fft); // Run FFT
83
84            float perEmpty = (float) res_fft.zCnt/SAMPLES;
85            if(perEmpty >= 0.95) // If 85% of buffer is zero,
                   consider it empty
86            {
87              quiet_cnt--; // Keep track of empty captures
88
89              if(quiet_cnt == 0)
```

```
90            {
91              sigDone_flg = 1; // Signal finished, adc no
                    longer needed
92            }
93          }
94          else // Reset quiet count
95            quiet_cnt = num_emptyBuf; // Reset quiet_cnt
96
97          if(xQueueSend(stats_mbx, &res_fft, 0)) txm_pkgs++;
98
99          else trans_tx_fail++;
100
101         if(adcDone_flg && uxQueueMessagesWaiting(buffer_mbx)
                == 0)
102         {
103           txmDone_flg = 1;
104           vTaskSuspend(NULL); // Suspend if ADC done and no
                more data to process
105         }
106       }
107
108     }
109
110
111     lastWake = xTaskGetTickCount();
```

```
112       vTaskDelayUntil(&lastWake, Period);
113     }
114 }
```

I.5 Analysis

```c
/*
 * analysis.c
 *
 *  Created on: Apr 5, 2023
 *      Author: Ty Freeman
 */

/* System Includes */
#include "FreeRTOS.h"
#include "queue.h"
#include "task.h"
#include "stm32l4xx_hal.h"

/* Lib Includes */
#include <stdio.h>
#include <stdlib.h>
#include "string.h"
#include "math.h"

/* User-Created Includes */
#include "global.h"
#include "transform.h"
#include "analysis.h"
```

```
24
25 #define wErr 2 // Window error correction factor
26 #define NPBw 1.5 // Noise Power BW for Hann window
27 #define T FS/SAMPLES // Time Step
28
29 extern QueueHandle_t stats_mbx;
30 extern QueueHandle_t res_mbx;
31
32 int ansy_tx_fail = 0;
33 int ansy_rx_fail = 0;
34
35 unsigned long Total_mult = 0;
36 unsigned long Total_add = 0;
37
38 int ansys_pkgs = 0;
39
40 extern _Bool txmDone_flg;
41 _Bool ansyDone_flg = 0;
42
43
44 void ansys_Task(void * pvParameters)
45 {
46   TickType_t lastWake = 0;
47   TickType_t Period = pdMS_TO_TICKS(10);
48   stats_t fft_res;
```

```
49
50    float *temp_buf = NULL;
51
52    while(1)
53    {
54      if(xQueueReceive(stats_mbx, &fft_res, Period)) // If
            analysis has been triggered
55      {
56        temp_buf = malloc((SMP_2+1)*sizeof(float)); // N/2+1
              buffer for calculations
57
58        ansys_pkgs++;
59        for(int i = 0; i < SMP_2+1; i++) // Convert to single
              sided
60        {
61          if(i==0)
62          {
63            fft_res.res_buf[i] = mag(fft_res.res_buf[i])/
                SAMPLES; // Normalize magnitude of DC component
64          }
65          else
66          {
67            fft_res.res_buf[i] = sqrt(2)*mag(fft_res.res_buf[i
                ])/SAMPLES; // Convert to Amplitude rms value
68          }
```

```
69
70          temp_buf[i] = creal(fft_res.res_buf[i])*wErr; //
               Correct windowed amplitude and transfer to smaller
                buffer
71        }
72
73        if(xQueueSend(res_mbx, temp_buf, 0)) ansys_pkgs++;//
             Send smaller buffer for printing
74
75        else ansy_tx_fail++;
76
77        free(temp_buf);
78
79        /* For averages at end of the program */
80        Total_mult += fft_res.mult_cnt;
81        Total_add  += fft_res.add_cnt;
82
83        if(txmDone_flg && uxQueueMessagesWaiting(stats_mbx) ==
              0)
84        {
85          ansyDone_flg = 1;
86          vTaskSuspend(NULL);
87        }
88      }
89
```

```
90      lastWake = xTaskGetTickCount();
91      vTaskDelayUntil(&lastWake, Period);
92
93    }
94  }
95
96
97
98
99
100 /**
101   * @brief Magnitude calculation of complex number
102   * @param N Complex number
103   * @retval None complex floating point number
104   */
105 float mag(float complex N)
106 {
107   float r2 = creal(N)*creal(N);
108   float i2 = cimag(N)*cimag(N);
109
110   return sqrt(r2 + i2);
111 }
```

I.6 Global Header

```
1  /*
2   * global.h
3   *
4   *  Created on: Jan 31, 2023
5   *      Author: Ty Freeman
6   */
7
8  #ifndef INC_GLOBAL_H_
9  #define INC_GLOBAL_H_
10
11 #include "complex.h"
12
13 #define PI 3.14159265358979323846
14 #define SYS_FREQ 80000000
15 #define SAMPLES 256
16 #define FS 5000
17 #define dt 1/FS
18 #define SMPx2 SAMPLES*2
19 #define SMP_2 SAMPLES/2
20
21 extern unsigned int pkg_cnt;
22 extern int txm_pkgs;
23 extern int ansys_pkgs;
```

```
24
25 /* Control Flags */
26 extern _Bool ansy_flg;
27 extern _Bool pkgRdy_flg;
28 extern _Bool adcDone_flg;
29 extern _Bool sigDone_flg;
30 extern _Bool bufEmpty_flg;
31 extern _Bool resRdy_flg;
32
33 /* Fail Counters */
34 extern int adc_tx_fail;
35 extern int trans_tx_fail;
36 extern int trans_rx_fail;
37 extern int ansy_tx_fail;
38 extern int ansy_rx_fail;
39
40 extern double step;
41
42 enum  Priority{
43   PriorityIdle,           ///< priority: idle (lowest)
44   PriorityLow,                ///< priority: low
45   PriorityBelowNormal,          ///< priority: below normal
46   PriorityNormal,         ///< priority: normal (default)
47   PriorityAboveNormal,          ///< priority: above normal
48   PriorityHigh,           ///< priority: high
```

```
49    PriorityRealtime,           ///< priority: realtime (
        highest)
50    PriorityError = 0x84         ///< system cannot determine
        priority or thread has illegal priority
51  };
52
53  typedef struct stats{
54    unsigned int pkg_num;
55    unsigned long mult_cnt;
56    unsigned long add_cnt;
57    int zCnt;
58    float complex res_buf[SAMPLES];
59  }stats_t;
60
61
62  #endif /* INC_GLOBAL_H_ */
```

I.7 FFT

```c
/*
 * fft.c
 *
 *  Created on: Nov 7, 2022
 *      Author: Ty Freeman
 */

/* System Includes */
#include <fft.h>
#include "FreeRTOS.h"
#include "task.h"
#include "stm32l4xx_hal.h"

/* lib Includes */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "math.h"

/* User-Created Includes */

float complex twexp = 2*PI*I/SAMPLES; // Twiddle exponent e
    ^(2PIj/N)
```

```
23
24 void FFT(stats_t *results)
25 {
26   int a = SMP_2; // Dual-Node distance factor
27   int m = 0; // Twiddle power
28
29   float complex x  = 0; // Primary summation term
30   float complex xp = 0; // Secondary summation term
31
32   results->mult_cnt = 0; // Reset operations counter
33   results->add_cnt = 0;
34
35   int stages = log_2(SAMPLES); // Calculate number of stages
       necessary
36
37   for(int j = 1; j <= stages; j++)
38   {
39     for(int k = 0; k < SAMPLES; k++)
40     {
41       if(!(k & a)) // Remove redundant computations
42       {
43         m = bit_reverse(stages, k >> (stages - j)); //
             Calculate twiddle power
44         x = results->res_buf[k];
45         xp = cexp(twexp*m)*results->res_buf[k+a];
```

```
46
47          results->res_buf[k] = x + xp;
48          results->res_buf[k + a] = x - xp;
49          results->mult_cnt += 1; // One complex
               multiplication
50          results->add_cnt += 2; // Two complex additions
51        }
52      }
53
54      a >>= 1; // Change Dual-Node distance
55    }
56
57    /* In Place Array Re-Indexing */
58    for(int i = 0; i < SAMPLES; i++)
59    {
60      int p = bit_reverse(stages, i);
61
62      if(i < p) // Only swap elements once
63      {
64        float complex n = results->res_buf[i];
65        results->res_buf[i] = results->res_buf[p];
66        results->res_buf[p] = n;
67      }
68
69      if(creal(results->res_buf[i]) == 0)
```

```
70          results->zCnt++;
71      }
72  }
73
74  /**
75      * @brief Bit reversal algorithm
76      * @param sz Number of bits in number
77      * @param index Current index value to be reversed
78      * @retval int Bit reversed index
79      */
80  int bit_reverse(int sz, int index)
81  {
82      int p = 0;
83
84      for(int i = 0; i <= sz; i++)
85      {
86          if(index & (1 << (sz - i)))
87          {
88              p |= 1 << (i - 1);
89          }
90      }
91
92      return p;
93  }
94
```

```
95  /**
96    * @brief Base-2 logarithm
97    * @param N Number
98    * @retval int Number of bits in N
99    */
100 int log_2(unsigned int N)
101 {
102   int pow = 0;
103
104   while(N)
105   {
106     N >>= 1;
107     pow++;
108   }
109
110   return pow - 1;
111 }
```

I.8 FFT Test Main

```
1  /* USER CODE BEGIN Header */
2  /**
3   ******************************************************************************

4   * @file           : main.c
5   * @brief          : Main program body
6   ******************************************************************************

7   * @attention
8   *
9   * Copyright (c) 2023 STMicroelectronics.
10  * All rights reserved.
11  *
12  * This software is licensed under terms that can be found
       in the LICENSE file
13  * in the root directory of this software component.
14  * If no LICENSE file comes with this software, it is
       provided AS-IS.
15  *
16  ******************************************************************************

17  */
18 /* USER CODE END Header */
```

```
19 /* Includes
      ------------------------------------------------------------------
      */
20 #include "main.h"
21
22 /* Private includes
      ----------------------------------------------------------
      */
23 /* USER CODE BEGIN Includes */
24 #include "stdio.h"
25 #include "stdlib.h"
26 #include "fft.h"
27 #include "math.h"
28 #include <inttypes.h>
29 /* USER CODE END Includes */
30
31 /* Private typedef
      -----------------------------------------------------------
      */
32 /* USER CODE BEGIN PTD */
33
34 /* USER CODE END PTD */
35
36 /* Private define
      ------------------------------------------------------------
```

```
   */
/* USER CODE BEGIN PD */
#define STR_SZ 10
#define fc 261.63
#define fg 783.99
#define fe 659.25
#define TOREAL 3.21/4096
#define TIM_PERIOD 1e6/FS
/* USER CODE END PD */

/* Private macro
   -----------------------------------------------------------------
   */
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables
   ---------------------------------------------------------
   */
ADC_HandleTypeDef hadc1;

TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim6;

```

```
57 UART_HandleTypeDef huart2;
58 DMA_HandleTypeDef hdma_usart2_tx;
59
60 /* USER CODE BEGIN PV */
61 _Bool full = 0;
62 _Bool done = 0;
63 uint8_t tim_turnover = 0;
64 stats_t fft_res;
65 uint16_t RWM[SAMPLES]; // Capture buffer
66 /* USER CODE END PV */
67
68 /* Private function prototypes
      -----------------------------------------------*/
69 void SystemClock_Config(void);
70 static void MX_GPIO_Init(void);
71 static void MX_DMA_Init(void);
72 static void MX_USART2_UART_Init(void);
73 static void MX_TIM6_Init(void);
74 static void MX_ADC1_Init(void);
75 static void MX_TIM3_Init(void);
76 /* USER CODE BEGIN PFP */
77
78 /* USER CODE END PFP */
79
80 /* Private user code
```

```
    ------------------------------------------------------------
    */
/* USER CODE BEGIN 0 */
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
  HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
  HAL_UART_DMAStop(huart);
  done = 1;
}

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) //
    ADC callback function
{
  static int idx = 0;
  RWM[idx] = HAL_ADC_GetValue(hadc); // Acquire ADC value
  if(idx == SAMPLES-1) // Check index value. Restart?
  {
    full = 1;
    HAL_TIM_PWM_Stop(&htim3, TIM_CHANNEL_1);
  }
  else
    idx++;
}
/* USER CODE END 0 */

```

```
103 /**
104   * @brief  The application entry point.
105   * @retval int
106   */
107 int main(void)
108 {
109   /* USER CODE BEGIN 1 */
110   unsigned char dblSpace[] = "\n\n\r";
111   /* USER CODE END 1 */
112
113   /* MCU Configuration
        --------------------------------------------------------
        */
114
115   /* Reset of all peripherals, Initializes the Flash
        interface and the Systick. */
116   HAL_Init();
117
118   /* USER CODE BEGIN Init */
119
120   /* USER CODE END Init */
121
122   /* Configure the system clock */
123   SystemClock_Config();
124
```

```
125   /* USER CODE BEGIN SysInit */
126
127   /* USER CODE END SysInit */
128
129   /* Initialize all configured peripherals */
130   MX_GPIO_Init();
131   MX_DMA_Init();
132   MX_USART2_UART_Init();
133   MX_TIM6_Init();
134   MX_ADC1_Init();
135   MX_TIM3_Init();
136   /* USER CODE BEGIN 2 */
137   HAL_UART_Transmit(&huart2, dblSpace, sizeof(dblSpace), 2);
138   HAL_ADC_Start_IT(&hadc1); // Start ADC
139   HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1); // Start ADC
        trigger timer
140
141   /* USER CODE END 2 */
142
143   /* Infinite loop */
144   /* USER CODE BEGIN WHILE */
145   while (1)
146   {
147     if(full)
148     {
```

```
149         HAL_ADC_Stop_IT(&hadc1);
150         float *temp_buf = NULL;
151         char pBuf[(SMP_2+1)*STR_SZ];
152
153         for(int i = 0; i < SAMPLES; i++)
154         {
155           float hann = 0.5-0.5*cos(2*PI*i/SAMPLES); //
                Calculate Hanning window coefficient
156 //          fft_res.res_buf[i] = sin(2*PI*200*i/FS); // Single
    -Tone 200 Hz
157 //          fft_res.res_buf[i] = (sin(2*PI*1000*i/FS)+1) +
    0.5*(cos(2*PI*200*i/FS)+1); // Dual-Tone 200 Hz and 1 kHz
158 //          fft_res.res_buf[i] = ((sin(2*PI*fc*i/FS)) + (sin
    (2*PI*fg*i/FS)) + (sin(2*PI*fe*i/FS))); // C Chord triad
159           fft_res.res_buf[i] = TOREAL*(RWM[i]); // ADC
                Implementation
160           fft_res.res_buf[i] *= hann; // Apply Hanning window
                and convert to complex number
161         }
162
163
164         classic_FFT(&fft_res);
165
166         full = 0;
167
```

```
168         temp_buf = malloc((SMP_2+1)*sizeof(float));
169
170         for(int i = 0; i < (SMP_2+1); i++)
171         {
172           if(i==0)
173           {
174             fft_res.res_buf[i] = mag(fft_res.res_buf[i])/
                  SAMPLES; // Normalize magnitude of DC component
175           }
176           else
177           {
178             fft_res.res_buf[i] = sqrt(2)*mag(fft_res.res_buf[i
                  ])/SAMPLES; // Convert to Amplitude rms value
179           }
180           temp_buf[i] = creal(fft_res.res_buf[i])*2; //
                Correct windowed amplitude and transfer to smaller
                 buffer
181
182           if(temp_buf[i] > 9 && temp_buf[i] < 100)
183             sprintf(pBuf+i*STR_SZ, "%.5f,", temp_buf[i]);
184
185           else if(temp_buf[i] > 99)
186             sprintf(pBuf+i*STR_SZ, "%.4f,", temp_buf[i]);
187
188           else
```

```
189             sprintf(pBuf+i*STR_SZ, "%.6f,", temp_buf[i]);
190
191           HAL_UART_Transmit_DMA(&huart2, pBuf, sizeof(pBuf));
192         }
193
194         free(temp_buf);
195
196       }
197
198       if(done)
199       {
200         HAL_UART_Abort(&huart2);
201         printf("\n\n\r***** STATS *****\n\r");;
202         printf("Time per transform:\t%f uS\n\r", (float)
              fft_res.time/80);
203         printf("Mult Ops per transform:\t%d\n\r", fft_res.
              mult_cnt);
204         printf("Add Ops per transform:\t%d\n\n\r", fft_res.
              add_cnt);
205         exit(1);
206       }
207       /* USER CODE END WHILE */
208
209       /* USER CODE BEGIN 3 */
210   }
```

```
211   /* USER CODE END 3 */
212 }
213
214 /**
215   * @brief System Clock Configuration
216   * @retval None
217   */
218 void SystemClock_Config(void)
219 {
220   RCC_OscInitTypeDef RCC_OscInitStruct = {0};
221   RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
222
223   /** Configure the main internal regulator output voltage
224   */
225   if (HAL_PWREx_ControlVoltageScaling(
        PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
226   {
227     Error_Handler();
228   }
229
230   /** Initializes the RCC Oscillators according to the
        specified parameters
231   * in the RCC_OscInitTypeDef structure.
232   */
233   RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
```

```
234   RCC_OscInitStruct.HSIState = RCC_HSI_ON;
235   RCC_OscInitStruct.HSICalibrationValue =
        RCC_HSICALIBRATION_DEFAULT;
236   RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
237   RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
238   RCC_OscInitStruct.PLL.PLLM = 1;
239   RCC_OscInitStruct.PLL.PLLN = 10;
240   RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
241   RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
242   RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
243   if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
244   {
245     Error_Handler();
246   }
247
248   /** Initializes the CPU, AHB and APB buses clocks
249   */
250   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|
        RCC_CLOCKTYPE_SYSCLK
251                                 |RCC_CLOCKTYPE_PCLK1|
                                    RCC_CLOCKTYPE_PCLK2;
252   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
253   RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
254   RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
255   RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
```

```
256
257   if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct,
          FLASH_LATENCY_4) != HAL_OK)
258   {
259     Error_Handler();
260   }
261 }
262
263 /**
264   * @brief ADC1 Initialization Function
265   * @param None
266   * @retval None
267   */
268 static void MX_ADC1_Init(void)
269 {
270
271   /* USER CODE BEGIN ADC1_Init 0 */
272
273   /* USER CODE END ADC1_Init 0 */
274
275   ADC_MultiModeTypeDef multimode = {0};
276   ADC_ChannelConfTypeDef sConfig = {0};
277
278   /* USER CODE BEGIN ADC1_Init 1 */
279
```

```
280   /* USER CODE END ADC1_Init 1 */
281
282   /** Common config
283   */
284   hadc1.Instance = ADC1;
285   hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
286   hadc1.Init.Resolution = ADC_RESOLUTION_12B;
287   hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
288   hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
289   hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
290   hadc1.Init.LowPowerAutoWait = DISABLE;
291   hadc1.Init.ContinuousConvMode = DISABLE;
292   hadc1.Init.NbrOfConversion = 1;
293   hadc1.Init.DiscontinuousConvMode = DISABLE;
294   hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIG_T3_TRGO;
295   hadc1.Init.ExternalTrigConvEdge =
         ADC_EXTERNALTRIGCONVEDGE_RISING;
296   hadc1.Init.DMAContinuousRequests = DISABLE;
297   hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
298   hadc1.Init.OversamplingMode = DISABLE;
299   if (HAL_ADC_Init(&hadc1) != HAL_OK)
300   {
301     Error_Handler();
302   }
303
```

```
304   /** Configure the ADC multi-mode
305   */
306   multimode.Mode = ADC_MODE_INDEPENDENT;
307   if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode)
          != HAL_OK)
308   {
309     Error_Handler();
310   }
311
312   /** Configure Regular Channel
313   */
314   sConfig.Channel = ADC_CHANNEL_1;
315   sConfig.Rank = ADC_REGULAR_RANK_1;
316   sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
317   sConfig.SingleDiff = ADC_SINGLE_ENDED;
318   sConfig.OffsetNumber = ADC_OFFSET_NONE;
319   sConfig.Offset = 0;
320   if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
321   {
322     Error_Handler();
323   }
324   /* USER CODE BEGIN ADC1_Init 2 */
325
326   /* USER CODE END ADC1_Init 2 */
327
```

```
328 }
329
330 /**
331   * @brief TIM3 Initialization Function
332   * @param None
333   * @retval None
334   */
335 static void MX_TIM3_Init(void)
336 {
337
338   /* USER CODE BEGIN TIM3_Init 0 */
339
340   /* USER CODE END TIM3_Init 0 */
341
342   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
343   TIM_MasterConfigTypeDef sMasterConfig = {0};
344   TIM_OC_InitTypeDef sConfigOC = {0};
345
346   /* USER CODE BEGIN TIM3_Init 1 */
347
348   /* USER CODE END TIM3_Init 1 */
349   htim3.Instance = TIM3;
350   htim3.Init.Prescaler = 80 - 1;
351   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
352   htim3.Init.Period = (int) floor(TIM_PERIOD);
```

```
353   htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
354   htim3.Init.AutoReloadPreload =
          TIM_AUTORELOAD_PRELOAD_DISABLE;
355   if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
356   {
357     Error_Handler();
358   }
359   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
360   if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig)
          != HAL_OK)
361   {
362     Error_Handler();
363   }
364   if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
365   {
366     Error_Handler();
367   }
368   sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
369   sMasterConfig.MasterSlaveMode =
          TIM_MASTERSLAVEMODE_DISABLE;
370   if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &
          sMasterConfig) != HAL_OK)
371   {
372     Error_Handler();
373   }
```

```
374   sConfigOC.OCMode = TIM_OCMODE_PWM1;
375   sConfigOC.Pulse = 65535;
376   sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
377   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
378   if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC,
        TIM_CHANNEL_1) != HAL_OK)
379   {
380     Error_Handler();
381   }
382   /* USER CODE BEGIN TIM3_Init 2 */
383
384   /* USER CODE END TIM3_Init 2 */
385   HAL_TIM_MspPostInit(&htim3);
386
387 }
388
389 /**
390   * @brief TIM6 Initialization Function
391   * @param None
392   * @retval None
393   */
394 static void MX_TIM6_Init(void)
395 {
396
397   /* USER CODE BEGIN TIM6_Init 0 */
```

```
398
399   /* USER CODE END TIM6_Init 0 */
400
401   TIM_MasterConfigTypeDef sMasterConfig = {0};
402
403   /* USER CODE BEGIN TIM6_Init 1 */
404
405   /* USER CODE END TIM6_Init 1 */
406   htim6.Instance = TIM6;
407   htim6.Init.Prescaler = 0;
408   htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
409   htim6.Init.Period = 65535;
410   htim6.Init.AutoReloadPreload =
        TIM_AUTORELOAD_PRELOAD_DISABLE;
411   if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
412   {
413     Error_Handler();
414   }
415   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
416   sMasterConfig.MasterSlaveMode =
        TIM_MASTERSLAVEMODE_DISABLE;
417   if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &
        sMasterConfig) != HAL_OK)
418   {
419     Error_Handler();
```

```
420   }
421   /* USER CODE BEGIN TIM6_Init 2 */
422
423   /* USER CODE END TIM6_Init 2 */
424
425 }
426
427 /**
428   * @brief USART2 Initialization Function
429   * @param None
430   * @retval None
431   */
432 static void MX_USART2_UART_Init(void)
433 {
434
435   /* USER CODE BEGIN USART2_Init 0 */
436
437   /* USER CODE END USART2_Init 0 */
438
439   /* USER CODE BEGIN USART2_Init 1 */
440
441   /* USER CODE END USART2_Init 1 */
442   huart2.Instance = USART2;
443   huart2.Init.BaudRate = 115200;
444   huart2.Init.WordLength = UART_WORDLENGTH_8B;
```

```
445   huart2.Init.StopBits = UART_STOPBITS_1;
446   huart2.Init.Parity = UART_PARITY_NONE;
447   huart2.Init.Mode = UART_MODE_TX_RX;
448   huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
449   huart2.Init.OverSampling = UART_OVERSAMPLING_16;
450   huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
451   huart2.AdvancedInit.AdvFeatureInit =
        UART_ADVFEATURE_NO_INIT;
452   if (HAL_UART_Init(&huart2) != HAL_OK)
453   {
454     Error_Handler();
455   }
456   /* USER CODE BEGIN USART2_Init 2 */
457
458   /* USER CODE END USART2_Init 2 */
459
460 }
461
462 /**
463   * Enable DMA controller clock
464   */
465 static void MX_DMA_Init(void)
466 {
467
468   /* DMA controller clock enable */
```

```
469     __HAL_RCC_DMA1_CLK_ENABLE();
470
471     /* DMA interrupt init */
472     /* DMA1_Channel7_IRQn interrupt configuration */
473     HAL_NVIC_SetPriority(DMA1_Channel7_IRQn, 0, 0);
474     HAL_NVIC_EnableIRQ(DMA1_Channel7_IRQn);
475
476 }
477
478 /**
479     * @brief GPIO Initialization Function
480     * @param None
481     * @retval None
482     */
483 static void MX_GPIO_Init(void)
484 {
485     GPIO_InitTypeDef GPIO_InitStruct = {0};
486 /* USER CODE BEGIN MX_GPIO_Init_1 */
487 /* USER CODE END MX_GPIO_Init_1 */
488
489     /* GPIO Ports Clock Enable */
490     __HAL_RCC_GPIOC_CLK_ENABLE();
491     __HAL_RCC_GPIOH_CLK_ENABLE();
492     __HAL_RCC_GPIOA_CLK_ENABLE();
493     __HAL_RCC_GPIOB_CLK_ENABLE();
```

```
494
495     /*Configure GPIO pin Output Level */
496     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0|LD2_Pin,
            GPIO_PIN_RESET);
497
498     /*Configure GPIO pin : B1_Pin */
499     GPIO_InitStruct.Pin = B1_Pin;
500     GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
501     GPIO_InitStruct.Pull = GPIO_NOPULL;
502     HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
503
504     /*Configure GPIO pins : PA0 LD2_Pin */
505     GPIO_InitStruct.Pin = GPIO_PIN_0|LD2_Pin;
506     GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
507     GPIO_InitStruct.Pull = GPIO_NOPULL;
508     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
509     HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
510
511 /* USER CODE BEGIN MX_GPIO_Init_2 */
512 /* USER CODE END MX_GPIO_Init_2 */
513 }
514
515 /* USER CODE BEGIN 4 */
516 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
517 {
```

```
518   /* USER CODE BEGIN Callback 1 */
519   if (htim->Instance == TIM6)
520   {
521     tim_turnover++;
522   }
523   /* USER CODE END Callback 1 */
524 }
525 /* USER CODE END 4 */
526
527 /**
528   * @brief  This function is executed in case of error
          occurrence.
529   * @retval None
530   */
531 void Error_Handler(void)
532 {
533   /* USER CODE BEGIN Error_Handler_Debug */
534   /* User can add his own implementation to report the HAL
          error return state */
535   __disable_irq();
536   while (1)
537   {
538   }
539   /* USER CODE END Error_Handler_Debug */
540 }
```

```
541
542 #ifdef  USE_FULL_ASSERT
543 /**
544   * @brief  Reports the name of the source file and the
        source line number
545   *         where the assert_param error has occurred.
546   * @param  file: pointer to the source file name
547   * @param  line: assert_param error line source number
548   * @retval None
549   */
550 void assert_failed(uint8_t *file, uint32_t line)
551 {
552   /* USER CODE BEGIN 6 */
553   /* User can add his own implementation to report the file
        name and line number,
554      ex: printf("Wrong parameters value: file %s on line %d\
          r\n", file, line) */
555   /* USER CODE END 6 */
556 }
557 #endif /* USE_FULL_ASSERT */
```

I.9 MATLAB Simulation and Comparison Environment

```
clear
close all
clc

%% Environment Variables
N = 4096;
sz = N/2+1;
fs = 5000;
fc = 261.63;
fg = 783.99;
fe = 659.25;
dt = 1/fs;
t = 0:dt:(N-1)*dt;
i = 0:N-1;
f = linspace(0,fs/2,N/2+1);
h = hann(N)';

%% Signal Generation
% x = sin(2*pi*200*t) + 1;
x = (sin(2*pi*1000*t)+1) + 0.5*(cos(2*pi*200*t)+1);
% x = (sin(2*pi*fc*t) + sin(2*pi*fg*t) + sin(2*pi*fe*t)); %
    Create C Chord Sinusoid

```

```
23 T = load_data("Data_Files\SR_Real\SR_ADC-1k-200_4096.csv",
       sz);
24
25 xUnwin = x;
26 x = h.*x; % Apply Hanning Window
27
28 X = abs(fft(x,N))/N; % Normalized Magnitude of fft output
29 X = X(1:N/2+1); % Single-sided conversion
30 X(2:end-1) = sqrt(2).*X(2:end-1); % Amplitude Vrms for non-
       DC components
31 % X = 2*X; % Window Amplitude Correction Factor
32
33 %% Results plotting
34 figure
35 plot(t,xUnwin);
36 title("Unwindowed Signal in Time Domain")
37 xlabel("Time (s)")
38 ylabel("Amplitude (V)")
39
40 figure
41 plot(t,x);
42 title("Windowed Signal in Time Domain")
43 xlabel("Time (s)")
44 ylabel("Amplitude (V)")
45
```

```
figure
subplot(2,1,1)
plot(f,X);
title("MATLAB FFT Results")
xlim([-1 fs/2])
xlabel("Frequency (Hz)")
ylabel("Amplitude (Vrms)")

subplot(2,1,2)
plot(f, T);
title("CTP FFT Results")
xlim([-1 fs/2])
xlabel("Frequency (Hz)")
ylabel("Amplitude (Vrms)")

function C = load_data(CDataPath, CN)

    %% Import Data From C Program
    opts = delimitedTextImportOptions("NumVariables", CN);

    % Specify range and delimiter
    opts.DataLines = [3, 3];
    opts.Delimiter = ",";
    [vartypes{1, 1:CN}] = deal('double');
    opts.VariableTypes = vartypes;
```

```
71
72      % Specify file level properties
73      opts.ExtraColumnsRule = "ignore";
74      opts.EmptyLineRule = "read";
75      opts.ConsecutiveDelimitersRule = "join";
76
77      % Import the data
78      C = readmatrix(CDataPath, opts);
79
80      % Clear temporary variables
81      clear opts
82  end
```