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

Ty Freeman
tkf1604@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation

This Master's Project is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@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 ix
List of Tables xiii

1 Introduction 1
  1.1 Research Goals .................................................. 2
  1.2 Contributions .................................................... 3
  1.3 Organization ...................................................... 3

2 Bibliographical Research 4
  2.1 A brief history of the FFT ...................................... 4
  2.2 FFT Derivation ................................................... 6
    2.2.1 DFT Derivation for $N = 4$ .................................. 6
    2.2.2 Pattern Extraction .......................................... 11
      2.2.2.1 Relating to the values of $x_t$ .......................... 11
      2.2.2.2 Relating to the Roots of Unity ......................... 12
      2.2.2.3 Putting the Patterns Together ......................... 13
    2.2.3 FFT for $N = 8$ .............................................. 14
## Contents

2.3 Alternate FFT Algorithms .............................................. 17
2.4 Continuous Flow FFTs ............................................... 19
  2.4.1 Windowing Continuous Flow FFT’s ............................. 20
2.5 Real-Time Operating Systems ..................................... 23
  2.5.1 Modern RTOS .................................................. 23
  2.5.2 Problems with an RTOS ...................................... 24
  2.5.3 Use of an RTOS .............................................. 25

3 Program Architectures ............................................... 27
  3.1 Data Path Overview ................................................ 27
  3.2 \textit{read\_Task} .................................................... 29
    3.2.1 Top of Program ............................................. 32
    3.2.2 End of Data Path .......................................... 33
    3.2.3 End of Program ............................................ 34
    3.2.4 UART Rx Callback ......................................... 35
    3.2.5 UART Tx Callback .......................................... 36
  3.3 \textit{adc\_Task} ..................................................... 36
    3.3.1 ADC Main Functionality ................................... 37
    3.3.2 Ending Conditions .......................................... 39
    3.3.3 ADC Conversion Callback ................................ 40
  3.4 \textit{txm\_Task} ..................................................... 40
  3.5 \textit{ansys\_Task} .................................................. 45
  3.6 FFT Implementation .............................................. 49
    3.6.1 Helper Functions .......................................... 51
  3.7 Global Header ..................................................... 53
## Contents

### I Source Code

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.1 Main</td>
<td>I-1</td>
</tr>
<tr>
<td>I.2 UART</td>
<td>I-26</td>
</tr>
<tr>
<td>I.3 ADC</td>
<td>I-35</td>
</tr>
<tr>
<td>I.4 Transform</td>
<td>I-41</td>
</tr>
<tr>
<td>I.5 Analysis</td>
<td>I-47</td>
</tr>
<tr>
<td>I.6 Global Header</td>
<td>I-52</td>
</tr>
<tr>
<td>I.7 FFT</td>
<td>I-55</td>
</tr>
<tr>
<td>I.8 FFT Test Main</td>
<td>I-60</td>
</tr>
<tr>
<td>I.9 MATLAB Simulation and Comparison Environment</td>
<td>I-85</td>
</tr>
</tbody>
</table>
# List of Figures

2.1  $N = 2$ Butterfly Diagram .................................................. 16  
2.2  $N = 8$ Butterfly Diagram .................................................... 17  
2.3  Rectangular Filter in Time and Frequency Domain ....................... 21  
2.4  Hanning Window in Time Domain and Frequency Domain ............... 23  
3.1  Program Flowchart ............................................................... 29  
3.2  *read_Task* Signal Flow Diagram ............................................. 31  
3.3  Start of *read_Task* ............................................................ 32  
3.4  End of *read_Task* .............................................................. 33  
3.5  End of Program Statistics Printing .......................................... 34  
3.6  UART Rx Callback .............................................................. 35  
3.7  UART Tx Callback Function .................................................. 36  
3.8  Signal Flowchart of *adc_Task* ............................................... 37  
3.9  *adc_Task* Main Functionality Block ...................................... 38  
3.10  *adc_Task* Wrap-up Portion ................................................ 39  
3.11  ADC Conversion Callback Function ....................................... 40  
3.12  Transform Task Initialization Phase ..................................... 41  
3.13  *stats_t* Typedef Definition ............................................... 41
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14</td>
<td>Transform Task Main Functionality</td>
<td>43</td>
</tr>
<tr>
<td>3.15</td>
<td>Signal Flowchart for <em>txm_task</em></td>
<td>44</td>
</tr>
<tr>
<td>3.16</td>
<td><em>ansys_Task</em> Signal Flow Diagram</td>
<td>46</td>
</tr>
<tr>
<td>3.17</td>
<td>Analysis Task Function</td>
<td>48</td>
</tr>
<tr>
<td>3.18</td>
<td>User Defined Function for Calculating Magnitude of Complex Numbers</td>
<td>48</td>
</tr>
<tr>
<td>3.19</td>
<td>Butterfly Loops</td>
<td>50</td>
</tr>
<tr>
<td>3.20</td>
<td>Results Buffer Reordering</td>
<td>51</td>
</tr>
<tr>
<td>3.21</td>
<td>Bit Reversal Algorithm</td>
<td>52</td>
</tr>
<tr>
<td>3.22</td>
<td>$\log_2$ Function</td>
<td>53</td>
</tr>
<tr>
<td>3.23</td>
<td>Test Program Signal Creation</td>
<td>55</td>
</tr>
<tr>
<td>3.24</td>
<td>Test Program Results Manipulation</td>
<td>55</td>
</tr>
<tr>
<td>3.25</td>
<td>Test Program Ending</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>MATLAB Environment Variables</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>MATLAB Generated Data Importing</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>MATLAB Math and Data Plotting</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Terminal Output for 200 Hz Single Frequency Test Validation</td>
<td>62</td>
</tr>
<tr>
<td>4.5</td>
<td>200 Hz Windowed Test Signal</td>
<td>63</td>
</tr>
<tr>
<td>4.6</td>
<td>MATLAB and Test Program 200 Hz Results</td>
<td>63</td>
</tr>
<tr>
<td>4.7</td>
<td>MATLAB and Test Program 200 Hz Results with 4096 Data Points</td>
<td>64</td>
</tr>
<tr>
<td>4.8</td>
<td>MATLAB and Test Program 200 Hz Real Signal</td>
<td>65</td>
</tr>
<tr>
<td>4.9</td>
<td>MATLAB and Test Program 200 Hz Real Signal, N = 4096</td>
<td>66</td>
</tr>
<tr>
<td>4.10</td>
<td>200 Hz Windowed Sine Waveform</td>
<td>67</td>
</tr>
<tr>
<td>4.11</td>
<td>MATLAB and Test Program Dual-Tone Results</td>
<td>68</td>
</tr>
<tr>
<td>4.12</td>
<td>MATLAB and Test Program Dual-Tone Results, N = 4096</td>
<td>69</td>
</tr>
</tbody>
</table>
4.13 MATLAB and Test Program Dual-Tone Real Signal Results . . . . . . . . . . 70
4.14 MATLAB and Test Program Dual-Tone Real Signal Results, N = 4096 . . . . 71
4.15 Real Dual-Tone Signal from Signal Generator . . . . . . . . . . . . . . . . . 72
4.16 MATLAB Generated Dual-Tone Signal . . . . . . . . . . . . . . . . . . . . 72
4.17 C Chord Triad Test Signal . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.18 MATLAB and Test Program Triad Results . . . . . . . . . . . . . . . . . . 74
4.19 MATLAB and Test Program Triad Results, N = 4096 . . . . . . . . . . . . . 75
4.20 FFT of Triad Signal, N = 256, Fs = 44.1k . . . . . . . . . . . . . . . . . . . 75
4.21 FFT of Triad Signal, N = 4096, Fs = 44.1k . . . . . . . . . . . . . . . . . . . 76
4.22 Frequency Content of Audio File, N = 256 . . . . . . . . . . . . . . . . . . . 77
4.23 Frequency Content of Vocal Audio File . . . . . . . . . . . . . . . . . . . . 78
4.24 Frequency Content of Audio File Using Full File . . . . . . . . . . . . . . . 79
4.25 Data Path Validation Test 1: Limit = 5 . . . . . . . . . . . . . . . . . . . . 81
4.26 Data Path Validation Test 2: Limit = 35 . . . . . . . . . . . . . . . . . . . . 82
4.27 Data Path Validation Test 3: Limit = 150 . . . . . . . . . . . . . . . . . . . . 82
4.28 Data Path Validation Test 1: Limit = 5, N = 128 . . . . . . . . . . . . . . . . 83
4.29 Data Path Validation Test 1: Limit = 35, N = 128 . . . . . . . . . . . . . . . . 83
4.30 Data Path Validation Test 1: Limit = 150, N = 128 . . . . . . . . . . . . . . 84
4.31 Terminal Printing Results, Limit = 4 . . . . . . . . . . . . . . . . . . . . . 85
4.32 Terminal Printing Results, Limit = 4 . . . . . . . . . . . . . . . . . . . . . 85
4.33 Program Statistics for Test 1 . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.34 C Program Spectrogram of 50Hz-2kHz Sweep in 500 ms . . . . . . . . . . . 87
4.35 MATLAB Spectrogram of 50Hz-2kHz Sweep in 500 ms . . . . . . . . . . . . 87
4.36 C Program Spectrogram of 50Hz-2kHz Sweep in 5s with 500 mV offset . . . 89
4.37 MATLAB Spectrogram of 50Hz-2kHz Sweep in 5s with 500 mV offset . . . 89
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.38</td>
<td>Program Statistics for Test 3 Before Adjustment</td>
<td>90</td>
</tr>
<tr>
<td>4.39</td>
<td>Program Statistics for Test 3 After Adjustment</td>
<td>90</td>
</tr>
<tr>
<td>4.40</td>
<td>C Program Spectrogram of 1Hz-200Hz Sweep with 500 mV offset</td>
<td>91</td>
</tr>
<tr>
<td>4.41</td>
<td>MATLAB Spectrogram of 1Hz-200Hz Sweep with 500 mV offset</td>
<td>92</td>
</tr>
<tr>
<td>4.42</td>
<td>C Program Spectrogram of 1Hz-200Hz Sweep in 5s with 500 mV offset</td>
<td>93</td>
</tr>
<tr>
<td>4.43</td>
<td>MATLAB Spectrogram of 1Hz-200Hz Sweep with 500 mV offset</td>
<td>93</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Enumeration of Eq. (2.8) in Base 10 Truth Table Form</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Enumeration of Eq. (2.9) in Base 10 Truth Table Representation</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Rearranged Tbl. 2.1 Without Primary $m$ Values</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Revision of Tbl. 2.3 using Eq. 2.10</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Revision of Tbl. 2.2 using Eq. 2.10</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>Relative Efficiencies of Varying Radix FFT’s</td>
<td>19</td>
</tr>
</tbody>
</table>
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 $N \log(N)$ 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 project 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
2.1 A brief history of the FFT

A 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 vein 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

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

(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,

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

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

where $W$ represents the $N^{th}$ complex root of unity: $e^{j2\pi/N}$. 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

\[
k = 2k_1 + k_0 \quad n = 2n_1 + n_0 \quad \text{(2.4)}
\]

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

\[
X(n_1, n_0) = \sum_{k_1=0}^{1} \sum_{k_0=0}^{1} x_0(k_1, k_0)W^{(2n_1+n_0)(2k_1+k_0)} \quad \text{(2.5)}
\]

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

\[ [W^4]^{n_1k_1} = [e^{j2\pi4/4}]^{n_1k_1} = 1^{n_1k_1} = 1 \]

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 since \(k_1\) 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.
2.2 FFT Derivation

\[ x_1(0,0) = x_0(0,0)W^0 + x_0(1,0)W^0 \]
\[ x_1(0,1) = x_0(0,1)W^0 + x_0(1,1)W^0 \]
\[ x_1(1,0) = x_0(0,0)W^0 + x_0(1,0)W^2 \]
\[ x_1(1,1) = x_0(0,1)W^0 + x_0(1,1)W^2 \] (2.8)

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

<table>
<thead>
<tr>
<th></th>
<th>( k_1 = 0 )</th>
<th>( k_1 = 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1(n_0,k_0) )</td>
<td>( x_0(k_1,k_0) )</td>
<td>( m )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ x_2(n_0,n_1) = \sum_{k_0=0}^{4} x_1(n_0,k_0)W^{(2n_1+n_0)(k_0)} \] (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.
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.

\[
\begin{align*}
X(0,0) &= X(0) = x_2(0) = x_2(0,0) \\
X(0,1) &= X(1) = x_2(2) = x_2(1,0) \\
X(1,0) &= X(2) = x_2(1) = x_2(0,1) \\
X(1,1) &= X(3) = x_2(3) = x_2(1,1)
\end{align*}
\]

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
2.2 FFT Derivation

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>
<thead>
<tr>
<th>$k_1 = 0$</th>
<th>$k_1 = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1(n_0, k_0)$</td>
<td>$x_0(k_1, k_0)$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

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.
2.2 FFT Derivation

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

$$W^m = -W^{\left(\frac{N}{2}\right)^m} \quad (2.10)$$

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

$$W^2 = e^{i2\pi/4} = e^{i\pi} = -1 = -W^0$$

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 $\left(\frac{N}{2}\right)m_1 = m_2$ 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.
2.2 FFT Derivation

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

<table>
<thead>
<tr>
<th></th>
<th>$k_1 = 0$</th>
<th>$k_1 = 1$</th>
<th>$x_0(k_1, k_0)$</th>
<th>$x_0(k_1, k_0)$</th>
<th>$x_0(k_1, k_0)$</th>
<th>Sign of $W$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>+</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>+</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>$k_0 = 0$</th>
<th>$k_0 = 1$</th>
<th>$x_1(n_0, k_0)$</th>
<th>$x_1(n_0, k_0)$</th>
<th>$x_1(n_0, k_0)$</th>
<th>Sign of $W$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>+</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $\gamma$ as the current summation or stage indexed from 0:

\begin{align*}
x_{t+1}(n) &= x_t(n) + x_t(n+a)W^m \\
x_{t+1}(n+a) &= x_t(n) - x_t(n+a)W^m
\end{align*}

(2.11)  
(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

\begin{align*}
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
\end{align*}

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

\begin{align*}
k &= 4k_2 + 2k_1 + k_0 \\
n &= 4n_2 + 2n_1 + n_0
\end{align*}

(2.13)

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

\[
W^{(4n_2+2n_1+n_0)k_2}W^{(4n_2+2n_1+n_0)k_1}W^{(4n_2+2n_1+n_0)k_0}
\]

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

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

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

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

\[
x_3(n_0,n_1,n_2) = \sum_{k_0=0}^{1} x_1(n_0,n_1,k_0)W^{(4n_2+2n_1+n_0)k_0}
\]  
(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](image)

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.


2.4 Continuous Flow FFTs

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

Table 2.6: Relative Efficiencies of Varying Radix FFT’s

<table>
<thead>
<tr>
<th>$r$</th>
<th>$\frac{r}{\log_2 r}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1.88</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2.15</td>
</tr>
<tr>
<td>6</td>
<td>2.31</td>
</tr>
<tr>
<td>7</td>
<td>2.49</td>
</tr>
<tr>
<td>8</td>
<td>2.67</td>
</tr>
<tr>
<td>9</td>
<td>2.82</td>
</tr>
<tr>
<td>10</td>
<td>3.01</td>
</tr>
</tbody>
</table>
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 \( t = \frac{N}{2} \times T \), 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](image)
The window utilized in this paper is a Hanning window which is defined as:

\[ h(n) = 0.5 \times (1 - \cos(2\pi \frac{n}{N})) \], \quad n = 0, \ldots, N - 1 \quad (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].
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
2.5 Real-Time Operating Systems

will have more jitter but can still usually meet a required deadline, while hard RTOSs can deterministically meet their required deadlines [#open_sourceRTOS]. 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. 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
3.1 Data Path Overview

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 $\frac{N}{2} + 1$ 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 *readTask*

The *readTask* 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
readTask 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 readTask can be seen in Fig. 3.2.
Figure 3.2: *read_Task* Signal Flow Diagram
3.2 read_Task

```c
void read_Task(void * pvParameters)
{
  TickType_t lastWake = 0;
  TickType_t Period = pdMS_TO_TICKS(5);
  float fft_res[SMO_2+1];
  char TxBuf[(SMO_5)+STR_S2];
  int avg_N2 = 0;
  int avg_N1 = 0;

  HAL_UART_Transmit(huart2, caret, sizeof(caret), 2); // Print starting CMD caret

  while(1)
  {
    HAL_UARTEx_ReceiveIdle_DMA(huart2, &xdbuf, 1); // Begin DMA
    HAL_DMA_DISABLE_IT(huart2_dma, DMA_IT_HT);

    if(cr_flg)
    {
      cr_flg = 0; // Clear carriage return flag
      ttl_pages = stoi(nbuf); // Extract user set limit
      if(ttl_pages < sizeof(nbuf)) // Clear receive buffer
        HAL_UART_Transmit(huart2, &nbuf, sizeof(nbuf), 1); // Send buffer
      xTaskCreate(read_Task, "send", 104, (void *) ttl_pages, PriorityNormal, 4);  
    }
  }
}
```

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 \textit{adc\_Task}.

### 3.2.2 End of Data Path

The \textit{read\_Task} sits idle again after beginning the \textit{adc\_Task} instance. At this point, the \textit{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 \textit{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 \textit{if\text{-}statement} at line 139 in Fig. 3.4 returns false if no objects are waiting in the \textit{res\_mbx} queue.

```c
if(xQueueReceive(res_mb, fft_res, 0) && bufRdy_flg)  // Results ready and Tx buffer clear
{
    uart_pkgs++;
    for(int i=0; i< SMP_2+1; i++)
        { /* Maintain standard 10 bytes of data after printf */
            if(fft_res[i] > 9 && fft_res[i] < 100)
                sprintf(TxBuf+i*STR_SZ, "%3.1f", fft_res[i]);
            else if(fft_res[i] > 99)
                sprintf(TxBuf+i*STR_SZ, "%4.1f", fft_res[i]);
            else
                sprintf(TxBuf+i*STR_SZ, "%5.2f", fft_res[i]);
        }
    HAL_UART_Transmit_DMA(huart2, (unsigned *) TxBuf, sizeof(TxBuf));  // Start DMA
    bufRdy_flg = 0;  // tx buffer being used by new data
}
```

Figure 3.4: End of \textit{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 \textit{for loop} formats the incoming floating point data into a 10-
3.2 read_Task

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.

Figure 3.5: End of Program Statistics Printing
3.2 read_Task

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 \textit{pkgs\_sent} counter and then checks ending conditions. For example, if the analysis task has signaled it has completed its functionality through the \textit{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 \textit{PROG\_END} flag.

![Figure 3.7: UART Tx Callback Function](image)

3.3 \textit{adc\_Task}

Like the \textit{uart\_Task}, the \textit{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.

![Signal Flowchart of *adc_Task*](image)

**Figure 3.8: Signal Flowchart of *adc_Task***

### 3.3.1 ADC Main Functionality
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 Section 3.4.

This section begins with the setting of the \( \text{adcDone\_flg} \), 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.

```c
if(lim_flg || sigDone_flg) // Two possible ending conditions
{
    /* Turn off Timer and ADC before deleting the adc task */
    HAL_TIM_PWM_Stop(htim3, TIM_CHANNEL_1);
    HAL_ADC_Stop_IT(hadc3);
    adcDone_flg = 1; // Signal program that adc is done
    if(sigDone_flg) pkg_cnt -= num_emptyBuf; // Remove empty buffers from total count
    sigDone_flg = 0; // Clear signal flag
    lim_flg = 0; // Clear limit flag
    vTaskDelete(NULL);
}
```

Figure 3.10: \( \text{adc\_Task} \) 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.

```c
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) // ADC callback function
{
    RWM[idx] = HAL_ADC_GetValue(&hadc1); // Acquire ADC value
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5); // Toggle LED
    if(idx -- SAMPLES-1) // Check Index value. Restart?
    {
        full = 1;
        idx = SMP_2; // Only saving half of samples for window overlapping
    }
    else
        idx++;
}
```

Figure 3.11: ADC Conversion Callback Function

3.4 \textit{txm\_Task}

Unlike the previous two tasks, the transform task, \textit{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 \textit{quiet\_cnt} representing the counter used in the transform task to track the number of empty packages seen. The \textit{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.

```c
void tnm_Task(void *pvParameters)
{
    TickType_t lastWake = 0;
    TickType_t Period = pdMS_TO_TICKS(20);
    uint16_t *rec = NULL;
    stats_t res_fft;

    tm2full = (float) SAMPLES/FS; // Time it takes to fill one buffer worth of samples
    quiet_cnt = (int) floor(3/tm2full); // 3 seconds of silence signals end of incoming signal
    num_emptyBuf = quiet_cnt;

    while(1)
    {
    }
}

typedef struct stats{
    unsigned int pkg_num;
    unsigned long mult_cnt;
    unsigned long add_cnt;
    int zCnt;
    float complex res_buf[SAMPLES];
} stats_t;
```

Figure 3.12: Transform Task Initialization Phase

![Typedef Definition](image)

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
3.4 txm_Task

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.
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 \textit{fft\_res} variable, which receives data from the stats mailbox, is a struct of the typedef \textit{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 \textit{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:

\begin{equation}
A_{V_{rms}}(i) = \frac{\text{magnitude}(FFT(i))}{N}, \quad i = 0;
\end{equation}

For non-DC components:

\begin{equation}
A_{V_{rms}}(i) = \sqrt{2} \times \frac{\text{magnitude}(FFT(i))}{N}, \quad i = 1, 2, 3, \ldots, \frac{N}{2} + 1;
\end{equation}

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 \textit{mag} function in lines 63 and 67 is defined in the same \textit{analysis.c} file (1.5) and can be seen in Fig. 3.18. After the completion of this \textit{for-loop}, the data is finished being processed and sent into the results mailbox queue, \textit{res\_mbx}. With the data being sent away, the temporary buffer is freed. Once the \textit{ansys\_Task} sees the \textit{txmDone\_flg} is set and it processed all data that was passed to it, the analysis task signals it has finished with the \textit{ansyDone\_flg} and suspends itself to limit the context switching occurring during the final print statements.
3.5 ansys_Task

```c
// void ansys_task(void * pParameters)
{
    TickType_t lastMake = 0;
    TickType_t period = pdMS_TO_TICKS(10);
    stats_t fft_res;

    float *temp_buf = NULL;

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

            for(int i = 0; i < N/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] = sqrtC1*mag(fft_res.res_buf[i])/SAMPLES; // Convert to Amplitude real value
                }
            }

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

            if(nQueueSend(res_mbx, temp_buf, 0) != -1) ansys_pkpop++; // Send smaller buffer for printing
        }

        else ansys_task fail++;

        free(temp_buf);

        // For averages at end of the program */
        Total_real += fft_res.real_cnt;
        Total_imag += fft_res.imag_cnt;

        if(tsmDone_flg && nQueueMessagesWaiting(stats_mbx) == 0)
        {
            ansysDone_flg = 1;
            vTaskDelay(pdMS_TO_TICKS(400));
        }
    }
}
```

Figure 3.17: Analysis Task Function

```c
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.
3.6 FFT Implementation

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.
3.6 FFT Implementation

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.
The second helper function is an implementation of the log2() 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.
3.7 Global Header

Many of the important program signals are defined or shared through a header file named `global.h`. 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
3.8 C Test Program

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.

Figure 3.24: Test Program Results Manipulation
3.8 C Test Program

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.

```c
if (done)
{
    HAL_UART_Abort(&huart2);
    printf("\n\n\n***** STATS *****\n\n");
    printf("Time per transform: %f us\n", (float) fft_res.time/80);
    printf("Mul Ops per transform: %d\n", fft_res.mul_cnt);
    printf("Add Ops per transform: %d\n", fft_res.add_cnt);
    exit(1);
}
```

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.

Notice in Fig. 4.1 the built-in $\text{hann()}$ 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 $\text{fft()}$ 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.
4.1 MATLAB Validation Environment

values of $V_{rms}$. 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.

```matlab
function C = load_data(CDataPath, N)

    % Import Data From C Program
    opts = delimitedTextImportOptions("NumVariables", N);
    opts.DataLines = [3, 3];
    opts.Delimiter = ",,;
    [vartypes{1, 1:N}] = deal('double');
    opts.VariableTypes = vartypes;

    % Specify file level properties
    opts.ExtraColumnsRule = "ignore";
    opts.EmptyLineRule = "read";
    opts.ConsecutiveDelimitersRule = "join";

    % Import the data
    C = readmatrix(CDataPath, opts);

    % Clear temporary variables
    clear opts

end
```

Figure 4.2: MATLAB Generated Data Importing
4.1 MATLAB Validation Environment

```matlab
T = load_data("Data_Files\SR_Synth\SR_Dual-Tone.csv", sz);
X = abs(fft(x,N))/N; % Normalized Magnitude of fft output
X = X(1:N/2+1); % Single-sided conversion
X(2:end-1) = sqrt(2).*X(2:end-1); % Amplitude Vrms for non-DC components

%% Results plotting
figure
plot(t,xUnwin);
title("UnWindowed Signal in Time Domain")
xlabel("Time (s)")
ylabel("Amplitude (V)")

figure
plot(t,x);
title("Windowed Signal in Time Domain")
xlabel("Time (s)")
ylabel("Amplitude (V)")

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("CTR FFT Results")
xlim([-1 fs/2])
xlabel("Frequency (Hz)")
ylabel("Amplitude (Vrms)")
```

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
4.2 Test Program 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.

![MATLAB FFT Results](image1)

![CIP FFT Results](image2)

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-sigaled tests.

![MATLAB and Test Program 200 Hz Real Signal](image)

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.
4.2 Test Program Validation

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:

\[ x = (\sin(2\pi \times 1000) + 1) + 0.5 \times (\cos(2\pi \times 200) + 1) \]  

(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.

![200 Hz Windowed Sine Waveform](image)

Figure 4.10: 200 Hz Windowed Sine Waveform
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.
4.2.2 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.
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.
4.2 Test Program Validation

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 Test Program Validation

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.

![MATLAB FFT Results](image1)

![CTP FFT Results](image2)

**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.
4.2 Test Program Validation

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.

![MATLAB FFT Results](image)

![CTF FFT Results](image)

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.

![MATLAB FFT Results](image1)

![CFIT FFT Results](image2)

**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.

![Frequency Content of Audio File Using Full File](image)

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 \textit{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 \textit{uart\_Task} will also only increment its counter once since it only receives data. However, there is another counter found in the \textit{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.
4.3 Main Program Validation

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.
4.3 Main Program Validation

***** STATS *****
Average Time per transform: 13969.599609 uS
Average Multi Ops per transform: 1024
Average Add Ops per transform: 2048
Number of Rx Failures: 0
Number of Tx Failures: 0
Packages seen by Task:

<table>
<thead>
<tr>
<th>Task</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>35</td>
</tr>
<tr>
<td>TXM</td>
<td>70</td>
</tr>
<tr>
<td>ANSY</td>
<td>70</td>
</tr>
<tr>
<td>UART</td>
<td>35</td>
</tr>
<tr>
<td>SENT</td>
<td>34</td>
</tr>
</tbody>
</table>

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

***** STATS *****
Average Time per transform: 14031.426758 uS
Average Multi Ops per transform: 1024
Average Add Ops per transform: 2048
Number of Rx Failures: 0
Number of Tx Failures: 0
Packages seen by Task:

<table>
<thead>
<tr>
<th>Task</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>150</td>
</tr>
<tr>
<td>TXM</td>
<td>300</td>
</tr>
<tr>
<td>ANSY</td>
<td>300</td>
</tr>
<tr>
<td>UART</td>
<td>150</td>
</tr>
<tr>
<td>SENT</td>
<td>148</td>
</tr>
</tbody>
</table>

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.

```
***** STATS *****
Average Time per transform: 33562.199219 µS
Average Mul Ops per transform: 448
Average Add Ops transform: 896
Number of Rx Failures: 0
Number of Tx Failures: 0
Packages seen by Task:
   ADC  5
   TXM  10
   AMSY 10
   UART  5
   SENT  5
```

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

```
***** STATS *****
Average Time per transform: 33782.601562 µS
Average Mul Ops per transform: 448
Average Add Ops transform: 896
Number of Rx Failures: 0
Number of Tx Failures: 0
Packages seen by Task:
   ADC  35
   TXM  70
   AMSY 79
   UART  35
   SENT  36
```

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

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.

---

#### Table 4.3.2.1

<table>
<thead>
<tr>
<th>Task</th>
<th>Packages seen</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>150</td>
</tr>
<tr>
<td>IXM</td>
<td>300</td>
</tr>
<tr>
<td>ANSY</td>
<td>300</td>
</tr>
<tr>
<td>UART</td>
<td>150</td>
</tr>
<tr>
<td>SENT</td>
<td>157</td>
</tr>
</tbody>
</table>

**Figure 4.30: Data Path Validation Test 1: Limit = 150, N = 128**
4.3 Main Program Validation

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 $V_{rms}^2/Hz$. 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](image)
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.
4.3 Main Program Validation

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](image)

![Figure 4.39: Program Statistics for Test 3 After Adjustment](image)
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
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.
4.3 Main Program Validation

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 $\mu$V 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


veloping Higher Radix FFT Algorithms,” in *APCCAS 2006 - 2006 IEEE Asia Pacific 

tional Academy of Sciences of the United States of America*, vol. 73, no. 4, pp. 1005–6, 
Apr. 1976.


Time-Varying Amplitude and Frequency Characteristic of Sub-Synchronous Oscillation 
Based on Sliding Window FFT,” in *2018 China International Conference on Electricity 

lation 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.


Appendix I

Source Code

I.1  Main

/* USER CODE BEGIN Header */
/**
 * @file               : main.c
 * @brief              : Task, queue, peripheral initialization
       and starting of RTOS scheduler

 * @attention
 * *
 * Copyright (c) 2022 STMicroelectronics.
 * All rights reserved.
 */
* This software is licensed under terms that can be found in the LICENSE file
* in the root directory of this software component.
* If no LICENSE file comes with this software, it is provided AS-IS.

**************************************************************************
/* USER CODE END Header */
/* Includes */
------------------------------------------------------------------
#include "main.h"

/* Private includes */
-----------------------------------------------
/* USER CODE BEGIN Includes */

#include "FreeRTOS.h"
#include "timers.h"
#include "queue.h"
#include "semphr.h"
#include "event_groups.h"
#include "task.h"

// General Includes
#include <stdlib.h>
#include <stdio.h>
#include <complex.h>
#include "math.h"

// File Specific Includes
#include "global.h"
#include "transform.h"
#include "analysis.h"
#include "uart.h"
#include "adc.h"

/* USER CODE END Includes */

/* Private typedef

* /

/* USER CODE BEGIN PTD */

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

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 */
/* Private user code
   -----------------------------------------------
   */

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
   * @brief The application entry point.
   * @retval int
   */

int main(void)
{
    /* USER CODE BEGIN 1 */

    /* USER CODE END 1 */

    /* MCU Configuration
       -----------------------------------------------
       */

    /* Reset of all peripherals, Initializes the Flash
       interface and the Systick. */
HAL_Init();

/* USER CODE BEGIN Init */

/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_USART2_UART_Init();
MX_ADC1_Init();
MX_TIM6_Init();
MX_TIM3_Init();

/* USER CODE BEGIN 2 */
HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED); // Calibrate ADC on startup

/* USER CODE END 2 */
/ * USER CODE BEGIN RTOS_MUTEX */
/* add mutexes, ... */
/* USER CODE END RTOS_MUTEX */

/* USER CODE BEGIN RTOS_SEMAPHORES */
/* add semaphores, ... */
/* USER CODE END RTOS_SEMAPHORES */

/* USER CODE BEGIN RTOS_TIMERS */
/* start timers, add new ones, ... */
/* USER CODE END RTOS_TIMERS */

/* USER CODE BEGIN RTOS_QUEUES */
stats_mbx = xQueueCreate(5, sizeof(stats_t));
if(stats_mbx == NULL)
{
    printf("Could not build stats mailbox\n\r");
    exit(1);
}

buffer_mbx = xQueueCreate(2, SAMPLES*sizeof(uint16_t));
if(buffer_mbx == NULL)
{
    printf("Could not build buffer mailbox\n\r");
    exit(1);
res_mbx = xQueueCreate(20, (SMP_2+1)*sizeof(float));
if(res_mbx == NULL)
{
    printf("Could not build results mailbox\n\r");
    exit(1);
}

/* USER CODE END RTOS_QUEUES */

/* Create the thread(s) */
/* USER CODE BEGIN RTOS_THREADS */
xTaskCreate(read_Task, "rdr", 1024, NULL, PriorityNormal, &rdr);
xTaskCreate(txm_Task, "tf", 10000, NULL, PriorityHigh, &tf);
xTaskCreate(ansys_Task, "ansys", 2048, NULL, PriorityNormal, &ansys);

/* USER CODE END RTOS_THREADS */

/* Start scheduler */
/* Infinite loop */
/* USER CODE BEGIN WHILE */
vTaskStartScheduler();

while (1) {
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    }
    /* USER CODE END 3 */
}

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void) {
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output voltage */
    if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
    {
        RCC_OscInitTypeDef RCC_OscInitStruct = {0};
        RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

        /** Configure the main internal regulator output voltage */
        if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
Error_Handler();
}

/** Initializes the RCC Oscillators according to the
 * specified parameters
 */
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLM = 1;
RCC_OscInitStruct.PLL.PLLN = 10;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
    Error_Handler();
}

/** Initializes the CPU, AHB and APB buses clocks */
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;

RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK) {
    Error_Handler();
}

/**
 * @brief ADC1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_ADC1_Init(void)
{

    /* USER CODE BEGIN ADC1_Init 0 */

    /* USER CODE END ADC1_Init 0 */
/ * USER CODE END ADC1_Init 0 */

ADC_MultiModeTypeDef multimode = {0};
ADC_ChannelConfTypeDef sConfig = {0};

/* USER CODE BEGIN ADC1_Init 1 */

/** Common config */

/*
 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = DISABLE;
 hadc1.Init.NbrOfConversion = 1;
 hadc1.Init.DiscontinuousConvMode = DISABLE;
 hadc1.Init.ExternalTrigConv = ADCEXTERNALTRIG_T3_TRGO;
 hadc1.Init.ExternalTrigConvEdge =
     ADCEXTERNALTRIGCONVEDGE_RISING;*/

/* USER CODE END ADC1_Init 1 */
hadc1.Init.DMAContinuousRequests = DISABLE;

hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;

hadc1.Init.OversamplingMode = DISABLE;

if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
    Error_Handler();
}

/** Configure the ADC multi-mode */

multimode.Mode = ADC_MODE_INDEPENDENT;

if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
{
    Error_Handler();
}

/** Configure Regular Channel */

sConfig.Channel = ADC_CHANNEL_1;

sConfig.Rank = ADC_REGULAR_RANK_1;

sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;

sConfig.SingleDiff = ADC_SINGLE_ENDED;

sConfig.OffsetNumber = ADC_OFFSET_NONE;

sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}

/* USER CODE BEGIN ADC1_Init 2 */

/* USER CODE END ADC1_Init 2 */

} }

/**
 * @brief TIM3 Initialization Function
 * @param None
 * @retval None
 */

static void MX_TIM3_Init ( void )
{

/* USER CODE BEGIN TIM3_Init 0 */

/* USER CODE END TIM3_Init 0 */

TIM_ClockConfigTypeDef sClockSourceConfig = {0};
TIM_MasterConfigTypeDef sMasterConfig = {0};
TIM_OC_InitTypeDef sConfigOC = {0};
/* USER CODE BEGIN TIM3_Init 1 */

htim3.Instance = TIM3;
htim3.Init.Prescaler = 80 - 1;
htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
htim3.Init.Period = (int) floor(TIM_PERIOD);
htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;

if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
{
    Error_Handler();
}

sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig)
    != HAL_OK)
{
    Error_Handler();
}

if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
{
    Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
sMasterConfig.MasterSlaveMode =
    TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &
    sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.Pulse = 0;
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC,
    TIM_CHANNEL_1) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN TIM3_Init 2 */
/* USER CODE END TIM3_Init 2 */

/* brief TIM6 Initialization Function */
* @param None
* @retval None
*/

static void MX_TIM6_Init(void)
{

TIM_MasterConfigTypeDef sMasterConfig = {0};

TIM_MasterConfigTypeDef sMasterConfig = {0};

 TIM_MasterConfigTypeDef sMasterConfig = {0};

if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
{
  Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN TIM6_Init 2 */
/* USER CODE END TIM6_Init 2 */

/**
 * @brief USART2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_USART2_UART_Init ( void )
{
    /* USER CODE BEGIN USART2_Init 0 */
    /* USER CODE END USART2_Init 0 */
/* USER CODE BEGIN USART2_Init 1 */

/* USER CODE END USART2_Init 1 */

huart2.Instance = USART2;

huart2.Init.BaudRate = 115200;

huart2.Init.WordLength = UART_WORDLENGTH_8B;

huart2.Init.StopBits = UART_STOPBITS_1;

huart2.Init.Parity = UART_PARITY_NONE;

huart2.Init.Mode = UART_MODE_TX_RX;

huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;

huart2.Init.Oversampling = UART_OVERSAMPLING_16;

huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;

huart2.AdvancedInit.AdvFeatureInit =
    UART_ADVFEATURE_NO_INIT;

if (HAL_UART_Init(&huart2) != HAL_OK)
{
    Error_Handler();
}

/* USER CODE BEGIN USART2_Init 2 */

/* USER CODE END USART2_Init 2 */
/**
 * Enable DMA controller clock
 */

static void MX_DMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Channel6_IRQHandler interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel6_IRQn, 5, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel6_IRQn);

  /* DMA1_Channel7_IRQHandler interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel7_IRQn, 5, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel7_IRQn);

}

/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */

static void MX_GPIO_Init(void)
GPIO_InitTypeDef GPIO_InitStruct = {0};

/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

/* GPIO Ports Clock Enable */
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOH_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOB_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);

/*Configure GPIO pin : B1_Pin */
GPIO_InitStruct.Pin = B1_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : LD2_Pin */
GPIO_InitStruct.Pin = LD2_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */
/* USER CODE END 4 */
/**
 * @brief  Period elapsed callback in non blocking mode
 * @note This function is called when TIM1 interrupt took place, inside
 * HAL_TIM_IRQHandler(). It makes a direct call to
 * HAL_IncTick() to increment
 * a global variable "uwTick" used as application time base.
 *
 * @param htim : TIM handle
 * @retval None
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    /* USER CODE BEGIN Callback 0 */
    /* USER CODE END Callback 0 */
if (htim->Instance == TIM1) {
  HAL_IncTick();
}

/* USER CODE BEGIN Callback 1 */

/* USER CODE END Callback 1 */

/**
 * @brief This function is executed in case of error occurrence.
 * @retval None
 */

void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}
#ifdef USE_FULL_ASSERT

/**
 * @brief Reports the name of the source file and the
 * source line number where the assert_param error has occurred.
 *
 * @param file: pointer to the source file name
 *
 * @param line: assert_param error line source number
 *
 * @retval None
 */

void assert_failed (uint8_t *file, uint32_t line)
{
   /* USER CODE BEGIN 6 */
   /* User can add his own implementation to report the file
    * name and line number,
    * ex: printf("Wrong parameters value: file %s on line %d\n
    * r
    *\n", file, line) */
   /* USER CODE END 6 */
}

#endif /* USE_FULL_ASSERT */
I.2 UART

/*
 * uart.c
 *
 * Created on: Nov 10, 2022
 * Author: Ty Freeman
 */

#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "event_groups.h"
#include "stm32l4xx_hal.h"

#include <stdlib.h>
#include <stdio.h>
#include "string.h"
#include "math.h"

#include "global.h"
#include "uart.h"
#include "adc.h"

#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;
_Bool bufRdy_flg = 1;
_Bool PROG_END = 0;
.extern _Bool ansyDone_flg;

int Tx_fails = 0;
int Rx_fails = 0;
char nbuf[5];

int ttl_pkgs = 0;
int pkgs_sent = 0;
int uart_pkgs = 0;

/* Callback for UART receiver. Every 1 character triggers this callback function which does light processing of value*/
void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t size)
{
    static int index = 0;

    HAL_UART_Transmit(&huart2, &rxbuf, 1, 2); //Echo character

    if(!cr_flg)
    {
        if((rxbuf == '\n' || rdbuf == '\r') && rd_flg) //If
enter key has been hit

    {
      cr_flg = 1; // Set the flag which is checked in the
      // reader task below
      rd_flg = 0; // Clear trigger flag
      index = 0;
      HAL_UART_Transmit(&huart2, cr, sizeof(cr), 2); // Echo
      // character
    }

    else if ((rxbuf == 'g' || rxbuf == 'G') && !rd_flg) // g
      or G for Go
        {
        rd_flg = 1; // Ready for read task
      }

    else if (rxbuf > 47 && rxbuf < 58 && index < 5)
        {
        nbuf[index] = rxbuf; // Collect up to 5 numbers in
        // this buffer
        index++; // Increment buffer index
      }

    else if (rxbuf == 's' || rxbuf == 'S') // s or S for
      // seconds
        {
        timLim_flg = 1; // User specified time limit
      }
else if(rxbuf != '"') // If not acceptable letter
{
    HAL_UART_Transmit(&huart2, bkspc, sizeof(bkspc), 2);
    // Auto backspace
}
}
}

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) // UART Tx DMA transfer complete callback
{
    pkg_sent++; // Keep track of packages sent
    if(ansyDone_flg && uxQueueMessagesWaitingFromISR(res_mbx) == 0) // Need adc done and all data sent
    {
        PROG_END = 1; // Signal program end
        HAL_UART_AbortTransmit(huart); // Turn off continuous printing
    }
    else
    {
        bufRdy_flg = 1; // Signal buffer ready for new data
    }
}

void read_Task(void * pvParameters)
TickType_t lastWake = 0;
TickType_t Period = pdMS_TO_TICKS(5);

float fft_res[SMP_2+1];
char TxBuf[(SMP_2+1)*STR_SZ];

int avg_MC = 0;
int avg_AC = 0;

HAL_UART_Transmit(&huart2, caret, sizeof(caret), 2); // Print starting CMD caret

while(1)
{
    HAL_UARTEx_ReceiveToIdle_DMA(&huart2, &rxbuf, 1); // Begin DMA
    __HAL_DMA_DISABLE_IT(&hdma_usart2_rx, DMA_IT_HT);

    if(cr_flg)
    {
        cr_flg = 0; // Clear carriage return flag
        ttl_pkgs = atoi(nbuf); // Extract user set limit
        memset(nbuf,0,sizeof(nbuf)); // Clear number buffer
if(timLim_flg) ttl_pkgs = (int) floor(ttl_pkgs/tm2full);

HAL_UART_Transmit(&huart2, cr, sizeof(cr), 2);

xTaskCreate(adc_Task, "adc", 1024, (void *) &ttl_pkgs,
            PriorityNormal, &adc);

if(xQueueReceive(res_mbx, fft_res, 0) && bufRdy_flg) // Results ready and Tx buffer clear
{
    uart_pkgs ++;

    for(int i=0; i<SMP_2+1; i++)
    {
        /* Maintain standard 10 bytes of data after sprintf */
        if(fft_res[i] > 9 && fft_res[i] < 100)
            sprintf(TxBuf+i*STR_SZ, "%.5f," , fft_res[i]);
        else if(fft_res[i] > 99)
            sprintf(TxBuf+i*STR_SZ, "%.4f," , fft_res[i]);
        else
            sprintf(TxBuf+i*STR_SZ, "%.6f," , fft_res[i]);
HAL_UART_Transmit_DMA(&huart2, (unsigned *) TxBuf, sizeof(TxBuf)); // Start DMA
bufRdy_flg = 0; // Tx buffer being used by new data

if(PROG_END) // All data printed and ADC done
{
    PROG_END = 0; // Clr flag
    avg_MC = ceil((float) Total_mult / pkg_cnt); // Calculate average number of multiplications per transform
    avg_AC = ceil((float) Total_add / pkg_cnt); // Calculate average number of additions per transform
    Tx_fails = trans_tx_fail + ansy_tx_fail + adc_tx_fail;
    Rx_fails = trans_rx_fail + ansy_rx_fail;

    /* Print Stats Message Block */
    printf("\n\n\r***** STATS *****\n\n\r"),
    printf("Average Mult Ops per transform:\t%d\n\r",}
avg_MC);

printf("Average Add Ops transform: \%d\n\n", avg_AC);

printf("Number of Rx Failures: \%d\n", Rx_fails);
printf("Number of Tx Failures: \%d\n", Tx_fails);
printf("Packages seen by Task: \n\r");
printf("\t\tADC \t\%d\n\r", pkg_cnt);
printf("\t\tTXM \t\%d\n\r", txm_pkgs);
printf("\t\tANSY \t\%d\n\r", ansys_pkgs);
printf("\t\tUART \t\%d\n\r", uart_pkgs);
printf("\t\tSENT \t\%d\n\r", pkgs_sent);
}

lastWake = xTaskGetTickCount();
vTaskDelayUntil(&lastWake, Period);
}
I.3 ADC

/* 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) //
46     // ADC callback function
47 {
48    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
53         overlapping
54     }
55     else
56         idx++;
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
64         trigger timer
65         HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, 1); // Turn on LED to
66         signal ADC start
67
68         uint16_t *package = NULL;
pkg_cnt = 0; // Reset for repeat trials
quiet_cnt = num_emptyBuf; // Reset quiet_cnt number at
  start of ADC task

/* User Set Limit Signals */
_Bool lim_set = 0;
int *pkg_lim = (int *) pvParameters;
if(*pkg_lim > 0) // Pull pkg limit from parameter
{
  lim_set = 1;
}

while(1)
{
  if(full)
  {
    package = (uint16_t *)malloc(SAMPLES*sizeof(uint16_t)); // Create package buffer
    memcpy(package, RWM, SAMPLES*sizeof(uint16_t)); // Move values from capture to package buffer
    memmove(RWM, RWM+SMP_2, SMP_2*sizeof(uint16_t)); // Shift capture buffer values down for window overlap
  
  
  if(xQueueSend(buffer_mbx, package, 1)) // Send package
to transform task for processing
{
  pkg_cnt++; // Count number of packages processed
  pkgRdy_flg = 1; // Data package in mailbox ready for transform
}

else adc_tx_fail += 1; // If failed to post, keep track of failure

if(lim_set && pkg_cnt == *pkg_lim) lim_flg = 1; // Signal time to stop

full = 0; // Reset full flg

free(package); // Release package space

if(lim_flg || sigDone_flg) // Two possible ending conditions
{
  /* Turn off Timer and ADC before deleting the adc task */
  HAL_TIM_PWM_Stop(&htim3, TIM_CHANNEL_1);
  HAL_ADC_Stop_IT(&hadc1);
}
adcDone_flg = 1; // Signal program that adc is done
if(sigDone_flg) pkg_cnt -= num_emptyBuf; // Remove empty buffers from total count

sigDone_flg = 0; // Clear signal flag
lim_flg = 0; // Clear limit flag
vTaskDelete(NULL);
}

lastwake = xTaskGetTickCount();
vTaskDelayUntil(&lastwake, pdMS_TO_TICKS(30)); // Task suspension for 30 ms
}
I.4 Transform

/*
 * 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"
```c
#include "fft.h"

#define TOREAL 3.3/4096
#define TODIG 4096/3.3

extern QueueHandle_t buffer_mbx;
extern QueueHandle_t stats_mbx;

/* Data Transmission Failure Flags */
int trans_tx_fail = 0;
int trans_rx_fail = 0;

/* Program Control Flags */
_Bool ansy_flg = 0;
_Bool sigDone_flg = 0;
_Bool txmDone_flg = 0;

/* Program Ending Variables */
float tm2full;
unsigned int quiet_cnt;
int num_emptyBuf;

int txm_pkgs = 0;

void txm_Task(void * pvParameters)
```
{  
  TickType_t lastWake = 0;  
  TickType_t Period = pdMS_TO_TICKS(20);  

  uint16_t *rec = NULL;  

  stats_t res_fft;  

  tm2full = (float) SAMPLES/FS; // Time it takes to fill one  
                             // buffer worth of samples  
  quiet_cnt = (int) floor(3/tm2full); // 3 seconds of  
                                     // silence signals end of incoming signal  
  num_emptyBuf = quiet_cnt;  

  while(1)  
  {  
    if(pkgRdy_flg)  
    {  
      res_fft.pkg_num = pkg_cnt; // Find current package  
                                // number  
      rec = (uint16_t *)malloc(SAMPLES*sizeof(uint16_t)); //  
                                // Create reception buffer  

      if(xQueueReceive(buffer_mbx, rec, 0) != pdTRUE)  
        trans_rx_fail++; // Receive data buffer from mailbox
else {
    txm_pkgs++;
    pkgRdy_flg = 0; // Reset pkgRdy flag

    for(int i = 0; i < SAMPLES; i++)
        {
            float hann = 0.5 - 0.5 * cos(2*PI*i/SAMPLES); // Calculate Hanning window coefficient
            res_fft.res_buf[i] = (float)rec[i]*hann*TOREAL + 0*I; // Apply Hanning window and convert to complex number
        }

    free(rec); // Free temporary transfer buffer
    FFT(&res_fft); // Run FFT

    float perEmpty = (float)res_fft.zCnt/SAMPLES;
    if(perEmpty >= 0.95) // If 85% of buffer is zero, consider it empty
        {
            quiet_cnt--; // Keep track of empty captures
        }
    if(quiet_cnt == 0)
{  
sigDone_flg = 1; // Signal finished, adc no longer needed  
}

else // Reset quiet count  
    quiet_cnt = num_emptyBuf; // Reset quiet_cnt

if(xQueueSend(stats_mbx, &res_fft, 0)) txm_pkgs++;

else trans_tx_fail++;

if(adcDone_flg && uxQueueMessagesWaiting(buffer_mbx) == 0)
{
    txmDone_flg = 1;
    vTaskSuspend(NULL); // Suspend if ADC done and no more data to process
}

lastWake = xTaskGetTickCount();
vTaskDelayUntil(&lastWake, Period);
I.5  Analysis

/*
 * 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"
# define wErr 2 // Window error correction factor
#define NPBw 1.5 // Noise Power BW for Hann window
#define T FS/SAMPLES // Time Step

extern QueueHandle_t stats_mbx;
extern QueueHandle_t res_mbx;

int ansy_tx_fail = 0;
int ansy_rx_fail = 0;

unsigned long Total_mult = 0;
unsigned long Total_add = 0;

int ansys_pkgs = 0;

extern _Bool txmDone_flg;
_Bool ansyDone_flg = 0;

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

} /* 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);
}

else ansy_tx_fail++;
free(temp_buf);
lastWake = xTaskGetTickCount();
vTaskDelayUntil(&lastWake, Period);
}
}

/**
 * @brief Magnitude calculation of complex number
 * @param N Complex number
 * @retval None complex floating point number
 */
float mag(float complex N)
{
    float r2 = creal(N)*creal(N);
    float i2 = cimag(N)*cimag(N);
    return sqrt(r2 + i2);
}
I.6 Global Header

/*
 * global.h
 *
 * Created on: Jan 31, 2023
 * Author: Ty Freeman
 */

#ifndef INC_GLOBAL_H_
define INC_GLOBAL_H_

#include "complex.h"

#define PI 3.14159265358979323846
#define SYS_FREQ 80000000
#define SAMPLES 256
#define FS 5000
#define dt 1/FS
#define SMPx2 SAMPLES*2
#define SMP_2 SAMPLES/2

extern unsigned int pkg_cnt;
extern int txm_pkgs;
extern int ansys_pkgs;
/* Control Flags */
extern _Bool ansy_flg;
extern _Bool pkgRdy_flg;
extern _Bool adcDone_flg;
extern _Bool sigDone_flg;
extern _Bool bufEmpty_flg;
extern _Bool resRdy_flg;

/* Fail Counters */
extern int adc_tx_fail;
extern int trans_tx_fail;
extern int trans_rx_fail;
extern int ansy_tx_fail;
extern int ansy_rx_fail;

extern double step;

enum Priority{
    PriorityIdle,      ///< priority: idle (lowest)
    PriorityLow,      ///< priority: low
    PriorityBelowNormal,  ///< priority: below normal
    PriorityNormal,    ///< priority: normal (default)
    PriorityAboveNormal,  ///< priority: above normal
    PriorityHigh,      ///< priority: high
PriorityRealtime, //< priority: realtime (highest)

PriorityError = 0x84 //< system cannot determine
               priority or thread has illegal priority

};

typedef struct stats {
    unsigned int pkg_num;
    unsigned long mult_cnt;
    unsigned long add_cnt;
    int zCnt;
    float complex res_buf[SAMPLES];
} stats_t;

#endif /* INC_GLOBAL_H_ */
I.7 FFT

1 /*
2  * fft.c
3  *
4  * Created on: Nov 7, 2022
5  * Author: Ty Freeman
6  */
7
8 /* System Includes */
9  #include <fft.h>
10  #include "FreeRTOS.h"
11  #include "task.h"
12  #include "stm32l4xx_hal.h"
13
14 /* lib Includes */
15  #include <stdio.h>
16  #include <stdlib.h>
17  #include <string.h>
18  #include "math.h"
19
20 /* User-Created Includes */
21
22 float complex twexp = 2*PI*I/SAMPLES; // Twiddle exponent e
23             ^\frac{2\pi i}{N}
void FFT(stats_t *results) {
    int a = SMP_2; // Dual-Node distance factor
    int m = 0; // Twiddle power

    float complex x = 0; // Primary summation term
    float complex xp = 0; // Secondary summation term

    results->mult_cnt = 0; // Reset operations counter
    results->add_cnt = 0;

    int stages = log_2(SAMPLES); // Calculate number of stages necessary

    for(int j = 1; j <= stages; j++) {
        for(int k = 0; k < SAMPLES; k++) {
            if(!(k & a)) // Remove redundant computations
                { 
                m = bit_reverse(stages, k >> (stages - j)); // Calculate twiddle power
                x = results->res_buf[k];
                xp = cexp(twexp*m)*results->res_buf[k+a];

            }
        }
    }
}
46  results->res_buf[k] = x + xp;
47  results->res_buf[k + a] = x - xp;
48  results->mult_cnt += 1; // One complex multiplication
49  results->add_cnt += 2; // Two complex additions
50  }
51  }
52  }
53  a >>= 1; // Change Dual-Node distance
54  }
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)
results->zCnt++;
}
}

/**
 * @brief Bit reversal algorithm
 * @param sz Number of bits in number
 * @param index Current index value to be reversed
 * @retval int Bit reversed index
 */

int bit_reverse(int sz, int index)
{
    int p = 0;

    for(int i = 0; i <= sz; i++)
    {
        if(index & (1 << (sz - i)))
        {
            p |= 1 << (i - 1);
        }
    }

    return p;
}
/**
 * @brief Base-2 logarithm
 * @param N Number
 * @retval int Number of bits in N
 */

int log_2(unsigned int N)
{
    int pow = 0;

    while(N)
    {
        N >>= 1;
        pow ++;
    }

    return pow - 1;
}
I.8 FFT Test Main

/* USER CODE BEGIN Header */
/**

**********************************************************************
* @file : main.c
* @brief : Main program body
**********************************************************************

* @attention

* Copyright (c) 2023 STMicroelectronics.
* All rights reserved.
*
* This software is licensed under terms that can be found
* in the LICENSE file
* in the root directory of this software component.
* If no LICENSE file comes with this software, it is
* provided AS-IS.
*
**********************************************************************

*/

/* USER CODE END Header */
/* Includes */

#include "main.h"

/* Private includes */

/* USER CODE BEGIN Includes */
#include "stdio.h"
#include "stdlib.h"
#include "fft.h"
#include "math.h"
#include <inttypes.h>
/* USER CODE END Includes */

/* Private typedef */

/* USER CODE BEGIN PTD */
/* USER CODE END PTD */

/* Private define */

/* USER CODE BEGIN define */
/* USER CODE END define */
I.8 FFT Test Main

37 /* USER CODE BEGIN PD */
38 #define STR_SZ 10
39 #define fc 261.63
40 #define fg 783.99
41 #define fe 659.25
42 #define TOREAL 3.21/4096
43 #define TIM_PERIOD 1e6/FS
44 /* USER CODE END PD */
45
46 /* Private macro

   ------------------------------------------------------------------------

   */
47 /* USER CODE BEGIN PM */
48
49 /* USER CODE END PM */
50
51 /* Private variables

   ------------------------------------------------------------------------

   */
52 ADC_HandleTypeDef hadc1;
53
54 TIM_HandleTypeDef htim3;
55 TIM_HandleTypeDef htim6;
UART_HandleTypeDef huart2;
DMA_HandleTypeDef hdma_usart2_tx;

/* USER CODE BEGIN PV */
_Bool full = 0;
_Bool done = 0;
uint8_t tim_turnover = 0;
stats_t fft_res;
uint16_t RWM[SAMPLES]; // Capture buffer
/* USER CODE END PV */

/* Private function prototypes-----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_TIM6_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM3_Init(void);
/* USER CODE BEGIN PFP */
/* USER CODE END PFP */

/* 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 )
{
    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 */
1.8 FFT Test Main

/**
 * @brief The application entry point.
 * @retval int
 */

int main(void)
{
    /* USER CODE BEGIN 1 */
    unsigned char dblSpace[] = "\n\n\r";
    /* USER CODE END 1 */

    /* MCU Configuration
     --------------------------------------------------------
     */

    /* Reset of all peripherals, Initializes the Flash
     interface and the Systick. */
    HAL_Init();

    /* USER CODE BEGIN Init */

    /* USER CODE END Init */

    /* Configure the system clock */
    SystemClock_Config();
}
/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_USART2_UART_Init();
MX_TIM6_Init();
MX_ADC1_Init();
MX_TIM3_Init();

/* USER CODE BEGIN 2 */
HAL_UART_Transmit(&huart2, dblSpace, sizeof(dblSpace), 2);
HAL_ADC_Start_IT(&hadc1); // Start ADC
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1); // Start ADC
  trigger timer

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  if(full)
  {

HAL_ADC_Stop_IT(&hadc1);

float *temp_buf = NULL;
char pBuf[(SMP_2+1)*STR_SZ];

for(int i = 0; i < SAMPLES; i++)
{
    float hann = 0.5 - 0.5*cos(2*PI*i/SAMPLES); // Calculate Hanning window coefficient
    // fft_res.res_buf[i] = sin(2*PI*200*i/FS); // Single-Tone 200 Hz
    // 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
    // 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
    fft_res.res_buf[i] = TOREAL*(RWM[i]); // ADC Implementation
    fft_res.res_buf[i] *= hann; // Apply Hanning window and convert to complex number
}

classic_FFT(&fft_res);

full = 0;
temp_buffer = malloc((SMP_2 + 1) * sizeof(float));

for (int i = 0; i < (SMP_2 + 1); i++)
{
  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_buffer[i] = creal(fft_res.res_buf[i]) * 2; // Correct windowed amplitude and transfer to smaller buffer

  if (temp_buffer[i] > 9 && temp_buffer[i] < 100)
    sprintf(pBuf + i * STR_SZ, "%.5f", temp_buffer[i]);
  else if (temp_buffer[i] > 99)
    sprintf(pBuf + i * STR_SZ, "%.4f", temp_buffer[i]);
  else

I.8 FFT Test Main

```c
    sprintf(pBuf+i*STR_SZ, "%.6f", temp_buf[i]);

    HAL_UART_Transmit_DMA(&huart2, pBuf, sizeof(pBuf));
```

```c
    free(temp_buf);
```

```c
    if( done )
    {
        HAL_UART_Abort(&huart2);
        printf("\n\n\r ***** STATS *****\n\r");
        printf("Time per transform:\t%f uS\n\r", (float)fft_res.time/80);
        printf("Mult Ops per transform:\t%d\n\r", fft_res.mult_cnt);
        printf("Add Ops per transform:\t%d\n\r", fft_res.add_cnt);
        exit(1);
    }
```

/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */

```c
```

```c```
/* USER CODE END 3 */

} /* USER CODE BEGIN 4 */

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
RCC_OscInitTypeDef RCC_OscInitStruct = {0};
RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

/** Configure the main internal regulator output voltage
*/
if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
Error_Handler();
}

/** Initializes the RCC Oscillators according to the specified parameters
 * in the RCC_OscInitTypeDef structure.
 */
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
RCC_ClkInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;

RCC_OscInitStruct.OscillatorSource = RCC_OSCILLATORTYPE_HSI;
RCC_OscInitStruct.HSISource = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLM = 1;
RCC_OscInitStruct.PLL.PLLN = 10;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
    Error_Handler();
}

/** Initializes the CPU, AHB and APB buses clocks */
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK |
                            RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
    Error_Handler();
}

/**
 * @brief ADC1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_ADC1_Init(void)
{

    /* USER CODE BEGIN ADC1_Init 0 */

    /* USER CODE END ADC1_Init 0 */

    ADC_MultiModeTypeDef multimode = {0};
    ADC_ChannelConfTypeDef sConfig = {0};

    /* USER CODE BEGIN ADC1_Init 1 */

    /* USER CODE END ADC1_Init 1 */
/** USER CODE END ADC1_Init 1 */

/** Common config */

hadc1.Instance = ADC1;

hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;

hadc1.Init.Resolution = ADC_RESOLUTION_12B;

hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;

hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;

hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;

hadc1.Init.LowPowerAutoWait = DISABLE;

hadc1.Init.ContinuousConvMode = DISABLE;

hadc1.Init.NbrOfConversion = 1;

hadc1.Init.DiscontinuousConvMode = DISABLE;

hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIG_T3_TRGO;

hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;

hadc1.Init.DMAContinuousRequests = DISABLE;

hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;

hadc1.Init.OversamplingMode = DISABLE;

if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
    Error_Handler();
}

/** Configure the ADC multi-mode */
multimode.Mode = ADC_MODE_INDEPENDENT;
if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
{
    Error_Handler();
}

/** Configure Regular Channel */
sConfig.Channel = ADC_CHANNEL_1;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN ADC1_Init 2 */
/* USER CODE END ADC1_Init 2 */
static void MX_TIM3_Init(void)
{

/* USER CODE BEGIN TIM3_Init 0 */

/* USER CODE END TIM3_Init 0 */

TIM_ClockConfigTypeDef sClockSourceConfig = {0};
TIM_MasterConfigTypeDef sMasterConfig = {0};
TIM_OC_InitTypeDef sConfigOC = {0};

/* USER CODE BEGIN TIM3_Init 1 */

/* USER CODE END TIM3_Init 1 */

htim3.Instance = TIM3;
htim3.Init.Prescaler = 80 - 1;
htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
htim3.Init.Period = (int) floor(TIM_PERIOD);
htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
{
    Error_Handler();
}

sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
{
    Error_Handler();
}

if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
{
    Error_Handler();
}

sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
sConfigOC.OCMode = TIM_OCMODE_PWM1;

sConfigOC.Pulse = 65535;

sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;

sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;

if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
{
    Error_Handler();
}

/* USER CODE BEGIN TIM3_Init 2 */

/* USER CODE END TIM3_Init 2 */

HAL_TIM_MspPostInit(&htim3);

}
TIM_MasterConfigTypeDef sMasterConfig = {0};

sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;

if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}

if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
{
    Error_Handler();
}
I.8 FFT Test Main

```c
} /* USER CODE BEGIN TIM6_Init 2 */

/* USER CODE END TIM6_Init 2 */

} /* USER CODE END TIM6_Init 2 */

/**
 * @brief USART2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_USART2_UART_Init ( void )
{
    /* USER CODE BEGIN USART2_Init 0 */

    /* USER CODE END USART2_Init 0 */

    /* USER CODE BEGIN USART2_Init 1 */

    /* USER CODE END USART2_Init 1 */

    huart2.Instance = USART2;
    huart2.Init.BaudRate = 115200;
    huart2.Init.WordLength = UART_WORDLENGTH_8B;
```
huart2.Init.StopBits = UART_STOPBITS_1;
huart2.Init.Parity = UART_PARITY_NONE;
huart2.Init.Mode = UART_MODE_TX_RX;
huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart2.Init.Oversampling = UART_OVERSAMPLING_16;
huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
if (HAL_UART_Init(&huart2) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN USART2_Init 2 */

/* USER CODE END USART2_Init 2 */

/**
 * Enable DMA controller clock
 * /
static void MX_DMA_Init(void)
{
    /* DMA controller clock enable */
__HAL_RCC_DMA1_CLK_ENABLE();

/* DMA interrupt init */
/* DMA1_Channel7_IRQn interrupt configuration */
HAL_NVIC_SetPriority(DMA1_Channel7_IRQHandler, 0, 0);
HAL_NVIC_EnableIRQ(DMA1_Channel7_IRQHandler);

}

/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    /* USER CODE BEGIN MX_GPIO_Init_1 */
    /* USER CODE END MX_GPIO_Init_1 */

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOH_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
/* Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0 | LD2_Pin, GPIO_PIN_RESET);

/* Configure GPIO pin : B1_Pin */
GPIO_InitStruct.Pin = B1_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

/* Configure GPIO pins : PA0 LD2_Pin */
GPIO_InitStruct.Pin = GPIO_PIN_0 | LD2_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{

/* USER CODE BEGIN Callback 1 */
if (htim->Instance == TIM6) {
    tim_turnover++;
}
/* USER CODE END Callback 1 */

/* USER CODE END 4 */

/** *
 * @brief This function is executed in case of error occurrence.
 * @retval None *
 */
void Error_Handler(void) {
    /* USER CODE BEGIN Error_Handler_Debug */
    __disable_irq();
    while (1) {
        /* USER CODE END Error_Handler_Debug */
    }
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    __disable_irq();
    while (1) {
    }
    /* USER CODE END Error_Handler_Debug */
}
# ifdef USE_FULL_ASSERT
/**
 * @brief Reports the name of the source file and the source line number where the assert_param error has occurred.
 * @param file: pointer to the source file name
 * @param line: assert_param error line source number
 * @retval None
 */

void assert_failed(uint8_t *file, uint32_t line)
{
    /* USER CODE BEGIN 6 */
    /* User can add his own implementation to report the file name and line number,
    ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
    /* USER CODE END 6 */
}

#endif /* USE_FULL_ASSERT */
%% 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 \texttt{T = load\_data("Data\_Files\SR\_Real\SR\_ADC\_1k\_200\_4096.csv", sz);} \\
24 \\
25 \texttt{xUnwin = x;} \\
26 \texttt{x = h.*x; \% Apply Hanning Window} \\
27 \\
28 \texttt{X = abs(fft(x,N))/N; \% Normalized Magnitude of fft output} \\
29 \texttt{X = X(1:N/2+1); \% Single-sided conversion} \\
30 \texttt{X(2:end-1) = sqrt(2).*X(2:end-1); \% Amplitude Vrms for non-DC components} \\
31 \texttt{\% X = 2*X; \% Window Amplitude Correction Factor} \\
32 \\
33 \texttt{\% Results plotting} \\
34 \texttt{figure} \\
35 \texttt{plot(t,xUnwin);} \\
36 \texttt{title("Unwindowed Signal in Time Domain")} \\
37 \texttt{xlabel("Time (s)")} \\
38 \texttt{ylabel("Amplitude (V)")} \\
39 \\
40 \texttt{figure} \\
41 \texttt{plot(t,x);} \\
42 \texttt{title("Windowed Signal in Time Domain")} \\
43 \texttt{xlabel("Time (s)")} \\
44 \texttt{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;
% Specify file level properties
opts.ExtraColumnsRule = "ignore";
opts.EmptyLineRule = "read";
opts.ConsecutiveDelimitersRule = "join";

% Import the data
C = readmatrix(CDataPath, opts);

% Clear temporary variables
clear opts
end