

# FFT Manual

Leviticus Norman, Veloria Pannell, Dylan Brooks

August 2024

## 1 Overview

This tutorial shows how to configure the PSoC to perform a fast Fourier transform and read the results over a Serial connection. This tutorial was made for the big board. The particular FFT algorithm we are using is from *Numerical Recipes in C*. It takes an array and overwrites it with the FFT of the array. Read carefully. There are many places for you to configure the PSoC incorrectly or use the algorithm incorrectly.

You can find instructions on how to set up the PSoC to convert floats to strings in section [3.1](#).

## 2 Hardware

The project only uses two pins to talk over UART. These are the Rx pin, which is connected to P3.1 and the Tx pin, which is connected to P3.0. Refer to the wiring diagram below:

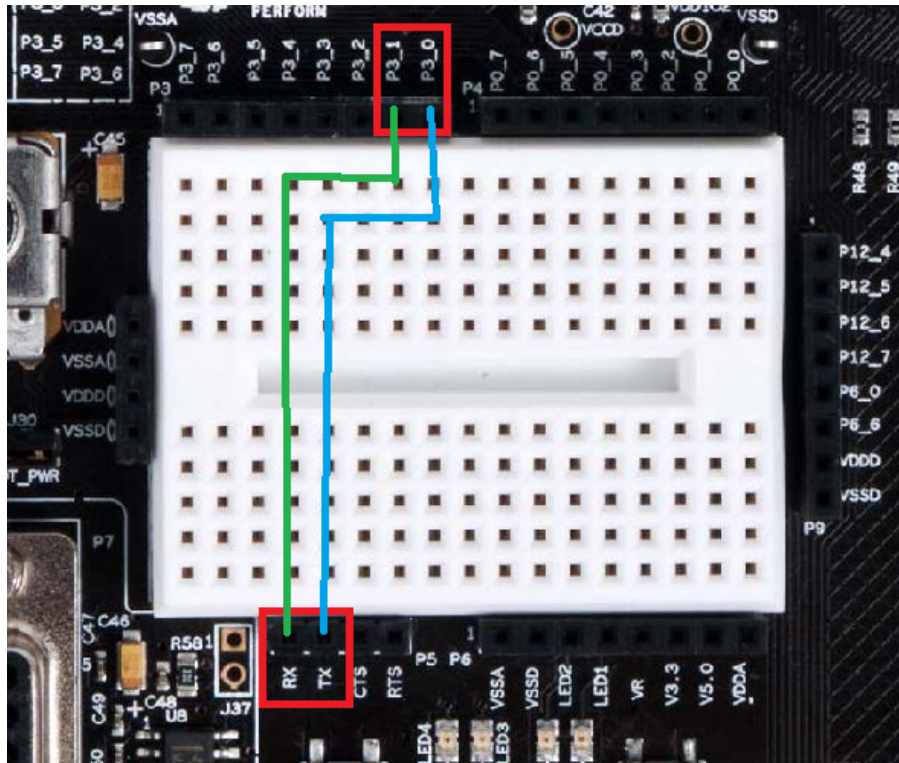


Figure 1: Pinout

### 3 Cypress Schematic

The project only uses a UART block. The UART block is configured to 9600 bps as it has been for the other UART connections within the class.

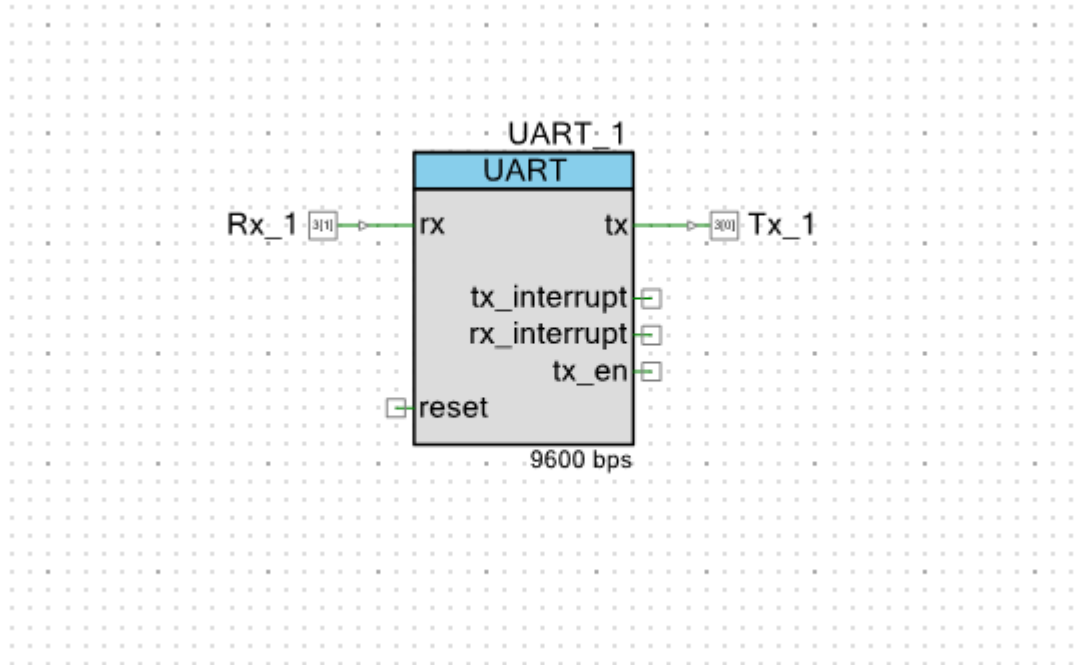
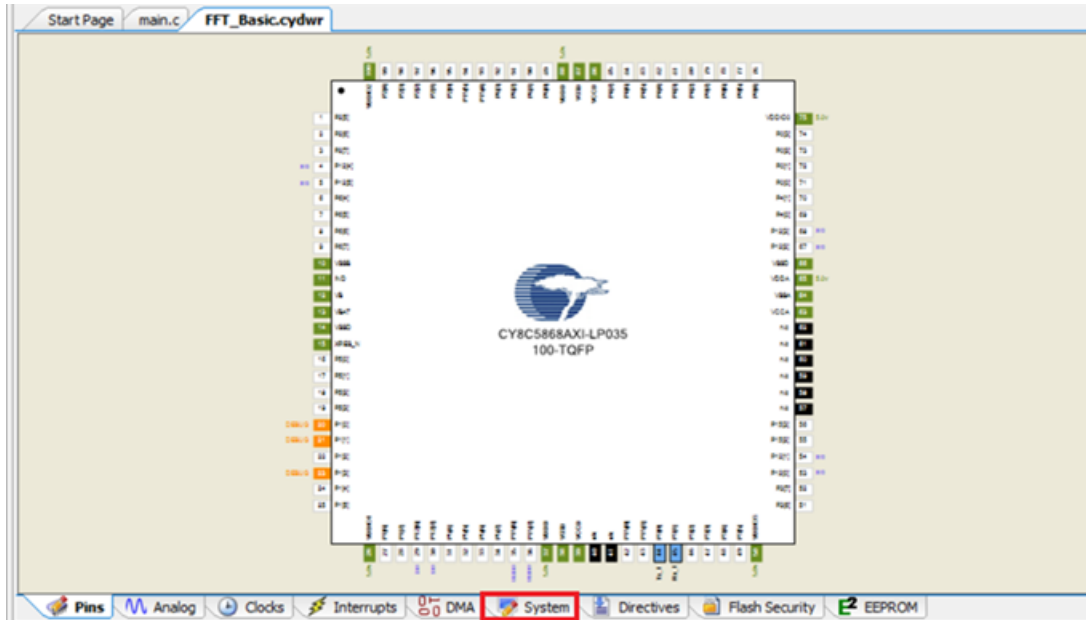


Figure 2: Cypress Schematic

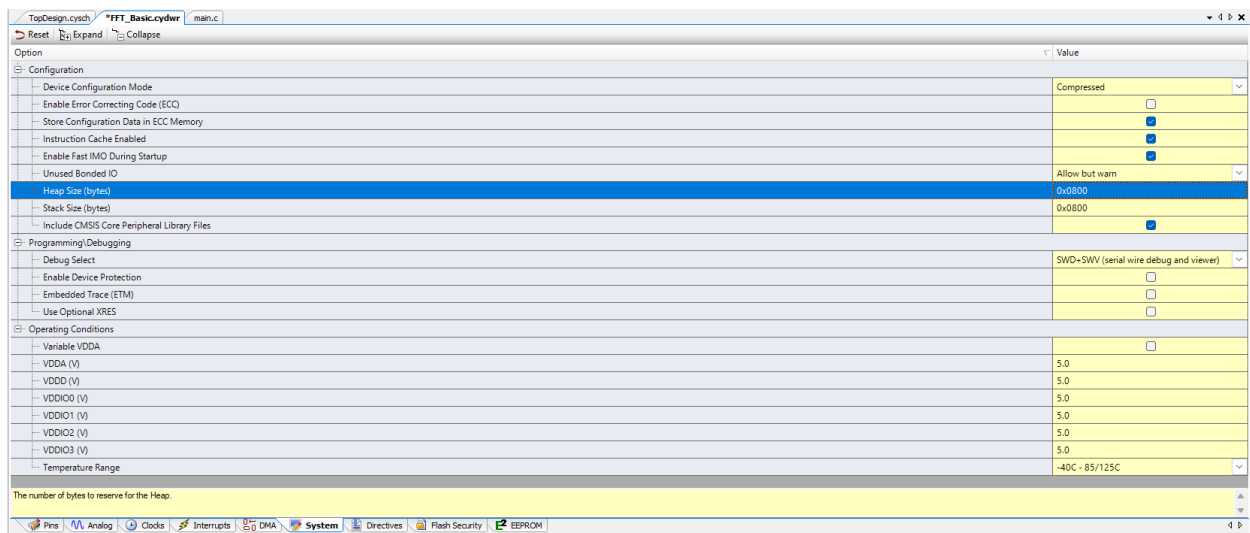
#### 3.1 WARNING IMPORTANT READ!!!

By default, the PSoC cannot directly cast floats to strings. One way to do this is with the `sprintf()` function in the C standard input and output library (`stdio.h`). Casting a string to a float is a memory expensive operation and microcontrollers are often memory constrained, so PSoC Creator has removed `sprintf()` from the `stdio` library by default. Thus, if the user would like to utilize `sprintf()` in their program, they need to inform the PSoC Creator compiler to include `sprintf()` at compile time and to allocate more of the PSoC's internal memory to the heap for calculation purposes. First we need to increase the heap size so we have enough dynamic memory to perform float to string conversion. Then we're going to change some settings in our compiler so that our PSoC has the necessary library to convert floats to strings. (Note: In the project provided with this document these steps have already been done. If you create a new project DO NOT forget to follow these instructions. Failure to do so will not provide error messages. Instead your program will compile and program normally, but just not work without any clear indication as to why.)

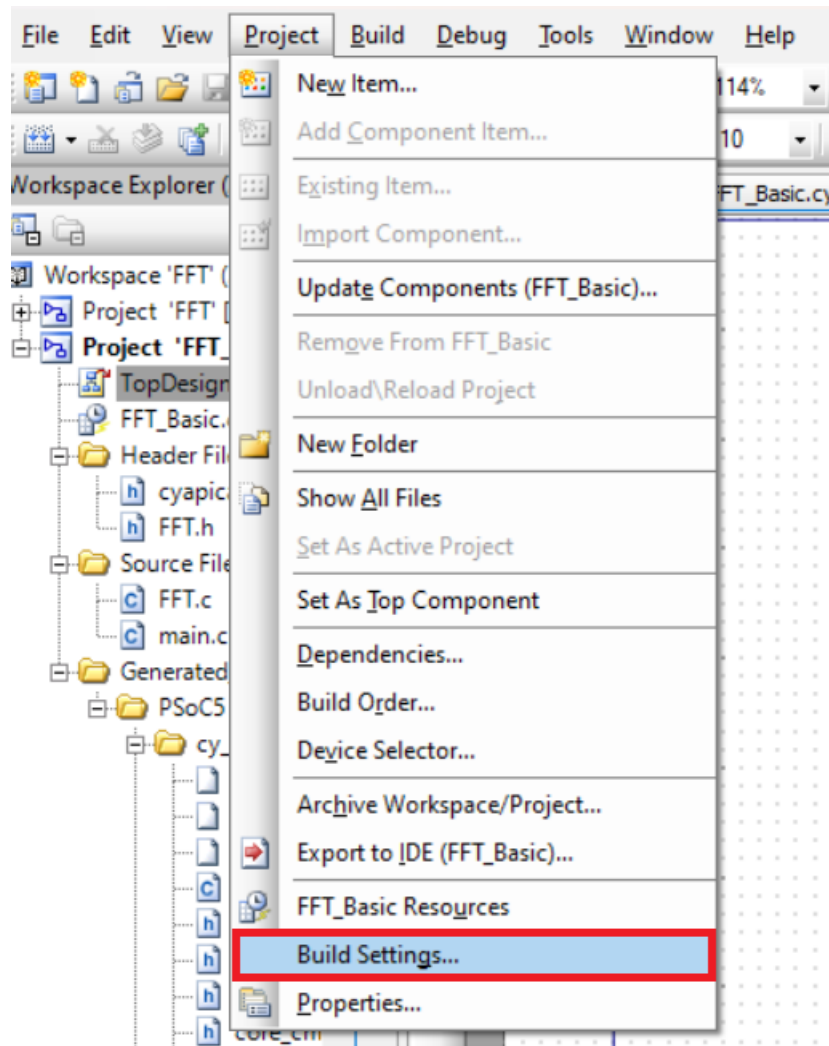
1. Navigate to PSoC creator and load the project
2. Open the Design Wide Resources file (ends in .cydwr)
3. Navigate to the section labeled System



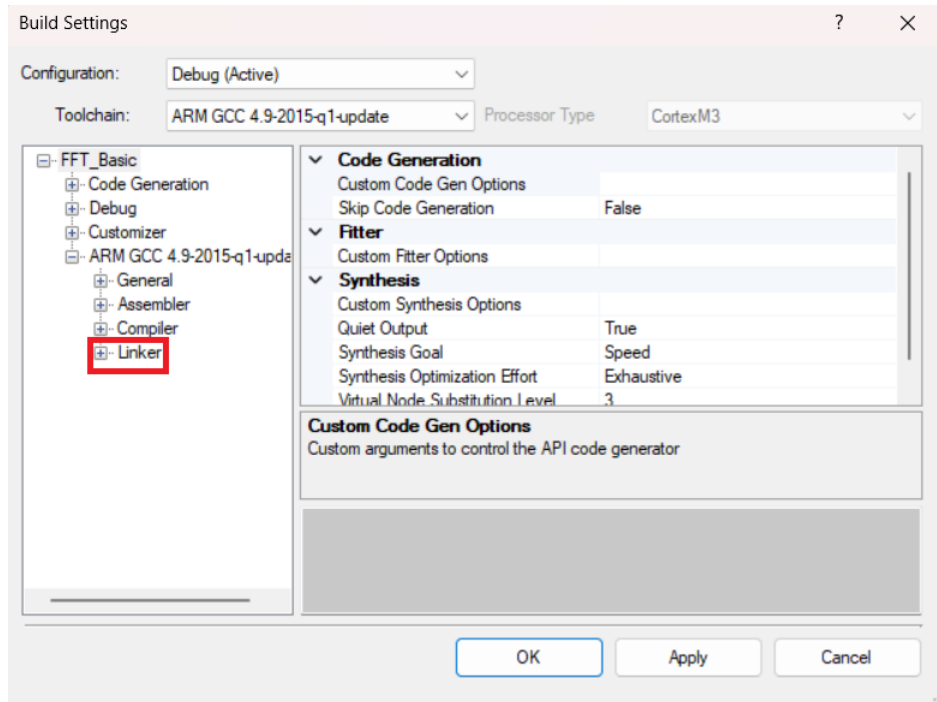
4. Find the heap size and change the value to “0x0800” (Note: If you are experiencing abnormal behaviour when performing the FFT on larger sized arrays you can try further increasing the heap size)



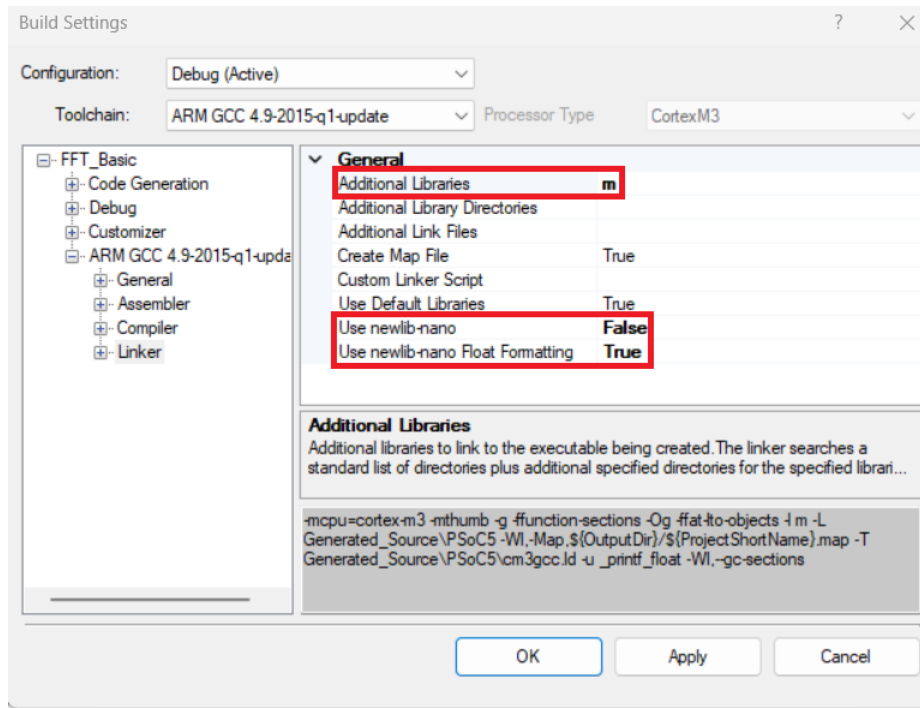
5. Click on **Project** → **Build Settings**



6. In the left hand menu, navigate to **ARM GCC 4.9-2015-q1-update** → **Compiler** → **Linker**



7. Under **Additional Libraries** include "m" to include the C math library. Additionally, ensure **Use newlib-nano** is set to **False** and **Use newlib-nano Float Formatting** is set to **True**.

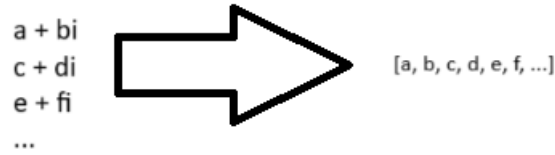


## 4 FFT

You can find the FFT algorithm we use as well as others in *Numerical Recipes in C*. It was written by N. M Brenner originally published in *Three Fortran Programs that Perform the Cooley-tukey Fourier Transform* at Lincoln Labs. We have implemented the most general algorithm that will work for any real or complex signal whose length is an integer power of two.

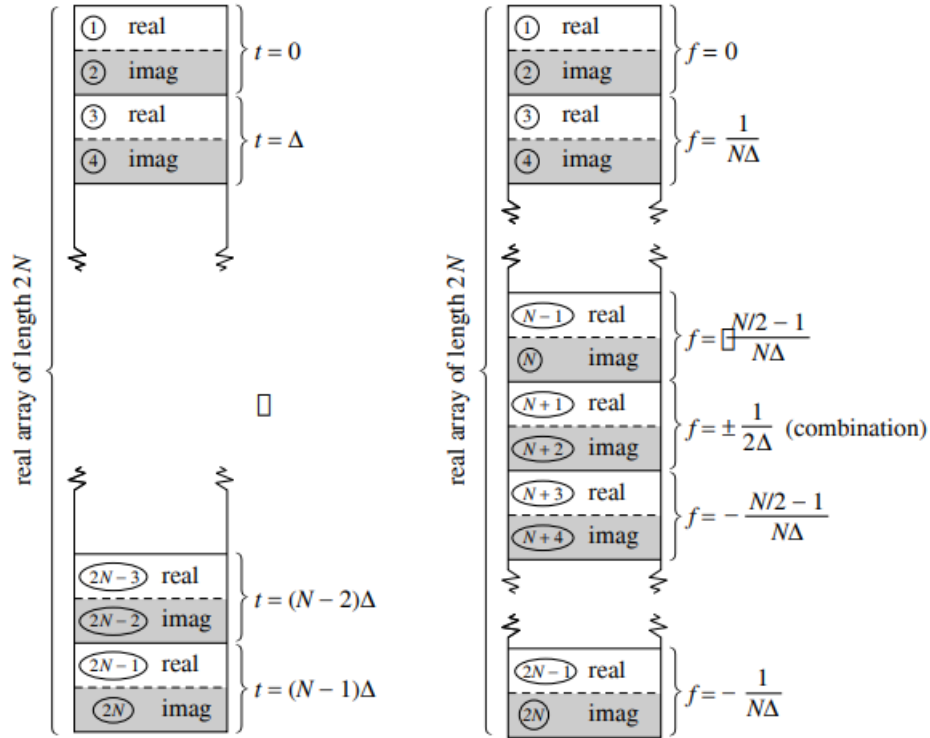
The FFT takes three inputs, a complex array containing the sampled signal, the number of samples, and sign. The sign parameter is an integer indicating if you're performing the FFT or the inverse FFT. 1 corresponds to the FFT and -1 corresponds to the inverse FFT.

In order to represent complex numbers in C we initialize an array that is twice the length of our signal. Elements at even indices of the array corresponds to real values, while elements at odd indices correspond to imaginary values. An example of this notation is shown below.



The length of the signal must be an integer power of two. If your signal is not an integer power of two, you can append zeroes to the end of your signal until it is.

The order of the output of the FFT is not intuitive. Refer to the diagram below to understand the output of the FFT, where  $N$  refers to the length of the signal and  $\Delta$  refers to the time between samples.



## 5 Firmware

As long as you follow ALL other instructions in this manual, using the FFT is relatively simple.

1. Begin by defining your constants and arrays to store your signals (Note: Sampling Frequency is not used in the function `four1()`; it is only used in later functions that write the results).

```
float DCData[2*N];
float HarmonicData[2*N];
float HarmonicData2[2*N];
float NyquistData[2*N];
float CosineData[2*N];
float CosineData2[2*N];

#define N 8
#define samplingFrequency 8
#define pi 3.141592653589793
```

2. Fill your arrays, keeping in mind that elements at even indices refer to the real part of your signal, while elements at odd indices refer to the imaginary part (Note: Most signals are real, which means your array will most likely contain zero in the odd elements).

```
for (i = 0; i < 2*N; i++)
{
    if (i%2 == 0)
    {
        DCData[i] = 1.0;
        HarmonicData[i] = sin((pi/8)*i);
        HarmonicData2[i] = sin((pi/4)*i);
        NyquistNumber*=-1;
        NyquistData[i] = NyquistNumber;
        CosineData[i] = cos((pi/8)*i);
        CosineData2[i] = cos((pi/4)*i);
    }
    else
    {
        DCData[i] = 0.0;
        HarmonicData[i] = 0.0;
        HarmonicData2[i] = 0.0;
        NyquistData[i] = 0.0;
        CosineData[i] = 0.0;
        CosineData2[i] = 0.0;
    }
}
```

3. Define the other parameters for the FFT, paying close attention to the types the algorithm calls for. Remember length is the length of the signal, not the length of the array (Note: The length of the array should be twice the length of the signal).

```
int sign = 1;
unsigned long length = N;
```

4. Finally pass the array and parameters to `four1()`.

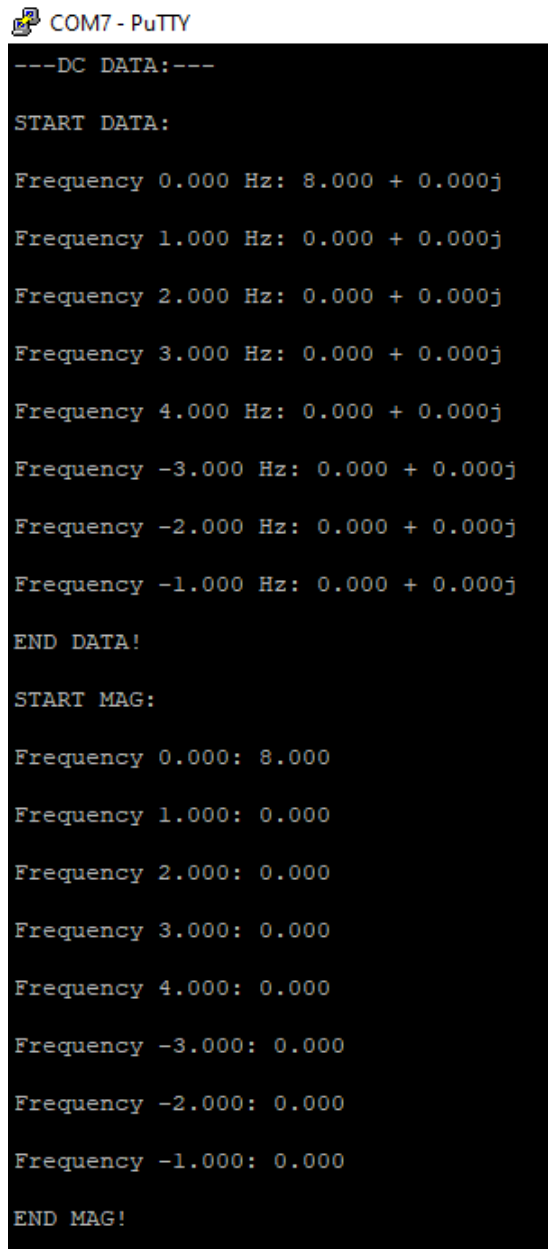
```
four1(DCData-1, length, sign);
four1(HarmonicData-1, length, sign);
four1(HarmonicData2-1, length, sign);
four1(NyquistData-1, length, sign);
four1(CosineData-1, length, sign);
four1(CosineData2-1, length, sign);
```



## 6 Seeing the Result

In order to see the output of the FFT we have included two functions for writing the results of the FFT over a serial link, `WriteData()` and `WriteMagnitude()`. These two algorithms convert the output of the FFT from floats to strings, calculate the frequency each point corresponds to, and writes it over the Serial Port.

Both functions take three inputs: the output array of the FFT, the length of the signal (not the array), and the sampling frequency. The output is sent over the serial port, where it should be plugged into a computer with a terminal emulator of your choice (PuTTY, TeraTerm, etc.). `WriteData()` simply writes the output of the FFT, while `WriteMagnitude()` calculates the Magnitude for each point before writing over serial. An example of the output in a terminal emulator is below.

A screenshot of a PuTTY terminal window titled "COM7 - PuTTY". The terminal displays the output of an FFT function. It starts with a separator line "----DC DATA:---". Then it says "START DATA:". This is followed by nine lines of complex number output: "Frequency 0.000 Hz: 8.000 + 0.000j", "Frequency 1.000 Hz: 0.000 + 0.000j", "Frequency 2.000 Hz: 0.000 + 0.000j", "Frequency 3.000 Hz: 0.000 + 0.000j", "Frequency 4.000 Hz: 0.000 + 0.000j", "Frequency -3.000 Hz: 0.000 + 0.000j", "Frequency -2.000 Hz: 0.000 + 0.000j", "Frequency -1.000 Hz: 0.000 + 0.000j", and "Frequency -0.000 Hz: 0.000 + 0.000j". After this, it says "END DATA!". Then it says "START MAG:". This is followed by nine lines of magnitude output: "Frequency 0.000: 8.000", "Frequency 1.000: 0.000", "Frequency 2.000: 0.000", "Frequency 3.000: 0.000", "Frequency 4.000: 0.000", "Frequency -3.000: 0.000", "Frequency -2.000: 0.000", "Frequency -1.000: 0.000", and "Frequency -0.000: 0.000". Finally, it says "END MAG!".

```
COM7 - PuTTY
----DC DATA:---
START DATA:
Frequency 0.000 Hz: 8.000 + 0.000j
Frequency 1.000 Hz: 0.000 + 0.000j
Frequency 2.000 Hz: 0.000 + 0.000j
Frequency 3.000 Hz: 0.000 + 0.000j
Frequency 4.000 Hz: 0.000 + 0.000j
Frequency -3.000 Hz: 0.000 + 0.000j
Frequency -2.000 Hz: 0.000 + 0.000j
Frequency -1.000 Hz: 0.000 + 0.000j
Frequency -0.000 Hz: 0.000 + 0.000j
END DATA!
START MAG:
Frequency 0.000: 8.000
Frequency 1.000: 0.000
Frequency 2.000: 0.000
Frequency 3.000: 0.000
Frequency 4.000: 0.000
Frequency -3.000: 0.000
Frequency -2.000: 0.000
Frequency -1.000: 0.000
Frequency -0.000: 0.000
END MAG!
```

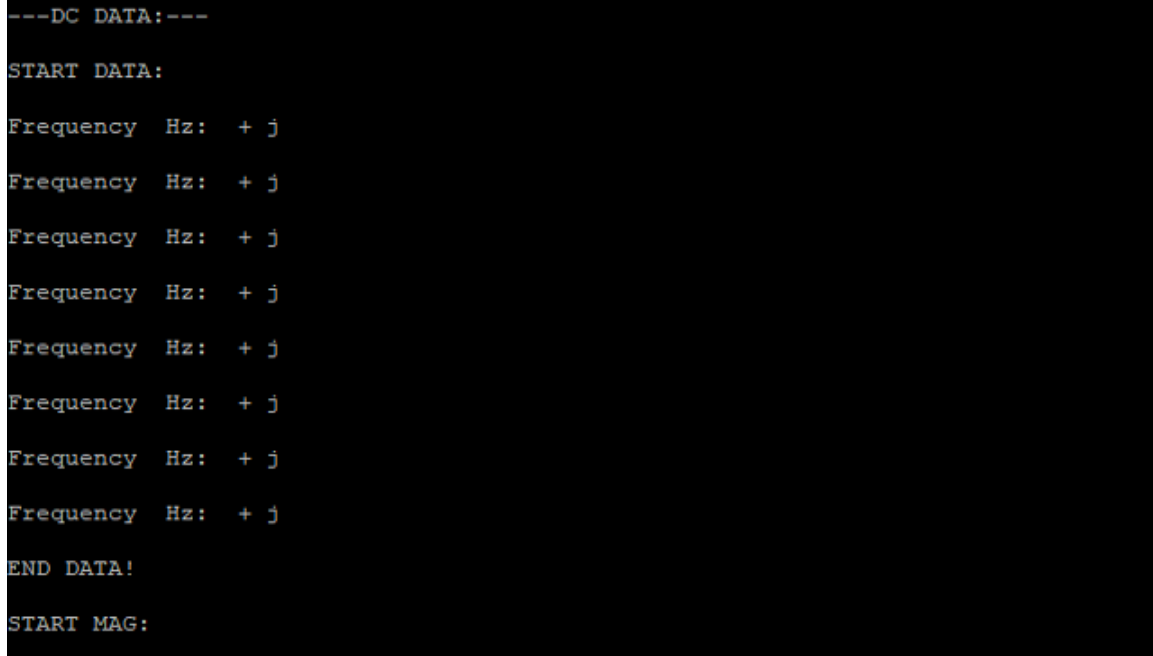
## 7 Helpful Links

- <https://numerical.recipes/> Website for Numerical Recipes in C. We used the four1 algorithm in the Third Edition. There are detailed descriptions of how the algorithm works, alternative FFTs, and a general background in FFTs. You can read the Third Edition for free.
- <https://cplusplus.com/reference/cstdio/sprintf/> Documentation for `sprintf`. For more details on all of `sprintf()`'s parameters refer to <https://cplusplus.com/reference/cstdio/printf/>

## 8 Addendum and Debugging

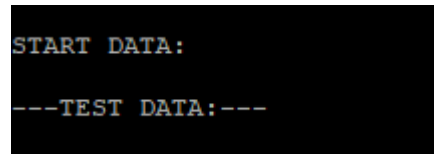
If you're having difficulty with the output of the FFT refer to the images and captions below to better understand what could be causing your problem.

- If the output of your FFT looks like the image below you probably didn't set **Use newlib-nano Float Formatting** from **False** to **True**. Refer to step 7 in section 3.1.



```
---DC DATA:---  
  
START DATA:  
  
Frequency Hz: + j  
Frequency Hz: + j  
Frequency Hz: + j  
Frequency Hz: + j  
Frequency Hz: + j  
Frequency Hz: + j  
Frequency Hz: + j  
Frequency Hz: + j  
Frequency Hz: + j  
  
END DATA!  
  
START MAG:
```

- If the output of your FFT looks like the image below you probably don't have enough memory to perform `sprintf()`. Make sure you followed step 4 in section 3.1. If you did and are still experiencing problems try increasing your heap size even further.



```
START DATA:  
  
---TEST DATA:---
```