Memoria Descriptiva de la Patente de Innovación sobre: "ANALIZADOR DE ESPECTRO"

A. A. Baldini, N. H. Gimenez, T. A. Pentacolo, L. D. Rispoli.

baldiniaxel@gmail.com, nicolas.hernan.gimenez@gmail.com, pentacoloagustin@gmail.com, rispoli_luciano@yahoo.com.ar

Resumen - La presente publicación instituye una memoria descriptiva, acerca del proceso de investigación y desarrollo sobre un Analizador de Espectro. El mismo es llevado a cabo por los suscribientes para la citada catedra de estudios. Se aborda lo relacionado al marco teórico y el desarrollo de un dispositivo en el rango de modelo de prueba, mediante la utilización de Arduino Due.

Palabras Claves – Analizador de Espectros Digital, Arduino Due, Filtro Hamming, FFT.

1. CONSIDERACIONES GENERALES

A. Definición

Un analizador de espectro es un instrumento de medición de señales eléctricas, el cual nos permite registrar y visualizar el espectro de potencia de una señal eléctrica en la entrada, en función de la frecuencia. Es decir, nos permite ver de forma cuantificada las distintas intensidades frecuenciales de la señal, sin importar el origen de la misma (eléctrica, acústica u óptica.), en el dominio de la frecuencia.

Además, no permiten medir el índice de modulación de las señales, ruido, potencia, entre otras cosas.

Dicho espectro suele ser mostrado a través de una pantalla, pero también pueden guardarse los datos obtenidos para analizarlos posteriormente (dependiendo del equipo).

Fig. 1 Analizador de Espectro Owon XSA1032-TG

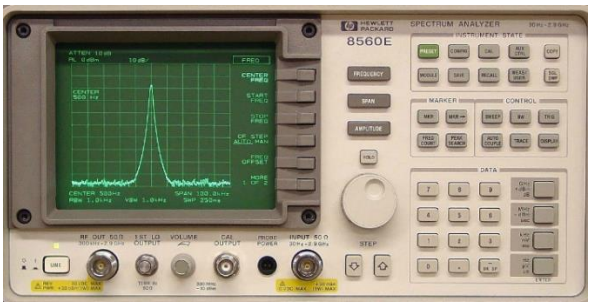


Fig. 2 Espectro de una señal sinusoidal

En la Fig. 2, vemos el espectro de una señal sinusoidal. Si bien esta debería ser una función delta, al tratarse de señales reales; en la realidad, se aproximan a las señales teóricas. Es normal que existan otros componentes frecuencias; sin embargo, se ve claramente que existe un pico en la frecuencia de 500 [Hz].

B. Principales Características

Para empezar, tratamos con un dispositivo que analiza las señales en el dominio frecuencial, un dominio infinito, por ende, la principal especificación es la **acotación frecuencial** que presenta el dispositivo, es decir el ancho de banda que podrá medir. Además, trabajamos con señales eléctricas donde lo que varía es la tensión en función del tiempo, y todos los dispositivos que trabajan con tensiones tienen **máximos de tensión permitidos** antes de que el dispositivo se dañe, por lo que es importante saber cuáles son las **tensiones de entrada máxima** del dispositivo. Cabe aclarar que generalmente se trabaja con señales de información que no se suele trabajar con altas tensiones, pero siempre es importante conocer los límites de nuestros instrumentos para evitar problemas en su operación.

Otra especificación de importancia es la **impedancia de entrada**, ya que, al trabajar con señales de información raramente trabajamos con corrientes altas, al menos en los puntos que utilizamos para medir, esto significa que si nuestro instrumento tiene una impedancia de entrada muy baja comparado al punto que queremos medir, estaremos alterando la señal, despreciando la misma. Por esto es importante tener en cuenta nuestra impedancia de entrada y de ser posible realizar las mediciones en puntos donde la impedancia

del circuito sea considerablemente menor que el del instrumento, de esta forma lograremos medir correctamente las características frecuenciales de la señal.

Otras especificaciones importantes son, el tipo de instrumento, si es **analógico**, **hibrido** o **digital**. En caso de ser digital o hibrido, la posibilidad de **utilizar diferentes ventanas** para analizar las señales. Luego de esto, ya podemos entrar en especificaciones de comodidad, que permita tener pre-configuraciones para analizar los datos en la pantalla, que nos permita guardar la señal o información de la misma, con la mayor comodidad.

C. Modelos en el mercado argentino

Solo se muestran los modelos de fácil acceso a compra por estar a la venta directa sin requerir pedido al exterior y limitaciones por el estado cambiario de moneda.

Fig. 6 Siglent SSA3000X.(U$D 2600 - U$D 1550)

Fig. 7 Owon XSA1000TG.(U$D2400 - U$D 1350)

2. MARCO TEÓRICO

A. Esquema Básico

El analizador de espectros, realiza un análisis de señales dinámicas. Esto implica transformar una señal en el dominio del tiempo al dominio de la frecuencia. Es dinámico, porque la señal varía con el tiempo a medida que realizamos la transformación.

Para esto se emplea el algoritmo de **Transformada rápida de Fourier (FFT)**.

El diagrama de bloques básico de un analizador de señales dinámicas o analizador de espectros se muestra en la siguiente figura.

Fig. 3 Esquema general de un Analizador de Espectro

1) Muestreo y digitalización: La señal en el dominio del tiempo debe ser muestreada; antes de ser digitalizada y provista al procesador FFT. Las muestras deben ser una réplica de la señal original y, por lo tanto, la velocidad de muestreo es importante. Debe confirmar el **Teorema de Muestreo** de **Shannon** y los criterios de **Nyquist**.

Después de muestrear la señal, la digitalización se realiza mediante el **convertidor de analógico a digital (ADC)**. El **muestreador** debe procesar la entrada exactamente en el momento correcto y debe mantener con precisión el voltaje de entrada medido en este momento hasta que el ADC haya terminado su conversión. El ADC debe tener alta resolución y linealidad.

Por ejemplo, para 70 [dB] de rango dinámico, el ADC debe tener al menos 12 [bits] de resolución y la mitad de linealidad de bits menos significativa.

Habiendo digitalizado cada muestra, se toma N muestras consecutivas, igualmente espaciadas, lo que se llama **registro de tiempo**, como nuestra entrada al procesador FFT. Para facilitar la simplificación, generalmente tomamos N como un múltiplo de 2.

2) Procesado por FFT: El **procesador FFT** transforma el registro de tiempo de entrada en un bloque completo de líneas de frecuencia. Esto se ejemplifica en la siguiente figura:

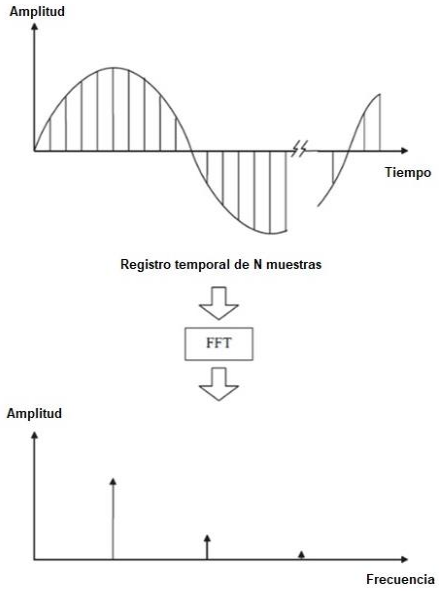


Fig. 4 Esquema general de un procesamiento por FFT

Como la FFT transforma todo el bloque de registro de tiempo en un bloque de líneas de dominio de frecuencia.

No puede haber un resultado de dominio de frecuencia válido hasta que se complete el registro de dominio de tiempo. Sin embargo, una vez que el registro de tiempo se llena inicialmente, en la siguiente instancia de muestreo, la muestra más antigua puede descartarse y la nueva muestra puede ser agregado. Esto, a su vez, permite que todas las muestras se desplacen en el registro de tiempo como se muestra en la siguiente imagen.

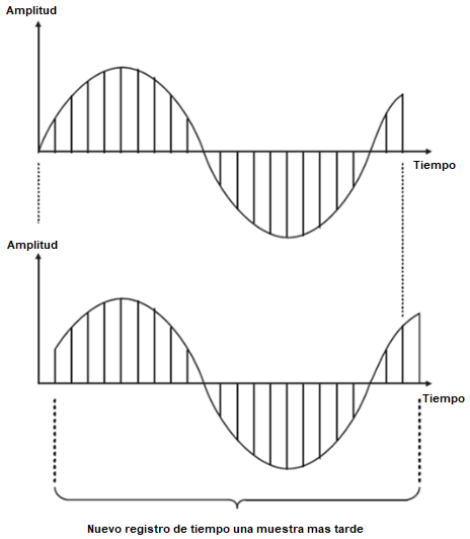


Fig. 5 Muestras desplazadas en el registro Tiempo

Finalmente, el bloque de líneas de dominio de frecuencia que sale del procesador FFT se muestra a través del **display**.

A. Esquema Avanzado

Nos dedicaremos a centrar la pesquisa en los Analizadores de espectro digitales.

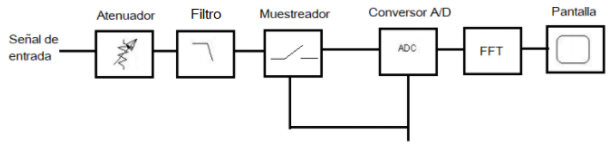


Fig. 8 Diagrama en bloques simplificado de un analizador de espectro

La Fig. 8 muestra el diagrama en bloques de un analizador de espectro. La señal de entrada pasa primero a través de un **atenuador variable**, proporcionando varios rangos de adecuación de la señal. Posteriormente, la señal se filtra mediante un **filtro paso bajo**; para eliminar el contenido de altas frecuencias las cuales son superiores al rango de frecuencias del instrumento.

Posteriormente, la señal es muestreada y convertida a formato digital mediante la combinación de un **muestreador** (en inglés: **SAMPLER**) y un **convertidor de señal analógico a digital** (**ADC**).

Finalmente, se calcula su espectro mediante **un módulo de FFT** para luego ser visualizado en la **pantalla del analizador** (**DISPLAY**).

B. Atenuador

Los atenuadores, se utilizan para reducir la amplitud o potencia de una señal sin distorsionar la forma de onda de la misma; el cual se coloca entre la señal de entrada y nuestro instrumento.

La entrada del analizador de espectros, en el diagrama de bloques, contiene un atenuador con el objetivo de controlar la señal que se aplica al circuito.

Si el nivel de señal es demasiado grande, los circuitos distorsionarán la señal generando productos de distorsión que se añaden a la señal de entrada (Fig. 9).

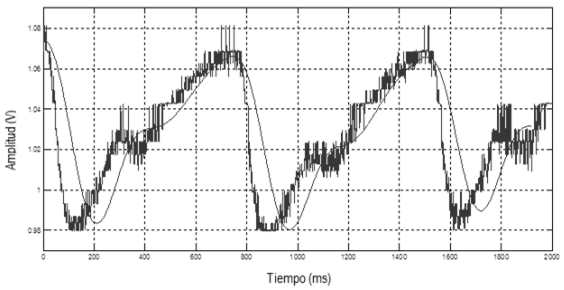


Fig. 9 Espectro de una señal sinusoidal

Si el nivel de señal es demasiado pequeño, la misma puede resultar enmascarada por el ruido presente en el analizador.

Algunos instrumentos proporcionan una característica de auto-rango que automáticamente selecciona la mejor atenuación.

C. Filtros

En esta ocasión, se aprovechó la utilización de un amplificador operacional para hacer un filtro activo pasa bajo de segundo orden; el cual el circuito integrado **LM358** tiene un ancho de banda máximo de 1 [MHz]. Esta característica sera la limitante del amplificador operacional según datasheet.

La principal característica de colocar este amplificador operacional es tener una alta impedancia a la entrada, para no distorsionar o consumir energía de la señal a medir.

Por otra parte, este filtro es el encargado de discriminar las frecuencias altas que quedaran fuera del rango de operación del instrumento.

El filtro activo implementado (Fig. 10) se denomina **Sallen-Key** de segundo orden y su circuito general es como se muestra en la siguiente imagen.

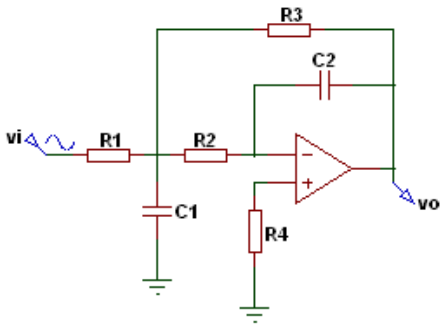


Fig. 10 El filtro activo Sallen-Key

Deduciendo del circuito anterior, podemos obtener su función de transferencia que será:

$$\frac{vo}{vi}(s) = \frac{\left(1+\frac{R_b}{R_a}\right)\frac{1}{C_1C_2R_1R_2}}{s^2 + s\left(\frac{1}{C_1R_2}+\frac{1}{C_1R_1}-\frac{R_b}{C_2R_2R_a}\right)+\frac{1}{C_1C_2R_1R_2}} \quad (1)$$

El mismo fue simulado y calculado de la siguiente manera:

$$fc = \frac{1}{2\pi\sqrt{R3.R4.C1.C2}} \quad (2)$$

La frecuencia de corte la fijaremos en 10 [MHz] debido a que no es un filtro ideal y posee una banda de transición, por lo tanto, para obtener el rango de trabajo adecuado para nuestro instrumento fijamos una frecuencia de corte superior.

Los capacitores C2 y C1 adoptamos el valor de un 1[nF].

Adoptamos R3 = R4 y despejando:

$$R3 = R4 = \frac{1}{2\pi.fc.\sqrt{C1.C2}} \quad (3)$$

En la siguiente imagen podemos observar la simulación y la respuesta en frecuencia del mismo.

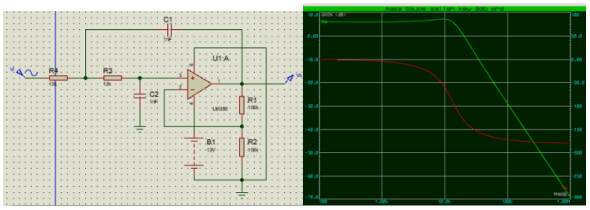


Fig. 11 Simulación y respuesta en frecuencia.

Como se observa en la Fig. 11, la curva de color verde es la fase, y la curva de color roja es la salida del filtro.

D. Muestreo y Digitalización

Como en todos los instrumentos y dispositivos que trabajen con señales analógicas de forma digital, el analizador de espectro requiere de alguna etapa que se encargue de convertir las señales analógicas en digitales. Se mencionó previamente que este dispositivo es llamados Conversor de Señal Analógico a Digital (Analog to Digital Converter (ADC)). El mismo cuentan con dos propiedades importantes y que determinan su calidad, estas son:

- **Resolución de cuantificación**
- **Frecuencia de muestreo**

La primera determina la exactitud que posee el ADC a la hora de obtener los valores de amplitud de la señal analógica, mientras que la segunda determina el número de muestras que toma el ADC en un segundo.

La principal limitación que presenta la frecuencia de muestreo en la mayoría de los instrumentos es respecto a la frecuencia máxima para la cual el instrumento es sensible (esta sensibilidad se debe a un filtro pasa bajos que elimina todos los armónicos en la señal que producirían solapamiento (**aliasing)**).

En los analizadores de espectro digitales la frecuencia de muestreo también cumple este rol; pero, además, la resolución del analizador de espectro depende de esta frecuencia, así como también de la cantidad de muestras utilizadas.

Cabe aclarar, que la manera de interpretar el termino “resolución” es diferente en instrumentos que trabajan en el campo temporal respecto a instrumentos que trabajan en el campo frecuencial como el analizador de espectro, ya que en los primeros hablamos de la

resolución de la señal muestreada, mientras que en los segundos hablamos de la cantidad de frecuencias que podemos analizar; definiéndose de esta manera la **resolución frecuencial**.

También se afecta la resolución de la señal muestreada en los instrumentos de campo frecuencial, pero en menor medida, ya que, si bien queremos saber los valores de potencia, muchas veces lo que buscamos es obtener información sobre las frecuencias utilizando como referencia la potencia. Por ejemplo, lo que nos interesa es saber si existe un pico en una frecuencia f_1 y obtener un valor aproximado de su potencia. Por ende, es preferible una mayor resolución frecuencial sacrificando un poco de resolución en cuanto a la potencia calculada.

La resolución frecuencial depende tanto de la frecuencia de muestreo como de la cantidad de valores procesados (cantidad de muestras).

Suponiendo una frecuencia de muestreo fija (lo cual ocurre ya que no se puede variar la misma dependiendo de la señal de entrada), lo anterior nos da una idea de que mientras más muestras tome nuestro instrumento mejor será la resolución. Si bien esto en principio es cierto, hay que tener en cuenta que estamos trabajando con dispositivos electrónicos, los cuales tienen limitaciones a la hora de realizar cálculos. Por ende, un valor de muestras excesivamente alto solo haría que el sistema deje de funcionar de forma eficiente. En consecuencia, hay que ensayar buscando el mayor número de muestras para las cuales nuestro procesador pueda aplicar la FFT en un tiempo menor a un periodo de muestreo, de esta forma cuando entre un nuevo dato podemos procesarlo.

Como ya se dijo, a veces conviene sacrificar resolución en la señal para ganar resolución frecuencial, esto significa en vez de trabajar con un ADC de 10 [bits], los cuales nos darán enteros de 0 a 1023, nos conviene trabajar con un ADC de 8 [bits], es decir de 0 a 255; de esta forma tanto la conversión durante el muestreo así como el tiempo de cálculo de la FFT disminuirá considerablemente, permitiéndonos utilizar con una misma frecuencia de muestreo una mayor cantidad de datos, lo que significa una mayor resolución frecuencial.

La forma de tomar los datos constara de tener un vector de 2^{γ} valores [1], los cuales están inicializados en "0", y cumplen el rol de una cola de datos. Es decir, cada vez que entra un nuevo dato, todos rotan y el último elemento de la cola se elimina.

E. Transformada Rápida de Fourier

1) *Historia*: El primer antecedente te la Transformada Rápida de Fourier (FFT) no fue encontrado hasta al haberse redescubierto el algoritmo en la década del 60.

Muchos investigadores han hecho uso en diferentes áreas por fuera de la matemática, pero con el interés solo de resolver su problema físico e incluso sin divulgación por considerarlo de poca validez para otras áreas. Es el caso de P. Rudnick, que lo desarrollo su mejora para un estudio en su área de oceanografía, basándose en un paper de G. C. Danielson & C. Lanczos (1942) usado en el área de difracción de rayo X; y posteriormente, una vez que vio el auge de el algoritmo planteado por J.W. Cooley & J. W.Tuckey en 1965.

Para cuando Rudnick trato de establecer que había desarrollado el método de forma personal, sin contacto con Cooley y Tuckey; la popularización hizo que nadie lo tuviera en cuenta para el crédito, a amen de Cooley y Tuckey reconocieron que era el método de ellos y que se descubrió por forma separada uno tiempo antes.

También la historia, dio también como autor a C. F. Gauss que descubrió el método un siglo antes (1805). Su algoritmo fue desestimado por él y publicado en una recopilación póstuma. Él lo uso para describir la trayectoria del asteroide Pallas y la forma escrita estaba relacionada con la forma de Euler en términos de cosenos y senos.

La moraleja de esto es: **tener debidamente documentado todo**, uno nunca sabe para qué puede servir o servir y mejorar otra área, tan o más importante que la nuestra.

Luego del paper de Cooley y Tukey en 1965, muchos investigadores incursionaron a investigar esta área obteniendo mejoras cada vez más importantes y en área de señales n-dimensionales.

A continuación, un listado cronológico de los hitos y autores en descubrimientos de la FFT.

[1] Cfr. Sección 2. E: *Transformada Rápida de Fourier*

Investigador	Fecha
C.F. Gauss	1805
F.Carlini	1828
A.Smith	1846
J. D. Everett	1860
C. Runge	1903
K. Stumpff	1939
G. C. Danielson & C. Lanczos	1942
L. H. Thomas	1948
Teorema del resto chino	
I.J. Good	1958
Teorema del resto chino	
J. W. Cooley & J. W. Tukey	1965
P. Rudnick	
C. M. Rader & Brenner	1968
Convolucion Ciclica por dos FFT ordinarias	
L. Bluestein	1968
Algoritmo (Chirp-Z) Transformada CZT	
S. Winograd	1976
(Otras formas Convolucion)	
G. Bruun	1978
Factorización polinomial recursiva inusual en potencias de 2	
H. Murakami	1996
Tamaños compuestos arbitrarios	
J.B. Birdsong & N. I. Rummelt	2016
Cuadrículas hexagonales FFT Bidimencional	

Fig. 12 Hitos Historicos

2) La Transformada Rápida de Fourier:

a) Introducción: La Transformada Rápida de Fourier (Fast Fourier Transform: F.F.T.) es un algoritmo matemático, que nos permite hallar la Transformada Discreta de Fourier, de una forma más veloz debido a una importante reducción de cálculos.

A su vez, este eficiente algoritmo (en términos de tiempo), puede ser adaptado para recuperar la señal en dominio tiempo sin utilizar la ecuación de síntesis en el sentido clásico. El uso de esta implementación del algoritmo se la conoce como Transformada Rápida Inversa de Fourier (Invert Fast Fourier Transform: L.F.F.T.).

Nosotros explicaremos los algoritmos desarrollados por J.W. Cooley y J.W. Tukey en 1965.

b) Algoritmo de Cooley-Tukey: Recordando que la transformada discreta de Fourier posee la siguiente formula:

$$X\left(e^{i\omega}\right)=\sum_{n=-\infty}^{\infty}x[n]\cdot e^{-i\omega n} \quad (4)$$

Podemos simplificarla, teniendo en cuenta que $\omega=\frac{2\pi k}{N}$. Remplazando en la ecuación anterior y considerando que x[n] es una función periódica con período N obtenemos:

$$X\left(k\right)=\sum_{n=0}^{N}x[n]\cdot W_N^{n\cdot k} \quad (5)$$

Aquí, si nosotros empezáramos a desarrollar tradicionalmente tendríamos que, por cada punto de n e N, hacer N multiplicaciones y sumarlas, o sea un total de N*N multiplicaciones y N(N-1) adiciones.

Para el desarrollo del algoritmo, se tiene en cuenta que el período N pude presentarse en las siguientes formas:

$$\bullet\ N=2^{\gamma} \quad (6)$$

N, se puede expresar como potencia de 2. $\gamma \geq 1 \in \aleph$.

$$\bullet\ N=r_1\cdot r_2\cdot r_3\cdots r_n \quad (7)$$

N, se puede expresar como producto de r_n,, donde los r_n, son factores de N.

$$\bullet\ N=c \quad (8)$$

N, es un número primo c.

c) Caso $N=2^{\gamma}$: Para este proceso representaremos a k y a n, como funciones de γ variables binarias que adoptan valores de 0 y 1.

$k=0,1,2,3\ldots \Rightarrow k=\{k_{\gamma-1}, k_{\gamma-2}, \ldots, k1, k0\}=00\ldots00, 00\ldots01, 00\ldots10, 00\ldots11,\ldots$

$n=0,1,2,3\ldots \Rightarrow n=\{n_{\gamma-1}, n_{\gamma-2}, \ldots, n1, n0\}=00\ldots00, 00\ldots01, 00\ldots10, 00\ldots11,\ldots$

(9)

Ósea:

$$k=2^{\gamma-1}k_{\gamma-1}+2^{\gamma-2}k_{\gamma-2}+\ldots+k_0$$
$$n=2^{\gamma-1}n_{\gamma-1}+2^{\gamma-2}n_{\gamma-2}+\ldots+n_0$$

(10)

Visto esto, remplazando en la ecuación de la D.F.T. obtenemos:

$$X\left(k_{\gamma-1},k_{\gamma-2},\cdots,k_0\right)=\sum_{n_0=0}^{1}\sum_{n_1=0}^{1}\cdots\sum_{n_{\gamma-1}=0}^{1}x\left[n_{\gamma-1},n_{\gamma-2},\cdots,n_0\right]\cdot W^{p}$$

(11)

Donde p es:

$$p=\left(2^{\gamma-1}k_{\gamma-1}+2^{\gamma-2}k_{\gamma-2}+\ldots+k_0\right)\left(2^{\gamma-1}n_{\gamma-1}+2^{\gamma-2}n_{\gamma-2}+\ldots+n_0\right)$$

(12)

A continuación, operaremos con W^P, optimizándolo para un mejor análisis.

$$W^p = W^{n\cdot k} = W^{\left(2^{\gamma-1}k_{\gamma-1}+2^{\gamma-2}k_{\gamma-2}+...+k_0\right)\cdot\left(2^{\gamma-1}n_{\gamma-1}+2^{\gamma-2}n_{\gamma-2}+...+n_0\right)} = \quad (13)$$

Por lo que distribuiremos multiplicando la potencia, para luego enfocaremos en el primer término de la productoria, que conforma a W^P.

$$W^{\left(2^{\gamma-1}k_{\gamma-1}+2^{\gamma-2}k_{\gamma-2}+...+k_0\right)\cdot\left(2^{\gamma-1}n_{\gamma-1}\right)} \quad (14)$$

obteniendo

$$= W^{2^{\gamma-1}\cdot k_0\cdot n_{\gamma-1}} \quad (15)$$

porque:

$$W^{\gamma-1} = W^N = 1 \quad (16)$$

Seguidamente, tomaremos el segundo término de la productoria que conforma W^P con lo que concluimos que:

$$W^{\left(2^{\gamma-1}k_{\gamma-1}+2^{\gamma-2}k_{\gamma-2}+...+k_0\right)\cdot\left(2^{\gamma-2}n_{\gamma-2}\right)} \quad (17)$$

obteniendo

$$= W^{2^{\gamma-2}k_0\cdot n_{\gamma-2}} \quad (18)$$

El proceso de análisis de los demás términos de la productoria, continua de forma análoga con cada uno, quedando las expresiones más simplificadas.

3) Módulo FFT: En el apartado de diagrama Fig. 8 el modulo respectivo realiza los cálculos por medio de un programa de interracion que permite resolver el mismo tomando los datos del anterior modulo y poniéndolo disponible para el display.

A veces los datos de entrada pueden ser divididos electrónicamente para poder operar de forma eficiente y hacer cálculos simultáneos como se muestra en la siguiente imagen, donde la mitad de los datos es procesado por un sub modulo y el resto por otro. Arrojando datos parciales que luego si son combinados en una FFT Total.

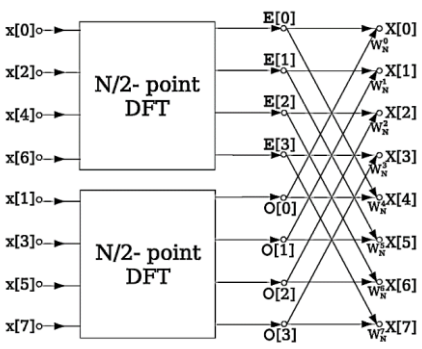


Fig. 13 Diezmado en tiempo con submodulos.

En cierta forma esto emula el concepto inicial de Cooley-Tukey. Este subdividía la señal y operaba.

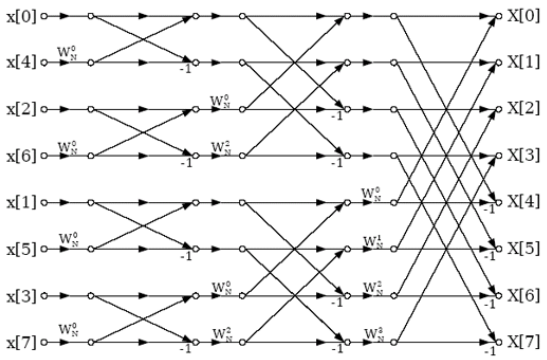


Fig. 14 Diezmado en tiempo.

Teniendo como mínimo una operación entre dos puntos denominada operación Mariposa.

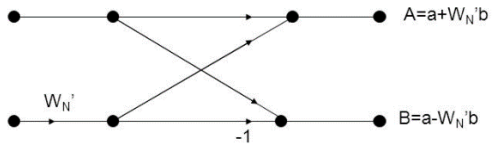


Fig. 15 Operación Mariposa.

De ahí que el algoritmo toma el nombre de método de las Mariposas dado a que la forma visual asemeja a una mariposa.

La forma de calcular se usó de forma manual y en el ámbito digital se usa para acelerar el cálculo por hardware antes del procesamiento por software. Aunque se debe considerar los efectos de lag por combinación de ambos.

4) *Aplicaciones fuera del campo matemático*: Las aplicaciones abiertas por la creciente demanda computacional a mediados de los 70, provoco variados campos de estudios que al día de hoy siguen siendo campos de investigación. A saber:

•Software de grabación digital, muestreo, síntesis aditiva y corrección de tono

•Multiplicación rápida de números enteros grandes y polinomios

•Multiplicación eficiente de matriz-vector para Toeplitz , circulante y otras matrices estructuradas

•Algoritmos de filtrado (ver métodos de superposición-añadir y superponer-guardar)

•Algoritmos rápidos para transformaciones de seno o coseno discretas (por ejemplo, DCT rápido utilizado

para codificación y decodificación JPEG y MPEG / MP3)

•Aproximación rápida de Chebyshev

•Ecuaciones en diferencias

•Cálculo de distribuciones isotópicas

•Modulación y demodulación de símbolos de datos complejos utilizando multiplexación por división de frecuencia ortogonal (OFDM) para 5G, LTE, Wi-Fi, DSL y otros sistemas de comunicación modernos

entre otros.

F. LA FFT Y FUNCIONES VENTANAS.

a) *Introducción:* La transformada rápida de Fourier (FFT) opera sobre un registro de tiempo de longitud finita en un intento de aproximarse la transformada de Fourier, que integra sobre un tiempo infinito.

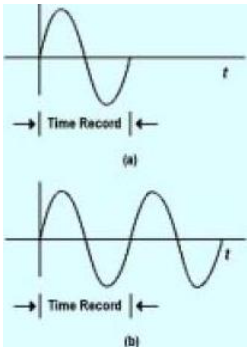


Fig. 16 Señal senoidal con el registro de tiempo.

Se puede observar en la Fig. 16 (a) la forma de onda de la señal encaja perfectamente con el registro de tiempo. Luego en Fig. 16 (b) podemos ver cuando la señal se replica nuevamente, no se introduce ningún transitorio.

La forma de onda y la fase de la señal pueden introducir cambios transitorios cuando la señal es replicada nuevamente como se muestra en la Fig. 17 (b).

Como podemos ver la forma de onda no encaja perfectamente con el registro de tiempo Fig. 17 (a).

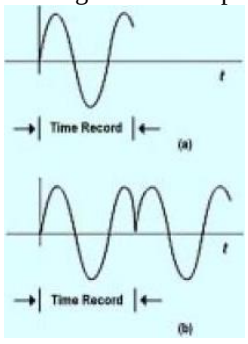


Fig. 17 Forma de onda y fase de la señal anterior replicada

Luego cuando replicamos la señal nuevamente, se introducirán transitorios que producen pérdidas o leakage en el dominio de la frecuencia.

Un modo de solucionar esta pérdida es forzar la señal a que termine en cero en los extremos del registro de tiempo, de forma cuando el registro de tiempos se replique no surjan transitorios. El forzado de la señal a cero se realiza multiplicando el registro de tiempo por una función de ventana (window o enventanado).

La forma de la ventana que elijamos para corregir esta perdida, aplicara distintos efectos dándonos distintos resultados, cediendo o ganando ciertas características.

Es muy importante conocer como afectara la ventana a nuestro resultado final.

Otro uso de las funciones ventanas es tratar de reducir la dispersión espectral y tratar de mantener el lóbulo principal lo más estrecho posible para conseguir una buena resolución espectral.

Nosotros veremos algunas de las ventanas más conocidas.

b) Tipos de Ventana:

Ventana Hanning (Hann): Esta función es una de la más utilizada en procesado digital de señal. Las muestras de tiempo se ponderan mediante la función Hanning, proporcionando una transición suave a cero en los extremos del registro de tiempo, por tanto, el registro de muestras no producirá un transitorio cuando sea replicado por el algoritmo FFT.

Aunque la forma de onda en el dominio del tiempo ha cambiado, después de aplicar la función Hanning, su contenido en frecuencia permanece prácticamente inalterado.

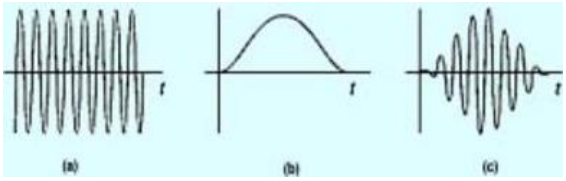


Fig. 18 (a) El registro de tiempo original. (b) La ventana Hanning. (c) El registro de tiempo después de aplicar la función de ventana Hanning.

La línea espectral asociada con la sinusoide se extiende una pequeña cantidad en el dominio de la frecuencia como muestra la Figura 19.

La ventana Hanning pone en la balanza entre la precisión en amplitud y la resolución en frecuencia y comparada con otras funciones de ventana proporciona la mejor resolución en frecuencia, teniendo como costo es la reducción de la exactitud en amplitud.

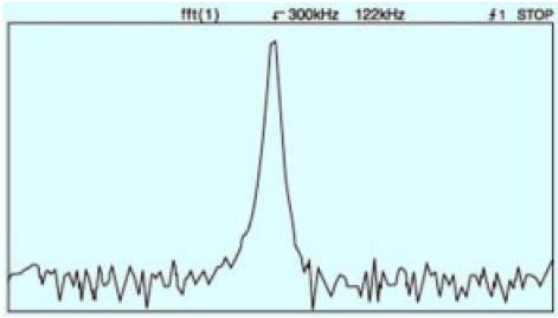


Fig. 19 La ventana Hanning produce líneas espectrales relativamente estrechas.

Ventana Flattop: Esta ventana tiene una banda de paso más plana y reduce las diferencias en amplitud entre las muestras minimizando el error en amplitud.

La ventada Flattop se considera muy precisa en amplitud, teniendo un error máximo de amplitud de 0.1 dB (1%).

Como contrapartida, la resolución en frecuencia es menor que en el caso de la función Hanning como podemos ver en la Fig, 20.

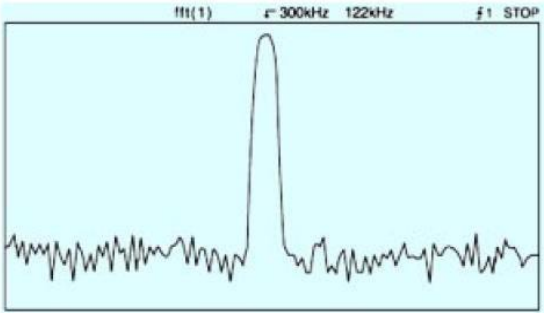


Fig. 20 La ventana Flattop genera líneas espectrales más anchas.

Ventana Uniform: La ventana Uniform no se considera realmente una ventana ya que no altera las muestras. Se usa en aquellas formas de onda que tienen el mismo valor en los dos extremos.

Ventana Exponencial: Una de las ventajas del analizador FFT es que se puede usar para medir el contenido en frecuencia de señales transitorias rápidas.

En este tipo de ventana, la porción del principio de la señal no se altera, mientras que se fuerza a cero al final del registro de tiempo.

Esto resulta muy conveniente en el análisis de señales transitorias.

c) Selección de una función ventana.

La mayor parte de las medidas requerirán el uso de funciones Hanning o Flattop. Estas dos funciones de ventana son las más usuales en medidas de análisis de espectros.

Elegir entre estas dos ventanas supone un compromiso entre resolución en frecuencia y exactitud en amplitud como explicamos anteriormente.

La ventana Hanning se utiliza cuando necesitamos la mejor resolución en frecuencia.

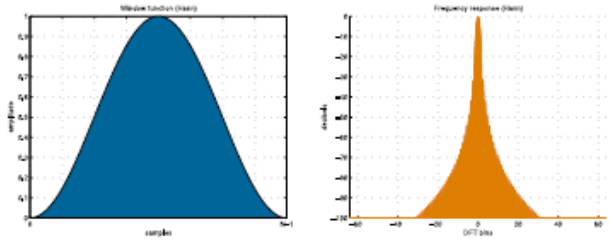
La ventana Flattop da mejores resultados en la resolución de amplitud.

La ventana Uniforme se usa cuando se puede garantizar que no habrá efectos de fuga o pérdidas.

La ventana Exponencial se usa cuando la señal a medir es un transitorio.

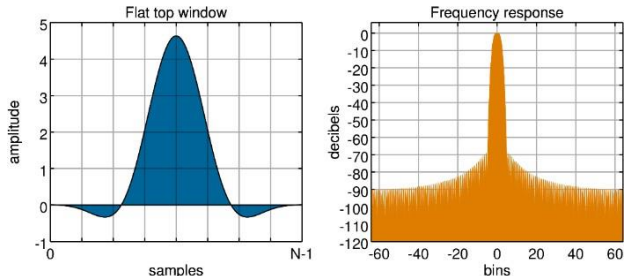
Estas dos últimas ventanas se deben considerar solo en situaciones especiales.

d) Comparación de ventanas.



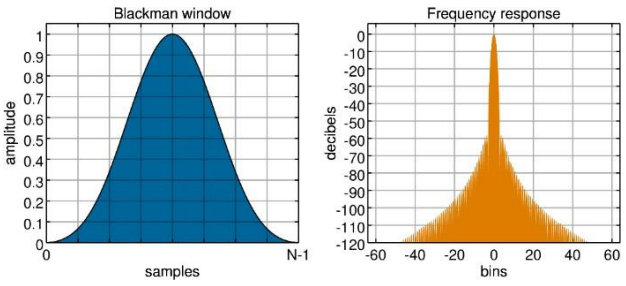
$$v(n) = a0 - a1 \cdot \cos\left(\frac{2\pi n}{N-1}\right)$$
$$a0 = 0{,}5; a1 = 0{,}5$$

Fig. 21 Ventana Hann



$$v(n) = a0 - a1 \cdot \cos\left(\frac{2\pi n}{N-1}\right) + a2 \cdot \cos\left(\frac{4\pi n}{N-1}\right) - a3 \cdot \cos\left(\frac{6\pi n}{N-1}\right) + a4 \cdot \cos\left(\frac{8\pi n}{N-1}\right)$$
$$a0 = 1;\ a1 = 1{,}93;\ a2 = 1{,}29; a3 = 0{,}388; a4 = 0{,}032$$

Fig. 22 Ventana Flat top



$$v(n) = a0 - a1 \cdot \cos\left(\frac{2\pi n}{N-1}\right) + a2 \cdot \cos\left(\frac{4\pi n}{N-1}\right)$$
$$a0 = 0{,}42; a1 = 0{,}5; a2 = 0{,}08$$

Fig. 23 Ventana Blackman

e) Ejemplos de aplicación de ventanas

Se enlistan las aplicaciones y ventana típica para ello.

Contenido de la señal	Ventana
Onda sinusoidal o combinación de ondas sinusoidales.	Hann
Onda sinusal (la precisión de amplitud es importante).	Flat Top
Señal aleatoria de banda estrecha (Narrowband) (datos de la vibración).	Hann
Banda ancha aleatoria (Ruido blanco).	Uniform
Ondas sinusoidales muy espaciadas.	Uniform
Señal de repuesta (golpe de martillo).	Exponencial
Dos tonos con frecuencias cercanas y amplitudes casi iguales.	Uniform
Mediciones precisas de amplitud de un solo tono.	Flat Top

Fig. 24 Aplicaciones

3. IMPLEMENTACIÓN DEL PROTOTIPO

Para la presente implementación se seleccionó la placa Arduino Due.

A. Características de la Placa Arduino Due

a) Características Principales

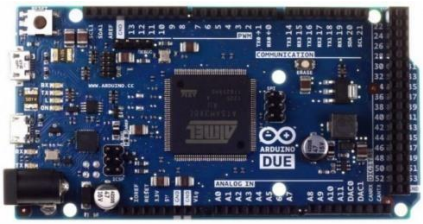


Fig. 25 – Arduino Due.

- El microcontrolador que integra a esta placa es el Atmel SAM3X8E ARM Cortex-M3 de 32 [bits].
  Esto hace que tengamos potencia de cálculos (realiza cálculos de forma más rápida) comparados con los AVR de 8 [bit]; que es el que poseen la mayoría de las placas similares.
- Posee una velocidad de reloj, o de respuesta, mucho más elevada que los otros Arduino (84 [Mhz]).
- Posee 2 DAC o pines de salidas analógicas con una resolución de 12 bits (4096 niveles) a una alta frecuencia de muestreo.
- Posee 2 puertos miniUSB de los cuales uno se utiliza para programar y el otro es un puerto nativo que se utiliza para conectar un mouse, teclado, etc.

b) Otras caracteristicas

- Este dispositivo funciona a 3.3 [V], en vez de los 5[V] de las otras placas. Esto lo hace compatible directamente con muchos dispositivos de Arduino que funcionan a ese voltaje.
- Voltaje recomendado de entrada (pin Vin): 7-12[V].
- 4 pines para comunicación serie (Rx y Tx)
- Pines E/S digitales: 54, de los cuales 12 son PWM.
- 4 UART (puertas seriales por hardware)
- 11 pines E/S analógicos.
- Posee un botón para borrar la programación del microcontrolador.

c) Ventajas y Desventajas Arduino Due.

Ventajas:

- Procesador muy potente (ARM Cortex-M3 de 32 [bits]).
- Su velocidad de relog de 84 [Mhz].

Desventajas:

- Entradas digitales a 3.3[V] (no se podría usar los que piden 5[V]).
- No posee memoria EEPROM (no se puede guardar resultados en variables y apagar la placa, conservando los datos).

B. Consideraciones previas a la programación

a) Descomposición en Tiempo

La descomposición en tiempo es un artilugio para ordenar los datos presentes en el vector de entrada; de forma en que el dato en una terminada posición de la muestra pueda ser convocado de forma más eficiente a la hora de efectivizar los cálculos del espectro.

A efectos de cálculo tradicional (la forma gráfica desarrollada por Cooley-Tukey) figura 26, se observa que se necesita los datos de entrada en un ordenamiento especial. Para conseguirlo, se van reordenando los datos de entrada aplicando el criteriode posición par e impar de la muestra; hasta llegar al punto en que ninguna parte de la muestra se puede separar en dos partes más.

El estado final sería un reordenamiento de los datos originales en una forma; donde los datos parecerían estar mezclados al azar, si no fuese porque aplicamos el criterio de posición par e impar.

Partiendo de un ejemplo de 16 posiciones, el resultadolo observamos en la siguiente figura:

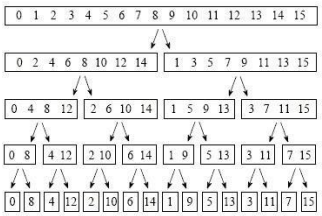


Fig. 26 Desdoblamiento del vector datos.

Si, en consecuencia, representamos las posiciones porsus valores homólogos en binario; se obtiene esta comparación:

0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

Fig. 27 Simetría Binaria del resultado del reordenamiento para dieciséis (16) datos.

Por inspección visual de la figura anterior, encontramos que el reordenamiento aplicado posee una relación a nivel binario, entre el vector de entrada que contiene los datos de la señal con el vector final necesario para agilizar el cálculo del espectro.

Este ordenamiento produce una relación entre posiciones, donde una posición está relacionada por reflexión de su marcador binario.

Tomemos el ejemplo de la posición doce (12), que en binario se expresa como 1100. Al reflejar este número binario se obtiene 0011, lo que da como resultado la posición tres (3). Por lo tanto, se manda el dato contenido en la posición 3 a la posición 12 del nuevo vector arreglado.

Esto se observa en todas las posiciones, aunque hay algunas que no varían dada su simetría y se le llaman puntos fijos de la muestra. Las mismas son la destacadas en color amarillo en las imágenes.

El mismo trabajo podemos aplicar a un vector de ocho datos obteniendo:

0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Fig.28 Simetría Binaria del resultado del reordenamiento para ocho (8) datos.

Comparando la figura 27 con la figura 28, tenemos que el resultado es el mismo que dividir en dos la muestra y reordenado por separado como si fuesen ambos vectores de 8 datos y luego alternando uno con otro.

Por lo que gráficamente podemos reordenar los datos de esta forma.

Sin embargo, para la aplicación computacional, nos conviene mantener el aspecto binario y controlar los bits reflejados de las posiciones.

b) Síntesis del Dominio Frecuencia: Una vez obtenido el vector necesario, el esquema de aplicación de cálculo era el siguiente:

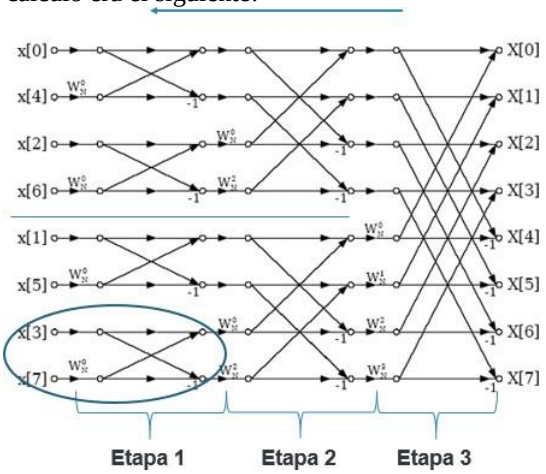


Fig. 29 Algoritmo grafico de Cooley-Tukey para ocho (8) datos.

De la imagen se desprende que, el caculo mínimo es entre dos puntos a los que se le aplica la operación mariposa como pre cálculo.

La etapa uno (1) termina al completar las dos operaciones mariposa por cada sub DFT. Por lo tanto, se necesita un Loop para controlar el cálculo mariposa como también otro Loop para controlas las sub DFT de la etapa dos (2).

Un tercer Loop es necesario para que controle el cálculo general ya se considera el caso donde N=2□ Por lo tanto, en esta sección podemos ver que se respeta el siguiente diagrama de flujo:

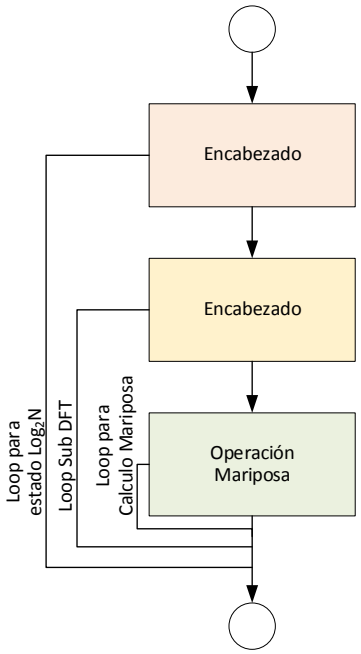


Fig. 30 Diagrama de flujo para la síntesis del dominio frecuencia.

c) *Diagrama de Flujo Para el cálculo de FFT:* Recapitulado, el diagrama total considerando el reordenamiento de datos necesario resulta el ejemplificado en la figura siguiente:

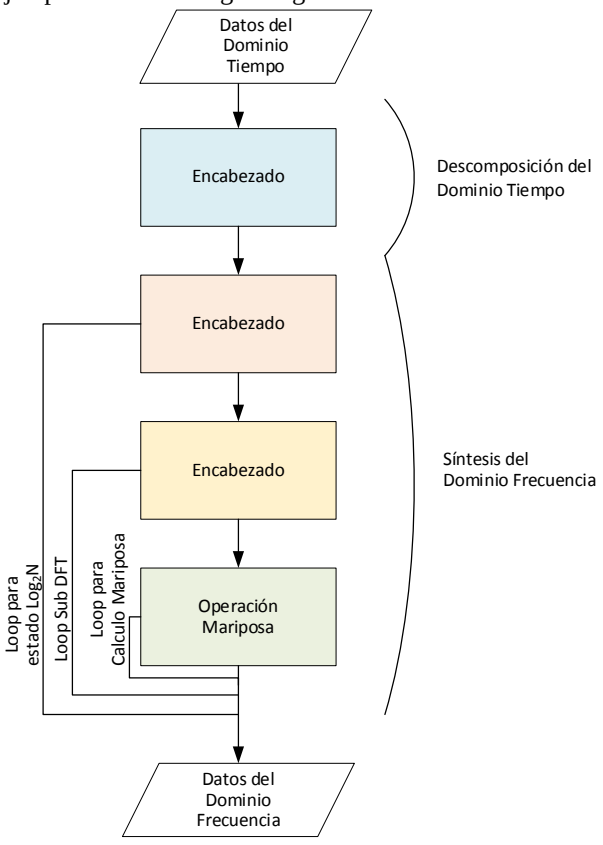


Fig. 31 Diagrama de flujo para el cálculo de la FFT.

En este aspecto, los bloques encabezados representan operaciones básicas entre componentes sin toma de decisiones o interacciones. Las únicas interacciones están marcadas entre los bloques pertinentes.

d) Implementación de la FFT en C++: En cuanto a las implementaciones del algoritmo en C++, forma parte de un objeto estandarizado hace años.

En la sección apéndice A.0, se encuentra el código del objeto donde se marcó las secciones mencionadas previamente. En la sección apéndice A.2 y A.3 la versión para Arduino Due

e) *Programación General*: En lo que respecta al código implementado; el mismo fue diseñado para ser utilizado en un Arduino Due.

Con respecto al código, su implementación y funcionamiento, fue más sencillo que el esperado; y su funcionamiento fue óptimo. Se tiene que agregar algunas cosas que faltan para que funcione completamente, como tener en cuenta la variación de los potenciómetros utilizados como atenuadores y modificar la potencia en base a esto.

En la sección apéndice A.1, se encuentra el código detallado completamente, con las funciones que cumplen cada parte del código, cabeceras y cuerpo respectivamente.

Las interrupciones se encuentran en la sección A.4.

C. Esquemático y PCB

El proyecto lo hemos basado la realización de un shield de Arduino Due; por esta razón nos hemos limitado a realizar un circuito que entre en el tamaño de este.

Respecto a la funcionalidad, toda la parte de cálculo es realizada por el procesador. Por ende, la parte circuito cumple únicamente la función de adaptar la señal de entrada a los niveles de tensión y frecuencia correspondientes para su posterior análisis.

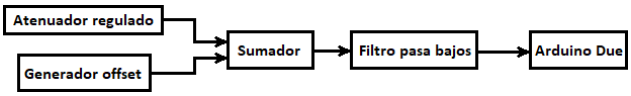


Fig. 32 Esquema del circuito de adaptación de señal de entrada para Arduino Due.

Observando el esquema del circuito que se presenta en la Fig. 32, podemos identificar los siguientes bloques:

a) *Atenuador regulado:* Esta etapa se resume básicamente en un control de volumen de la entrada. Esto nos permite que, si la señal de entrada tiene una amplitud mayor a la permitida por el Arduino Due, la atenuemos hasta obtener una señal medible.

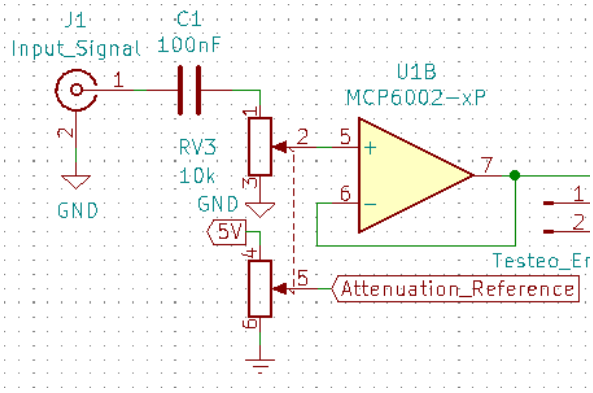


Fig. 33 Esquemático de un Atenuador Regulado.

Como vemos en la Fig. 33, tenemos la entrada con una ficha que nos permite utilizar cable mallado, lo cual nos da la ventaja de suprimir parte del ruido que pueda ingresar por este. Luego filtramos la componente continua de la misma, y seguidamente nos encontramos con el control de volumen.

Se logra apreciar que este control de volumen, utiliza un potenciómetro doble. Esto se decidió de esta forma ya que el segundo potenciómetro nos permite controlar una tensión que variará de 5[V] a 0[V] de manera proporcional a la atenuación o al volumen de la señal de entrada, por lo tanto, el procesador sabrá que tanto se ha atenuado la señal utilizando la señal **Attenuation_Reference**.

Por último, disponemos de un seguidor de tensión para evitar cualquier problema con las corrientes.

b) *Generador de Offset:* El resultado de la etapa anterior es una señal alterna con su amplitud reducida de ser necesario, ahora bien, lo que necesitamos es que dicha señal alterna se encuentre en el rango de 0 a 5[V], para ello debemos.

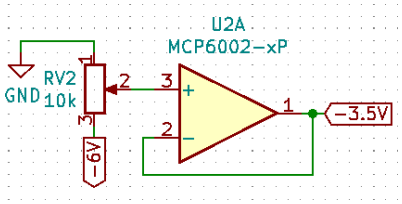


Fig. 34 Esquemático de un generador de offset.

Por lo tanto, esta etapa es un control de volumen que nos brinda una tensión negativa regulable; de forma que logremos restarle la tensión seleccionada. Ahora bien, el circuito se alimenta con 12[V] y 0[V]; por ende, para obtener la tensión negativa lo que se hace es trabajar con una masa virtual. La misma es equivalente a 6[v], por lo cual los 12[V] pasan a ser +6[V] y la masa de la alimentación pasa a ser -6[V]. Esto se decidió para mejorar el funcionamiento de los operacionales.

Sin embargo, el juego de masas nos trajo un nuevo problema: el Arduino está trabajando entonces entre la masa real y la masa virtual, razón por la que necesitamos que la señal se encuentre en dicho rango.

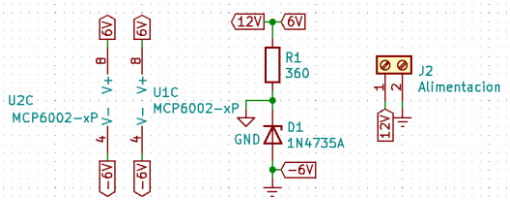


Fig. 35 Adaptación de rango de tensión de las señales

c) *Sumador:* En esta etapa se realiza: la suma de la señal de offset generada previamente; la cual, al ser negativa resulta en una resta en vez de una suma. De esta manera, podemos regular sobre que tensión se encuentra nuestra señal.

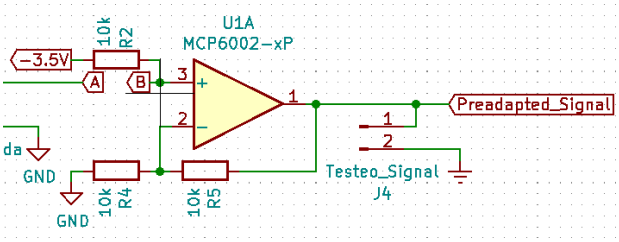


Fig. 36 Esquemático de un Sumador.

Se logra apreciar que falta una resistencia en la imagen anterior (la cual se encontraría conectada entre A y B). Para esto, se ha utilizado un potenciómetro doble de la misma forma que en la etapa inicial; y esto es debido a que la señal puede ser muy pequeña. Por ende, no atenuaremos su señal, sino que nos interesara amplificarla. Para esto sirve este potenciómetro. A su vez, también tenemos una referencia para saber que tanto estamos amplificando la señal.

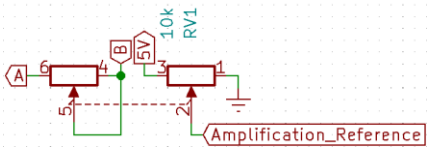


Fig. 37 Esquemático de indicador de porcentaje de amplificación.

Al igual que se ha explicado antes, la señal **Amplification_Reference** sirve para que el procesador sepa cuanto se ha amplificado la señal. Utilizando tanto esta señal como la Attenuation_Reference se puede calcular la potencia real de la señal analizada.

d) *Filtro Pasa Bajos:* La última etapa antes del Arduino, consiste en un filtro pasa bajo anti-aliasing, el cual nos limitara las componentes frecuenciales de la señal, tanto para reducir la potencia del ruido como para evitar el aliasing producido por el efecto de la digitalización.

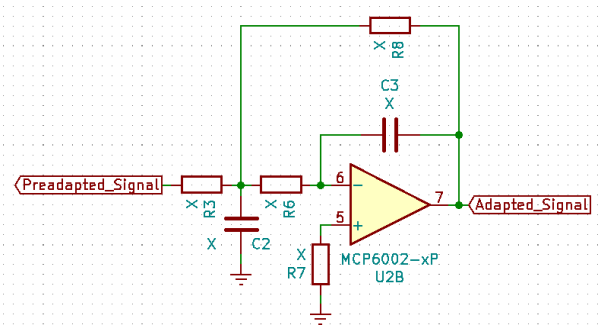


Fig. 38 Esquemático de filtro Pasa Bajos.

Como se logra apreciar, no se ha calculado aún los valores del filtro y esto se debe a que no se ha determinado exactamente cuál será la frecuencia de muestreo del Arduino, pero consta básicamente de un filtro pasa bajos activo de segundo orden, el cual tendrá como frecuencia de corte la mitad de la frecuencia de muestreo.

e) *Tentativa disposición de pines:*

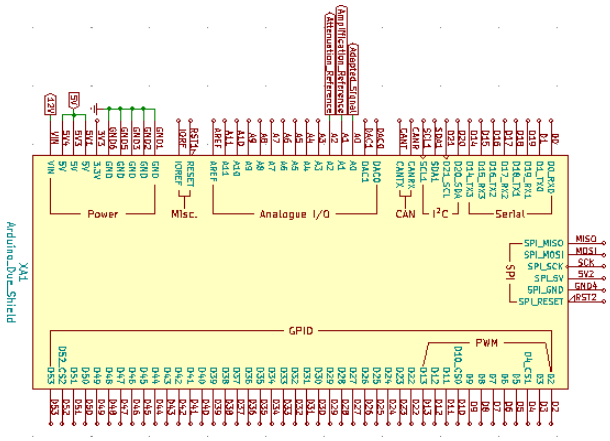


Fig 39 Esquemático de Arduino Due.

Aquí vemos la posible distribución de pines a utilizar dentro del shield de Arduino Due.

f) *PCB:* El esquemático del circuito fue diseñado en KiCad; para luego plasmarse en Proteus la creación del PCB en cuestión. Esto ocurrió debido a algunos problemas que nos surgieron al momento de obtener una visión 3D del circuito, que necesitábamos para el diseño de la carcasa desarrollado en el siguiente apartado.

En el apéndice A.5 y A.6se puede observar ambos esquemáticos completos; tanto en Proteus como en KiCad.

A continuación, se pueden observar el PCB y su versión 3D que obtuvimos a través de Proteus:

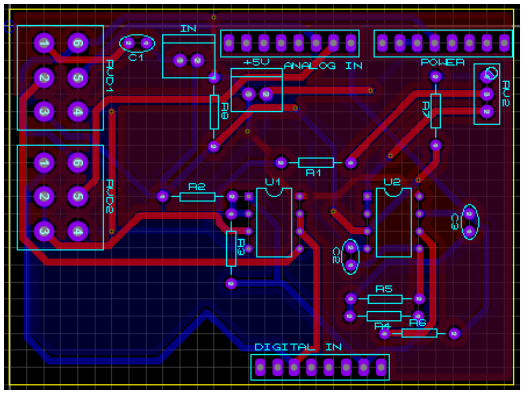


Fig. 40 PCB del Analizador de Espectro (AE)

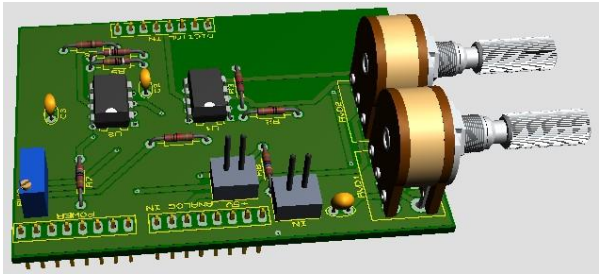


Fig. 41 Diseño 3D del circuito principal del AE.

D. Diseño de la carcasa

El diseño de la carcasa del Analizador de Espectros se realizó mediante un programa de uso gratuito FreeCAD.

El programa es utilizado ampliamente para la creación de objetos destinados a ser impresos con una Impresora 3D.

A continuación, se presenta la simulación de la carcasa completa de nuestro instrumento:

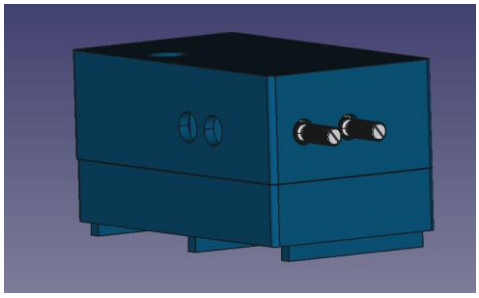


Fig. 42 Carcasa completa del AE - Vista posterior.

a) *Carcasa Superior:* Esta parte del instrumento, posee varios detalles entre ellos dos agujeros en la parte izquierda, que corresponden a los pines de entrada y salida de la señal a analizar en el Analizador de Espectro. Un orificio superior utilizado para variar el preset que se encuentra en el circuito 3D presentado. Por último, los dos círculos frontales en donde se ubicarían los potenciómetros.

Todos estos detalles se pueden observar en las figuras presentadas a continuación:

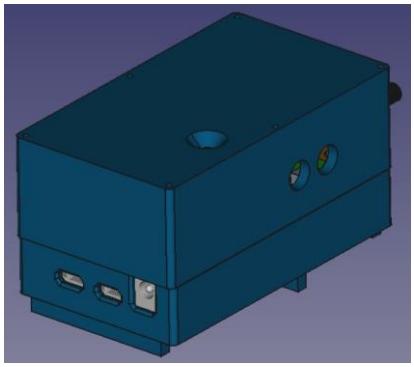


Fig. 43 Carcasa completa del AE - Vista posterior

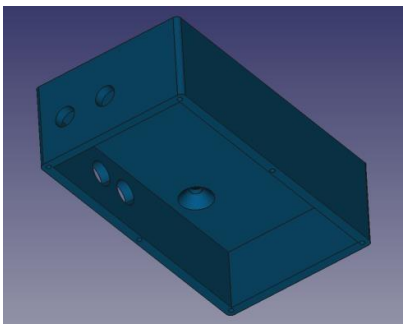


Fig. 44 Carcasa superior - Vista frontal.

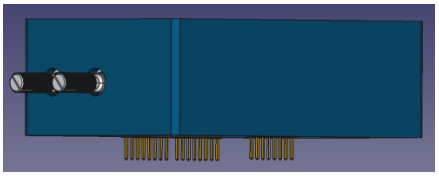


Fig. 45 Carcasa superior - Vista lateral.

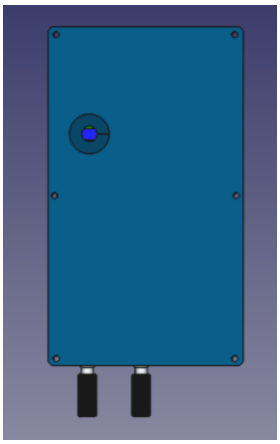


Fig. 46 Carcasa superior - Vista superior.

Además, también tiene orificios en los bordes de 1[mm] de radio que se ven claramente en la Fig. 46, utilizados para ajustar la presión de la carcasa.

b) *Carcasa Inferior*: La parte inferior de la carcasa es la principal del instrumento, ya que mediante un soporte fija a nuestro Arduino Due. Este posee ranuras horizontales destinados a la ventilación del instrumento.

En las siguientes figuras, se puede observar cómo se fijaría el microcontrolador a esta base y el diseño completo de la misma.

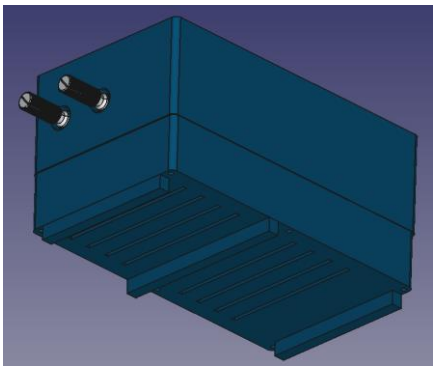


Fig. 47 Carcasa completa del AE - Vista Inferior

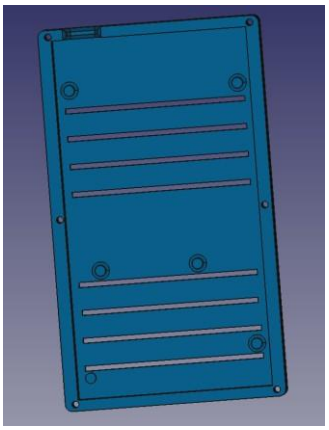


Fig. 48 Carcasa Inferior -Vista superior.

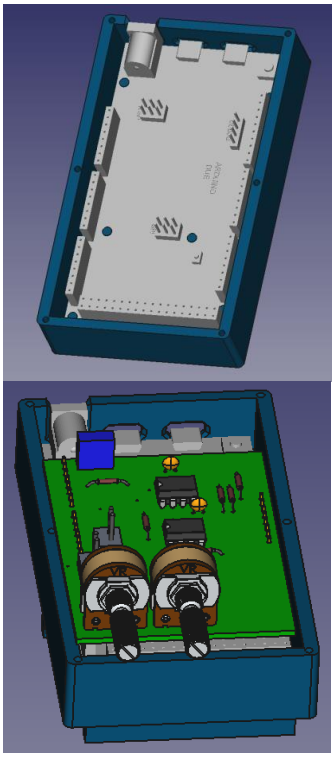


Fig. 49 Carcasa Inferior con un Arduino Due - Vista superior.

También posee orificios en la parte posterior del mismo, donde se ubicarían los microUSB y alimentación del Arduino Due.

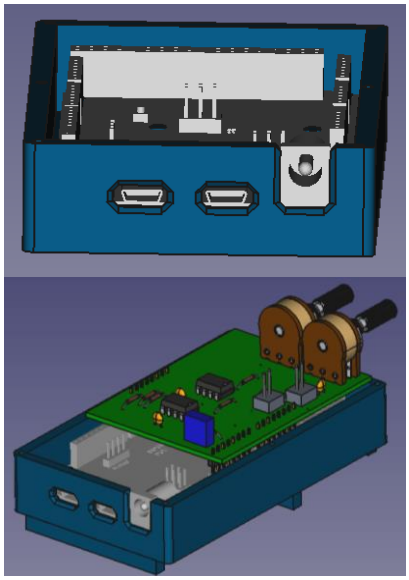


Fig. 50 Carcasa Inferior con Arduino due -Vista posterior.

E. Ensayo del circuito

Luego de las simulaciones y comprobar el correcto funcionamiento del instrumento, pasamos al armado del circuito en una protoboard.

A continuación, presentamos algunas imágenes tomadas durante la prueba del circuito del Analizador de Espectros. Entre estas se encuentran la fuente que alimenta el circuito y generador de señales utilizado para ingresar una señal.

También se observa un osciloscopio que se utilizó para verificar la correcta señal de entrada.

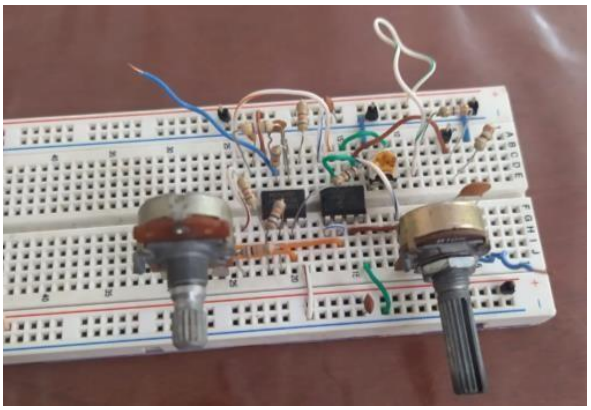


Fig. 51 Desdoblamiento del vector datos.

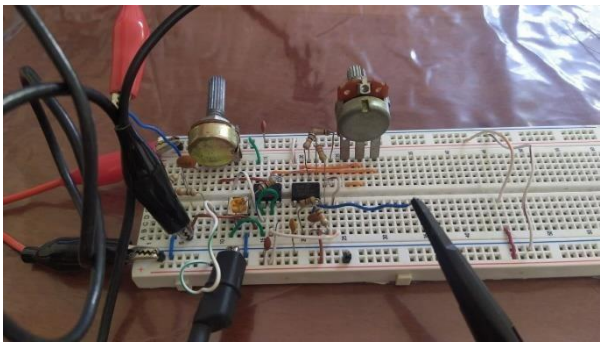


Fig. 52 Desdoblamiento del vector datos

a) *Instrumentos utilizados:* Para el ensayo se consiguieron los siguientes elementos de laboratorio.

- Osciloscopio: Tektronix TDS2014B
- Generador de Funciones: OWON AG1022
- Fuente de Alimentación: ATTEN TPR3003T-3C

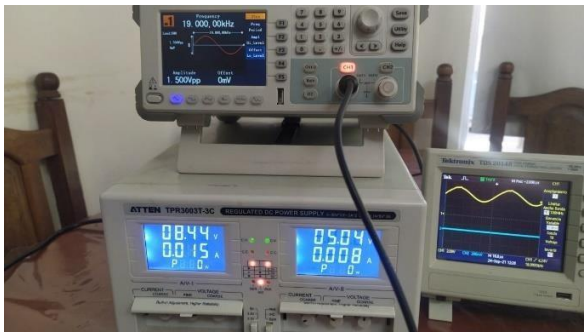


Fig. 53 Generador de funciones. Fuente de alimentación. Osciloscopio.

b) *Señal Senoidal de entrada:* En esta prueba se ingresa a nuestro circuito una señal senoidal de 19[KHz].

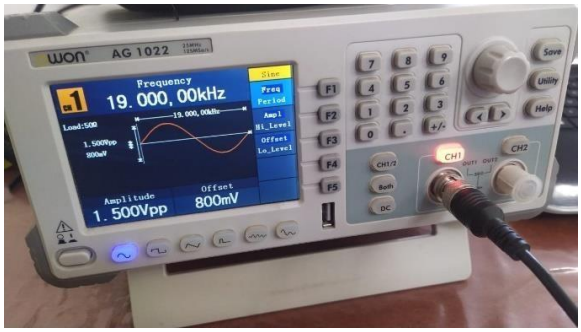


Fig. 54 Señal de entrada Senoidal de 19 [kHz]

Resultados:

```
10:53:06.214 -> 0.000000Hz 753.6386
10:53:06.214 -> 3125.000000Hz 345.6363
10:53:06.214 -> 6250.000000Hz 4.4832
10:53:06.214 -> 9375.000000Hz 7.7686
10:53:06.214 -> 12500.000000Hz 12.4270
10:53:06.214 -> 15625.000000Hz 1499.0552
10:53:06.214 -> 18750.000000Hz 3886.4622
10:53:06.214 -> 21875.000000Hz 1984.6310
10:53:06.214 -> 25000.000000Hz 74.4571
10:53:06.214 -> 28125.000000Hz 7.1692
10:53:06.214 -> 31250.000000Hz 10.0088
10:53:06.214 -> 34375.000000Hz 2.0985
10:53:06.214 -> 37500.000000Hz 10.4355
10:53:06.214 -> 40625.000000Hz 4.3815
10:53:06.214 -> 43750.000000Hz 17.7846
10:53:06.214 -> 46875.000000Hz 6.4387
10:53:06.254 ->
```

Fig. 55 Prueba con señal senoidal.

c) *Señal Cuadrada de entrada:* En esta prueba se ingresa a nuestro circuito una señal cuadrada de 19 [KHz].

Fig. 56 Señal de entrada cuadrada de 19 [kHz]

Resultados:

```
10:51:49.676 -> 0.000000Hz 785.4028
10:51:49.676 -> 3125.000000Hz 618.9616
10:51:49.676 -> 6250.000000Hz 1002.4152
10:51:49.676 -> 9375.000000Hz 577.0889
10:51:49.676 -> 12500.000000Hz 272.0759
10:51:49.676 -> 15625.000000Hz 2102.5047
10:51:49.676 -> 18750.000000Hz 4979.2779
10:51:49.676 -> 21875.000000Hz 2414.7472
10:51:49.676 -> 25000.000000Hz 158.7166
10:51:49.676 -> 28125.000000Hz 572.8392
10:51:49.676 -> 31250.000000Hz 778.8649
10:51:49.676 -> 34375.000000Hz 620.4381
10:51:49.676 -> 37500.000000Hz 193.4371
10:51:49.676 -> 40625.000000Hz 1002.9170
10:51:49.676 -> 43750.000000Hz 1662.9111
10:51:49.676 -> 46875.000000Hz 751.4846
10:51:49.676 ->
```

Fig. 57 Prueba con señal cuadrada

Resultados Generales de los ensayos:

Se comprobó el correcto funcionamiento del circuito.

Surgieron problemas con respecto al control del offset mediante el preset. Esto se soluciona colocando un seguidor de tensión a la tensión del offset para que esta no se modifique con el resto de los componentes del circuito.

4. Implementación del Modelo de Prueba

A. Programación General

La programación para esta instancia es básicamente sin cambios; pudiéndose observar las diferencias entre los apéndices A y B en los apartados respectivos, los que se encuentran debidamente comentados.

Se omitieron directamente los bloques sin cambios entre una sección y otra; por lo que las que quedaron solo en la sección A, y no figuran en la B, son estados finales para este proyecto. En la sección C del apéndice se pueden observar los diagramas de flujo que gobiernan tanto al Arduino como al MatLAB; el cual en esta instancia gráfica lo aportado por el Arduino a través de la comunicación serie.

B. Esquemático y PCB

Sin cambios desde el prototipo.

C. Diseño de la carcasa

Sin cambios desde el prototipo.

D. Ensayo del circuito

No se pudo ejercer el respectivo ensayo debido a falla del operacional y su imposibilidad para la fecha de entrega de restituirlo, dado que era importado.

5. CONCLUSIONES

Se logró crear un dispositivo que mida las componentes espectrales; el cual fue de implementación relativamente fácil, pudiendo ser un DIY (Do It Yourself , Hágalo usted mismo).

También, se logró aumentar el potencial de una placa Arduino Due, mediante el uso de una placa de extensión (Un shield); la cual, es una buena alternativa frente a costosos equipos, en general.

Dentro de las posibles mejoras podemos citar:

- La creación de una interfaz gráfica que amplié el potencial del dispositivo.
- La posibilidad de cambiar el tipo de ventana que se utiliza.
- La posibilidad de centrar el espectro en la posición deseada.
- La posibilidad de convertir la amplitud a escala en [db]
- Y la de mostrar información adicional mediante cálculos automáticos.

RECONOCIMIENTOS

Los autores, agradecen los consejos, ideas y contribuciones de:

- Esp. Ing. Facundo Cuestas
- Bioing. José Miguel Pentácolo
- Dr. Fabio Miguel Vincitorio

para la realización de este trabajo. En nombre del equipo expresamos nuestro profundo agradecimiento.

REFERENCIAS

Consideraciones Generales

[1] https://es.wikipedia.org/wiki/Analizador_de_espectro

[2] https://ingenieriaelectronica.org/analizador-de-espectros-definicion-tipos-y-caracteristicas/

[3]http://files.owon.com.cn/specifications/OWON%20XSA1000%20Series%20Spectrum%20Analyzer%20technical%20spec.s.pdf

Marco Teórico:

Atenuador:

[4]http://tutorialesdeelectronicabasica.blogspot.com/2019/03/atenuador-bridged-t-tutorial-de.html

[5] http://xq2dwo.blogspot.com/2016/04/atenuador-de-rf.html/

Filtros:

[6] https://www.electronics-tutorials.ws/filter/filter_5.html

Muestreo y Digitalización:

[7] MILONE, Diego H. y Otros.; "Introducción a las señales y sistemas discretos" Version Digital, Argentina; 2009; Editorial UNER

Transformada Rapida de Fourier:

[8] BANKS, Stephen P.; "Signal Processing, Image Processing and Pattern Recognition"; Great Britain; 1990; Editorial Universidad de Cambridge.

[9] BRIGHAM, Oran E.; "The Fast Fourier Transform and its Application"; U.S.A.; 1988; Editorial Prentice Hall S.A..

[10] BURRNS, Sydney C. y otros; "Ejercicios de Tratamiento de la Señal Utilizando MatLab V.4., Un Enfoque Práctico"; España; 1998; Editorial Prentice Hall S.A..

[11] JACKSON, Leland B.; "Signal, Systems and Transforms"; U.S.A.; 1988; Editorial Addison Wesley S.A..

[12] PAPOULIS, Athanasios; "Sistemas Digitales y Analógicos, Transformadas de Fourier, Estimación Espectral"; España; 1978;Editorial Marcombo Boixareu Editores

[13] RAMIREZ CORTES Juan Manuel y Otros (marzo-abril 1998). "El algoritmo de la transformada rápida de Fourier y sú controvertido origen". Revista Ciencia y desarrollo, N°139, Volumen XXIV, 70-77

[14] Chapter 12: The Fast Fourier Transform. https://www.dspguide.com/ch12/

[15] Digilentic - Lab 7b: Digital Spectrum Analyzer. https://reference.digilentinc.com/learn/courses/unit-7- lab7b/start

Ventanas:

[16] https://es.wikipedia.org/wiki/Ventana_(funci%C3%B3n)

Implementación del Prototipo

[17] https://www.hwlibre.com/arduinodue/#Que_es_Arduino_Due

[18] http://manueldelgadocrespo.blogspot.com/p/arduino-due.html

[19] Librería FFT: https://www.arduinolibraries.info/libraries/arduino-fft

[20] Librerias TIMER: https://platformio.org/lib/show/11467/SAMDUE_TimerInterrupt/installation

[21] Diseño de la carcasa mediante FreeCAD: http://diwo.bq.com/course/curso-de-introduccion-a- freecad/

APÉNDICES

ANALIZADOR DE ESPECTRO

Programación. Esquemático

Facultad Regional Paraná, Universidad Tecnológica Nacional, Argentina.

```
1   /*
2       FUNCION EN C++ DE LA FFT
3       Catedra: MEDIDAS ELECTRONICAS II
4       Docentes: C. Cappelletti, M. F. Krenz
5       Grupo N°: 3
6       Alumnos: A. A. Baldini, N. H. Gimenez, T. A. Pentacolo, L. D.
        Rispoli.
7   */
8
9   //FFT_____________________________________________________________
10
11  void FFT(int16_t* bufferRe, int16_t* bufferIm)
12  {
13  /*
14    En este ejemplo estandarizado, el programa, mediante loops,
      Calcula la FFT.
15  */
16
17  // Declaraciones: Variables de control de loop y tamaño de vectores___
18   int16_t bl;        // Tamaño actual de la mariposa.
19   int16_t p;         // Tamaño actual de la mariposa dividido por dos.
20   int16_t k;         // Nivel actual.
21   int16_t m;         // Contador de coeficiente.
22   int16_t i;         // Coeficientes de control para el primer punto.
23   int16_t j;         // Coeficientes de control para el segundo punto.
24   int16_t wRe;       // Coeficiente de giro actual (Parte Real).
25   int16_t wIm;       // Coeficiente de giro actual (Parte Imaginaria).
26   int16_t temp;
27
28   bl = FFT_SIZE;     // Tamaño actual de la mariposa.
29   p = FFT_SIZE >> 1; // Tamaño actual de la mariposa dividido por dos.
30   k = 0;             // Numero de nivel.
31
32  // loop_____________________________________________________________
33   while (p > 0)
34    {
35
36    for (m=0; m<p; m++)    // Loop sub-DFT.
37     {
38         j = m << k;       // Coeficientes de control.
39
40         wRe = twiddleCoefficients[(j + FFT_SIZE / 4)];
41         wIm = -twiddleCoefficients[j];
42
43         for (i=m; i<FFT_SIZE; I += bl)   // Loop calculo de mariposa.
44          {
45              j = i + p;  // i - Coeficientes de control para el
                primer punto.
46                          // j - Coeficientes de control para el
                            segundo punto.
47
48              FFT2(bufferRe+i, bufferIm+i, bufferRe+j, bufferIm+j,
                wRe, wIm); // Operación mariposa.
49          }
```

```
50        }
51
52      k++;
53      p >>= 1;
54      bl >>= 1;
55
56      }
57
58    // Ordena el dominio frecuencia_____________________________________________
59     for (i=1; i<(FFT_SIZE - 1); i++)
60      {
61
62          j = bitPermutationTable[i];
63
64          if (j <= i)
65           {
66              continue;
67            }
68
69          temp = bufferRe[i];
70          bufferRe[i] = bufferRe[j];
71          bufferRe[j] = temp;
72
73          temp = bufferIm[i];
74          bufferIm[i] = bufferIm[j];
75          bufferIm[j] = temp;
76      }
77
78    }
79
80    //FFT2: Operación Mariposa_____________________________________________
81
82    void FFT2(int16_t *ReA, int16_t *ImA, int16_t *ReB, int16_t *ImB,
      int16_t Wr, int16_t Wi)
83    {
84    // Declaraciones_____________________________________________________
85     int w0, w1, w2, w3, w6;
86
87    /*
88      Escala para Q1.15. La notación Q, como la define Texas Instruments,
89      consiste en la letra Q seguida de un par de números m . n , donde m
90      es el número de bits utilizados para la parte entera del valor y n
91      es el número de bits fraccionarios.
92    */
93
94     w0 = *ReA >> 1;
95     w1 = *ImA >> 1;
96     w2 = *ReB >> 1;
97     w3 = *ImB >> 1;
98
99    // Camino suma_____________________________________________________
100
101   //Guarda el resultado real. w6 = w0 + w2;
102    *ReA = (int16_t) (w0 + w2);
```

```
103
104   //Guarda el resultado imaginario. w7 = w1 + w3;
105    *ImA = (int16_t) (w1 + w3);
106
107   // Camino resta_____________________________________________________________
108
109    w0 = w0 - w2;
110    w1 = w1 - w3;
111
112    w2 = w0 * Wr;           // Multiplicar por coeficiente.
113    w6 = w1 * Wi;
114    w2 = (w2 - w6) >> 15;
115
116    *ReB = (int16_t) w2 ;  // Guardar resultado W3 aquí (Parte Real).
       w2 = w2 >> 15;
117
118    w2 = w0 * Wi;           // Multiplicar por coeficiente.
119    w6 = w1 * Wr;
120    w2 = (w2 + w6) >> 15;
121
122    *ImB = (int16_t) w2;   // Guardar resultado W3 aquí (Parte
       Imaginaria).
123   }
```

```
1  /*
2      MODELO DE PRUEBA de ANALIZADOR DE ESPECTRO
3      Catedra: MEDIDAS ELECTRONICAS II
4      Docentes: C. Cappelletti, M. F. Krenz
5      Grupo N°: 3
6      Alumnos: A. A. Baldini, N. H. Gimenez, T. A. Pentacolo, L. D.
       Rispoli.
7  */
8
9  /*
10   En este ejemplo, el programa (mediante un loop) habilita la
     conversión de datos por el ADC; para luego, aplicar el filtro
     Hamming a los datos.
11   Una vez limpio esto, se procede al cálculo de la respectiva FFT y
     preparamos el espectro para solo ver el Diagrama de Amplitud.
12 Finalmente se procede a mostrar el mismo.
13 */
14
15 //Declaraciones_____________________________________________________
16
17 #include <arduinoFFT.h>
18 #include "SAMDUETimerInterrupt.h"
19
20 uint16_t Timer_Index = 0;            // Identificador del Timer.
21
22 const uint16_t samples = 32;         // Cantidad de muestras a
   analizar.
23 const double Fm = 1E5;               // Frecuencia de muestreo.
24
25 arduinoFFT FFT = arduinoFFT();       // Creo el objeto FFT.
26
27 double vReal[samples];               // Vector que contendrá los
   valores adquiridos en el ADC.
28 double vAnalog[samples];             // Vector que contendrá los
   valores reales calculados en la FFT.
29 double vImag[samples];               // Vector que contendrá los
   valores imaginarios.
30
31 //TimerHandler1_____________________________________________________
32
33 void TimerHandler1()
34 /*
35  *   TimerHandle1();
36  *   En este método se produce la rotación de los datos adquiridos
37  *   y se añade al inicio del vector el nuevo dato adquirido.
38  */
39 {
40   PIOB->PIO_ODSR = 1<<25;               // Escribo un 1 en el pin 25
     del puerto B.
41
42   for (int i = samples; i > 0; i--)     // Rotamos el vector.
43   {
44     vReal[i] = vReal[i-1];
45   }
```

```
46
47      vReal[0] = analogRead(A7);              // Leemos el nuevo dato.
48
49      PIOB->PIO_ODSR = 0;                     // Escribo un 0 en el pin 25
        del puerto B.
50    }
51
52    //setup_____________________________________________________________________
53
54    void setup() {
55      PIOB->PIO_PER = (1<<25);
        // Configuramos el puerto B para controlarlo por PIO.
56      PIOB->PIO_OER = (1<<25);
        // Establecemos el pin 25 del puerto B como salida.
57
58      analogReadResolution(10);
        // Establecemos la resolución del ADC en 10 bits.
59
60      for (int i = 0; i < samples; i++)
        // Inicializo los vectores en 0.
61      {
62        vReal[i] = 0;
63        vAnalog[i] = 0;
64      vImag[i] = 0;
65      }
66
67      DueTimerInterrupt dueTimerInterrupt = DueTimer.getAvailable();
        // Creamos un objeto DueTimerInterrupt (administra interrupción de
        Timer).
68      dueTimerInterrupt.attachInterruptInterval(10, TimerHandler1);
        // Establecemos una interrupción cada 10 [us], obteniendo una
        frecuencia de 100[kHz].
69      Timer_Index = dueTimerInterrupt.getTimerNumber();
        // Obtenemos el identificador del Timer.
70
71      Serial.begin(115200);
        // Inicializamos el puerto serie.
72      while(!Serial);
        // Esperamos a que finalice la inicialización.
73      Serial.println("Ready");
74      adc_start(ADC);
        // Iniciamos la conversión del ADC.
75    }
76
77    //loop_____________________________________________________________________
78
79    void loop() {
80      DueTimerPtr[Timer_Index].stopTimer();
        // Pausamos el Timer.
81      for (int i = 0; i < samples; i++)
82      {
83        vAnalog[i] = vReal[i] - 512;
          // Copiamos los valores adquiridos hasta el momento y eliminamos
          la componente de continua.
```

```
84        vImag[i] = 0;
          // Limpiamos los valores imaginarios.
85      }
86      DueTimerPtr[Timer_Index].startTimer();
        // Activamos nuevamente el Timer.
87      FFT.Windowing(vAnalog, samples, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
        // Aplicamos la ventana Hamming al los datos adquiridos.
88      FFT.Compute(vAnalog, vImag, samples, FFT_FORWARD);
        // Aplicamos la FFT a los datos adquiridos.
89      FFT.ComplexToMagnitude(vAnalog, vImag, samples);
        // Obtenemos la amplitud del espectro calculado.
90      PrintVector(vAnalog, (samples >> 1));
        // Transmitimos la magnitud del espectro calculado.
91      delay(10);
        // Esperamos 10 [ms].
92    }
93
94    //PrintVector_____________________________________________________________
95
96    void PrintVector(double *vData, uint16_t bufferSize)
97    /*
98     * PrintVector(double *vData, uint16_t bufferSize);
99     *  double *vData: Es el vector que apunta a los datos a imprimir.
100    *  uint16_t bufferSize: Cantidad de datos contenidos en el vector.
101    *
102    * Este método calcula el valor de frecuencia correspondiente a cada
       elemento
103    * del vector, y transmite tanto la frecuencia calculada como su
       valor de
104    * espectro.
105    */
106   {
107     for (uint16_t i = 0; i < bufferSize; i++)
        // Por cada elemento.
108     {
109       double abscissa;
110       abscissa = ((i * 1.0 * Fm) / samples);
          // Obtenemos el valor de frecuencia correspondiente.
111       Serial.print(abscissa, 6);
112       Serial.print("Hz");
113       Serial.print(" ");
114       Serial.println(vData[i], 4);
          // Transmitimos la información.
115     }
116     Serial.println();
117   }
```

```
1   /*
2
3       FFT libray
4       Copyright (C) 2010 Didier Longueville
5       Copyright (C) 2014 Enrique Condes
6
7       This program is free software: you can redistribute it and/or
        modify
8       it under the terms of the GNU General Public License as
        published by
9       the Free Software Foundation, either version 3 of the License, or
10      (at your option) any later version.
11
12      This program is distributed in the hope that it will be useful,
13      but WITHOUT ANY WARRANTY; without even the implied warranty of
14      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15      GNU General Public License for more details.
16
17      You should have received a copy of the GNU General Public License
18      along with this program.  If not, see
        <http://www.gnu.org/licenses/>.
19
20  */
21
22  #ifndef arduinoFFT_h /* Prevent loading library twice */
23  #define arduinoFFT_h
24  #ifdef ARDUINO
25      #if ARDUINO >= 100
26          #include "Arduino.h"
27      #else
28          #include "WProgram.h" /* This is where the standard Arduino
            code lies */
29      #endif
30  #else
31      #include <stdlib.h>
32      #include <stdio.h>
33      #ifdef  __AVR__
34          #include <avr/io.h>
35          #include <avr/pgmspace.h>
36      #endif
37      #include <math.h>
38      #include "defs.h"
39      #include "types.h"
40  #endif
41
42  #define FFT_LIB_REV 0x14
43  /* Custom constants */
44  #define FFT_FORWARD 0x01
45  #define FFT_REVERSE 0x00
46
47  /* Windowing type */
48  #define FFT_WIN_TYP_RECTANGLE 0x00 /* rectangle (Box car) */
49  #define FFT_WIN_TYP_HAMMING 0x01 /* hamming */
50  #define FFT_WIN_TYP_HANN 0x02 /* hann */
```

```
51  #define FFT_WIN_TYP_TRIANGLE 0x03 /* triangle (Bartlett) */
52  #define FFT_WIN_TYP_NUTTALL 0x04 /* nuttall */
53  #define FFT_WIN_TYP_BLACKMAN 0x05 /* blackman */
54  #define FFT_WIN_TYP_BLACKMAN_NUTTALL 0x06 /* blackman nuttall */
55  #define FFT_WIN_TYP_BLACKMAN_HARRIS 0x07 /* blackman harris*/
56  #define FFT_WIN_TYP_FLT_TOP 0x08 /* flat top */
57  #define FFT_WIN_TYP_WELCH 0x09 /* welch */
58  /*Mathematial constants*/
59  #define twoPi 6.28318531
60  #define fourPi 12.56637061
61  #define sixPi 18.84955593
62
63  #ifdef   AVR___
64      static const double _c1[]PROGMEM = {0.0000000000, 0.7071067812,
    0.9238795325, 0.9807852804,
65
    0.9951847267, 0.9987954562, 0.9996988187, 0.9999247018,
66
    0.9999811753, 0.9999952938, 0.9999988235, 0.9999997059,
67
    0.9999999265, 0.9999999816, 0.9999999954, 0.9999999989,
68
    0.9999999997};
69      static const double _c2[]PROGMEM = {1.0000000000, 0.7071067812,
    0.3826834324, 0.1950903220,
70
    0.0980171403, 0.0490676743, 0.0245412285, 0.0122715383,
71
    0.0061358846, 0.0030679568, 0.0015339802, 0.0007669903,
72
    0.0003834952, 0.0001917476, 0.0000958738, 0.0000479369,
73
    0.0000239684};
74  #endif
75  class arduinoFFT {
76  public:
77      /* Constructor */
78      arduinoFFT(void);
79      arduinoFFT(double *vReal, double *vImag, uint16_t samples,
    double samplingFrequency);
80      /* Destructor */
81      ~arduinoFFT(void);
82      /* Functions */
83      uint8_t Revision(void);
84      uint8_t Exponent(uint16_t value);
85
86      void ComplexToMagnitude(double *vReal, double *vImag, uint16_t
    samples);
87      void Compute(double *vReal, double *vImag, uint16_t samples,
    uint8_t dir);
88      void Compute(double *vReal, double *vImag, uint16_t samples,
    uint8_t power, uint8_t dir);
89      void DCRemoval(double *vData, uint16_t samples);
90      double MajorPeak(double *vD, uint16_t samples, double
```

```
        samplingFrequency);
 91     void MajorPeak(double *vD, uint16_t samples, double
        samplingFrequency, double *f, double *v);
 92     void Windowing(double *vData, uint16_t samples, uint8_t
        windowType, uint8_t dir);
 93
 94     void ComplexToMagnitude();
 95     void Compute(uint8_t dir);
 96     void DCRemoval();
 97     double MajorPeak();
 98     void MajorPeak(double *f, double *v);
 99     void Windowing(uint8_t windowType, uint8_t dir);
100
101 private:
102     /* Variables */
103     uint16_t _samples;
104     double _samplingFrequency;
105     double *_vReal;
106     double *_vImag;
107     uint8_t _power;
108     /* Functions */
109     void Swap(double *x, double *y);
110 };
111
112 #endif
113
```

```
1   /*
2
3       FFT libray
4       Copyright (C) 2010 Didier Longueville
5       Copyright (C) 2014 Enrique Condes
6
7       This program is free software: you can redistribute it and/or
        modify
8       it under the terms of the GNU General Public License as
        published by
9       the Free Software Foundation, either version 3 of the License, or
10      (at your option) any later version.
11
12      This program is distributed in the hope that it will be useful,
13      but WITHOUT ANY WARRANTY; without even the implied warranty of
14      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15      GNU General Public License for more details.
16
17      You should have received a copy of the GNU General Public License
18      along with this program.  If not, see
        <http://www.gnu.org/licenses/>.
19
20  */
21
22  #include "arduinoFFT.h"
23
24  arduinoFFT::arduinoFFT(void)
25  { // Constructor
26      #warning("This method is deprecated and may be removed on future
        revisions.")
27  }
28
29  arduinoFFT::arduinoFFT(double *vReal, double *vImag, uint16_t
    samples, double samplingFrequency)
30  {// Constructor
31      this->_vReal = vReal;
32      this->_vImag = vImag;
33      this->_samples = samples;
34      this->_samplingFrequency = samplingFrequency;
35  this->_power = Exponent(samples); 36
    }
37
38  arduinoFFT::~arduinoFFT(void)
39  {
40  // Destructor
41  }
42
43  uint8_t arduinoFFT::Revision(void)
44  {
45      return(FFT_LIB_REV);
46  }
47
48  void arduinoFFT::Compute(double *vReal, double *vImag, uint16_t
    samples, uint8_t dir)
```

```
49  {
50      #warning("This method is deprecated and may be removed on future
        revisions.")
51  Compute(vReal, vImag, samples, Exponent(samples), dir); 52}
53
54  void arduinoFFT::Compute(uint8_t dir)
55  {// Computes in-place complex-to-complex FFT /
56      // Reverse bits /
57      uint16_t j = 0;
58      for (uint16_t i = 0; i < (this->_samples - 1); i++) {
59          if (i < j) {
60              Swap(&this->_vReal[i], &this->_vReal[j]);
61              if(dir==FFT_REVERSE)
62                  Swap(&this->_vImag[i], &this->_vImag[j]);
63          }
64          uint16_t k = (this->_samples >> 1);
65          while (k <= j) {
66              j -= k;
67              k >>= 1;
68          }
69          j += k;
70      }
71      // Compute the FFT  /
72  #ifdef   AVR___
73      uint8_t index = 0;
74  #endif
75      double c1 = -1.0;
76      double c2 = 0.0;
77      uint16_t l2 = 1;
78      for (uint8_t l = 0; (l < this->_power); l++) {
79          uint16_t l1 = l2;
80          l2 <<= 1;
81          double u1 = 1.0;
82          double u2 = 0.0;
83          for (j = 0; j < l1; j++) {
84               for (uint16_t i = j; i < this->_samples; i += l2) {
85                      uint16_t i1 = i + l1;
86                      double t1 = u1 * this->_vReal[i1] - u2 *
                        this->_vImag[i1];
87                      double t2 = u1 * this->_vImag[i1] + u2 *
                        this->_vReal[i1];
88                      this->_vReal[i1] = this->_vReal[i] - t1;
89                      this->_vImag[i1] = this->_vImag[i] - t2;
90                      this->_vReal[i] += t1;
91               this->_vImag[i] += t2; 92
                 }
93               double z = ((u1 * c1) - (u2 * c2));
94               u2 = ((u1 * c2) + (u2 * c1));
95               u1 = z;
96          }
97  #ifdef   AVR___
98          c2 = pgm_read_float_near(&(_c2[index]));
99          c1 = pgm_read_float_near(&(_c1[index]));
```

```
100             index++;
101     #else
102             c2 = sqrt((1.0 - c1) / 2.0);
103             c1 = sqrt((1.0 + c1) / 2.0);
104     #endif
105             if (dir == FFT_FORWARD) {
106                 c2 = -c2;
107             }
108         }
109         // Scaling for reverse transform /
110         if (dir != FFT_FORWARD) {
111             for (uint16_t i = 0; i < this->_samples; i++) {
112                  this->_vReal[i] /= this->_samples;
113                  this->_vImag[i] /= this->_samples;
114             }
115         }
116     }
117
118     void arduinoFFT::Compute(double *vReal, double *vImag, uint16_t
        samples, uint8_t power, uint8_t dir)
119     {   // Computes in-place complex-to-complex FFT
120         // Reverse bits
121         #warning("This method is deprecated and may be removed on future
            revisions.")
122         uint16_t j = 0;
123         for (uint16_t i = 0; i < (samples - 1); i++) {
124             if (i < j) {
125                 Swap(&vReal[i], &vReal[j]);
126                 if(dir==FFT_REVERSE)
127                     Swap(&vImag[i], &vImag[j]);
128             }
129             uint16_t k = (samples >> 1);
130             while (k <= j) {
131                 j -= k;
132                 k >>= 1;
133             }
134             j += k;
135         }
136         // Compute the FFT
137     #ifdef   AVR___
138         uint8_t index = 0;
139     #endif
140         double c1 = -1.0;
141         double c2 = 0.0;
142         uint16_t l2 = 1;
143         for (uint8_t l = 0; (l < power); l++) {
144             uint16_t l1 = l2;
145             l2 <<= 1;
146             double u1 = 1.0;
147             double u2 = 0.0;
148             for (j = 0; j < l1; j++) {
149                  for (uint16_t i = j; i < samples; i += l2) {
150                        uint16_t i1 = i + l1;
151                        double t1 = u1 * vReal[i1] - u2 * vImag[i1];
```

```
152                     double t2 = u1 * vImag[i1] + u2 * vReal[i1];
153                     vReal[i1] = vReal[i] - t1;
154                     vImag[i1] = vImag[i] - t2;
155                     vReal[i] += t1;
156                     vImag[i] += t2;
157                 }
158                 double z = ((u1 * c1) - (u2 * c2));
159                 u2 = ((u1 * c2) + (u2 * c1));
160                 u1 = z;
161             }
162 #ifdef   AVR___
163             c2 = pgm_read_float_near(&(_c2[index]));
164             c1 = pgm_read_float_near(&(_c1[index]));
165             index++;
166 #else
167             c2 = sqrt((1.0 - c1) / 2.0);
168             c1 = sqrt((1.0 + c1) / 2.0);
169 #endif
170             if (dir == FFT_FORWARD) {
171                 c2 = -c2;
172             }
173         }
174         // Scaling for reverse transform
175         if (dir != FFT_FORWARD) {
176             for (uint16_t i = 0; i < samples; i++) {
177                 vReal[i] /= samples;
178                 vImag[i] /= samples;
179             }
180         }
181 }
182 
183 void arduinoFFT::ComplexToMagnitude()
184 { // vM is half the size of vReal and vImag
185     for (uint16_t i = 0; i < this->_samples; i++) {
186         this->_vReal[i] = sqrt(sq(this->_vReal[i]) +
        sq(this->_vImag[i]));
187     }
188 }
189 
190 void arduinoFFT::ComplexToMagnitude(double *vReal, double *vImag,
    uint16_t samples)
191 {   // vM is half the size of vReal and vImag
192     #warning("This method is deprecated and may be removed on future
    revisions.")
193     for (uint16_t i = 0; i < samples; i++) {
194         vReal[i] = sqrt(sq(vReal[i]) + sq(vImag[i]));
195     }
196 }
197 
198 void arduinoFFT::DCRemoval()
199 {
200     // calculate the mean of vData
201     double mean = 0;
202     for (uint16_t i = 0; i < this->_samples; i++)
```

```
203         {
204             mean += this->_vReal[i];
205         }
206         mean /= this->_samples;
207         // Subtract the mean from vData
208         for (uint16_t i = 0; i < this-> samples; i++)
209         {
210             this->_vReal[i] -= mean;
211         }
212     }
213
214     void arduinoFFT::DCRemoval(double *vData, uint16_t samples)
215     {
216         // calculate the mean of vData
217         #warning("This method is deprecated and may be removed on future
            revisions.")
218         double mean = 0;
219         for (uint16_t i = 0; i < samples; i++)
220         {
221             mean += vData[i];
222         }
223         mean /= samples;
224         // Subtract the mean from vData
225         for (uint16_t i = 0; i < samples; i++)
226         {
227             vData[i] -= mean;
228         }
229     }
230
231     void arduinoFFT::Windowing(uint8_t windowType, uint8_t dir)
232     {// Weighing factors are computed once before multiple use of FFT
233     // The weighing function is symetric; half the weighs are recorded
234         double samplesMinusOne = (double(this->_samples) - 1.0);
235         for (uint16_t i = 0; i < (this->_samples >> 1); i++) {
236             double indexMinusOne = double(i);
237             double ratio = (indexMinusOne / samplesMinusOne);
238             double weighingFactor = 1.0;
239             // Compute and record weighting factor
240             switch (windowType) {
241             case FFT_WIN_TYP_RECTANGLE: // rectangle (box car)
242                 weighingFactor = 1.0;
243                 break;
244             case FFT_WIN_TYP_HAMMING: // hamming
245                 weighingFactor = 0.54 - (0.46 * cos(twoPi * ratio));
246                 break;
247             case FFT_WIN_TYP_HANN: // hann
248                 weighingFactor = 0.54 * (1.0 - cos(twoPi * ratio));
249                 break;
250             case FFT_WIN_TYP_TRIANGLE: // triangle (Bartlett)
251                 #if defined(ESP8266) || defined(ESP32)
252                 weighingFactor = 1.0 - ((2.0 * fabs(indexMinusOne -
                    (samplesMinusOne / 2.0))) / samplesMinusOne);
253                 #else
254                 weighingFactor = 1.0 - ((2.0 * abs(indexMinusOne -
```

```
                (samplesMinusOne / 2.0))) / samplesMinusOne);
255             #endif
256             break;
257         case FFT_WIN_TYP_NUTTALL: // nuttall
258             weighingFactor = 0.355768 - (0.487396 * (cos(twoPi *
                ratio))) + (0.144232 * (cos(fourPi * ratio))) -
                (0.012604 * (cos(sixPi * ratio)));
259             break;
260         case FFT_WIN_TYP_BLACKMAN: // blackman
261             weighingFactor = 0.42323 - (0.49755 * (cos(twoPi *
                ratio))) + (0.07922 * (cos(fourPi * ratio)));
262             break;
263         case FFT_WIN_TYP_BLACKMAN_NUTTALL: // blackman nuttall
264             weighingFactor = 0.3635819 - (0.4891775 * (cos(twoPi *
                ratio))) + (0.1365995 * (cos(fourPi * ratio))) -
                (0.0106411 * (cos(sixPi * ratio)));
265             break;
266         case FFT_WIN_TYP_BLACKMAN_HARRIS: // blackman harris
267             weighingFactor = 0.35875 - (0.48829 * (cos(twoPi *
                ratio))) + (0.14128 * (cos(fourPi * ratio))) - (0.01168
                * (cos(sixPi * ratio)));
268             break;
269         case FFT_WIN_TYP_FLT_TOP: // flat top
270             weighingFactor = 0.2810639 - (0.5208972 * cos(twoPi *
                ratio)) + (0.1980399 * cos(fourPi * ratio));
271             break;
272         case FFT_WIN_TYP_WELCH: // welch
273             weighingFactor = 1.0 - sq((indexMinusOne -
                samplesMinusOne / 2.0) / (samplesMinusOne / 2.0));
274             break;
275         }
276         if (dir == FFT_FORWARD) {
277             this->_vReal[i] *= weighingFactor;
278             this->_vReal[this->_samples - (i + 1)] *= weighingFactor;
279         }
280         else {
281             this->_vReal[i] /= weighingFactor;
282             this->_vReal[this->_samples - (i + 1)] /= weighingFactor;
283         }
284     }
285 }
286
287
288 void arduinoFFT::Windowing(double *vData, uint16_t samples, uint8_t
    windowType, uint8_t dir)
289 {// Weighing factors are computed once before multiple use of FFT
290 // The weighing function is symetric; half the weighs are recorded
291     #warning("This method is deprecated and may be removed on future
        revisions.")
292     double samplesMinusOne = (double(samples) - 1.0);
293     for (uint16_t i = 0; i < (samples >> 1); i++) {
294         double indexMinusOne = double(i);
295         double ratio = (indexMinusOne / samplesMinusOne);
296         double weighingFactor = 1.0;
```

```
297             // Compute and record weighting factor
298             switch (windowType) {
299             case FFT_WIN_TYP_RECTANGLE: // rectangle (box car)
300                 weighingFactor = 1.0;
301                 break;
302             case FFT_WIN_TYP_HAMMING: // hamming
303                 weighingFactor = 0.54 - (0.46 * cos(twoPi * ratio));
304                 break;
305             case FFT_WIN_TYP_HANN: // hann
306                 weighingFactor = 0.54 * (1.0 - cos(twoPi * ratio));
307                 break;
308             case FFT_WIN_TYP_TRIANGLE: // triangle (Bartlett)
309                 #if defined(ESP8266) || defined(ESP32)
310                 weighingFactor = 1.0 - ((2.0 * fabs(indexMinusOne -
                    (samplesMinusOne / 2.0))) / samplesMinusOne);
311                 #else
312                 weighingFactor = 1.0 - ((2.0 * abs(indexMinusOne -
                    (samplesMinusOne / 2.0))) / samplesMinusOne);
313                 #endif
314                 break;
315             case FFT_WIN_TYP_NUTTALL: // nuttall
316                 weighingFactor = 0.355768 - (0.487396 * (cos(twoPi *
                    ratio))) + (0.144232 * (cos(fourPi * ratio))) -
                    (0.012604 * (cos(sixPi * ratio)));
317                 break;
318             case FFT_WIN_TYP_BLACKMAN: // blackman
319                 weighingFactor = 0.42323 - (0.49755 * (cos(twoPi *
                    ratio))) + (0.07922 * (cos(fourPi * ratio)));
320                 break;
321             case FFT_WIN_TYP_BLACKMAN_NUTTALL: // blackman nuttall
322                 weighingFactor = 0.3635819 - (0.4891775 * (cos(twoPi *
                    ratio))) + (0.1365995 * (cos(fourPi * ratio))) -
                    (0.0106411 * (cos(sixPi * ratio)));
323                 break;
324             case FFT_WIN_TYP_BLACKMAN_HARRIS: // blackman harris
325                 weighingFactor = 0.35875 - (0.48829 * (cos(twoPi *
                    ratio))) + (0.14128 * (cos(fourPi * ratio))) - (0.01168
                    * (cos(sixPi * ratio)));
326                 break;
327             case FFT_WIN_TYP_FLT_TOP: // flat top
328                 weighingFactor = 0.2810639 - (0.5208972 * cos(twoPi *
                    ratio)) + (0.1980399 * cos(fourPi * ratio));
329                 break;
330             case FFT_WIN_TYP_WELCH: // welch
331                 weighingFactor = 1.0 - sq((indexMinusOne -
                    samplesMinusOne / 2.0) / (samplesMinusOne / 2.0));
332                 break;
333             }
334             if (dir == FFT_FORWARD) {
335                 vData[i] *= weighingFactor;
336                 vData[samples - (i + 1)] *= weighingFactor;
337             }
338             else {
339                 vData[i] /= weighingFactor;
```

```
340                 vData[samples - (i + 1)] /= weighingFactor;
341             }
342         }
343     }
344
345     double arduinoFFT::MajorPeak()
346     {
347         double maxY = 0;
348         uint16_t IndexOfMaxY = 0;
349         //If sampling_frequency = 2 * max_frequency in signal,
350         //value would be stored at position samples/2
351         for (uint16_t i = 1; i < ((this->_samples >> 1) + 1); i++) {
352             if ((this->_vReal[i-1] < this->_vReal[i]) &&
                (this->_vReal[i] > this->_vReal[i+1])) {
353                 if (this->_vReal[i] > maxY) {
354                     maxY = this->_vReal[i];
355                     IndexOfMaxY = i;
356                 }
357             }
358         }
359         double delta = 0.5 * ((this->_vReal[IndexOfMaxY-1] -
            this->_vReal[IndexOfMaxY+1]) / (this->_vReal[IndexOfMaxY-1] -
            (2.0 * this->_vReal[IndexOfMaxY]) + this->_vReal[IndexOfMaxY+1]));
360         double interpolatedX = ((IndexOfMaxY + delta)   *
            this->_samplingFrequency) / (this->_samples-1);
361         if(IndexOfMaxY==(this->_samples >> 1)) //To improve calculation
            on edge values
362             interpolatedX = ((IndexOfMaxY + delta)   *
                this->_samplingFrequency) / (this->_samples);
363         // returned value: interpolated frequency peak apex
364         return(interpolatedX);
365     }
366
367     void arduinoFFT::MajorPeak(double *f, double *v)
368     {
369         double maxY = 0;
370         uint16_t IndexOfMaxY = 0;
371         //If sampling_frequency = 2 * max_frequency in signal,
372         //value would be stored at position samples/2
373         for (uint16_t i = 1; i < ((this->_samples >> 1) + 1); i++) {
374             if ((this->_vReal[i - 1] < this->_vReal[i]) &&
                (this->_vReal[i] > this->_vReal[i + 1])) {
375                 if (this->_vReal[i] > maxY) {
376                     maxY = this->_vReal[i];
377                     IndexOfMaxY = i;
378                 }
379             }
380         }
381         double delta = 0.5 * ((this->_vReal[IndexOfMaxY - 1] -
            this->_vReal[IndexOfMaxY + 1]) / (this->_vReal[IndexOfMaxY - 1]
            - (2.0 * this->_vReal[IndexOfMaxY]) + this->_vReal[IndexOfMaxY +
            1]));
382         double interpolatedX = ((IndexOfMaxY + delta)   *
            this->_samplingFrequency) / (this->_samples - 1);
```

```
383         if (IndexOfMaxY == (this->_samples >> 1)) //To improve
            calculation on edge values
384             interpolatedX = ((IndexOfMaxY + delta)  *
                this->_samplingFrequency) / (this->_samples);
385         // returned value: interpolated frequency peak apex
386         *f = interpolatedX;
387         #if defined(ESP8266) || defined(ESP32)
388         *v = fabs(this->_vReal[IndexOfMaxY - 1] - (2.0 *
            this->_vReal[IndexOfMaxY]) + this->_vReal[IndexOfMaxY + 1]);
389         #else
390         *v = abs(this->_vReal[IndexOfMaxY - 1] - (2.0 *
            this->_vReal[IndexOfMaxY]) + this->_vReal[IndexOfMaxY + 1]);
391         #endif
392     }
393
394     double arduinoFFT::MajorPeak(double *vD, uint16_t samples, double
        samplingFrequency)
395     {
396         #warning("This method is deprecated and may be removed on future
            revisions.")
397         double maxY = 0;
398         uint16_t IndexOfMaxY = 0;
399         //If sampling_frequency = 2 * max_frequency in signal,
400         //value would be stored at position samples/2
401         for (uint16_t i = 1; i < ((samples >> 1) + 1); i++) {
402             if ((vD[i-1] < vD[i]) && (vD[i] > vD[i+1])) {
403                 if (vD[i] > maxY) {
404                     maxY = vD[i];
405                     IndexOfMaxY = i;
406                 }
407             }
408         }
409         double delta = 0.5 * ((vD[IndexOfMaxY-1] - vD[IndexOfMaxY+1]) /
            (vD[IndexOfMaxY-1] - (2.0 * vD[IndexOfMaxY]) +
            vD[IndexOfMaxY+1]));
410         double interpolatedX = ((IndexOfMaxY + delta)  *
            samplingFrequency) / (samples-1);
411         if(IndexOfMaxY==(samples >> 1)) //To improve calculation on edge
            values
412             interpolatedX = ((IndexOfMaxY + delta)  * samplingFrequency)
                / (samples);
413         // returned value: interpolated frequency peak apex
414         return(interpolatedX);
415     }
416
417     void arduinoFFT::MajorPeak(double *vD, uint16_t samples, double
        samplingFrequency, double *f, double *v)
418     {
419         #warning("This method is deprecated and may be removed on future
            revisions.")
420         double maxY = 0;
421         uint16_t IndexOfMaxY = 0;
422         //If sampling_frequency = 2 * max_frequency in signal,
423         //value would be stored at position samples/2
```

```
424         for (uint16_t i = 1; i < ((samples >> 1) + 1); i++) {
425             if ((vD[i - 1] < vD[i]) && (vD[i] > vD[i + 1])) {
426                 if (vD[i] > maxY) {
427                     maxY = vD[i];
428                     IndexOfMaxY = i;
429                 }
430             }
431         }
432         double delta = 0.5 * ((vD[IndexOfMaxY - 1] - vD[IndexOfMaxY +
            1]) / (vD[IndexOfMaxY - 1] - (2.0 * vD[IndexOfMaxY]) +
            vD[IndexOfMaxY + 1]));
433         double interpolatedX = ((IndexOfMaxY + delta)   *
            samplingFrequency) / (samples - 1);
434         //double popo =
435         if (IndexOfMaxY == (samples >> 1)) //To improve calculation on
            edge values
436             interpolatedX = ((IndexOfMaxY + delta)   * samplingFrequency)
                / (samples);
437         // returned value: interpolated frequency peak apex
438         *f = interpolatedX;
439         #if defined(ESP8266) || defined(ESP32)
440         *v = fabs(vD[IndexOfMaxY - 1] - (2.0 * vD[IndexOfMaxY]) +
            vD[IndexOfMaxY + 1]);
441         #else
442         *v = abs(vD[IndexOfMaxY - 1] - (2.0 * vD[IndexOfMaxY]) +
            vD[IndexOfMaxY + 1]);
443         #endif
444     }
445
446     uint8_t arduinoFFT::Exponent(uint16_t value)
447     {
448         #warning("This method may not be accessible on future revisions.")
449         // Calculates the base 2 logarithm of a value
450         uint8_t result = 0;
451         while (((value >> result) & 1) != 1) result++;
452         return(result);
453     }
454
455     // Private functions
456
457     void arduinoFFT::Swap(double *x, double *y)
458     {
459         double temp = *x;
460         *x = *y;
461         *y = temp;
462     }
463
```

```
/****************************************************************************************************
  SAMDUETimerInterrupt.h
  For SAM DUE boards
  Written by Khoi Hoang

  Built by Khoi Hoang
  https://github.com/khoih-prog/SAMDUE_TimerInterrupt
  Licensed under MIT license

  Now even you use all these new 16 ISR-based timers,with their
  maximum interval practically unlimited (limited only by
  unsigned long miliseconds), you just consume only one SAM DUE
  timer and avoid conflicting with other cores' tasks.
  The accuracy is nearly perfect compared to software timers. The
  most important feature is they're ISR-based timers
  Therefore, their executions are not blocked by bad-behaving
  functions / tasks.
  This important feature is absolutely necessary for
  mission-critical tasks.

  Based on SimpleTimer - A timer library for Arduino.
  Author: mromani@ottotecnica.com
  Copyright (c) 2010 OTTOTECNICA Italy

  Based on BlynkTimer.h
  Author: Volodymyr Shymanskyy

  Version: 1.2.0

  Version Modified By   Date      Comments
  ------- -----------  ---------- -----------
  1.0.1   K Hoang      06/11/2020 Initial coding
  1.1.1   K.Hoang      06/12/2020 Add Change_Interval example. Bump
  up version to sync with other TimerInterrupt Libraries
  1.2.0   K.Hoang      10/01/2021 Add better debug feature. Optimize
  code and examples to reduce RAM usage
*****************************************************************************************************
*****************************************************/

#pragma once

#ifndef SAMDUETIMERINTERRUPT_H
#define SAMDUETIMERINTERRUPT_H

#if !( defined(ARDUINO_SAM_DUE) || defined(  SAM3X8E  ) )
  #error This code is designed to run on SAM DUE board / platform!
  Please check your Tools->Board setting.
#endif

#include "Arduino.h"
#include <inttypes.h>

#ifndef SAMDUE_TIMER_INTERRUPT_VERSION
```

```
44    #define SAMDUE_TIMER_INTERRUPT_VERSION        "SAMDUETimerInterrupt
      v1.2.0"
45  #endif
46
47  #include "TimerInterrupt_Generic_Debug.h"
48
49  #ifdef BOARD_NAME
50    #undef BOARD_NAME
51  #endif
52
53  #ifndef BOARD_NAME
54    #define BOARD_NAME        "SAM DUE"
55  #endif
56
57  /*
58      This fixes compatibility for Arduino Servo Library.
59      Uncomment to make it compatible.
60
61      Note that:
62          + Timers: 0,2,3,4,5 WILL NOT WORK, and will
63                    neither be accessible by Timer0,...
64  */
65  // #define USING_SERVO_LIB   true
66
67  #if USING_SERVO_LIB
68      // Arduino Servo library uses timers 0,2,3,4,5.
69    // You must have `#define USING_SERVO_LIB true` in your sketch.
70      #warning Using Servo Library, Timer0, 2, 3, 4 and 5 not available
71  #endif
72
73  #if defined TC2
74    #define NUM_TIMERS  9
75  #else
76    #define NUM_TIMERS  6
77  #endif
78
79  typedef void (*timerCallback)  ();
80
81  typedef struct
82  {
83      Tc *tc;
84      uint32_t channel;
85      IRQn_Type irq;
86  } DueTimerIRQInfo;
87
88  typedef struct
89  {
90      const char* tc;
91      uint32_t channel;
92      const char* irq;
93  } DueTimerIRQInfoStr;
94
95  // For printing info of selected Timer
96  const DueTimerIRQInfoStr TimersInfo[NUM_TIMERS] =
```

```
  {
    { "TC0", 0, "TC0_IRQn" },
    { "TC0", 1, "TC1_IRQn" },
    { "TC0", 2, "TC2_IRQn" },
    { "TC1", 0, "TC3_IRQn" },
    { "TC1", 1, "TC4_IRQn" },
    { "TC1", 2, "TC5_IRQn" },

#if defined(TC2)
    { "TC2", 0, "TC6_IRQn" },
    { "TC2", 1, "TC7_IRQn" },
    { "TC2", 2, "TC8_IRQn" },
#endif
};

class DueTimerInterrupt
{
  protected:

    // Represents the timer id (index for the array of
    DueTimerIRQInfo structs)
    const unsigned short _timerNumber;

    // Stores the object timer frequency
    // (allows to access current timer period and frequency):
    static double _frequency[NUM_TIMERS];

    // Make Interrupt handlers friends, so they can use _callbacks
    friend void TC0_Handler();
    friend void TC1_Handler();
    friend void TC2_Handler();
    friend void TC3_Handler();
    friend void TC4_Handler();
    friend void TC5_Handler();

#if defined(TC2)
    friend void TC6_Handler();
    friend void TC7_Handler();
    friend void TC8_Handler();
#endif

    static timerCallback _callbacks[NUM_TIMERS];

    // Store timer configuration (static, as it's fixed for every
    object)
    static const DueTimerIRQInfo Timers[NUM_TIMERS];

  public:

    DueTimerInterrupt(unsigned short timer) : _timerNumber(timer)
    {
      /*
        The constructor of the class DueTimerInterrupt
      */
```

```
149       }
150
151       static DueTimerInterrupt getAvailable()
          __attribute  ((always_inline))
152       {
153         /*
154           Return the first timer with no callback set
155         */
156
157         for (int i = 0; i < NUM_TIMERS; i++)
158         {
159           if (!_callbacks[i])
160           {
161             TISR_LOGWARN3(F("Using Timer("), i, F(") ="),
                TimersInfo[i].tc);
162             TISR_LOGWARN3(F("Channel ="), TimersInfo[i].channel, F(",
                IRQ ="), TimersInfo[i].irq);
163
164             return DueTimerInterrupt(i);
165           }
166         }
167
168         // Default, return Timer0;
169         return DueTimerInterrupt(0);
170       }
171
172       DueTimerInterrupt& attachInterruptInterval(double microseconds,
          timerCallback callback)   attribute  ((always_inline))
173       {
174         _callbacks[_timerNumber] = callback;
175
176         return startTimer(microseconds);
177       }
178
179       DueTimerInterrupt& attachInterrupt(float frequency,
          timerCallback callback)   attribute  ((always_inline))
180       {
181         return attachInterruptInterval((double) (1000000.0f /
          frequency), callback);
182       }
183
184       DueTimerInterrupt& attachInterrupt(timerCallback callback)
          __attribute  ((always_inline))
185       {
186         /*
187           Links the function passed as argument to the timer of the
            object
188         */
189
190         _callbacks[_timerNumber] = callback;
191
192         return *this;
193       }
194
```

```
195       DueTimerInterrupt& detachInterrupt()
          __attribute__((always_inline))
196       {
197         /*
198           Links the function passed as argument to the timer of the
              object
199         */
200
201         stopTimer(); // Stop the currently running timer
202
203         _callbacks[_timerNumber] = NULL;
204
205         return *this;
206       }
207
208       DueTimerInterrupt& startTimer(double microseconds= -1)
          __attribute__((always_inline))
209       {
210         /*
211           Start the timer
212           If a period is set, then sets the period and start the timer
213         If not period => default to 1Hz
214         */
215
216         if (microseconds > 0)
217           setPeriod(microseconds);
218
219         if (_frequency[_timerNumber] <= 0)
220           setFrequency(1);
221
222         NVIC_ClearPendingIRQ(Timers[_timerNumber].irq);
223         NVIC_EnableIRQ(Timers[_timerNumber].irq);
224
225         TC_Start(Timers[_timerNumber].tc, Timers[_timerNumber].channel);
226
227         return *this;
228       }
229
230       DueTimerInterrupt& restartTimer(double microseconds= -1)
          __attribute__((always_inline))
231       {
232         /*
233           Restart the timer
234           If a period is set, then sets the period and start the timer
235         If not period => default to 1Hz
236         */
237         // If not yet initialized, set 1Hz
238         if (_frequency[_timerNumber] <= 0)
239         {
240           setFrequency(1);
241         }
242         else if (microseconds < 0)
243         {
244           // Using previous settings if no argument (microseconds = -1)
```

```
245           setFrequency(_frequency[_timerNumber]);
246         }
247         else
248         {
249           setPeriod(microseconds);
250         }
251
252         NVIC_ClearPendingIRQ(Timers[_timerNumber].irq);
253         NVIC_EnableIRQ(Timers[_timerNumber].irq);
254
255         TC_Start(Timers[_timerNumber].tc, Timers[_timerNumber].channel);
256
257         return *this;
258       }
259
260       DueTimerInterrupt& stopTimer()   attribute  ((always_inline))
261       {
262         /*
263           Stop the timer
264         */
265
266         NVIC_DisableIRQ(Timers[_timerNumber].irq);
267
268         TC_Stop(Timers[_timerNumber].tc, Timers[_timerNumber].channel);
269
270         return *this;
271       }
272
273       DueTimerInterrupt& disableTimer()
274       {
275         return stopTimer();
276       }
277
278       // Picks the best clock to lower the error
279       static uint8_t bestClock(double frequency, uint32_t& retRC)
280       {
281         /*
282           Pick the best Clock, thanks to Ogle Basil Hall!
283
284           Timer    Definition
285           TIMER_CLOCK1   MCK /  2
286           TIMER_CLOCK2   MCK /  8
287           TIMER_CLOCK3   MCK / 32
288         TIMER_CLOCK4     MCK /128
289         */
290         const struct
291         {
292           uint8_t flag;
293           uint8_t divisor;
294         } clockConfig[] =
295         {
296           { TC_CMR_TCCLKS_TIMER_CLOCK1,    2 },
297           { TC_CMR_TCCLKS_TIMER_CLOCK2,    8 },
298           { TC_CMR_TCCLKS_TIMER_CLOCK3,   32 },
```

```
299           { TC_CMR_TCCLKS_TIMER_CLOCK4, 128 }
300         };
301
302         float ticks;
303         float error;
304         int clkId       = 3;
305         int bestClock   = 3;
306         float bestError = 9.999e99;
307
308         do
309         {
310           ticks = (float) SystemCoreClock / frequency / (float)
              clockConfig[clkId].divisor;
311           // error = abs(ticks - round(ticks));
312           error = clockConfig[clkId].divisor * abs(ticks -
              round(ticks)); // Error comparison needs scaling
313
314           if (error < bestError)
315           {
316             bestClock = clkId;
317             bestError = error;
318           }
319         } while (clkId-- > 0);
320
321         ticks = (float) SystemCoreClock / frequency / (float)
            clockConfig[bestClock].divisor;
322         retRC = (uint32_t) round(ticks);
323
324         return clockConfig[bestClock].flag;
325       }
326
327
328       DueTimerInterrupt& setFrequency(double frequency)
329       {
330         /*
331           Set the timer frequency (in Hz)
332         */
333
334         // Prevent negative frequencies
335         if (frequency <= 0)
336         {
337           frequency = 1;
338         }
339
340         // Remember the frequency — see below how the exact frequency
            is reported instead
341         //_frequency[_timerNumber] = frequency;
342
343         // Get current timer configuration
344         DueTimerIRQInfo timerIRQInfo = Timers[_timerNumber];
345
346         uint32_t rc = 0;
347         uint8_t clock;
348
```

```
349         // Tell the Power Management Controller to disable
350         // the write protection of the (Timer/Counter) registers:
351         pmc_set_writeprotect(false);
352
353         // Enable clock for the timer
354         pmc_enable_periph_clk((uint32_t)timerIRQInfo.irq);
355
356         // Find the best clock for the wanted frequency
357         clock = bestClock(frequency, rc);
358
359         switch (clock)
360         {
361           case TC_CMR_TCCLKS_TIMER_CLOCK1:
362             _frequency[_timerNumber] = (double)SystemCoreClock / 2.0 /
              (double)rc;
363             break;
364           case TC_CMR_TCCLKS_TIMER_CLOCK2:
365             _frequency[_timerNumber] = (double)SystemCoreClock / 8.0 /
              (double)rc;
366             break;
367           case TC_CMR_TCCLKS_TIMER_CLOCK3:
368             _frequency[_timerNumber] = (double)SystemCoreClock / 32.0
              / (double)rc;
369             break;
370           default: // TC_CMR_TCCLKS_TIMER_CLOCK4
371             _frequency[_timerNumber] = (double)SystemCoreClock / 128.0
              / (double)rc;
372             break;
373         }
374
375         // Set up the Timer in waveform mode which creates a PWM
376         // in UP mode with automatic trigger on RC Compare
377         // and sets it up with the determined internal clock as clock
          input.
378         TC_Configure(timerIRQInfo.tc, timerIRQInfo.channel,
          TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC | clock);
379
380         // Reset counter and fire interrupt when RC value is matched:
381         TC_SetRC(timerIRQInfo.tc, timerIRQInfo.channel, rc);
382
383         // Enable the RC Compare Interrupt.
384         timerIRQInfo.tc->TC_CHANNEL[timerIRQInfo.channel].TC_IER =
          TC_IER_CPCS;
385
386         // ... and disable all others.
387         timerIRQInfo.tc->TC_CHANNEL[timerIRQInfo.channel].TC_IDR =
          ~TC_IER_CPCS;
388
389         return *this;
390       }
391
392       DueTimerInterrupt& setPeriod(double microseconds)
          __attribute  ((always_inline))
393       {
```

```
394         /*
395           Set the period of the timer (in microseconds)
396         */
397
398         // Convert period in microseconds to frequency in Hz
399         double frequency = 1000000.0 / microseconds;
400
401         setFrequency(frequency);
402
403         return *this;
404       }
405
406       DueTimerInterrupt& setInterval(double microseconds)
          __attribute  ((always_inline))
407       {
408         return setPeriod(microseconds);
409       }
410
411       double getFrequency() const   attribute  ((always_inline))
412       {
413         /*
414           Get current time frequency
415         */
416         return _frequency[_timerNumber];
417       }
418
419       double getPeriod() const   attribute  ((always_inline))
420       {
421         /*
422           Get current time period
423         */
424         return 1.0 / getFrequency() * 1000000;
425       }
426
427       uint16_t getTimerNumber()
428       {
429         return _timerNumber;
430       }
431
432       bool operator== (const DueTimerInterrupt& rhs) const
          __attribute  ((always_inline))
433       {
434         return _timerNumber == rhs._timerNumber;
435       };
436
437       bool operator!= (const DueTimerInterrupt& rhs) const
          __attribute  ((always_inline))
438       {
439         return _timerNumber != rhs._timerNumber;
440       };
441   };
442
443
444
```

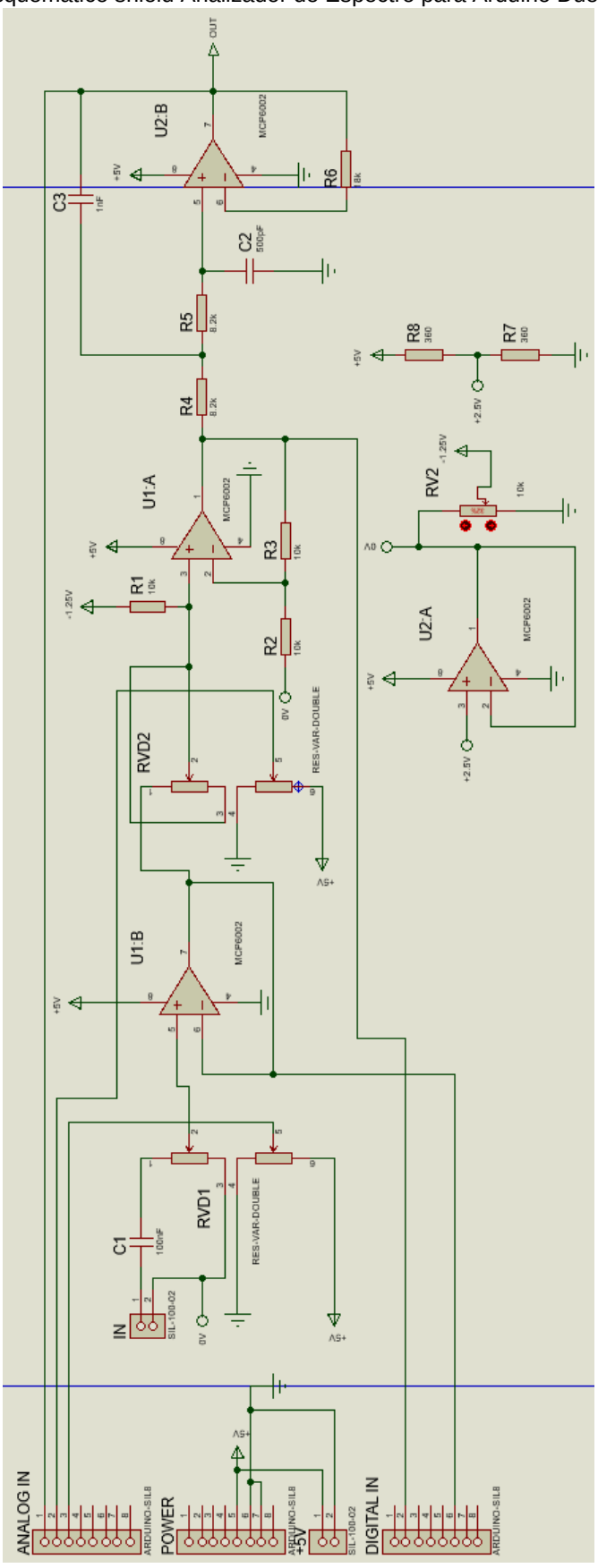
```
445  ////////////////////////////////////////////////////////////////////////
446
447  const DueTimerIRQInfo DueTimerInterrupt::Timers[NUM_TIMERS] =
448  {
449    { TC0, 0, TC0_IRQn },
450    { TC0, 1, TC1_IRQn },
451    { TC0, 2, TC2_IRQn },
452    { TC1, 0, TC3_IRQn },
453    { TC1, 1, TC4_IRQn },
454    { TC1, 2, TC5_IRQn },
455
456  #if defined(TC2)
457    { TC2, 0, TC6_IRQn },
458    { TC2, 1, TC7_IRQn },
459    { TC2, 2, TC8_IRQn },
460  #endif
461  };
462
463  // Fix for compatibility with Servo library
464  #if USING_SERVO_LIB
465    // Set _callbacks as used, allowing
       DueTimerInterrupt::getAvailable() to work
466    void (*DueTimerInterrupt::_callbacks[NUM_TIMERS])() =
467    {
468      (void (*)()) 1, // Timer 0 - Occupied
469      (void (*)()) 0, // Timer 1
470      (void (*)()) 1, // Timer 2 - Occupied
471      (void (*)()) 1, // Timer 3 - Occupied
472      (void (*)()) 1, // Timer 4 - Occupied
473      (void (*)()) 1, // Timer 5 - Occupied
474
475    #if defined(TC2)
476      (void (*)()) 0, // Timer 6
477      (void (*)()) 0, // Timer 7
478      (void (*)()) 0  // Timer 8
479    #endif
480    };
481
482  #else
483    void (*DueTimerInterrupt::_callbacks[NUM_TIMERS])() = {};
484  #endif
485
486  #if defined(TC2)
487    double DueTimerInterrupt::_frequency[NUM_TIMERS] = { -1, -1, -1,
       -1, -1, -1, -1, -1, -1};
488  #else
489    double DueTimerInterrupt::_frequency[NUM_TIMERS] = { -1, -1, -1,
       -1, -1, -1};
490  #endif
491
492  /*
493    Initializing all timers, so you can use them like this:
       Timer0.startTimer();
494  */
```

```
495   DueTimerInterrupt DueTimer(0);
496
497   DueTimerInterrupt Timer1(1);
498
499   // Fix for compatibility with Servo library
500   #if ( !USING_SERVO_LIB || !defined(USING_SERVO_LIB) )
501     DueTimerInterrupt Timer0(0);
502     DueTimerInterrupt Timer2(2);
503     DueTimerInterrupt Timer3(3);
504     DueTimerInterrupt Timer4(4);
505     DueTimerInterrupt Timer5(5);
506   #endif
507
508   #if defined(TC2)
509     DueTimerInterrupt Timer6(6);
510     DueTimerInterrupt Timer7(7);
511     DueTimerInterrupt Timer8(8);
512   #endif
513
514   DueTimerInterrupt DueTimerPtr[NUM_TIMERS] =
515   {
516   #if ( !USING_SERVO_LIB || !defined(USING_SERVO_LIB) )
517     Timer0,
518   #endif
519
520     Timer1,
521
522   #if ( !USING_SERVO_LIB || !defined(USING_SERVO_LIB) )
523     Timer2,
524     Timer3,
525     Timer4,
526     Timer5,
527   #endif
528
529   #if defined(TC2)
530     Timer6,
531     Timer7,
532     Timer8
533   #endif
534   };
535
536   ////////////////////////////////////////////////////////////////////////
      /
537
538   /*
539     Implementation of the timer _callbacks defined in
540
      arduino-1.5.2/hardware/arduino/sam/system/CMSIS/Device/ATMEL/sam3xa/
      include/sam3x8e.h
541   */
542   // Fix for compatibility with Servo library
543   #if ( !USING_SERVO_LIB || !defined(USING_SERVO_LIB) )
544
545   void TC0_Handler()
```

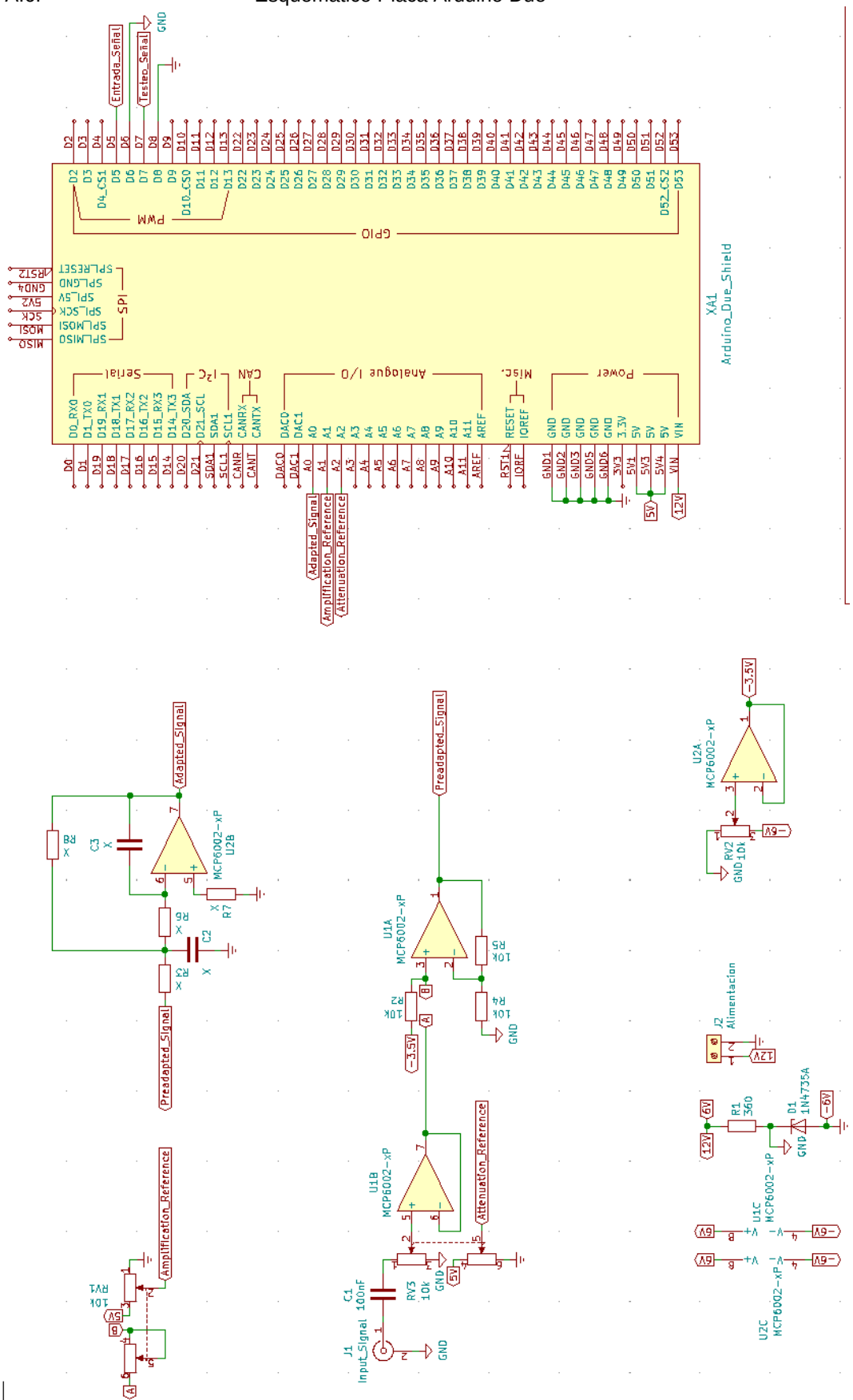
```
546   {
547     TC_GetStatus(TC0, 0);
548     DueTimerInterrupt::_callbacks[0]();
549   }
550
551   #endif
552
553   void TC1_Handler()
554   {
555     TC_GetStatus(TC0, 1);
556     DueTimerInterrupt::_callbacks[1]();
557   }
558
559   // Fix for compatibility with Servo library
560   #if ( !USING_SERVO_LIB || !defined(USING_SERVO_LIB) )
561
562   void TC2_Handler()
563   {
564     TC_GetStatus(TC0, 2);
565     DueTimerInterrupt::_callbacks[2]();
566   }
567
568   void TC3_Handler()
569   {
570     TC_GetStatus(TC1, 0);
571     DueTimerInterrupt::_callbacks[3]();
572   }
573
574   void TC4_Handler()
575   {
576     TC_GetStatus(TC1, 1);
577     DueTimerInterrupt::_callbacks[4]();
578   }
579
580   void TC5_Handler()
581   {
582     TC_GetStatus(TC1, 2);
583     DueTimerInterrupt::_callbacks[5]();
584   }
585   #endif
586
587   #if defined(TC2)
588
589   void TC6_Handler()
590   {
591     TC_GetStatus(TC2, 0);
592     DueTimerInterrupt::_callbacks[6]();
593   }
594
595   void TC7_Handler()
596   {
597     TC_GetStatus(TC2, 1);
598     DueTimerInterrupt::_callbacks[7]();
599   }
```

```
600
601   void TC8_Handler()
602   {
603     TC_GetStatus(TC2, 2);
604     DueTimerInterrupt::_callbacks[8]();
605   }
606   #endif
607
608   #endif     // SAMDUETIMERINTERRUPT_H
609
```

Apéndice A.5: Esquemático shield Analizador de Espectro para Arduino Due



Apéndice A.6: Esquemático Placa Arduino Due



Apéndice A.6: Esquemático Placa Arduino Due

```
1  /*
2      MODELO DE PRUEBA de ANALIZADOR DE ESPECTRO
3      Catedra: MEDIDAS ELECTRONICAS II
4      Docentes: C. Cappelletti, M. F. Krenz
5      Grupo N°: 3
6      Alumnos: A. A. Baldini, N. H. Gimenez, T. A. Pentacolo, L. D.
       Rispoli.
7  */
8
9  /*
10   En este ejemplo, el programa (mediante un loop) habilita la
     conversión de datos por el ADC; para luego, aplicar el filtro
     Hamming a los datos.
11   Una vez limpio esto, se procede al cálculo de la respectiva FFT y
     preparamos el espectro para solo ver el Diagrama de Amplitud.
12   Finalmente se procede a mostrar el mismo.
13 */
14
15 //Declaraciones_____________________________________________________
16 //Con cambios_____________________________________________________
17
18 #include <arduinoFFT.h>
19 #include "SAMDUETimerInterrupt.h"
20
21 #define pin_ADC  A7                 //Configuracion PIN OUT. Nueva
   Linea.
22 #define pin_atte A5                 //Configuración PIN OUT. Nueva
   Linea.
23 #define pin_gan A6                  //Configuración PIN OUT. Nueva
   Linea.
24
25 uint16_t Timer_Index = 0;           // Identificador del Timer.
26
27 const uint16_t samples = 32;        // Cantidad de muestras a
   analizar.
28 const double Fm = 1E5;              // Frecuencia de muestreo.
29
30 arduinoFFT FFT = arduinoFFT();      // Creo el objeto FFT.
31
32 double vReal[samples];              // Vector que contendrá los
   valores adquiridos en el ADC.
33 double vAnalog[samples];            // Vector que contendrá los
   valores reales calculados en la FFT.
34 double vImag[samples];              // Vector que contendrá los
   valores imaginarios.
35
36 //TimerHandler1_____________________________________________________
37 //Con cambios_____________________________________________________
38
39 void TimerHandler1()
40 /*
41  *   TimerHandle1();
42  *   En este método se produce la rotación de los datos adquiridos
43  *   y se añade al inicio del vector el nuevo dato adquirido.
```

```
44    */
45   {
46     PIOB->PIO_ODSR = 1<<25;               // Escribo un 1 en el pin 25
       del puerto B.
47
48     for (int i = samples; i > 0; i--)     // Rotamos el vector.
49     {
50       vReal[i] = vReal[i-1];
51     }
52
53     vReal[0] = analogRead(pin_ADC);       // Leemos el nuevo dato.
       Linea modificada
54
55     PIOB->PIO_ODSR = 0;                   // Escribo un 0 en el pin 25
       del puerto B.
56   }
57
58   //setup_____________________________________________________________
59
60   void setup() {
61     PIOB->PIO_PER = (1<<25);
       // Configuramos el puerto B para controlarlo por PIO.
62     PIOB->PIO_OER = (1<<25);
       // Establecemos el pin 25 del puerto B como salida.
63
64     analogReadResolution(10);
       // Establecemos la resolución del ADC en 10 bits.
65
66     for (int i = 0; i < samples; i++)
       // Inicializo los vectores en 0.
67     {
68       vReal[i] = 0;
69       vAnalog[i] = 0;
70       vImag[i] = 0;
71     }
72
73     DueTimerInterrupt dueTimerInterrupt = DueTimer.getAvailable();
       // Creamos un objeto DueTimerInterrupt (administra interrupción de
       Timer).
74     dueTimerInterrupt.attachInterruptInterval(10, TimerHandler1);
       // Establecemos una interrupción cada 10 [us], obteniendo una
       frecuencia de 100[kHz].
75     Timer_Index = dueTimerInterrupt.getTimerNumber();
       // Obtenemos el identificador del Timer.
76
77     Serial.begin(115200);
       // Inicializamos el puerto serie.
78     while(!Serial);
       // Esperamos a que finalice la inicialización.
79     Serial.println("Ready");
80     adc_start(ADC);
       // Iniciamos la conversión del ADC.
81   }
82
```

```
83  //loop_____________________________________________________________________
84  //Con cambios______________________________________________________________
85
86  void loop() {
87    DueTimerPtr[Timer_Index].stopTimer();
      // Pausamos el Timer.
88
89    for (int i = 0; i < samples; i++)
90    {
91      float atte = 1 - (float)analogRead(pin_atte) / (float)1024;
        // Obtenemos la referencia de la atenuación.
92      float gan = (float)1024 / (float)analogRead(pin_gan) ;
        // Obtenemos la referencia de la ganancia.
93      vAnalog[i] = vReal[i] - 512;
        // Copiamos los valores adquiridos hasta el momento y eliminamos
        la componente de continua.
94      vAnalog[i] = vAnalog[i] * 3.3 * atte * gan / (float)1024;
        // Convertimos los valores digitales a valores reales.
95      vImag[i] = 0;
        // Limpiamos los valores imaginarios.
96    }
97
98    DueTimerPtr[Timer_Index].startTimer();
      // Activamos nuevamente el Timer.
99    FFT.Windowing(vAnalog, samples, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
      // Aplicamos la ventana Hamming al los datos adquiridos.
100   FFT.Compute(vAnalog, vImag, samples, FFT_FORWARD);
      // Aplicamos la FFT a los datos adquiridos.
101   FFT.ComplexToMagnitude(vAnalog, vImag, samples);
      // Obtenemos la amplitud del espectro calculado.
102   PrintVector(vAnalog, (samples >> 1));
      // Transmitimos la magnitud del espectro calculado.
103   delay(100);
      // Esperamos 100 [ms].
104 }
105
106 //PrintVector______________________________________________________________
107 //Con cambios______________________________________________________________
108
109 void PrintVector(double *vData, uint16_t bufferSize)
110 /*
111  * PrintVector(double *vData, uint16_t bufferSize);
112  *  double *vData: Es el vector que apunta a los datos a imprimir.
113  *  uint16_t bufferSize: Cantidad de datos contenidos en el vector.
114  *
115  * Este método calcula el valor de frecuencia correspondiente a cada
     elemento
116  * del vector, y transmite tanto la frecuencia calculada como su
     valor de
117  * espectro.
118  */
119 {
120   for (uint16_t i = 0; i < bufferSize; i++)
      // Por cada elemento.
```

```
121     {
122       double abscissa;
123       abscissa = ((i * 1.0 * Fm) / samples);
          // Obtenemos el valor de frecuencia correspondiente.
124       Serial.print(abscissa, 6);
125       Serial.print(" ");
126       Serial.println(vData[i], 4);
          // Transmitimos la información.
127     }
128     Serial.println();
129   }
130
```

Apéndice B.1: **Programa Principal** **ReceptorMP.m**

```
1
2   %   MODELO DE PRUEBA de ANALIZADOR DE ESPECTRO
3   %   Catedra: MEDIDAS ELECTRONICAS II
4   %   Docentes: C. Cappelletti, M. F. Krenz
5   %   Grupo N°: 3
6   %   Alumnos: A. A. Baldini, N. H. Gimenez, T. A. Pentacolo, L. D.
    Rispoli.
7
8
9
10  % En este ejemplo, el programa inicia la comunicación para que el
    Arduino obtenga los datos a mostrar obtenidos desde nuestra placa.
11  % Finalmente se procede a mostrar el mismo.
12
13  % Inicialización del entorno de trabajo_____________________________
14
15  clc;
16  clear all;
17  close all;
18  commandwindow;
19
20  % Configuración con el usuario_____________________________________
21
22  puerto = input('Ingrese el puerto al que desea comunicarse: ej.
    COM1\n');
23  % Ingreso del puerto a comunicarse.
24
25  % Inicialización de las variables y objetos_____________________
26
27  analizador = serial(puerto, 'baudrate', 115200);
28  % Inicializa la comunicación con el puerto.
29  fopen(analizador);
30
31  freq = zeros(16, 1);
32  amp = zeros(16, 1);
33  h = figure; % Obtiene un Figure para plotear.
34  visor = [];
35  existe = 1;
36  tic
37
38  % Loop_____________________________________________________________
39
40  while (existe)
41  % Si el elemento figure existe en la pantalla.
42      i = 1;
43      while (i <= 16)
44          if (analizador.BytesAvailable >
            0)
45          % Mientras existan datos para leer.
46              data = 0;
47              while (length(data) <
                2)
48              % Y se obtengan dos datos (haciendo referencia a la
                frecuencia y a la amplitud).
```

```
49                  data = fscanf(analizador, '%f
                    %f\n');
50                  % Obtiene los datos.
51              end
52              if ((i == 1 && data(1) == 0) | i ~=
                1)
53              % Sincroniza los datos para que el primero sea el de la
                frecuencia 0[Hz].
54                  freq(i) = data(1);
55                  amp(i) = data(2);
56                  i = i+1;
57              end
58          end
59      end
60
61      if
        isempty(visor)
62      % Si el elemento figure esta vacío.
63          visor = plot(freq, amp, '-');   % Grafica los datos.
64          title('Espectro');
65          xlabel('Frecuencia [Hz]');
66          ylabel('Amplitud [V]');
67      else
68          set(visor, 'YData', amp, 'XData',
            freq);
69          % Si no está vacío, simplemente actualizo los datos.
70      end
71      drawnow;    % Actualiza la gráfica.
72      pause(0.1); % Espera 100 [ms] para volver a leer.
73      existe = ishandle(visor);
74  end
75
76  % Finaliza la aplicación_____________________________________________
77
78  toc
79  fclose(analizador); % Corta la conexión con el puerto.
80  disp('Se ha cerrado la conexión.');
```