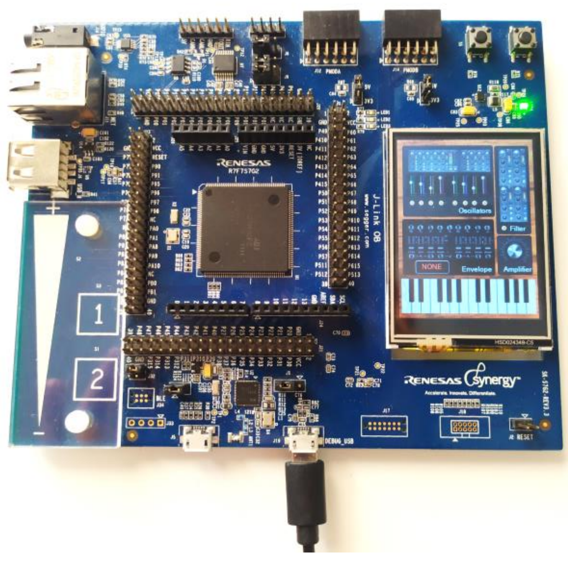
Trabajo Fin de Máster
Máster en Ingeniería de las Tecnologías Industriales



Implementación de un sintetizador MIDI en un microcontrolador de altas prestaciones.

Autor: María del Pilar Martínez Serrano
Tutor: Manuel Ángel Perales Esteve

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Máster

Máster en Ingeniería de las Tecnologías Industriales

Implementación de un sintetizador MIDI en un microcontrolador de altas prestaciones

Autor:

María del Pilar Martínez Serrano

Tutor:

Manuel Ángel Perales Esteve

Profesor Titular de Universidad

Dpto. de Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Máster: Implementación de un sintetizador MIDI en un microcontrolador de altas prestaciones

Autor: María del Pilar Martínez Serrano

Tutor: Manuel Ángel Perales Esteve

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2022

El Secretario del Tribunal

A mi familia

A mis amigos

AMDG

Agradecimientos

A mi madre, porque gracias a su ejemplo de trabajo y esfuerzo (y por supuesto a su insistencia), he logrado culminar este proyecto.

A mi familia, especialmente a Carmencita, María, Mapi y Santi, por tanto interés mostrado en la marcha de mi trabajo.

A Angelita, Blanca, Chío, Julia, Laura, María y Sofí, por su amistad incondicional, y por tantas sonrisas y lágrimas compartidas en este último año.

A Pablo, por venir a ser mi pilar en la etapa final de este proyecto y en cualquier otro que emprenda en el futuro.

A Manolo, por su infinita paciencia, y por prestarme una vez más su sintetizador, su tiempo y su antedespacho durante los últimos meses.

A mis compañeros de Atexis, en especial a Miri, por todo lo aprendido de ellos humana y profesionalmente.

A los que ya no están, por haber estado apoyándome desde Arriba.

A Dios, por poner a todas estas maravillosas personas a mi lado.

Pilar

Sevilla, 2022

Resumen

El nacimiento de los instrumentos musicales se remonta al origen de la humanidad. Los primeros cuernos, tambores y flautas datan de la era prehistórica, mas no tuvieron en principio una intencionalidad melódica sino que fueron concebidos para anunciar rituales, cazas o batallas. Se buscaba simplemente la emulación de sonidos naturales a partir de la percusión de objetos diversos. Progresivamente la riqueza del sonido y sus infinitas posibilidades sedujo al hombre, adquiriendo la música un papel fundamental en las diversas culturas. Así, a lo largo de la Historia se fue desarrollando un vasto número de obras de arte musical que ha llegado hasta nuestros días, así como el amplio abanico de instrumentos que permiten su interpretación. Llegado cierto punto de la Historia la pretensión de emulación de sonidos se comenzó a repetir, esta vez tratando de reproducir no solo los sonidos "naturales" sino buscando un acercamiento a aquellos generados por los instrumentos musicales clásicos. Es así como hicieron aparición los primeros instrumentos electrónicos.

Con este proyecto se pretende dar un pequeño paso más en la rápida evolución de dichos artefactos musicales, creando una versión mejorada del sintetizador Midisynth 2.0, concebido hace dos años durante mi Trabajo Fin de Grado. En esta ocasión el dispositivo se implementará sobre un microcontrolador de prestaciones superiores, que permitirá entre otras mejoras una mayor precisión en el cálculo de la onda, por incorporar artimética de punto flotante. La nueva placa cuenta con diversos periféricos integrados en la misma: pantalla LCD, pulsadores y panel de controles táctiles, además de conector de audio jack. Ello supone un avance respecto a la versión anterior, que incorporaba los distintos periféricos del sintetizador en una placa PCB diseñada para la aplicación específica del Midisynth 2.0. De este modo obtendremos un dispositivo integrado más compacto sobre un microcontrolador con potencia de cálculo superior, lo cual abrirá las puertas a futuras líneas de mejora para el sintetizador.

Abstract

The birth of musical instruments dates back to the origin of mankind. The first horns, drums and flutes arose in the Prehistoric Age, but originally they did not intend to be melodic, but rather to announce rituals, hunts or battles. The aim was simply to emulate natural sounds by percussion of various objects. Progressively, the richness of sound and its infinite possibilities seduced man, and music acquired a fundamental role in the different cultures. Thus, throughout History, a vast number of works of musical art have been developed, having survived to the present day, along with the wide range of instruments that allow their interpretation. At a certain point in History, the attempt to emulate sounds was repeated, this time trying to reproduce not only the sounds of the Nature but also seeking an approach to those sounds generated by classical musical instruments. This is how the first electronic instruments appeared.

The aim of this project is to take a small step forward in the rapid evolution of these musical devices, creating an improved version of the Midisynth 2.0 synthesiser, conceived two years ago during my Final Degree Project. This time the device will be implemented on a higher performance microcontroller, which will allow, in addition to other improvements, a higher precision in the calculation of the wave, by incorporating floating point arithmetic. The new board has various integrated peripherals: a LCD screen, some pushbuttons and a capacitive touch panel, as well as audio output type *jack*. This is an improvement over the previous version, which incorporated the various peripherals of the synthesiser on a PCB board designed for the specific application of Midisynth 2.0. This will result in a more compact embedded device on a microcontroller with superior computing power, which will open the door to future lines of improvement for the synthesiser.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xiv**

Índice de Tablas **xvii**

Índice de Figuras **xix**

Abreviaturas **xxiii**

1 Prefacio **12**
1.1 Estado del arte: Midisynth 2.0 *12*
1.2 Etapas del proyecto *14*
1.3 Alcance *15*
1.4 Estructura del documento *15*

2 Fundamentos teóricos **17**
2.1 Introducción a la síntesis de audio *17*
2.1.1 Módulos del sintetizador 18
2.1.2 Tipos de modulación 20
2.2 El estándar MIDI *23*
2.3 El sintetizador Midisynth 3.0 *24*

3 Búsqueda de alternativas hardware **27**
3.1 Placa de Desarrollo WiPy 3.0 *27*
3.1.1 Interrupciones 30
3.1.2 Testeo del DAC integrado 31
3.1.3 Pruebas con DAC externo MCP4921 33
3.1.4 Conclusiones 34
3.2 Plataforma Renesas Synergy™ *35*
3.2.1 Periféricos empleados 36
3.2.2 Elementos auxiliares 37

4 Herramientas Software **39**
4.1 E2 Studio *39*
4.2 Tera Term *39*
4.3 Guix Studio *40*
4.4 Capacitive Touch Workbench *40*
4.5 Atom *40*

5 Desarrollo del programa **43**
5.1 Especificaciones de funcionamiento *43*
5.2 Configuración de los módulos *46*

5.3 Diseño de la interfaz gráfica en GUIX Studio *50*
5.4 Etapas del programa *55*
5.4.1 Hilo principal 57
5.4.2 Interrupción MIDI 58
5.4.3 Hilo secundario - Control LCD 58
5.4.4 Interrupción DCO 59
5.4.5 Interrupción LFO 59
5.4.6 Interrupción panel táctil 60
5.5 Estructura del código *60*
6 Resultados **63**
7 Conclusiones **71**
Anexo I **73**
Anexo II **91**
Referencias **155**

ÍNDICE DE TABLAS

Tabla 2-1. Características del sonido. 18

Tabla 2-2. Mensajes tipo *note-on/note-off*. 24

Tabla 3-1. Especificaciones del SK-S7G2 [10]. 36

Tabla 5-1. Pinout del panel táctil. 48

Tabla 5-2. Pinout de la LCD. 49

ÍNDICE DE FIGURAS

Figura 1-1. Periféricos de control del Midisynt 2.0. 13
Figura 1-2. Placa WiPy 3.0 14
Figura 2-1. Modificación de la onda mediante modulación. 19
Figura 2-2. Módulos de un sintetizador. 19
Figura 2-3. Principales tipos de modulación. 20
Figura 2-4. Efecto trémolo. 21
Figura 2-5. Fases de la envolvente ADSR. 21
Figura 2-6. Trémolo vs. ADSR. 22
Figura 2-7. Efecto vibrato. 22
Figura 2-8. Efectos de modulación FM en función del Índice de modulación. 23
Figura 2-9. Conectores MIDI hembra y macho. 23
Figura 2-10. Estructura del mensaje MIDI. 24
Figura 2-11. Método de síntesis aditiva. 25
Figura 2-12. Parámetros del filtro pasa-banda [6]. 25
Figura 3-1. Diagrama de bloques de la WiPy 3.0 [7]. 27
Figura 3-2. Especificaciones de la placa WiPy 3.0. 28
Figura 3-3. Expansion Board 3.0 [8]. 28
Figura 3-4. Ventana de inicio ATOM. 29
Figura 3-5. Instalar plugin Pymakr. 29
Figura 3-6. Consola REPL de ATOM. 29
Figura 3-7. Testeo inicial del entorno ATOM. 30
Figura 3-8. Pruebas con el Timer. 30
Figura 3-9.Ploteo de senoidal básica en Python. 31
Figura 3-10. Generación de senoidal mediante el DAC integrado. 31
Figura 3-11. Otras pruebas. 32
Figura 3-12. Onda triangular generada por el DAC interno. 32
Figura 3-13. Encapsulado del MCP4921 [9]. 33
Figura 3-14. Conexiones del DAC externo. 33
Figura 3-15. Señal de salida del DAC y *Chip Select*. 33
Figura 3-16. Señal SCLK y CS del SPI. 34
Figura 3-17. Renesas Synergy SK-S7G2. 35
Figura 3-18. Sistema operativo de Tiempo Real, RTOS. 35
Figura 3-19. Placa PCB de adaptación MIDI. 37

Figura 3-20. Teclado *Icon iKeyboard 3X.* 37
Figura 5-1. Ventana de Inicio. 43
Figura 5-2. Menú principal. 44
Figura 5-3. Ventana de control DCO. 44
Figura 5-4. Slider para el control del desfase. 45
Figura 5-5. Ventana de control LFO y ventana de control ADSR. 45
Figura 5-6. Regulador de volumen. 46
Figura 5-7. Ventana de configuración de módulos. 47
Figura 5-8. Configuración de periféricos. 47
Figura 5-9. Configuración de pines. 48
Figura 5-10. Actualización de configuración HAL. 50
Figura 5-11. Carpeta de proyecto Guix. 50
Figura 5-12. Nuevo proyecto Guix. 51
Figura 5-13. Carpeta de archivos fuente Guix. 51
Figura 5-14. Ventana de la aplicación Guix Studio. 51
Figura 5-15. Crear ventanas del proyecto. 52
Figura 5-16. Pestaña *Properties* de los *Widgets.* 52
Figura 5-17. Configuración de Pixelmaps. 53
Figura 5-18. Editar *Screen Flow.* 53
Figura 5-19. System Event Trigger. 54
Figura 5-20. Child Signal Trigger. 54
Figura 5-21. Acción tipo transición de ventana. 54
Figura 5-22. Screen Flow del proyecto. 55
Figura 5-23. Diagrama de flujo del programa. 56
Figura 5-24. Directorio del proyecto. 61
Figura 6-1. Onda senoidal. 63
Figura 6-2. Onda cuadrada tipo 1. 64
Figura 6-3. Onda cuadrada tipo 2. 64
Figura 6-4. Onda tipo Guitarra. 64
Figura 6-5. Onda tipo Órgano. 65
Figura 6-6. Onda tipo Oboe. 65
Figura 6-7. Onda tipo *Hand-drawn.* 65
Figura 6-8. Onda tipo *Video Game.* 66
Figura 6-9. Onda senoidal + ona cuadrada tipo 1. 66
Figura 6-10. Onda senoidal + ona cuadrada tipo 2. 66
Figura 6-11. Onda de Órgano + Oboe 67
Figura 6-12. Ídem con desfase de valor 30. 67
Figura 6-13. Onda senoidal + cuadrada tipo 1 con desfase 50. 67
Figura 6-14. Onda senoidal con efecto trémolo. Envolvente senoidal. 68

Figura 6-15. Onda senoidal con efecto trémolo. Envolvente cuadrada. 69
Figura 6-16. Onda senoidal con efecto vibrato. Envolvente tipo “órgano”. 69
Figura 6-17. Onda senoidal con efecto vibrato. Envolvente cuadrada. 69
Figura 6-18. Onda senoidal con efecto FM. Envolvente senoidal. 70
Figura 6-19. Onda senoidal con efecto ADSR. 70
Figura 6-20. Detalle del efecto ADSR. 70

Abreviaturas

ADC	*Analog to Digital Converter*
AM	*Amplitude Modulation*
API	*Application Programming Interfaces*
BLE	*Bluetooth Low Energy*
CTW	*Capacitive Touch Workbench*
DCO	*Digitally Controlled Oscillator*
FM	*Frequency Modulation*
GUI	Grafical User Interface
IDE	*Integrated Development Environment*
IoT	*Internet of Things*
kbps	KiloBit Por Segundo
LCD	*Liquid Cristal Display*
MCU	*MicroController Unit*
MIDI	*Musical Instrument Digital Interface*
PCB	*Printed Circuit Board*
REPL	*Read-Eval-Print-Loop*
RTC	*Real-Time Clock*
RTOS	*Real Time Operative System*
RTOS	*Real-Time Operating System*
SPI	*Serial Peripheral Interface*
SSP	*Synergy Software Package*
ULP	*Ultra-Low Power*
VCO	*Voltage Controlled Oscillator*
CS	*Chip Select*
HAL	*Hardware Abstraction Layer*
CTSU	*Capacitive Touch Sensor Unit*

1 PREFACIO

To me, the synthesizers are equally natural instruments.
After all, what is more natural than electricity?.

- Vangelis -

Hace más de un siglo arrancó la rápida evolución de los instrumentos electrónicos, con la invención del *Théremin* en 1920, primer instrumento que producía melodías mediante corriente eléctrica. A este le siguieron el *Ondes Martenot*, el *Mark I*, el *MiniMoog* y muchos otros, hasta los sintetizadores actuales comercializados por marcas conocidas como Roland o Yamaha, que vulgarmente denotamos como "pianos electrónicos".

Sin embargo estos artefactos, cuyos sonidos han logrado ser tan ricos como los producidos por instrumentos clásicos, no son todavía reconocidos como instrumentos auténticos. Ello ha motivado la incansable innovación en este campo, pretendiendo lograr un acercamiento cada vez mayor al sonido de los instrumentos clásicos. El célebre compositor griego Vangelis comenta sobre esta controversia: "En mi opinión los sintetizadores son también instrumentos naturales; después de todo ¿qué hay más natural que la electricidad?".

Con esta misma pretensión, en el presente trabajo trataremos de desarrollar un sintetizador que emule los sonidos de distintos instrumentos musicales, basándonos en los conocimientos adquiridos durante la realización del Trabajo Fin de Grado, en 2020. Dicho proyecto consistió en la programación de un sintetizador MIDI (*Musical Instrument Digital Interface*) sobre el microcontrolador MSP430FR2355 de Texas Instruments. El dispositivo lo bautizamos como Midisynth 2.0 ya que se trataba de una segunda versión del sintetizador Midisynth, idea original de mi tutor, programado asimismo sobre otro microcontrolador de bajo consumo de Texas Instruments.

1.1 Estado del arte: Midisynth 2.0

Como acabamos de referir, nuestro punto de partida en este trabajo ha sido el proyecto Midisynth 2.0, sintetizador digital aditivo basado en tabla de ondas, que incorporaba cuatro efectos de modulación básicos para la reproducción del sonido, controlado mediante protocolo de comunicación MIDI.

El microcontrolador MSP430FR2355 era el encargado de realizar los cálculos de la síntesis de onda, programado en lenguaje C sobre el entorno de desarrollo Code Composer. El MCU (*MicroController Unit*) se

integraba sobre una placa de circuito impreso (PCB, *Printed Circuit Board*) junto a los periféricos necesarios para la recepción de las notas a reproducir, la salida del sonido sintetizado así como el control del dispositivo.

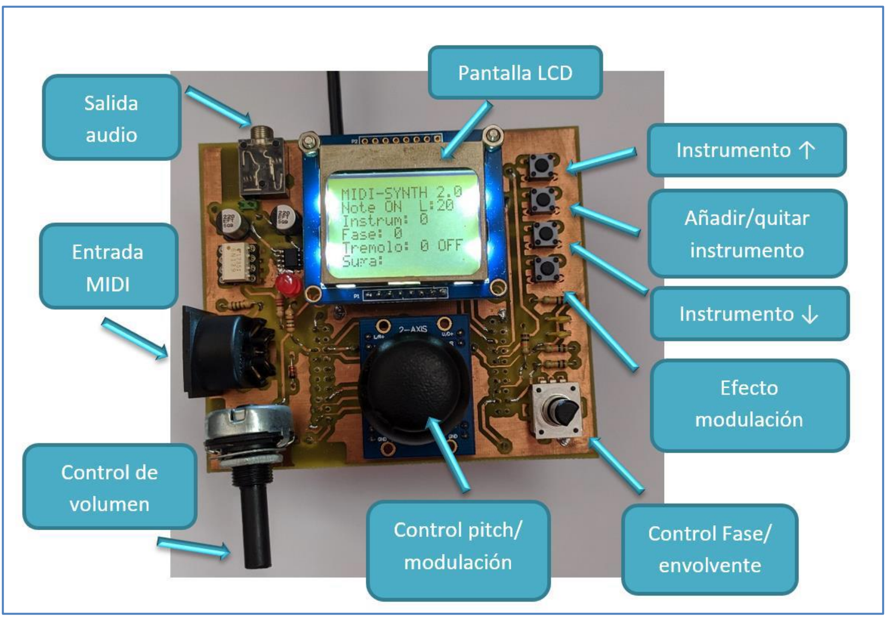


Figura 1-1. Periféricos de control del Midisynt 2.0.

Los elementos analógicos y digitales de entrada/salida al MCU eran los siguientes:

- **Conector hembra MIDI:** para la recepción de la secuencia de notas a reproducir mediante protocolo de comunicación MIDI.
- **Conector hembra tipo Jack:** salida de audio previamente tratada a través de un amplificador modelo TS482IDT en modo seguidor de tensión integrado asimismo en la placa.
- **Pantalla LCD monocroma:** la pantalla Nokia5110 de 48x48 píxeles funciona como interfaz gráfica para permitir el control del sintetizador por parte del usuario, conectada al MCU por puerto SPI (*Serial Peripheral Interface*), protocolo de comunicación serie síncrona.
- **Pulsadores:** controlan la selección de instrumentos para la síntesis aditiva así como el tipo de modulación activa.
- **Encoder rotativo:** la información sobre el giro recibida por sus pines tipo GPIO configuran el desfase asociado a las formas de onda propias de cada instrumento activo en la síntesis sonora.
- **Potenciómetro logarítmico:** el giro de este periférico se traduce en un divisor de tensión que regula el volumen del sintetizador.
- **Joystick:** se conecta al convertidor analógico/digital (*ADC*) del microcontrolador, el cual lee los valores de tensión asociados a cada eje y los traduce en valores digitales para la activación de la modulación (eje X) y la generación del efecto *pitch* (eje Y).

1.2 Etapas del proyecto

Desde un primer momento hemos buscado realizar en este trabajo una iteración más del sintetizador Midisynth referido anteriormente, tratando de buscar alternativas hardward para su implementación con la intención de mejorar las prestaciones del dispositivo y plantear alguna aplicación novedosa para el mismo.

Inicialmente surgió la idea de programar el Midisynth sobre la placa WiPy 3.0, plataforma de desarrollo IoT (*Internet of Things*) basada en un microcontrolador expressif ESP32 de doble núcleo, que incorpora conexión WiFi y Bluetooth. Esta placa funciona sobre el firmware Micropython, implementación del lenguaje de programación Python 3 optimizada para poder ejecutarse en microcontroladores [1]. Abriríamos así una línea novedosa de trabajo, programando en este lenguaje de alto nivel que se encuentra tan extendido en nuestros días en el sector de la industria y la ciencia.

Figura 1-2. Placa WiPy 3.0

Se planteaba además la posibilidad de replicar el sintetizador en varias de estas placas de bajo consumo, de modo que cada una de ellas reprodujera únicamente un tipo de timbre o forma de onda característica (en la version Midisynth 2.0 el programa ofrecía la posibilidad de seleccionar entre 8 timbres o instrumentos distintos). Aprovechando las prestaciones WiFi y Bluetooth de las nuevas placas podría lograrse la conexión de las mismas con un dispositivo “maestro” que enviara una partitura como mensajes MIDI a los distintos sintetizadores WiPy y coordinara la interpretación de la pieza como si de una orquesta se tratase.

Las dos etapas del proyecto serían: por un lado la programación del sintetizador Midisynth sobre la WiPy, traduciéndolo a lenguaje Micropython; y a continuación la implementación de las conexiones WiFi/Bluetooth entre los dispositivos “maestro-esclavo”.

Sin embargo, tras el estudio de este microcontrolador y la realización de numerosas pruebas de programación sobre el mismo, se tuvo que descartar su uso en esta aplicación, por no contar con la velocidad de ejecución suficiente para la implementación del sintetizador.

De modo que abrimos una nueva línea de trabajo, seleccionando como segunda opción el microcontrolador del kit SK-S7G2 de Renesas Synergy, el cual incorpora como lenguaje de programación el estándar C, con el que ya estábamos familiarizados gracias al proyecto Midisynth 2.0. No obstante, acometer el proyecto desde cero sobre el nuevo dispositivo no ha sido tarea sencilla. Ello ha supuesto en primer lugar el estudio del funcionamiento de su complejo sistema de módulos, cuya configuración en muchos casos no es trivial, para desarrollar a continuación, de forma progresiva, las funciones del sintetizador sobre el mismo. Se ha seguido un planteamiento conservador en la incorporación de las distintas prestaciones, siendo necesario realizar numerosas versiones del programa.

1.3 Alcance

Una vez seleccionado el nuevo microcontrolador para la implementación del sintetizador Midisynth 3.0 el grueso del trabajo ha sido lograr las mismas prestaciones de la versión Midisynth 2.0 para el nuevo dispositivo. Dadas las prestaciones superiores del SK-S7G2 frente al microcontrolador de la versión anterior se abre un abanico de posibilidades para la implementación de mejoras en nuestra aplicación.

El hecho de incorporar numerosos periféricos en la propia placa, como la pantalla LCD táctil y el panel de pulsadores, ya supone un avance considerable respecto a la versión anterior: para incorporar los periféricos del sintetizador Midisynth 2.0 (pantalla LCD, joystick, encoder, conector jack, etc) fue necesario diseñar y fabricar una placa de adaptación PCB específica para el microcontrolador en cuestión. Además de ahorrarnos dicho trabajo, en esta ocasión obtendremos un dispositivo más compacto e integrado, con una interfaz más intuitiva, gracias a la novedosa pantalla táctil LCD a color.

Por otro lado, la placa SK-S7G2 incorpora como novedad aritmética de punto flotante, con lo que lograremos mayor potencia en el cálculo de la onda sonora. Como prestación adicional programaremos un módulo de filtrado, del que carecía la versión Midisynth 2.0, el cual permitirá afinar el timbre de la onda modificando los armónicos de la misma[1].

1.4 Estructura del documento

En el presente texto se pretende plasmar el trabajo de programación realizado para la creación del sintetizador Midisynth 3.0, junto a la explicación de los principios teóricos en los que se basa el proyecto.

Tras un capítulo introductorio sobre algunos conceptos de la síntesis de audio y el estándar de comunicaciones MIDI empleado en este campo, se detalla en el tercer capítulo el proceso de búsqueda de alternativas hardware para la implementación de nuestro sintetizador. En dicho capítulo se desarrolla el estudio efectuado sobre el microcontrolador de la placa IoT WiPy 3.0, descartado tras una serie de pruebas, y la selección final de la plataforma de desarrollo Renesas Synergy SK-S7G2 para la aplicación que buscábamos.

A continuación, el cuarto capítulo detalla las herramientas software de las que se ha hecho uso en el proyecto. El grueso de esta memoria se contiene en el capítulo 5, en el cual se ilustra la programación de la placa, exponiendo en primer lugar las especificaciones que se esperan del sintetizador desarrollado. Seguidamente se explica el proceso de configuración de los distintos periféricos empleados, así como la estructura del código y las funciones ejecutadas en cada punto del programa.

Finalmente, en los capítulos 6 y 7, se muestran los resultados obtenidos y se comentan varias conclusiones sobre el trabajo realizado. En los Anexos I y II, se recogen los códigos desarrollados para la placa WiPy y la placa SK-S7G2, respectivamente.

[1] Este concepto se ilustra en el siguiente capítulo.

2 FUNDAMENTOS TEÓRICOS

Music is nothing else but wild sounds
civilized into time and tune.

-Thomas Fuller-

En este capítulo ilustraremos el funcionamiento del sintetizador de audio digital, detallando sus principales elementos así como las posibilidades que ofrece para articular las ondas generadas y crear un amplio abanico de sonidos. Asimismo, se explicará el funcionamiento del protocolo de comunicaciones MIDI, estándar entre dispositivos electrónicos de producción musical. Muchos de estos conceptos ya fueron recogidos en mi Trabajo Fin de Grado [2] por lo que me limitaré a dar unas breves pinceladas sobre los aspectos básicos necesarios para la comprensión del texto, remitiendo al lector al referido documento si desea profundizar en estos temas.

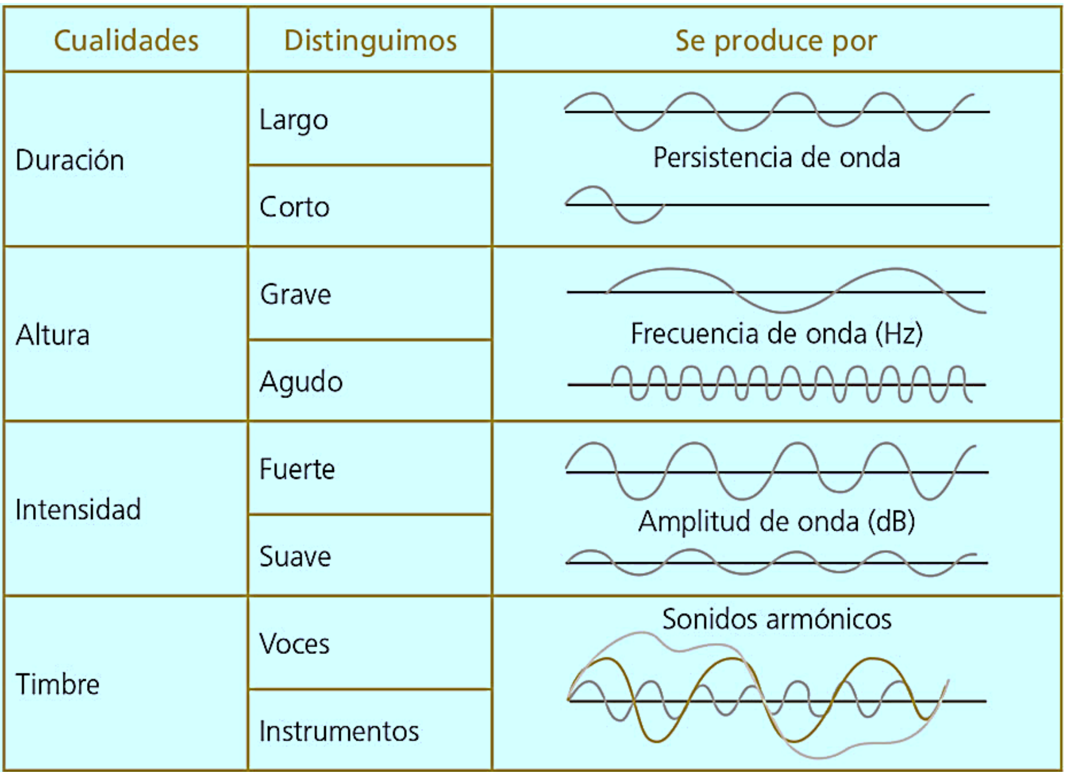
2.1 Introducción a la síntesis de audio

El sonido es el conjunto de vibraciones u ondas sonoras que se propagan en el aire hasta ser percibidas por nuestros oídos. Existen **cuatro características** en dichas ondas que nos permiten distinguir unos sonidos de otros: **la duración, la altura, la intensidad y el timbre.**

La duración es la persistencia de la onda en el tiempo, la cual distingue sonidos largos o cortos; la **altura** grave o aguda de los sonidos depende de la frecuencia de la onda, es decir, del número de veces que ésta se repite en el tiempo; la intensidad suave o fuerte de un sonido viene condicionada por la amplitude de la onda; y por ultimo, el timbre se relaciona con la forma de la onda.

Esta última nota característica varía según la fuente que origina la vibración: la vibración de nuestras cuerdas vocales, de un instrumento o del golpeo de cierto objeto darán lugar a ondas con distintos armónicos. De este modo cada instrumento posee su sonido particular, e incluso dentro de aquellos del mismo tipo, ninguno llega a sonar de forma idéntica, al igual que las voces humanas se distinguen unas de otras.

Se atribuye el adjetivo **"sintético"** a *aquellos productos que se obtienen por procedimientos industriales y que reproducen la composición y propiedades de uno natural* [3]. En el campo de estudio que nos ocupa aplicamos esta definición al **sonido sintético**, mediante el cual se trata de imitar las ondas sonoras reales, manipulando los cuatro rasgos distintivos que acabamos de describir: frecuencia, amplitud, duración y armónicos (timbre). [2]



Cualidades	Distinguimos	Se produce por
Duración	Largo	Persistencia de onda
	Corto	
Altura	Grave	Frecuencia de onda (Hz)
	Agudo	
Intensidad	Fuerte	Amplitud de onda (dB)
	Suave	
Timbre	Voces	Sonidos armónicos
	Instrumentos	

Tabla 2-1. Características del sonido[2].

2.1.1 Módulos del sintetizador

Un **sintetizador se compone de cuatro elementos** básicos para la generación de ondas básicas y su modificación progresiva hasta asemejarlas a las ondas naturales que trata de reproducir. Estos módulos se describen brevemente a continuación:

- **Oscilador principal:** constituye la primera etapa del proceso de síntesis, responsable de generar la onda básica que será modificada en módulos posteriores del sintetizador. Dicha onda se conoce como onda portadora o "carrier". El oscilador puede ser de dos tipos: *VCO ("Voltage Controlled Oscillator")* si produce valores de tensión, o *DCO ("Digitally Controlled Oscillator")* si genera la onda en forma de valores digitales que finalmente serán convertidos en tension por medio de un DAC. En un sintetizador puede haber uno o varios osciladores principales, cuyos sonidos se unirán en un modulo mezclador.

- **Oscilador de baja frecuencia:** genera la onda llamada "envolvente" que modifica la onda portadora mediante el proceso que conocemos como modulación. Su frecuencia tiene órdenes de magnitud menores que la onda principal. Este módulo de modificación puede situarse en cualquier punto del proceso de síntesis: justo después de cada DCO en caso de existir varios osciladores, tras la etapa de mezcla, o a la salida de los filtros.

[2] Imagen tomada de https://www.correodelmaestro.com/publico/html5022018/capitulo5/autorretrato_sonoro.html. Fecha de consulta: Diciembre de 2019.

En la figura se ilustran la onda envolvente, la onda de alta frecuencia o portadora, y en tercer lugar la onda resultante de la modulación:

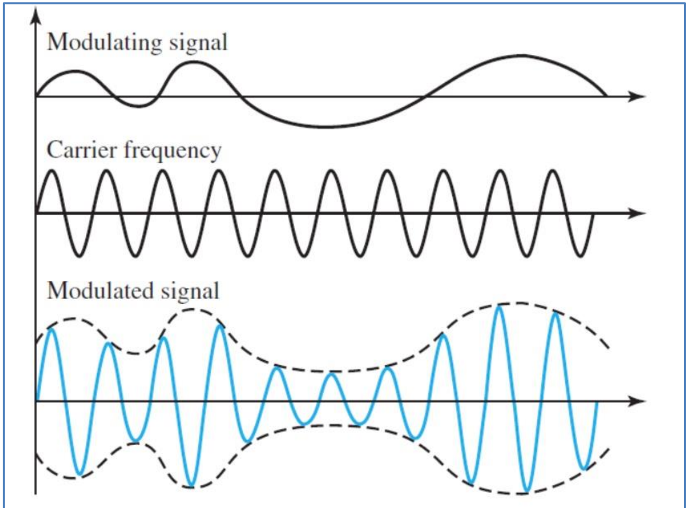


Figura 2-1. Modificación de la onda mediante modulación.

- **Filtro:** su papel es el de variar los ármónicos de la onda, de modo que se altere su timbre. También puede haber varios módulos de filtrado, antes y/o después de la etapa de mezcla.

- **Amplificador:** se trata de la última etapa del proceso, en la que la onda final ya perfilada es amplificada para regular su volumen (intensidad). Podrá ser analógico o digital, según si la onda ya toma valores de tensión o deba ser convertida en analógica por medio de un DAC colocado a la salida del amplificador.

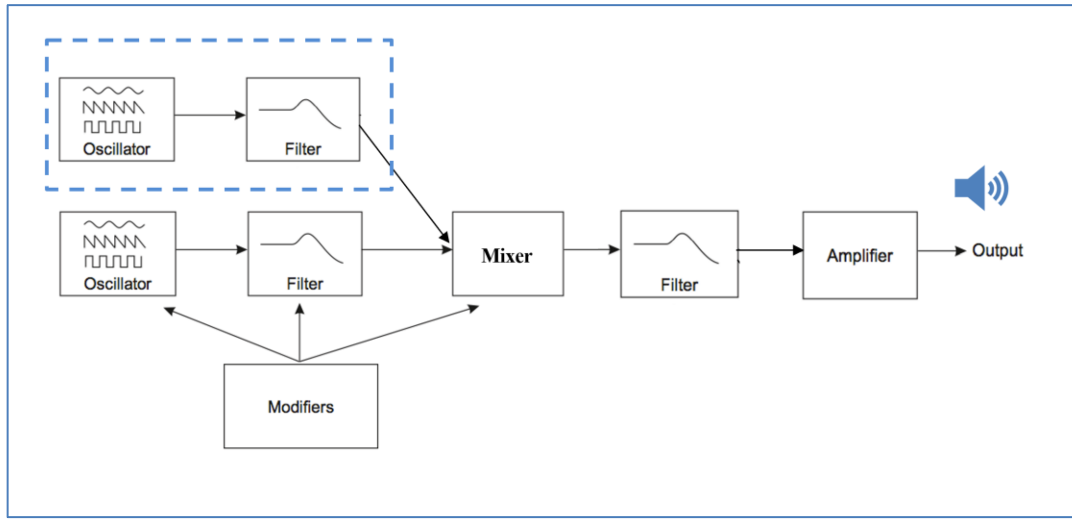


Figura 2-2. Módulos de un sintetizador[3].

[3] Los módulos encuadrados pueden repetirse n veces. Los filtros también son opcionales y su posición en el flujo es variable.

2.1.2 Tipos de modulación

Como hemos comentado, la onda en sus distintas fases del proceso de síntesis puede ser modificada mediante una etapa de modulación. En función de cómo afecte la onda envolvente a la onda modulada, se distinguen varios tipos de modulación:

- **Modulación en amplitud, AM** ("Amplitude Modulation"): la amplitud de la onda portadora es modificada periódicamente en función de la amplitud de la onda envolvente.

- **Modulación en frecuencia, FM** ("Frequency Modulation"): la frecuencia de la onda portadora evoluciona según la amplitud de la onda envolvente. El "índice de frecuencia" será el parámetro que relaciona la variación de la amplitude envolvente y la frecuencia de la onda moduladora. Como la frecuencia de la onda final será igual a la amplitud de la envolvente el ratio también puede verse como la relación entre sendas frecuencias:

$$\boldsymbol{I = \frac{\Delta\, Frecuencia\ onda\ modulada}{Frecuencia\ envolvente}}$$

  Si la variación de la frecuencia se expresa como $\boldsymbol{\Delta f = f_{carrier} + \Delta}$ siendo $\boldsymbol{\Delta}$ un término fijo que depende de la amplitud máxima de la envolvente, y dado que la envolvente se genera mediante un oscilador de menor frecuencia que la carrier, los índices de modulación típicosserán **I > 1.**

- **Modulación del filtro:** en este caso los parámetros del filtro son dependientes de los valores que toma la envolvente a cada instante.

  En la figura se ilustran de izquierda a derecha las modulaciones que acabamos de describir: AM, FM y modulación del filtro.

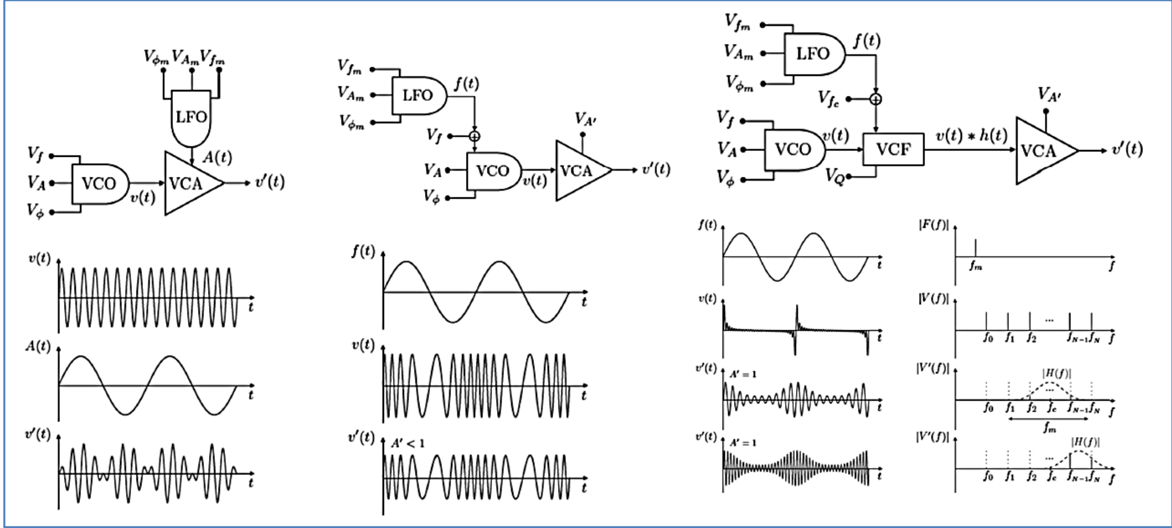


Figura 2-3. Principales tipos de modulación.

Dentro de cada tipo de modulación encontramos las siguientes variantes o efectos:

- **Trémolo:** modulación tipo **AM** con onda envolvente periódica, generalmente senoidal.

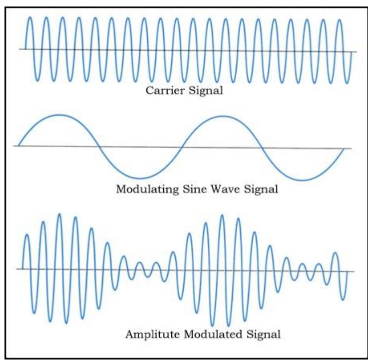


Figura 2-4. Efecto trémolo.

- **ADSR: caso especial de modulación AM** en la que la envolvente tiene una forma característica, ilustrada en la siguiente figura, con cuatro fases que varían según los parámetros A-D-S-R:
    - A ("Atack"): duración de la fase de ataque en la cual la amplitud aumenta.
    - D ("Decay"): duración de la fase en la que disminuye la amplitude.
    - S ("Sustain"): valor en porcentaje de la amplitud máxima (pico alcanzado en la fase de ataque) que tomará la amplitud durante la fase de sostenido.
    - R ("Release"): duración de la última etapa, en la cual la amplitud desciende finalmente hasta 0, "apagando" de esta forma el sonido.

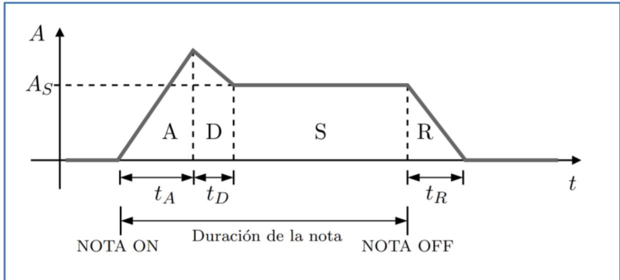


Figura 2-5. Fases de la envolvente ADSR.

El efecto ADSR difiere del trémolo en el hecho de que la forma de **la envolvente no se repite periódicamente, sino que ocupa el lapso de tiempo que dura cada nota** o sonido interpretado desde que es pulsada (señal nota ON) hasta poco después de ser liberada (señal nota OFF), de modo que en la fase *Release* se imita el eco o efecto de mitigación del sonido hasta que este desaparece. De forma similar, las fases de ataque y descenso emulan la fuerza inicial con las que son interpretadas ("atacadas") las notas.

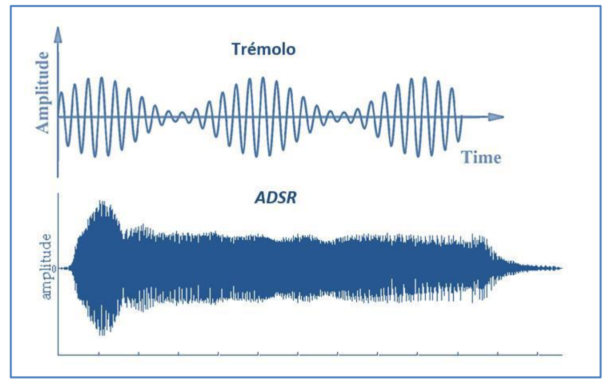


Figura 2-6. Trémolo vs. ADSR.

- **Wah-wah:** se denomina generalmente así a la modulación que afecta a los filtros del sintetizador.
- **Vibrato:** modulación tipo **FM** con onda envolvente periódica de baja frecuencia (LFO).

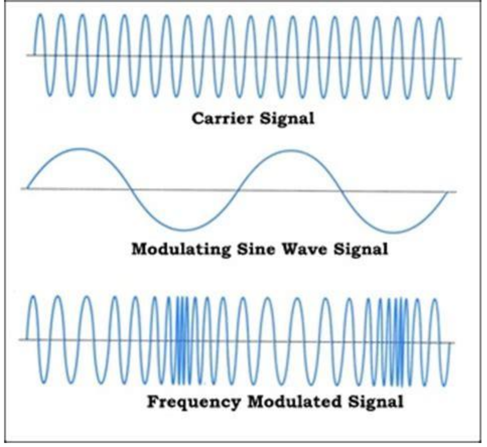


Figura 2-7. Efecto vibrato.

- **FM**: aunque recibe el mismo nombre que la modulación FM genérica, se trata de un caso especial en el cual la frecuencia de la envolvente toma valores próximos a la de la onda carrier, de forma que el oscilador secundario ya no es un LFO sino otro DCO.

  Esto resulta en índices de modulación próximos o menores que 1, y podrá provocar frecuencias instantáneas negativas en la onda modulada, que se traducirán en frecuencias positivas con cambios de fase de 180º. [4]

  Dichos cambios de fase conllevan una fuerte dependencia con la envolvente ya no solo en términos de frecuencia, sino también de amplitud, como puede observarse en el caso C de la siguiente figura:

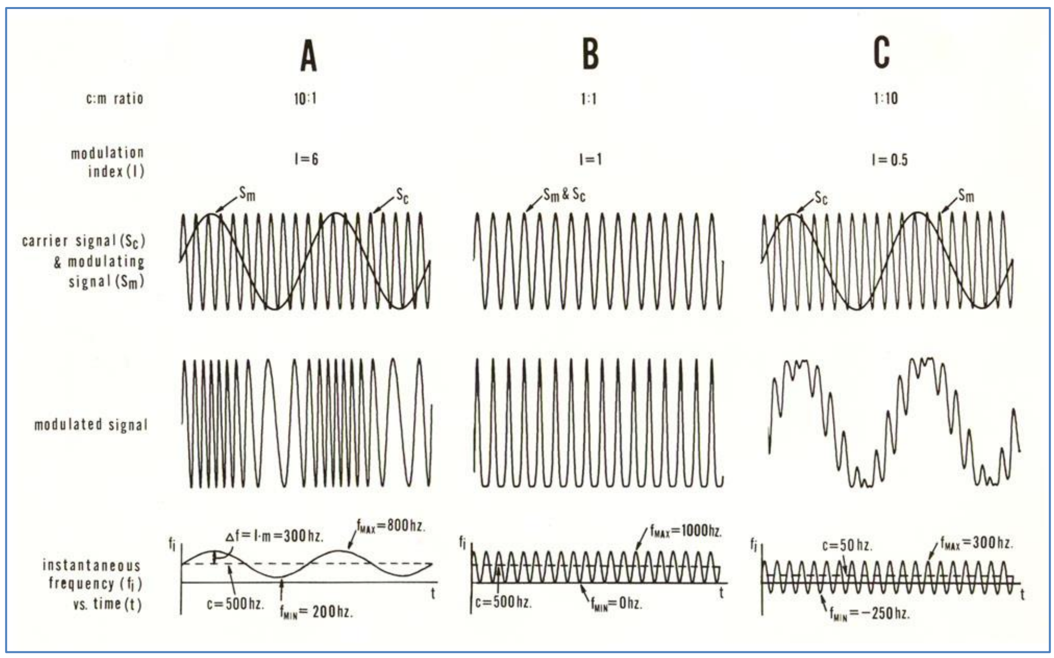


Figura 2-8. Efectos de modulación FM en función del Índice de modulación.

2.2 El estándar MIDI

MIDI es un estándar tecnológico que permite que distintos instrumentos musicales electrónicos, ordenadores y otros dispositivos comunicarse entre sí [4]. Utiliza un protocolo de comunicaciones puerto serie asíncrono, similar al UART, pero a una velocidad específica de 31.25 kbps. Sin embargo el MIDI no es simplemente un protocolo sino que comporta una interfaz digital de codificación de mensajes, además de cables de conexión propios: conectores de 5 clavijas tipo DIN.

Figura 2-9. Conectores MIDI hembra y macho.

La comunicación MIDI se basa en un sistema de mensajes cuya unidad es el **MIDI-byte, agrupación de 10 bits:** 1 bit de estado, 8 bits de datos y 1 bit de terminación. Los MIDI-bytes pueden ser de dos tipos: **bytes de estado o bytes de datos, distinguiéndose entre sí por el bit más significativo (MSB,** *Most Significant Bit*) que es igual a 1 en el bit de estado, y 0 en el bit de datos.

Un mensaje MIDI se compone de 3 MIDI-bytes, 1 byte de estado y 2 de datos. El útimo byte de datos es opcional de forma que el mensaje completo puede formarse con solo 2 MIDI-bytes.

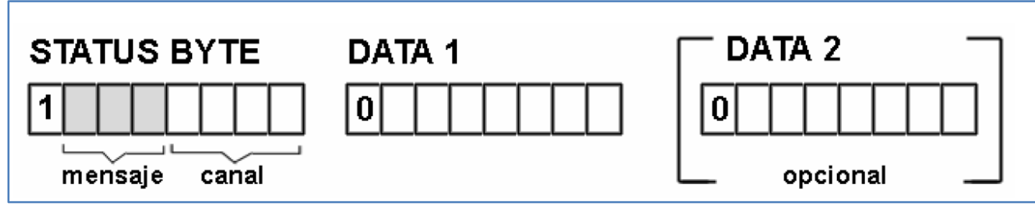


Figura 2-10. Estructura del mensaje MIDI.

El status-byte indica el tipo de mensaje y el canal al que va dirigido. Como hemos comentado el primer bit (MSB) será igual a 1, de forma que se cuenta con 3 bits para codificar $2^3 = 8$ tipos de mensajes y con 4 bits para direccionar $2^4 = 16$ canales distintos.

Entre los 8 tipos de mensajes codificables, los más importantes son los llamados *"note-on"* y *"note-off",* los cuales son generados por el controlador MIDI cuando se pulsa o se libera una nota. Los dos bits de datos que forman estos mensajes contienen información sobre la altura de la nota y la fuerza con la que debe ser reproducida (intensidad). Dichos mensajes básicos son los que contemplaremos en este proyecto, los cuales serán enviados al canal 0, por tanto el status byte tomará valores: "1001 0000" ó "1000 0000".

Tipo de mensaje	Status-Byte	Data 1	Data 2
Note-on	1 **0** **0** **1** n[4] n n n	Altura	Velocidad
Note-off	1 **0** **0** **0** n n n n	Altura	Velocidad

Tabla 2-2. Mensajes tipo *note-on/note-off.*

2.3 El sintetizador Midisynth 3.0

El sintetizador Midisynth 3.0 que implementamos en este proyecto será un **sintetizador digital aditivo por tabla de ondas**, al igual que su version anterior. Esto implica que contaremos con varios osciladores principales de tipo DCO (digital) cuyas ondas generadas se sumarán (aditivo) para articular la onda final. El oscilador leerá los puntos de la onda digital de unas tablas almacenadas en la memoria del microcontrolador.

[4] Las siglas nnnn indican los 4 bits para la codificación del canal.

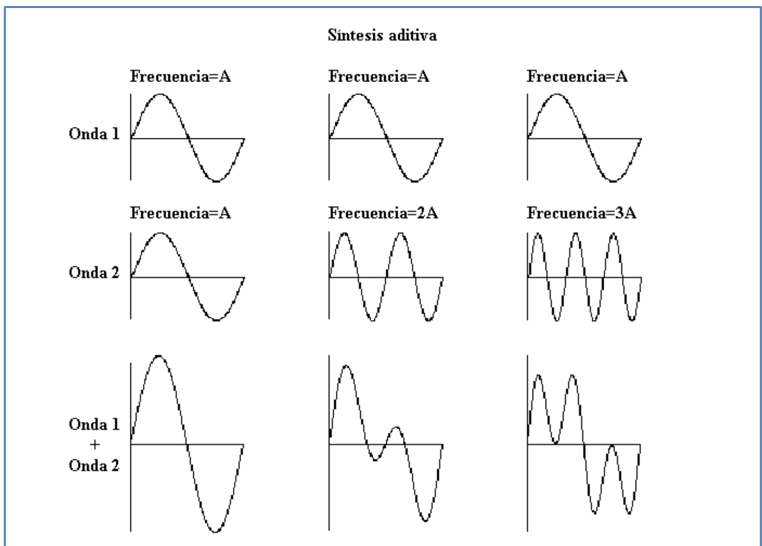


Figura 2-11. Método de síntesis aditiva[5].

Los osciladores principales trabajarán a **frecuencias características de 20kHz**, es decir, con periodos de trabajo **T = 1/20000Hz = 50 us**. Cada 50 microsegundos los DCO leerán un punto de la tabla de ondas, generando así las portadoras que se sumarán a continuación. Denotaremos a los osciladores como "instrumentos" puesto que cada uno de ellos leerá de la tabla una forma de onda distinta, asociada al timbre de un instrumento determinado. Podrán seleccionarse hasta 8 instrumentos para conformar la onda final fruto de la síntesis aditiva.

Una vez sumadas las ondas de los instrumentos activos, la onda final será **modificada por el módulo LFO**, en caso de estar la modulación activa, a frecuencias de 100kHz, esto es, **cada 100 milisegundos.** El sintetizador contará con 4 efectos de modulación posibles, de los descritos anteriormente: trémolo, vibrato, FM y ADSR.

Adicionalmente, se ha proyectado incorporar un filtro "pasa-banda" a la salida del modulo LFO, previo a la etapa de amplificación. Los parámetros de entrada serán la frecuencia de resonancia, f_0, el ancho de banda, w, y la ganancia, G.

En la figura se ilustra la respuesta en frecuencia del filtro: en el eje X la frecuencia en escala logarítmica, y en el eje Y la función de transferencia en decibelios.

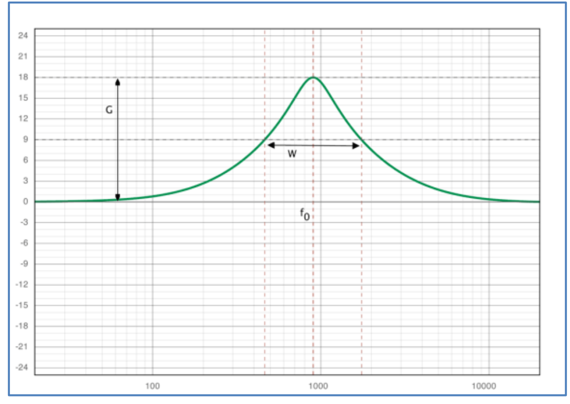


Figura 2-12. Parámetros del filtro pasa-banda [6].

Sin embargo, finalmente no se ha logrado el correcto funcionamiento de este módulo por lo que hemos optado por descartarlo en esta versión del sintetizador, quedando esta línea abierta para trabajo futuro.

[5] Imagen tomada de https://www.ehu.eus/acustica/espanol/musica/ineles/ineles.html. Fecha de consulta: Junio de 2022.

3 BÚSQUEDA DE ALTERNATIVAS HARDWARE

Los experimentos fallidos forman parte del proceso en igual medida que el experimento que funciona bien.

- Benjamin Franklin –

La idea inicial de este proyecto ha sido la implementación del sintetizador Midisynth 2.0, que desarrollé en el curso 2019-2020 para mi Trabajo Fin de Grado, utilizando en esta ocasión la placa de desarrollo Wipy, añadiendo asimismo nuevas prestaciones. Sin embargo, una vez exploradas las posibilidades de este dispositivo, se ha tenido que descartar su uso en esta aplicación, por no contar con la velocidad de ejecución necesaria para el desarrollo en tiempo real del sintetizador. El microcontrolador seleccionado finalmente para la creación de Midisynth 3.0 ha sido el SK-S7G2 de Renesas Synergy.

3.1 Placa de Desarrollo WiPy 3.0

Como ya comentamos en el capítulo introductorio, la placa WiPy 3.0 es una plataforma de desarrollo IoT (*Internet of Things*) basada en el microcontrolador expressif ESP32 de doble núcleo, que incorpora conexión WiFi y Bluetooth. El chip puede operar a velocidades de hasta 240MHz.

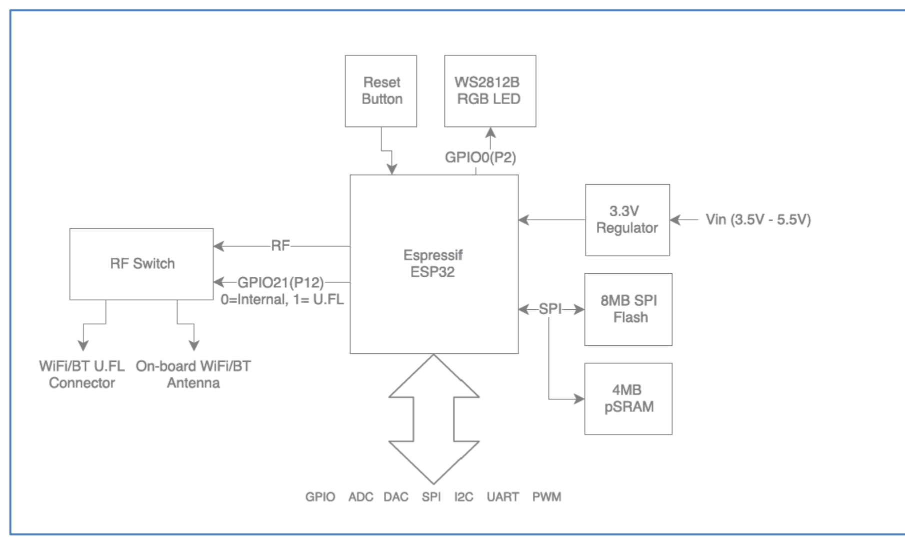


Figura 3-1. Diagrama de bloques de la WiPy 3.0 [7].

Las prestaciones más relevantes de este dispositivo se muestran en la tabla siguiente:

CPU	Microprocesador Xtensa® dual‑core 32‑bit.
	Procesador secundario *ULP* para el control de periféricos y GPIOs en modo de bajo consumo, a solo ~25uA.
	Sistema de cómputo en “punto flotante”.
	Sistema Python multi-hilo.
Memoria	RAM: 520KB *on-chip* + 4MB *(External QSPI RAM).*
	External flash: 8MB.
WiFi	802.11b/g/n 16mbps.
Bluetooth	Modo clásico y modo de bajo consumo *(BLE).*
Reloj	*RTC* a 150kHz.

Figura 3-2. Especificaciones de la placa WiPy 3.0.

Además cuenta con una variedad de elementos periféricos interesante para nuestra aplicación, entre otros, tres puertos serie UART, dos canales I2C y dos tipo SPI, salida PWM, temporizadores, convertidores ADC y DAC, y lector de tarjetas SD.

La WiPy 3.0 se monta sobre una placa de adaptación que permite la conectividad puerto serie UART con el puerto USB de un ordenador para cargar el código en el microcontrolador. Incorpora varios pulsadores y leds con los que trabajamos inicialmente implementando los primeros proyectos de prueba. Además integra una ranura para tarjetas microSD, que también hemos utilizado para extraer los datos de las ondas trazadas en el programa, para pruebas alternativas al uso de la salida *DAC*.

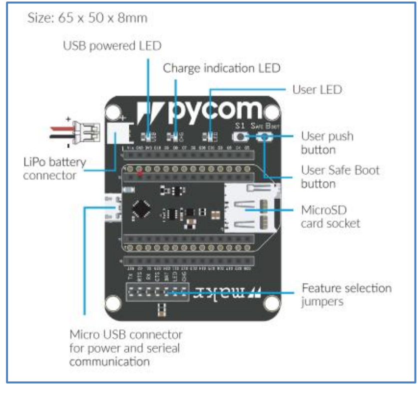


Figura 3-3. Expansion Board 3.0 [8].

La WiPy tiene habilitado el lenguaje Micropython, versión de Python adaptada para el uso en microcontroladores. El código se ejecuta utilizando el plugin PyMakr para Atom, software que ilustraremos en el capítulo 4. Una vez instalado el programa, lo abrimos y pulsamos CTRL+ALT+P para desplegar la barra de búsqueda; accedemos a la opción “Install Packages and Themes” y tecleamos “Pymakr” en el buscador de paquetes.

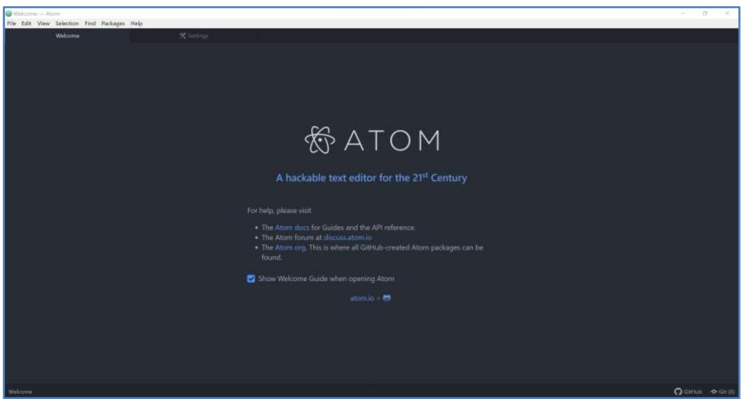


Figura 3-4. Ventana de inicio ATOM.

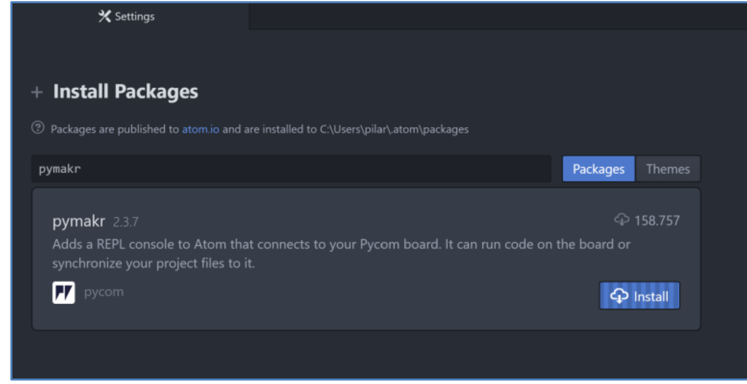


Figura 3-5. Instalar plugin Pymakr.

Si el paquete se ha instalado correctamente la placa se enlazará automáticamente con la consola REPL una vez la conectemos al portátil via USB.

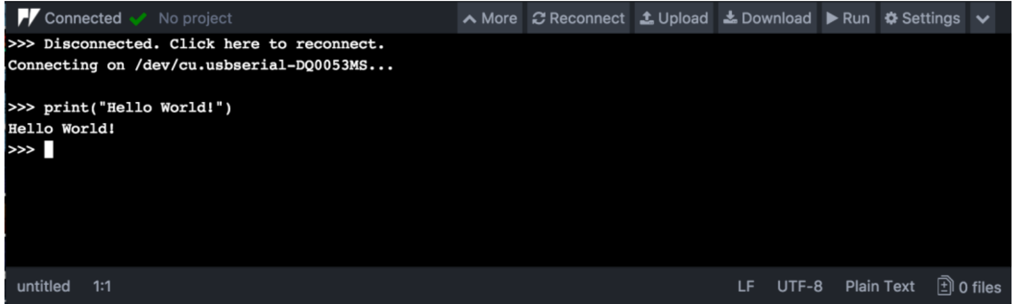


Figura 3-6. Consola REPL de ATOM.

Para comenzar a trabajar en ATOM debemos crear una carpeta de proyecto en nuestro PC en la cual se irán almacenando los distintos archivos de extensión .py, propios del software Micropython. Antes de comenzar la implementación del sintetizador, realizamos el conjunto de tutoriales recogidos en la web https://docs.pycom.io/tutorials/ para familiarizarnos con el entorno de programación y los módulos de Micropython.

Con los primeros proyectos comenzamos a manejar los GPIOs a través de los pulsadores y LEDs de la placa, logramos establecer una conexión WiFi, creamos un servidor FTP, probamos la conexión SPI, los convertidores ADC y DAC, los módulos de temporización (*timers* y *chrono*), y el sistema de archivos para lectura/escritura en la tarjeta SD.

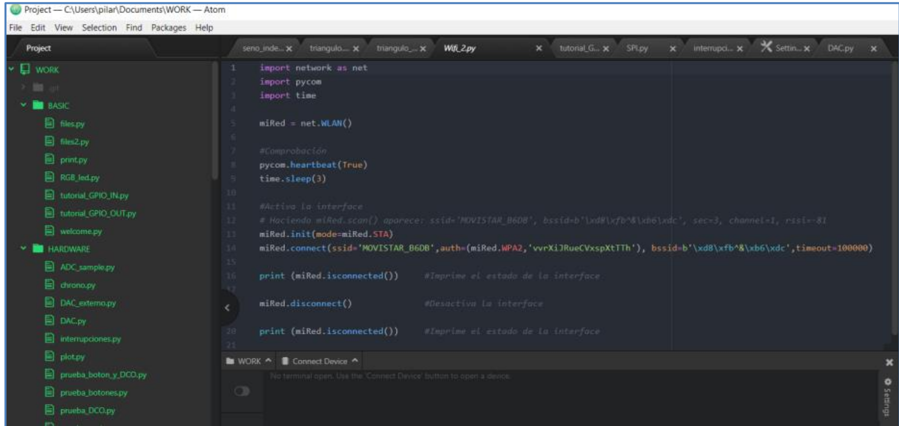


Figura 3-7. Testeo inicial del entorno ATOM.

Una vez exploradas todas las posibilidades del dispositivo, ya manejábamos todas las herramientas necesarias para crear nuestro sintetizador. No incluiremos todos estos códigos iniciales por no extender demasiado el presente texto.

3.1.1 Interrupciones

La primera prueba a realizar con la placa Wipy fue la generación de interrupciones temporizadas cada 50 microsegundos, es decir frecuencias de 20kHz, las cuales son necesarias para emular el oscilador principal del sintetizador (*DCO*). Mediante la clase *Timer* del módulo *Machine* ejecutamos el siguiente código:

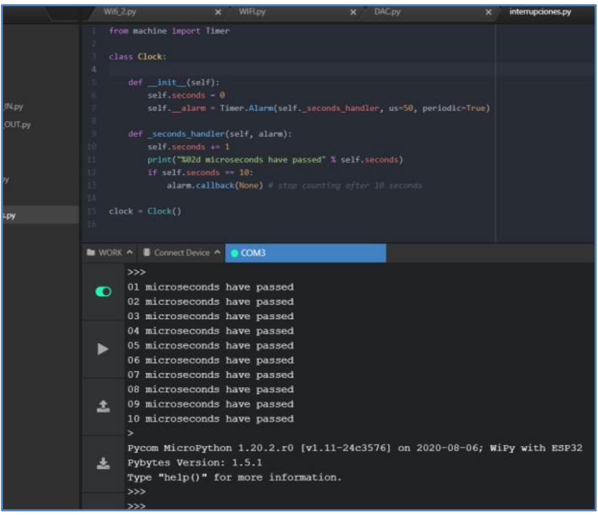


Figura 3-8. Pruebas con el Timer.

3.1.2 Testeo del DAC integrado

Para tener una constatación más real de la temporización, la siguiente prueba fue tratar de generar mediante interrupciones una senoide con una resolución de 20kHz y frecuencia audible, entre 200Hz y 2kHz. Los puntos de la onda se leerían de una tabla de valores enteros almacenada en el programa principal, para ejecutar el menor número de instrucciones posible dentro de la interrupción y alcanzar así la velocidad deseada. La forma de onda se debería poder observar mediante un osciloscopio, extrayendo la señal a través del convertidor DAC.

Antes de programar el DAC, tratamos de escribir la tabla de ondas en un archivo .txt dentro de la SD, la cual leemos a continuación directamente desde el PC y ploteamos mediante un script de Python clásico. Esto era necesario porque Micropython no incorpora el módulo "plot", ni muchos otros propios de Python, lo cual hizo bastante tedioso el trabajo dado que las alternativas a muchas funciones de Python eran muy limitadas, y debimos crearlas como funciones de usuario.

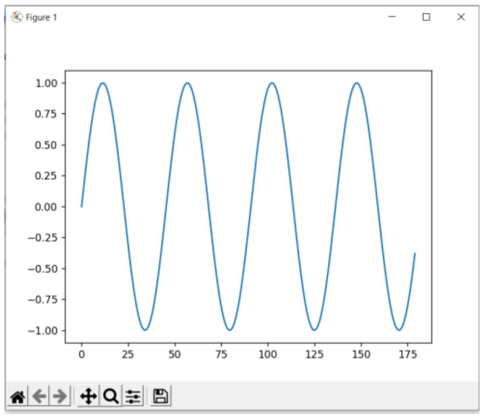


Figura 3-9.Ploteo de senoidal básica en Python.

A continuación, configuramos el DAC para extraer la onda generada y comprobar la frecuencia de las interrupciones. Observando el resultado en el osciloscopio, con una escala de tiempos de 20ms por división, se plotean aproximadamente unos 10 escalones por división, es decir, tendríamos una resolución de unos 2 milisegundos, bastante menor que los 50 microsegundos buscados.

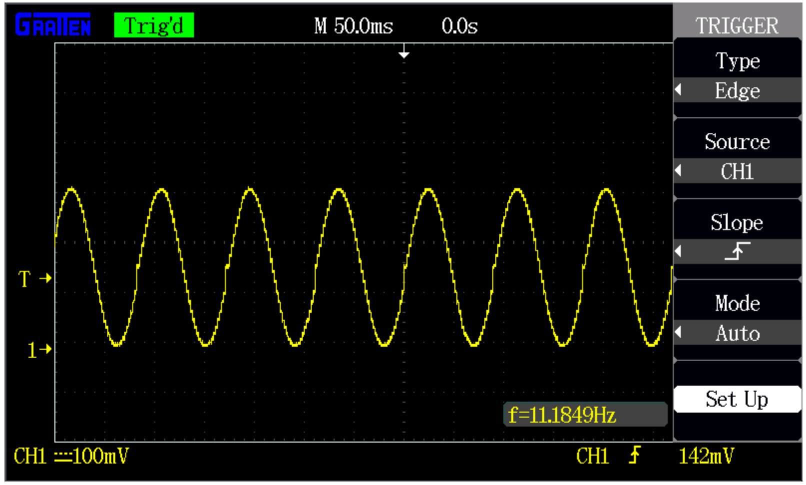


Figura 3-10. Generación de senoidal mediante el DAC integrado.

Otro problema que encontramos fue el hecho de que el MCU no era capaz de atender otra interrupción además de la DCO, la cual tratábamos de generar para la lectura de un pulsador. Posteriormente sería necesaria la generación de interrupciones adicionales, para la lectura de mensajes MIDI y otros periféricos del sintetizador, con lo cual los obstáculos de la WiPy iban *in crescendo.*

Realizando una serie de mejoras, entre otras reducir el número de instrucciones dentro de la interrupción al mínimo, y eliminando el uso de la función "print()" dentro de la misma[6], logramos una reducción del tiempo característico, pero no alcanzamos el objetivo de 50us de periodo. El tiempo de muestreo en esta prueba fue de unos 200us.

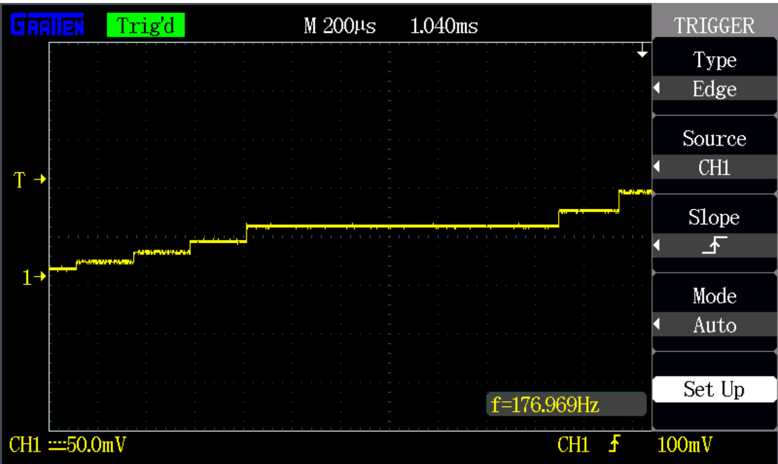


Figura 3-11. Otras pruebas.

En este punto, desconocíamos si la ralentización del programa se debía al funcionamiento del DAC o a la propia velocidad de ejecución de instrucciones por parte del intérprete de Micropython. Como prueba final, creamos un programa sencillo que enviaba una onda triangular al DAC, sin pausas ni interrupciones: un bucle tipo *while (1)* en el que se incrementaba una variable que enviábamos a la salida del DAC, reseteándola cuando desbordase. Se trataba de observar la salida del DAC por el osciloscopio una vez más para comprobar el tamaño de los escalones, que debiera ser del orden del tiempo de ejecución de dicha órdenes más el tiempo de procesamiento del DAC. Con este programa de instrucciones tan reducidas podríamos comprobar si el DAC era el culpable del retraso del programa.

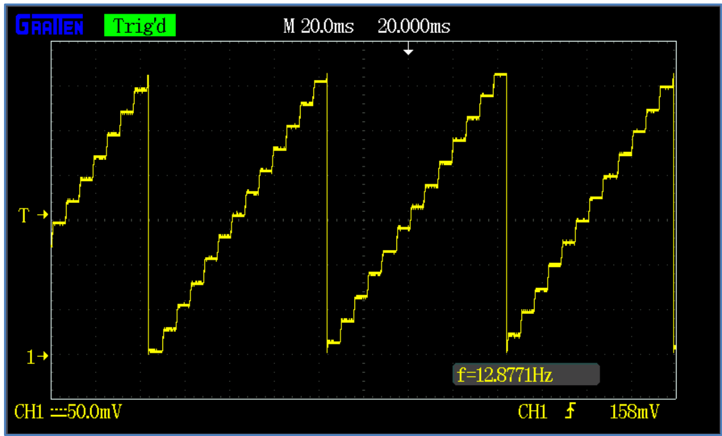


Figura 3-12. Onda triangular generada por el DAC interno.

La onda triangular resultante mostraba escalones del orden del milisegundo, con lo cual queda descartado definitivamente el DAC interno del microcontrolador para la aplicación.

[6] La consola no se debe utilizar en aplicaciones de Tiempo Real críticas, por ralentizar mucho la ejecución del programa.

3.1.3 Pruebas con DAC externo MCP4921

Una vez identificado el DAC integrado como obstáculo para nuestro proyecto, tratamos de incorporar un DAC externo a la placa vía puerto serie SPI. En este caso se trataba del DAC de 12 bits modelo MCP4921, del fabricante Microchip.

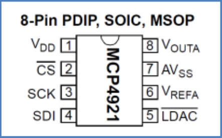


Figura 3-13. Encapsulado del MCP4921 [9].

Alimentamos el chip a 3.3V través de los pines 1 (Vdd) y 6 (Vrefa)y conectamos los pines 7 y 5 a la tierra del MCU. Los pines 2, 3 y 4 se conectan, respectivamente al *Chip Select (CS)*, el reloj y el bus de datos del SPI. Por medio del pin 8 observamos la salida del DAC mediante el osciloscopio. En la imagen se muestran las dos conexiones al osciloscopio, el cable verde para la salida del DAC y el cable marrón para la conexión a tierra.

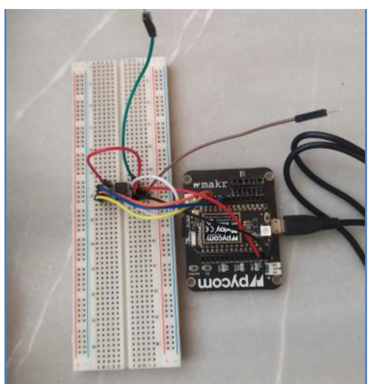


Figura 3-14. Conexiones del DAC externo.

De nuevo tratamos de generar la onda en diente de sierra para comprobar la velocidad del nuevo DAC. Para nuestra decepción, la triangular generada muestra una vez más escalones del orden del milisegundo. En la siguiente captura la señal triangular representa la salida del DAC y la señal amarilla el CS.

Figura 3-15. Señal de salida del DAC y *Chip Select.*

Observamos que el chip select muestra tiempos de permutación demasiado altos, con lo que realizamos como prueba final un bucle while (1) más sencillo aún, en el que únicamente se permuta el valor de este pin:

```
while True:
    cs.value(0)
    cs.value(1)
```

Incluso en este caso el pulso mínimo es de unos 12us, y ni siquiera tiene ancho fijo, tardando a veces casi el doble, como puede verse en la captura siguiente. La señal azul son los pulsos de reloj del SPI, que se generan en grupos de 16 flancos y la señal amarilla representa los flancos del CS:

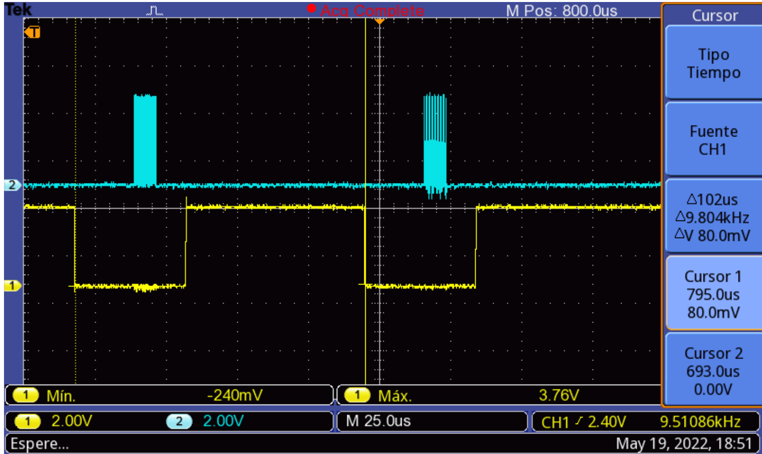


Figura 3-16. Señal SCLK y CS del SPI.

De esto deducimos que el microcontrolador tarda más de 100us en hacer dos operaciones sencillas y enviar el dato por SPI. De modo que si añadiésemos cualquier operación extra en el bucle, obtendríamos tiempos de ejecución que no eran operativos para nuestra aplicación de tiempo real.

3.1.4 Conclusiones

Pese a que inicialmente la placa no parecía imponer limitaciones hardware para la generación de ondas con resolución de 20kHz, el firmware Micropython ha hecho imposible estas operaciones. Ello se debe a que se trata de un intérprete de código, cuyas velocidades de ejecución son más lentas que los compiladores, puesto que ejecutan paso a paso el código fuente en tiempo real, al diferencia de los segundos, que efectúan una traducción a código máquina durante la preejecución del programa. Confirmamos con esto la conocida afirmación del empresario Bruce Craig: "el hardware es lo que hace a una máquina rápida; el software es lo que hace que una máquina rápida se vuelva lenta".

En definitiva, no era posible el uso de la WiPy 3.0 para la ejecución de operaciones con una frecuencia característica de 20kHz, por lo que quedaba descartada su uso para la creación del Midisynth 3.0. El siguiente paso por tanto sería barajar otras opciones de microcontroladores más adaptados a la aplicación que queríamos crear, y finalmente optamos por el kit Renesas S7G2 Synergy, que presentamos en la siguiente sección del capítulo.

Los códigos principales de esta etapa del proyecto se recogen en el Anexo I.

3.2 Plataforma Renesas Synergy™

Una vez descartada la placa WiPy por no contar con velocidades de ejecución suficientes para la aplicación, limitadas por el intérprete de Micropython, se ha optado por continuar el trabajo con el microcontrolador 240MHz Arm® Cortex®-M4, integrado en la placa de desarrollo SK-S7G2 de Renesas Synergy. Esta placa ha permitido finalmente la creación del sintetizador Midisynth 3.0 obteniendo resultados muy satisfactorios.

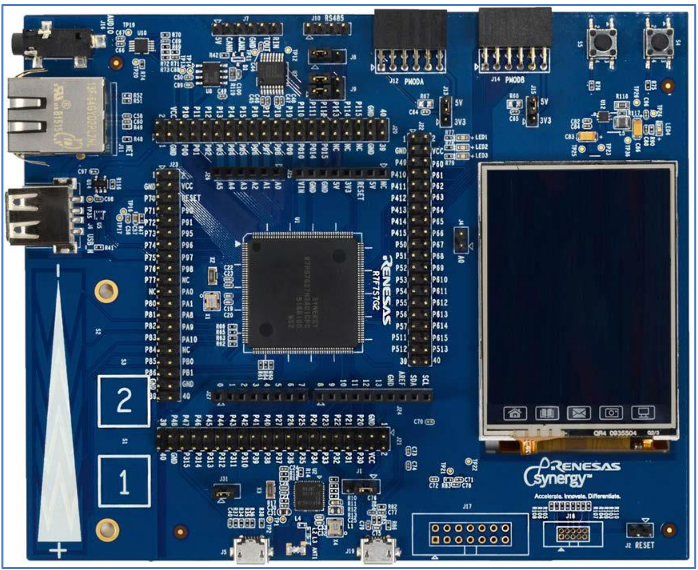


Figura 3-17. Renesas Synergy SK-S7G2.

El MCU del SK-S7G2 utiliza lenguaje estándar C, el cual admite mayores velocidades de ejecución por tratarse de un lenguaje compilado, **a pesar de contar con la misma velocidad del núcleo CPU que la tarjeta WiPy (240MHz).**

Además el microcontrolador es operado por un Sistema Operativo en Tiempo Real (*RTOS*, Real-Time Operating System), ideal para aplicaciones en las que el tiempo de ejecución sea crítico. La principal ventaja de usar los RTOS es su capacidad de ejecutar tareas de forma concurrente. Es decir, al mismo tiempo, lo cual permitirá manejar los recursos de una forma más eficiente [9].

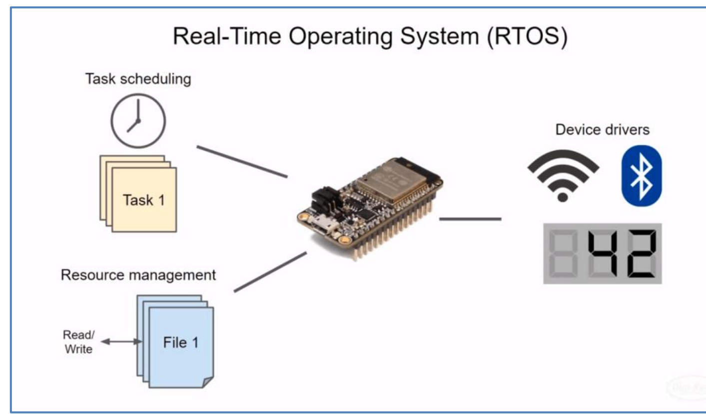


Figura 3-18. Sistema operativo de Tiempo Real, RTOS.

CPU	Microprocesador 240MHz Arm® Cortex®-M4.
	Modos de bajo consumo.
	Sistema de cómputo en “punto flotante”.
Memoria	640-KB SRAM.
	4-MB code flash memory.
Reloj	RTC con calendario desde el año 200 hasta 2099, con ajuste automático de años bisiestos.
	8 fuentes de reloj (on-chip oscillators)
Otros	Controlador de gráficos GLCDC, 14 temporizadores tipo PWM, 2 temporizadores asíncronos de uso general, convertidores analógicos, conectividad I2C, SPI y SCI.

Tabla 3-1. Especificaciones del SK-S7G2 [10].

El microcontrolador se encuentra integrado en la placa SK-S7G2 en un encapsulado tipo LQFP (“encapsulado cuadrado plano de perfil bajo” o *Low-profile Quad Flat Package*), con 176 pines. La placa provee una interfaz sencilla con los periféricos e incluye conectores USB y Ethernet, y salida de sudio tipo Jack. Incorpora como periféricos de interés, un panel táctil capacitivo con dos botones y un slider, y pantalla táctil LCD con resolución QVGA, *Quarter Video Graphics Array*, (240x320 px), además de varios LEDs y pulsadores mecánicos.

3.2.1 Periféricos empleados

Una diferencia notable entre este proyecto y su versión previa, Midisynth 2.0, es la integración de los periféricos en la propia placa SK-S7G2, de modo que no ha sido necesaria la fabricación de una *PCB* (Printed Circuit Board) específica como en la versión anterior. Se ha empleado la pantalla gráfica y los controles del panel táctil de la placa para la interacción y control del sintetizador por parte del usuario.

Los periféricos internos utilizados en nuestro proyecto han sido los siguientes:

- **2 temporizadores con interrupción**, para los osciladores DCO y LFO.
- **Convertidor digital-analógico, DAC.**
- **Puerto serie UART**, para la entrada de mensajes tipo MIDI.

Tanto los periféricos externos mencionados (LCD y pulsadores) como los internos del MCU se programan por medio de módulos o drivers, que funcionan como interfaz con el sistema operativo.

3.2.2 Elementos auxiliares

El único elemento hardware que hemos tenido que incorporar al SK-S7G2 ha sido el conector MIDI hembra tipo DIN de 180º. Se dispondrá integrado en una pequeña placa PCB junto a un optoacoplador para aislar la entrada, protegiendo así el equipo ante posibles picos de voltaje. En el header de 4 pines se conectan, de izquierda a derecha: alimentación (Vcc = 3.3V), tierra (GND), salida MIDI (no conectado) y entrada MIDI, conectado al canal 8 de la UART:

Figura 3-19. Placa PCB de adaptación MIDI.

Como entrada de mensajes MIDI para el control del sintetizador se ha utilizado el modelo de teclado *Icon iKeyboard 3X,* del fabricante Thomann. La salida de audio jack se puede conectar a un altavoz o unos auriculares para la reproducción de los sonidos sintetizados.

Figura 3-20. Teclado *Icon iKeyboard 3X.*

4 HERRAMIENTAS SOFTWARE

Existen dos tipos de lenguajes de programación:

por un lado, aquellos de los que la gente se queja constantemente;

por otro, los que nadie utiliza."

- Bjarne Stroustrup

En este capítulo describiremos brevemente los programas empleados para el desarrollo de nuestro sintetizador, desde el entorno la programación en código C++ hasta las aplicaciones necesarias para testear el correcto funcionamiento del mismo, así como su calibración y configuración.

4.1 E2 Studio

La principal herramienta de trabajo ha sido ***e2 Studio***, entorno de desarrollo IDE (*Integrated Development Environment)* de Renesas Synergy basado en la plataforma Eclipse, específica para el desarrollo de código en C++. Sobre este entorno se apoya el software SSP (*Synergy Software Package*), software comercial propio de Renesas que viene incluido en el precio del MCU y que se basa en el sistema operativo ThreadX®. Este último es un RTOS (*Real Time Operative System*) determinista multitarea basado en prioridades, que se combina con las librerías del SSP a través de un framework específico utilizando una API (*Application Programming Interfaces*). [3]

La versión empleada en este trabajo ha sido e² studio v7.5.1, la cual puede descargarse desde este enlace: Archived e² studio for Renesas Synergy™ | Renesas.

4.2 Tera Term

Para el testeo de la interfaz de comunicaciones puerto serie se ha utilizado el emulador de terminal TeraTerm, software de código abierto que nos permite establecer este tipo de comunicaciones en Microsoft Windows. A través del puerto USB testeamos el envío de mensajes entre el microcontrolador y el terminal, y una vez finalizadas las pruebas del protocolo UART los mensajes serán recibidos por el MCU desde el periférico de entrada MIDI.

En este proyecto se ha utilizado la última actualización del programa, v4.106, descargable desde la web Downloading File /74780/teraterm-4.106.exe - Tera Term - OSDN.

4.3 Guix Studio

Para el diseño de la interfaz gráfica que empleamos en la pantalla LCD se trabaja con el entorno de Desarrollo Guix Studio de Microsoft. Esta aplicación de escritorio permite la creación de gráficos o GUIs (Grafical User Interface) de forma visual, facilitando la generación posterior del código C++ a insertar en el programa principal del proyecto.

La version 5.6.1.0 que hemos empleado, puede encontrarse en el siguiente enlace: Archived GUIX™ Studio Releases for Renesas Synergy™ | Renesas

4.4 Capacitive Touch Workbench

Para el correcto funcionamiento del panel táctil capacitivo integrado en la placa es necesaria su calibración previa. Dicho proceso de ajuste se lleva a cabo empleando la herramienta CTW (*Capacitive Touch Workbench*), mediante la cual se crea un GUI en el que se asocian los botones y sliders disponibles a los canales asociados de la unidad táctil CTSU (*Capacitive Touch Sensing Unit*). Una vez regulados los umbrales de detección táctil mediante algunos test de interacción con el usuario, el programa genera el código asociado que se integra en el programa C++ que utiliza este periférico.

Esta aplicación puede obtenerse a través del enlace: Capacitive Touch Workbench for Renesas Synergy™ | Renesas.

4.5 Atom

Atom es un editor de texto para programación creado por la comunidad de software GitHub. Se trata de una aplicación de código abierto basado que soporta de forma predeterminada numerosos lenguajes, como C, C++, CSS, XML, PHP o Java. Adicionalmente, mediante el sistema de paquetes de licencia libre, mantenido por la comunidad de usuarios de GitHub, es posible la instalación de otros lenguajes, compiladores y debuggers. De este modo es posible conectar Atom con software de terceros posibilitando su utilización como entorno de programación IDE.

En nuestro caso hemos instalado el paquete Pymakr, que proporciona un entorno de programación interactivo tipo REPL (*Read-Eval-Print-Loop*) y permite la conexión con placas tipo Pycom como la Wipy para transferir archivos Micropython. La versión sobre la que hemos trabajado es la 1.46.0, obtenible en el repositorio github: Release 1.46.0 · atom/atom · GitHub.

5 DESARROLLO DEL PROGRAMA

Music is science more than art, and it is the main code of the universe.

- Vangelis -

El grueso del proyecto lo ha constituido la programación del sintetizador en el microcontrolador SK-S7G2, para lo cual se ha empleado principalmente el sotware e2 Studio y la aplicación Guix Studio, presentados en el capítulo cuarto. A continuación se detallará la configuración inicial de los drivers del microcontrolador y, seguidamente, el funcionamiento del programa C, exponiendo previamente las prestaciones que se esperan del dispositivo. Adicionalmente, se explicará de forma concisa el proceso de diseño de la interfaz gráfica y su vinculación con el código de control de la LCD en e2 Studio.

5.1 Especificaciones de funcionamiento

La pantalla LCD constituirá la interfaz para el manejo del sintetizador por parte del usuario. En ella se mostrarán distintas ventanas que permitirán ajustar los parámetros del proceso de síntesis. Al ejecutar el programa Midisynth 3.0 desde e2Studio se mostrará en primer lugar una ventana de inicio que dará acceso al menú principal mediante un simple toque en la pantalla.

Figura 5-1. Ventana de Inicio.

Desde el menú principal, que emula el estilo del sintetizador clásico Moog, se accede al otras tres ventanas para el control de los módulos principales que ya estudiamos en el capítulo 2, osciladores DCO, osciladores LFO y filtro. El módulo amplificador se controla desde el propio menú *Moog*.

Figura 5-2. Menú principal.

La primera etapa del proceso de síntesis es la generación de la onda principal, para lo que debemos seleccionar los osciladores principales accediendo a la ventana "control_instrumentos", dedicada al control de los DCO que formarán parte de la síntesis aditiva. Estos se identifican como 8 instrumentos, que pueden activarse mediante la pulsación de sus botones asociados en la LCD. Los instrumentos seleccionados se iluminarán en color verde.

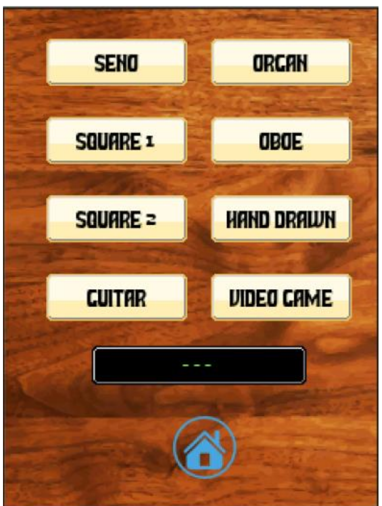


Figura 5-3. Ventana de control DCO.

Cuando la ventana DCO está activa en la LCD, el *slider* del panel táctil, situado en el lado izquierdo de la plaza, servirá para regular el desfase asociado a cada uno de los instrumentos activos. El valor recogido por este periférico se asociará al último instrumento que haya sido seleccionado. En caso de desear regular la fase de un instrumento distinto al último "activo" este deberá apagarse y encenderse de nuevo para tomar el control sobre su desfase.

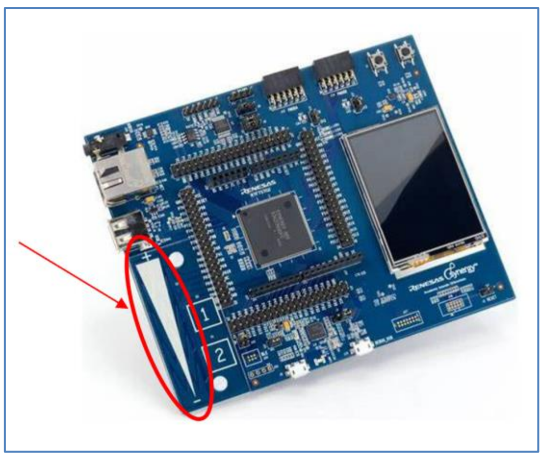


Figura 5-4. Slider para el control del desfase.

Al tocar de nuevo un pulsador que ya esté activo, este se apagará. Pulsando el icono "casa" se podrá regresar al menú principal, donde se señalizarán en verde los intrumentos activos, ordenados de izquierda a derecha, mediante unos indicadores luminosos.

A continuación podrá seleccionarse la envolvente deseada accediendo a la ventana "control_modulacion", entre los 4 módulos LFO disponibles: Trémolo, Vibrato, FM o ADSR. Los botones asociados a dichos efectos de modulación solo se accionarán en caso de estar activada la modulación, la cual se enciende mediante el pulsador ON/OFF situado en la zona superior de la pantalla. Al apagar el pulsador de modulación, se desactivará automáticamente cualquier modulación que se encontrara seleccionada.

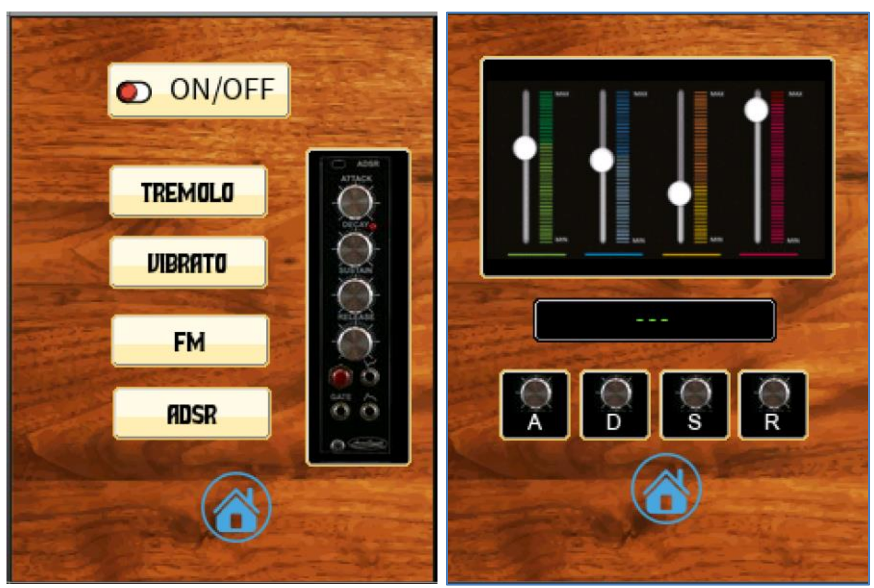


Figura 5-5. Ventana de control LFO y ventana de control ADSR.

Análogamente a la ventana DCO, cuando la ventana LFO se encuentra activa el slider funciona para regular distintos parámetros de la modulación que se encuentre activa. En caso del efecto FM, el slider controlará el índice de modulación. Si se activa la modulación ADSR, pulsando el botón situado a la derecha podremos ajustar los parámetros ADSR de la envolvente. Activando cada uno de ellos mediante un toque, estos se iluminarán en color verde, y el valor recogido por el slider ajustará los tiempos de las fases *Atack, Decay* y *Release*, y el valor de la amplitud en la fase *Sustain*, como porcentaje de la amplitud máxima.

De nuevo pulsando la "casa", volveremos al menú principal. La modulación activa se señalizará en el menú principal sobre un recuadro de texto.

En la zona superior derecha del menú *Moog* se encontrará el módulo de filtros. Pulsándolo accedemos a la ventana “control_filtros”, en la cual activaremos los mismos mediante el pulsador ON/OFF. Recuérdese que este módulo se ha tenido que eliminar del proyecto, pero se ha dejado el menú de opciones para posibles líneas futuras de trabajo; no obstante, aunque se modifiquen las variables asociadas a los controles del filtro, el efecto de este queda anulado en la onda final sintetizada.

Pulsando la “casa” volveremos al menú principal, y en caso de haber activado el filtro, quedará señalizado mediante un indicador luminoso de color verde.

Por último, el bloque de amplificación se controlará desde el propio menú principal: pulsando sobre icono del módulo (rueda de volumen), el texto del amplificador se iluminará en verde y el slider quedará activado para la regulación del volumen.

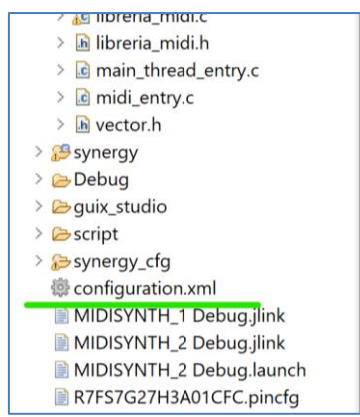
Figura 5-6. Regulador de volumen.

Una vez ajustado el funcionamiento del sintetizador, la generación del sonido deseado se accionará desde el teclado MIDI conectado a la placa mediante una PCB de adaptación. La información sobre la secuencia de notas interpretada por el usuario se envía al microcontrolador y viaja por el conjunto de bloque que hemos programado: el DCO genera el sonido principal con los intrumentos activados, el cual es articulado por el LFO y finalmente se amplifica hasta alcanzar la intensidad o volumen deseado.

5.2 Configuración de los módulos

Antes de comenzar a programar los periféricos internos del microcontrolador, es necesaria la configuración previa de sus módulos asociados. Los módulos son drivers que controlan los registros de los periféricos y proveen una interfaz para su programación a alto nivel; constituyen lo que se coonoce como *HAL (Hardware Abstraction Layer)*.

Para ello abrimos la ventana de configuración desde el directorio del proyecto y seleccionamos la pestaña threads. Dentro de ella, mediante la opción “New Thread” creamos los dos hilos del programa: el hilo principal (*Midisynth*) y el hilo de control de la LCD (*Main Thread*). El hilo HAL Common está ya creado por defecto en el proyecto.



Seleccionando cada uno de los hilos creados, configuramos sus módulos correspondientes, pulsando sobre la opción “New Stack”. Para el hilo Midisynth incluiremos los módulos: **Timer0**, para la interrupción DCO; **Timer1**, para el LFO; el **convertidor DAC0**, para la salida de de audio; un **puerto UART**, para la entrada MIDI; y los drivers asociados al panel táctil, ***CapTouch Button Framework*** y ***CapTouch Slider Framework.*** En la pestaña properties ajustaremos los parámetros de cada uno de ellos.

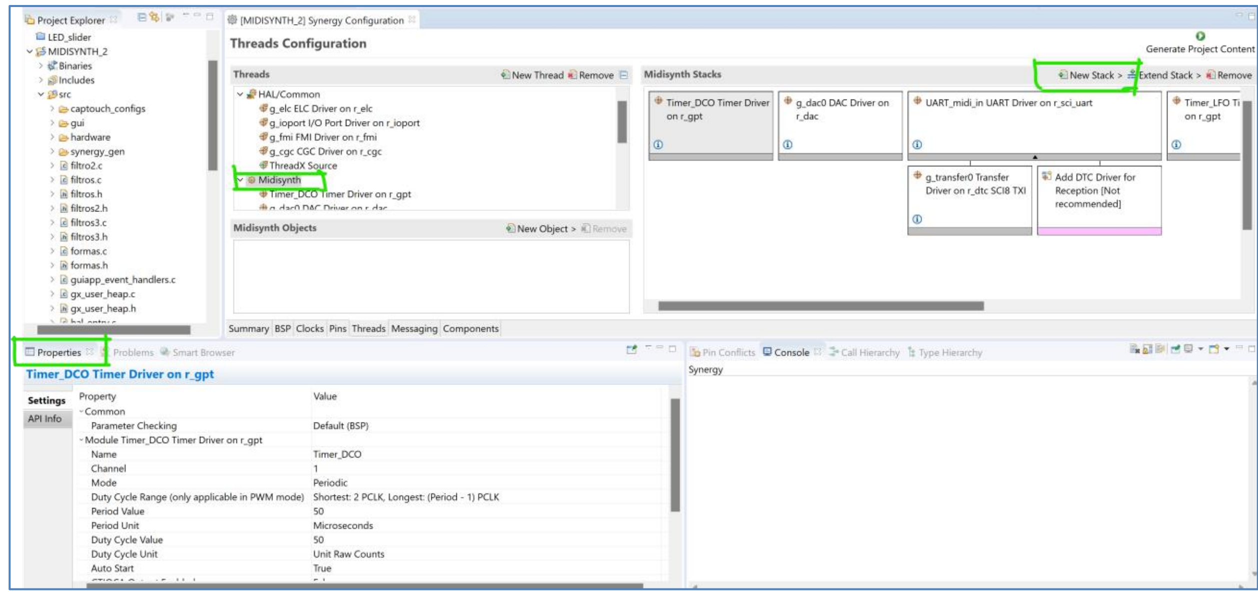


Figura 5-7. Ventana de configuración de módulos.

Por otro lado, en el hilo que gobierna la pantalla LCD (Main Thread) debemos incluir el driver que provee la interfaz gráfica, *GUIX on gx*, el driver SPI para establecer las comunicaciones con la pantalla, así como el driver *Touch Panel I2c* para detectar los eventos táctiles sobre la misma.

Una vez ajustados los parámetros de los módulos[7], activamos los pines asociados a los mismos, en la pestaña *Pins*. Para cada periférico el procedimiento será similar: desplegamos en primer lugar la opción "Peripherals", buscamos el periférico en cuestión y lo habilitamos (*Enabled*), seleccionando alguno de los posibles pines vinculados.

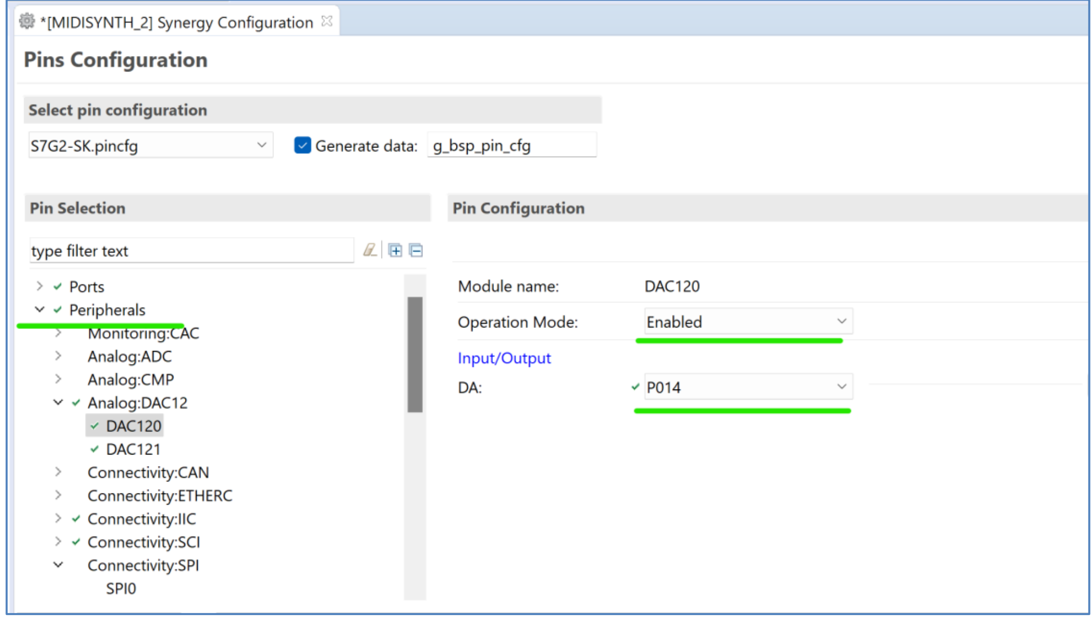


Figura 5-8. Configuración de periféricos.

[7] No ahondaremos en estos ajustes por no extendernos demasiado en el texto. Existen guías acerca de la configuración de cada módulo HAL en la galería de documentación de Renesas Synergy: Search | Renesas Electronics Corporation. Pueden encontrarse en este enlace, simplemente realizando la búsqueda "*Nombre_módulo* HAL module guide".

En principio al activar el pin, se notificará un error en color rojo, esto es porque el pin está inhabilitado, y debemos activarlo a continuación desplegando la opción Pins, y buscando el Pin que acabamos de seleccionar para el periférico en cuestión:

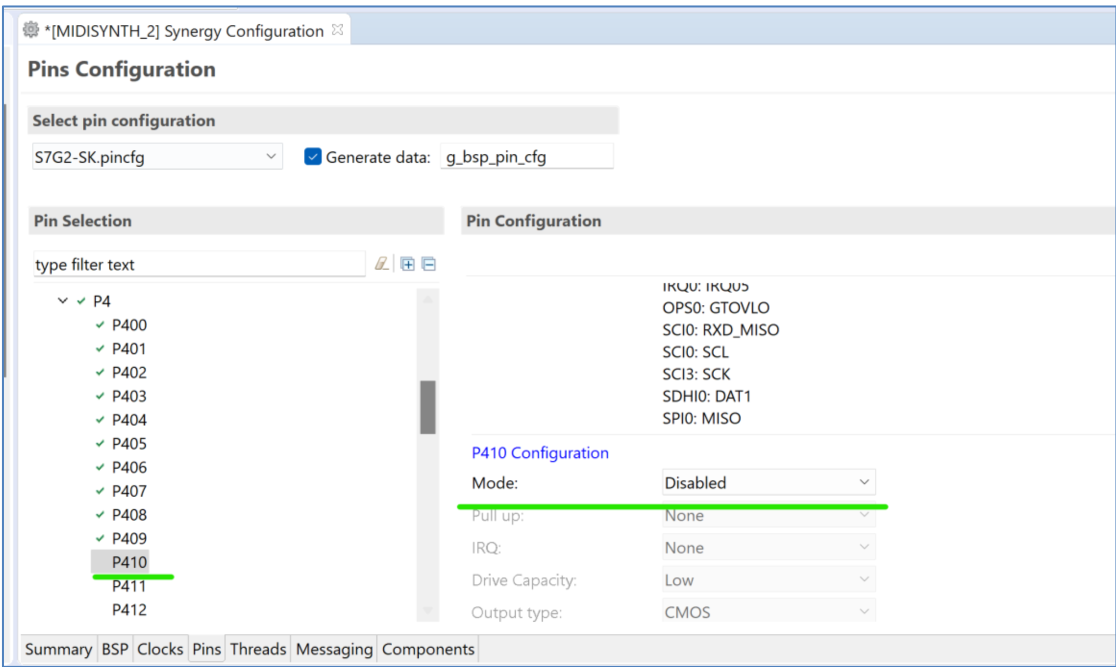


Figura 5-9. Configuración de pines.

El pinout de nuestro proyecto se ilustra en las siguientes tablas. Para el panel táctil (slider y botones) activaremos los pines del módulo CTSU0 (*Capacitive Touch Sensor Unit*):

	CTSU0	TSCAP	P205
PANEL TÁCTIL	CTSU0	TS00	P204
	CTSU0	TS01	P206
	CTSU0	TS02	P207
	CTSU0	TS04	P408
	CTSU0	TS05	P409
	CTSU0	TS10	P414
	CTSU0	TS11	P415

Tabla 5-1. Pinout del panel táctil.

Para la pantalla LCD los pines del driver de gráficos, del puerto SPI, del I2C y del panel táctil:

	PERIFÉRICO	INPUT/OUTPUT	PUERTO
PANTALLA TÁCTIL	SCI0	TXD_MOSI	P101
	SCI0	RXD_MISO	P100
	SCI0	SCK	P102
	SCI0	CTS_RTS_SS	P103
	IIC2	SDA	P511
	IIC2	SCL	P512
	LCD GPIO	LCD_WR	P115
	TOUCH PANEL GPIO	RESET	P609
	LCD GPIO	LCD_RESET	P610
	LCD GPIO	LCD_CS	P611
	GLCDC0	LCD_CLK	P900
	GLCDC0	LCD_DATA_00	P804
	GLCDC0	LCD_DATA_01	P803
	GLCDC0	LCD_DATA_02	P802
	GLCDC0	LCD_DATA_03	P606
	GLCDC0	LCD_DATA_04	P607
	GLCDC0	LCD_DATA_05	PA00
	GLCDC0	LCD_DATA_06	PA01
	GLCDC0	LCD_DATA_07	PA10
	GLCDC0	LCD_DATA_08	PA09
	GLCDC0	LCD_DATA_09	PA08
	GLCDC0	LCD_DATA_10	P615
	GLCDC0	LCD_DATA_11	P905
	GLCDC0	LCD_DATA_12	P906
	GLCDC0	LCD_DATA_13	P007
	GLCDC0	LCD_DATA_14	P908
	GLCDC0	LCD_DATA_15	P901
	GLCDC0	LCD_TCON0	P315
	GLCDC0	LCD_TCON1	P314
	GLCDC0	LCD_TCON2	P313

Tabla 5-2. Pinout de la LCD.

Para el correcto funcionamiento del panel táctil, además de la configuración de los pines que se acaba de exponer, es necesaria su calibración, utilizando la aplicación ***Capacitive Touch Workbench***, que ya mencionamos en el cuarto capítulo, siguiendo los pasos del tutorial que puede encontrarse en la galería de documentación de *Renesas*: Tuning the Capacitive Touch Solution (renesas.com).

Por último, la salida del **DAC0** estará asociada con el **pin P014** y el bus de lectura del canal 8 UART será el pin **P104 (RXD_MISO).**

Finalizada la configuración de módulos y pines, deberá actualizarse esta información en el microcontrolador, mediante la opción ***"Generate Project Content".***

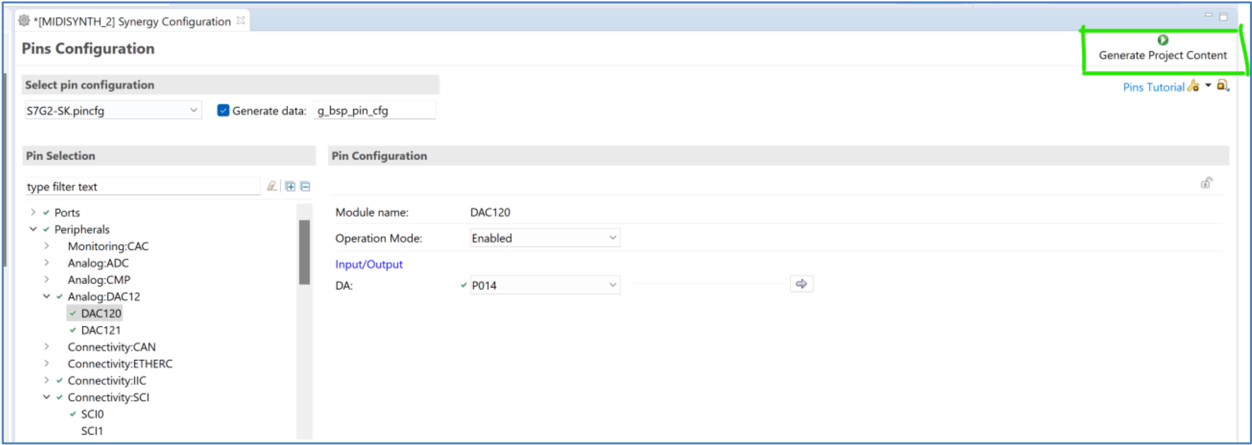


Figura 5-10. Actualización de configuración HAL.

5.3 Diseño de la interfaz gráfica en GUIX Studio

Para componer los gráficos asociados a los menús de control del sintetizador se utiliza la aplicación Guix Studio, que enlaza con el software e2Studio generando el código asociado a la configuración de las distintas ventanas gráficas. En primer lugar debe crearse una nueva carpeta en el directorio del proyecto, que nombraremos, por ejemplo, guix_studio:

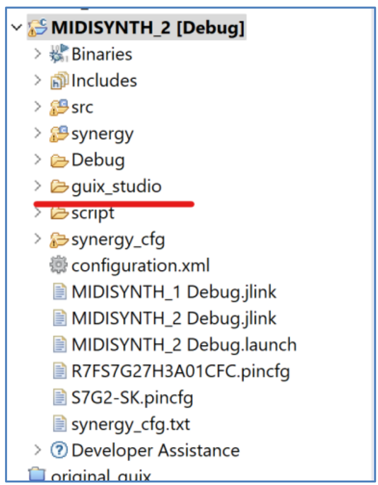


Figura 5-11. Carpeta de proyecto Guix.

A continuación crearemos un nuevo proyecto en la aplicación guix, cuyo *path* será la carpeta creada anteriormente:

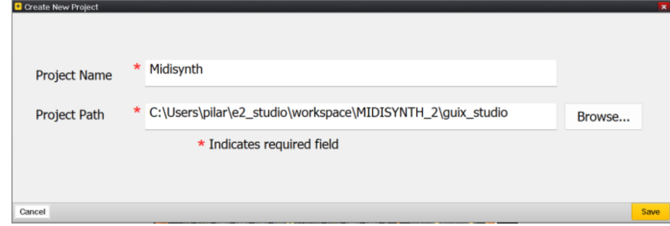


Figura 5-12. Nuevo proyecto Guix.

Adicionalmente, creamos otra carpeta vacía, esta vez dentro de la carpeta *src* del proyecto:

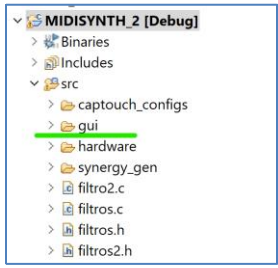
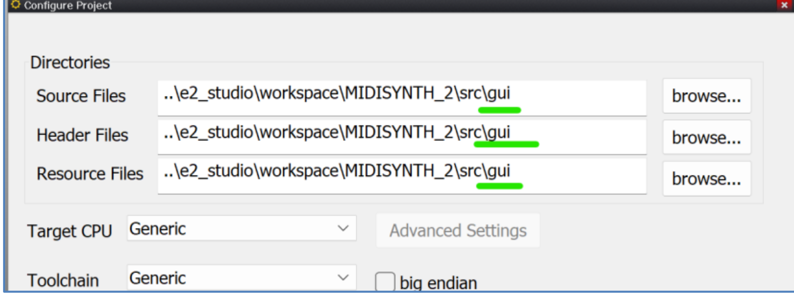


Figura 5-13. Carpeta de archivos fuente Guix.

En la configuración inicial el proyecto seleccionamos dicha carpeta como directorio para los archivos fuente, las cabeceras y los recursos generados por la aplicación:



La ventana de la aplicación tendrá el aspecto de la figura:

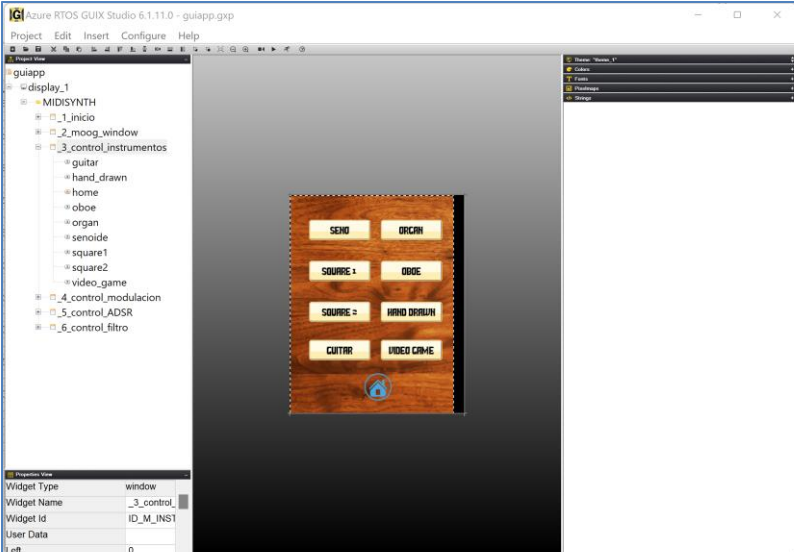


Figura 5-14. Ventana de la aplicación Guix Studio.

En el display raíz (display_1) creamos una nueva carpeta (*click derecho->insert->folder*) y dentro de ella insertamos las diferentes ventanas que controlan los módulos del sintetizador (*insert->window*).

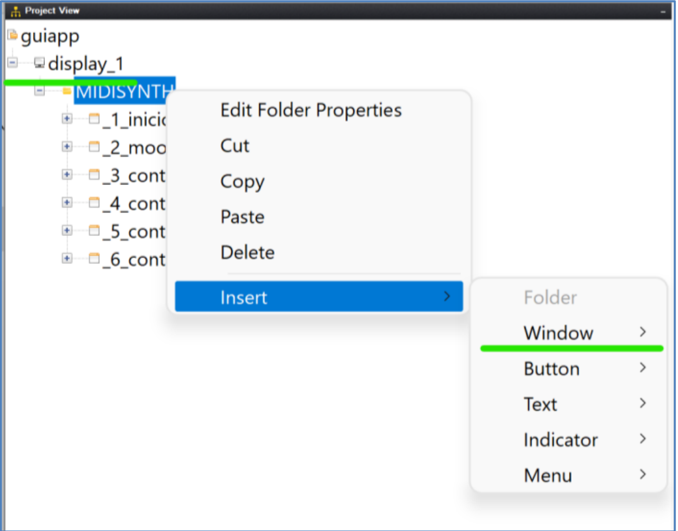


Figura 5-15. Crear ventanas del proyecto.

En cada ventana podremos insertar a su vez múltiples elementos, denominados *widgets*. Los empleados en este proyecto han sido los *widgets* tipo *Window, Button y Pixelmap Prompt*, pero la aplicación presenta muchos otras posibilidades.

La configuración de cada widget se lleva a cabo en la pestaña Properties, localizada en la zona izquierda de la aplicación. En ella se pueden ajustar los parámetros de diseño de los distintos elementos, y se les asignará un ID para identificarlos en el código del programa. Este ID deberá ser único para cada elemento que cuelgue de un mismo widget "padre", mas se trata de un atributo opcional que se incluye solo en aquellos elementos cuyos eventos táctiles queremos detectar; no será necesario por ejemplo para elementos decorativos de la interfaz gráfica.

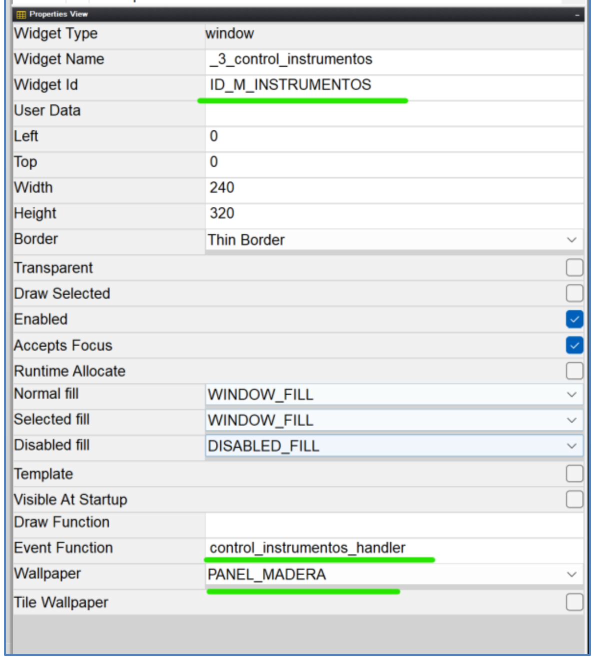


Figura 5-16. Pestaña *Properties* de los *Widgets.*

En los *Pixelmap Prompt* es posible seleccionar una imagen o fondo personalizado, que debe incluirse en la pestaña derecha *Pixelmaps->Custom*. Los pixelmaps empleados se muestran en la siguiente figura:

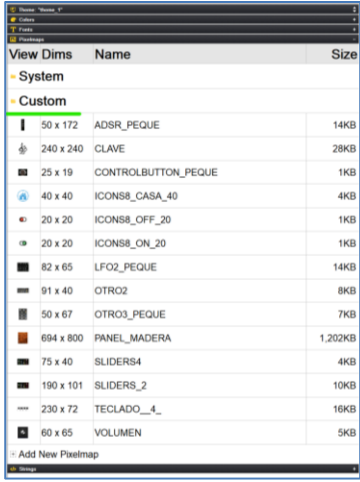


Figura 5-17. Configuración de Pixelmaps.

En los widgets tipo *Window*, asociados a los menús de control del sintetizador, incluimos un Event Handler, que será la rutina de interrupción a programar en e2 Studio para los eventos generados por los widgets "hijo" de cada ventana. Véase por ejemplo la figura 5.6 de la ventana "control de instrumentos".

Por otro lado, una manera sencilla de configurar la transición entre ventanas a partir de eventos tipo *click, es configurando el Screen Flow del proyecto, desde la pestaña de configuración:*

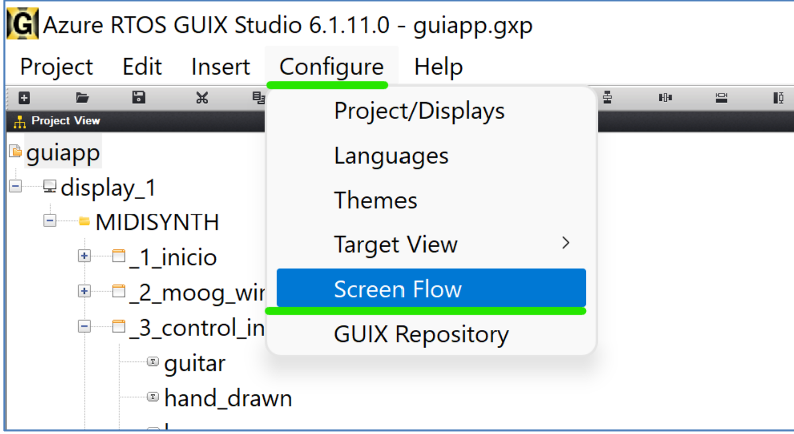


Figura 5-18. Editar *Screen Flow*.

Se abre un menú en el que se definen para cada ventana *Triggers* y *Actionss*, asociadas a los triggers anteriores. Pulsando botón derecho sobre el bloque de cada ventana, se crean primero nuevos triggers, que hemos asociado a dos tipos de eventos:

- La pulsación de cualquier punto de la pantalla (System Event -> GX_EVENT_PEN_UP).
- La pulsación de un widget "hijo" (Child Signal->ID_widget, GX_EVENT_CLICKED).

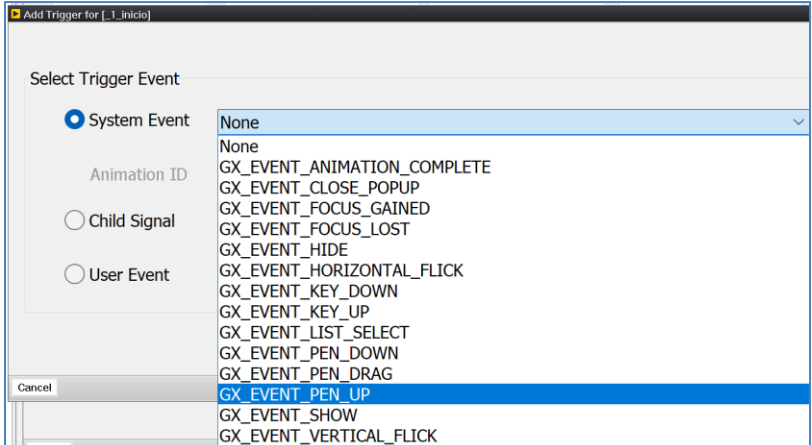


Figura 5-19. System Event Trigger.

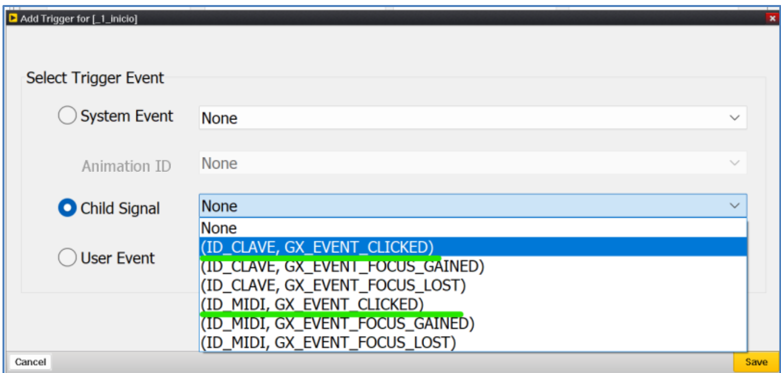


Figura 5-20. Child Signal Trigger.

Para cada trigger creado, creamos una nueva acción, tipo *Toggle,* y en el campo *Target* seleccionamos la ventana a la que queremos conmutar cuando se detecte el *Trigger*.

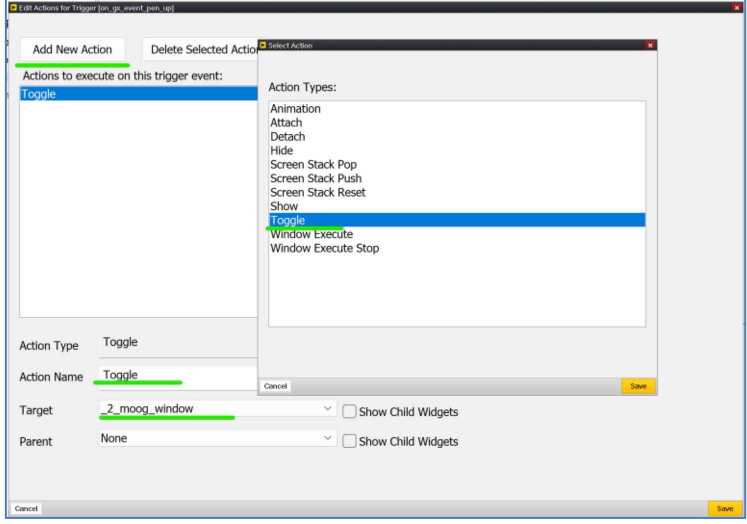


Figura 5-21. Acción tipo transición de ventana.

En el Screen Flow una vez configurado quedan indicadas las transiciones entre los bloque asociados a las distintas ventanas. El resto de transiciones dependen de combinaciones más complejas de eventos y variables, por lo que se definen a partir de las funciones tipo handler, que detallaremos en la sección 5.4.3:

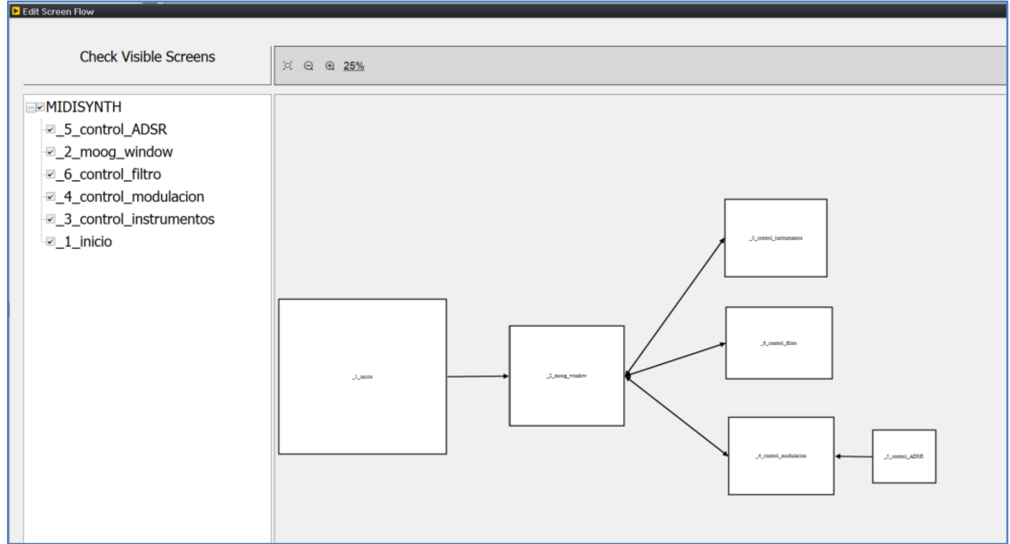
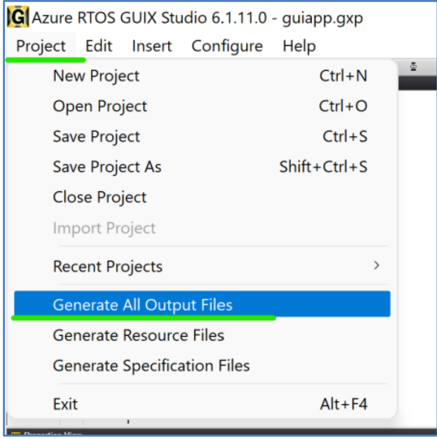


Figura 5-22. Screen Flow del proyecto.

Una vez diseñada la interfaz gráfica, se deberán generar los archivos fuente de configuración que quedarán incluidos automáticamente en el directorio del proyecto, dentro de la carpeta "gui" creada inicialmente. Para ello seleccionamos la opción Project->Generate All Output Files. Para actualizar dicha configuración en e2 Studio tendremos que volver a compilar el programa mediante la opción *Build.*



5.4 Etapas del programa

El programa se basa en un bucle *while(1)* contenido en el hilo principal del programa, que se ejecuta indefinidamente tras una etapa de configuración inicial, y que consiste en la comprobar contínuamente dos posibles eventos, notificados mediante variables *flag* asociadas:

- **Actualización de notas:** verifica si ha llegado una nota nueva o se ha apagado la nota que estaba sonando, es decir, comprueba si hay algún mensaje *note-ON/note-OFF* pendiente de procesar. La bandera que notifica este evento se activa desde la interrupción del puerto serie UART, asociado a la entrada MIDI.

- **Actualización de instrumentos:** revisa si ha habido cambios en la selección de los instrumentos, esto es, si se ha activado o desactivado alguno de los osciladores principales. En dicho caso se volverá a calcular la forma de onda básica, suma de las generadas por cada DCO activo. El aviso llega a través de una bandera que es generada desde el hilo secundario asociado al funcionamiento de la LCD.

Por otro lado, contaremos con varias rutinas de interrupción del bucle principal para la lectura de algunos periféricos y la generación temporizada de la onda. En total se producirán 4 tipos de interrupción:

1. **Interrupción DCO:** rutina temporizada cada 50 microsegundos (20kHz) para el cálculo de los distintos puntos que conforman la onda principal, mediante la lectura de la tabla de ondas, y su modificación a partir de la onda envolvente, en caso de estar activo el bloque LFO. Los valores digitales calculados se escribirán en el *DAC*, y se encontrarán en el rango [0,4095], dado que el convertidor tiene una resolución de 12 bits.

2. **Interrupción LFO:** rutina que calcula los puntos de la onda envolvente, con una resolución de 100kHz, es decir, se produce cada 100 microsegundos, cinco veces por cada interrupción DCO.

3. **Interrupción de la entrada MIDI:** esta rutina se ejecuta cuando se detecta la llegada de mensajes MIDI de tipo *note-ON/note-OFF* por el canal UART de comunicación puerto serie asociado. Se almacenan los valores de velocidad (intensidad) y altura de la nota, y se notifica al bucle principal mediante una variable tipo *flag* para que el mensaje sea procesado.

4. **Interrupciones del panel táctil (slider y botones):** se ejecutan cuando se detectan eventos sobre el panel táctil, en el slider o sobre los botones, respectivamente.

El hilo secundario, asociado al funcionamiento de la pantalla LCD, se ejecuta en paralelo con el principal, gracias al sistema operativo multi-hilo; en él se controlan distintos parámetros que son leídos en tiempo real por las interrupciones y por el bucle principal, como acabamos de comentar.

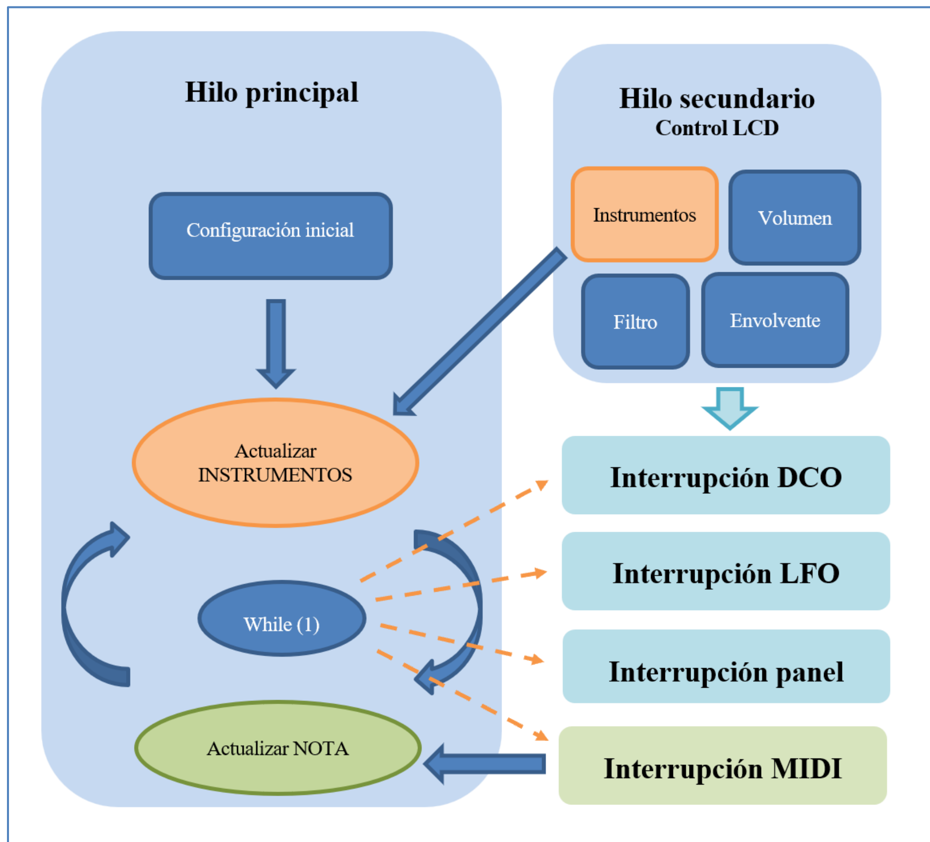


Figura 5-23. Diagrama de flujo del programa.

En las siguientes secciones se ilustran resumidamente las acciones ejecutadas en cada uno de los bloques del diagrama anterior.

5.4.1 Hilo principal

5.4.1.1 Configuración inicial

En esta primera etapa programamos los periféricos principales: habilitamos el canal 8 del módulo UART para la recepción de mensajes MIDI, activamos el timer 0 para la interrupción DCO, el timer 1 para la interrupción LFO y abrimos el canal 0 del DAC para la salida de audio.

Adicionalmente, asignamos valores iniciales a los parámetros ADSR: tiempos Atack, Decay y Release de 100 milisegundos, valor sustain del 80% y amplitudes asociadas a cada fase.

5.4.1.2 Actualización de notas

Este fragmento de código comprueba si se ha pulsado una tecla (nueva nota) o se ha liberado la tecla asociada al sonido que esta reproduciéndose, lo cual es notificado por la subida de la bandera `flag_Msg`.

- Si `flag_Msg` vale 1:
    - Bajamos la bandera: `flag_Msg=0`.
    - Si la tecla se ha pulsado, llega un mensaje de *Note_ON:*
        - Actualizar la nota actual almacenada: `nota_act = altura-OFFSET_NOTA`.

Donde la variable **nota_act representa el índice asociado al vector de frecuencias normalizadas**. El offset es una constante, igual a 36, valor de la primera "altura" contenida en nuestro vector de frecuencias normalizadas, de las 128 posibles enviadas por protocolo MIDI.

5.4.1.3 Actualización de instrumentos

Se comprueba si ha cambiado el número de instrumentos activos, detectando la subida de la bandera `flag_instrumento`, en cuyo caso se deberá recalcular la forma de onda aditiva, formada por las ondas asociadas a cada instrumento activo.

- Si `flag_instrumento` vale 1:
    - Bajamos la bandera: `flag_instrumento=0`.
    - Si el instrumento pulsado `(instrum_selec)` en pantalla estaba desactivado, lo activamos en el vector `instrum_activos[instrum_selec]`.
    - `En caso contrario, lo desactivamos: instrum_activos[instrum_selec] = 0.`
    - Por último, llamamos a la función que recalcula la onda.

5.4.1.4 Actualización de modulación

En este apartado del código sencillamente se reseteará la frecuencia de inicio de los efectos de modulación trémolo y vibrato. Esto se ejecutará en el caso de producirse un flanco en la bandera `flag_modulacion`, que es enviada desde el hilo de la LCD cuando se cambia el efecto de modulación activo, en cuyo caso:

- Bajamos la bandera: `flag_modulacion = 0`.
- Si el efecto activo es el Trémolo, la frecuencia de modulación se actualiza a `frecMod_ini_tremolo`, constante de valor 1MHz.
- Si el efecto activo es el Vibrato, la frecuencia de modulación se actualiza a `frecMod_ini_vibrato`, constante de valor 5MHz.

El resto de variables relacionadas con la modulación, se actualizarán directamente desde el hilo secundario,

como se verá en un apartado posterior.

5.4.2 Interrupción MIDI

De forma complementaria a las acciones anteriores, en la función de interrupción se ejecutan las órdenes siguientes, cuando se detecta la llegada de mensajes por el canal 8 de la UART:

- Si el mensaje contenido en el búfer tiene valor `0x00, se trata de un mensaje tipo Note_OFF:`
  - Subir la bandera `flag_Msg` para notificarlo.
  - Si el efecto ADSR estaba activo, comienza la fase Release de dicha modulación; el estado será `ADSR_RELEASE.`
  - En otro caso, el estado será `NO_HAY_NOTA.`
  - Resetear el contador de mensajes, `n = 0.`
- Si llega un mensaje de tipo Note_ON, de valor `0x90:`
  - Activar contador de mensajes, para esperar al mensaje de `Altura, n = 1.`
- Si llega otro mensaje, que contiene la `Altura (n vale 1):`
  - Guardar el valor de la altura de la nota.
  - Incrementar el contador de mensajes (ahora `n = 2`), para esperar el mensaje de velocidad o fuerza de ataque.
- Si llega otro mensaje, asociado a la velocidad, (`n vale 2`):
  - Guardar el valor de la velocidad.
  - Resetear el contador `n.`
  - Acualizar el estado: `estado = HAY_NOTA.`
  - Subir la bandera de mensaje MIDI: `flag_Msg = 1.`

5.4.3 Hilo secundario - Control LCD

En este hilo se actualizan variables que afectan tanto al bucle principal como a las interrupciones, gestionadas a través de las funciones tipo *event_handler* asociadas a cada una de las 4 ventanas gráficas asociadas a los menús de control y una función extra relacionada con el bloque regulador de volumen. Estas funciones son las siguientes:

- *UINT control_**instrumentos**_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)*
- *UINT control_**modulacion**_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)*
- *UINT control_**ADSR**_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)*
- *UINT control_**filtro**_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)*
- *UINT control_**volumen**_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)*

Estas funciones identifican los eventos GX_EVENT_CLICKED de cada uno de los widgets que contiene la ventana en cuestión, es decir, reconocen el botón que ha sido pulsado, llamando a otras funciones auxiliares a las que envían como argumentos de entrada los identificadores de dicho botón. Estas funciones auxiliares que hemos denotado como "update_" actualizan las variables globales asociadas y llaman a su vez a otras funciones que hemos creado para la actualización de color y texto de los distintos widgets, por ejemplo para

“encender” o “apagar” los indicadores luminosos en color verde.

5.4.4 Interrupción DCO

La función a ejecutar cada 50 microsegundos contiene las siguientes instrucciones:

- Resetear la variable auxiliar `DAC_aux`, que se irá modificando a lo largo del bucle hasta escribirla finalmente en el DAC.
- Si la variable de estado toma el valor `HAY_NOTA o bien ADSR_RELEASE` (indica que hay una nota con efecto ADSR y debe liberarse progresivamente por la llegada de un mensaje Note_OFF):
    - Leemos del vector que contiene la onda calculada, según la del punto actual, guardar en `DAC_aux.`
    - Si el efecto ADSR está inactivo: escalar con la variable `fuerza`.
    - Si el trémolo está activo, incrementar la fase según la frecuencia de la nota actual y multiplicar por el valor de la onda envolvente.
    - Si el efecto actual es el vibrato, incrementar la fase según la frecuencia de modulación.
    - Si está seleccionado el efecto FM:
        - Calculamos el valor de la envolvente, leyéndola de la tabla de muestras y escalándola debidamente.
        - Calculamos la frecuencia moduladora sumando a la frecuencia de la nota el valor de la envolvente que acabamos de calcular.
        - Incrementamos la fase según el valor de dicha frecuencia de modulación.
        - Incrementamos la fase de modulación sumándole la frecuencia de la nota actual desplazada en la tabla según el índice de modulación que haya sido seleccionado.
    - Si el efecto es el ADSR:
        - Escalamos según el valor ADSR calculado en la interrupción LFO.
        - Incrementamos la fase según la frecuencia de la nota actual.
    - En caso de que la modulación esté desactivada, simplemente incrementamos la fase.
    - Escalamos según el volumen seleccionado.
    - Saturamos para que el punto calculado no salga del rango [-128,128].
    - Escalamos la onda y sumamos un offset de 2048 para centrar la onda en el rango del DAC de 12 bits: [0,4095].
    - Escribimos el punto calculado de la onda en el DAC.

5.4.5 Interrupción LFO

Esta rutina temporizada cada 100 microsegundos es responsable de generar la onda encvolvente. En caso de haber alguna modulación activa, llevará a cabo las acciones siguientes:

- Si el trémolo se encuentra seleccionado:
    - Calcula el valor de la envolvente según la forma del instrumento envolvente.
    - Incrementa la fase moduladora según la frecuencia de modulación.
- Si tenemos efecto vibrato:[8]
    - Calcula la envolvente.

[8] Nótese que en la modulación FM estas mismas operaciones se efectúan dentro de la interrupción DCO, es decir, las envolvente se actualiza a una frecuencia 5 veces mayor, puesto que como hemos comentado, el FM no es más que un vibrato muy rápido.

- Calcula la frecuencia moduladora sumando a la frecuencia de la nota actual el valor de la envolvente.
- Incrementa la fase de modulación con la frecuencia moduladora recién calculada.

*Si el efecto es el FM, no se ejecuta ninguna orden.

- Si el efecto activo es el ADSR:
  - Si hay Note_ON, incrementar la variable contadora de tiempos y calcular el valor de la envolvente según la variable de tiempo se encuentre en el intervalo de ataque, de descenso o de sostenido, según las ecuaciones siguientes:

$$\text{Amplitud fase } \mathbf{A} = \frac{Amplitud\ alcanzada\ al\ final\ de\ la\ fase\ A}{Duración\ de\ la\ fase\ A} = \frac{1000}{tA}$$

$$\text{Amplitud fase } \mathbf{D} = \frac{Amplitud\ final\ fase\ A - Amplitud\ final\ fase\ D}{Duración\ de\ la\ fase\ D} = \frac{1000 - 10 * SUST(\%)}{tD}$$

$$\text{Amplitud fase } \mathbf{R} = \frac{Amplitud\ final\ fase\ S\ (constante) - Amplitud\ final\ fase\ R(0)}{Duración\ de\ la\ fase\ R} = \frac{10 * SUST(\%)}{tR}$$

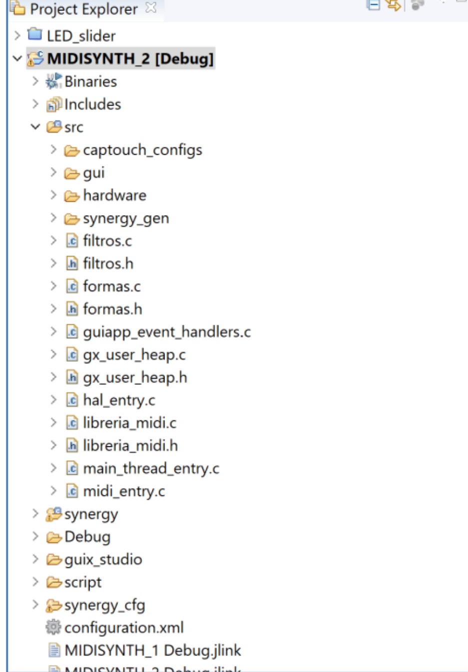
  - Si llega mensaje de note_OFF la nota debe liberarse, con lo que comienza la 4ª fase de la modulación (*Release*). Se calcula una envolvente que disminuye progresivamente su valor con una pendiente que depende del tiempo *tR* que haya sido ajustado.

5.4.6 Interrupción panel táctil

Para gestionar los eventos producidos en el panel táctil se definen dos rutinas de interrupción, asociadas al slider y a los botones, respectivamente.

En la rutina del slider, los valores leídos pueden relacionarse con el control de distintas variables:

- Si está activo el control del volumen (`volumen_ON`), el valor del slider se asocia a la variable `volumen`, escalándolo convenientemente.
- Si está activo el control de desfase de alguno de los instrumentos (`control_fase[instrum_selec]`), se actualiza la variable asociada y se invoca la función que recalcula la forma de onda aditiva.
- Si está activo el control de efecto:
  - Si tenemos efecto FM, el valor del slider actualiza el índice de modulación.
- Si está activo el control ADSR, que se acciona desde la ventana guix asociada al mismo, en función del botón activo (A, D, S ó R), actualizamos las variables asociadas a los mismos (`tA, tD, AmpSust y tR`).



5.5 Estructura del código

Los principales archivos del proyecto se alojan en la carpeta "src". En ella encontramos los archivos siguientes:

- **Carpeta captouch_configs:** contiene los archivos de calibración del panel táctil, que se generan automáticamente desde la aplicación *Capacitive Touch Workbench.*
- **Carpeta gui:** contiene las funciones de la pantalla táctil y sus parámetros. Estos archivos son creados desde el proyecto de la aplicación Guix Studio, con en que se encuentra enlazada la carpeta.
- **Synergy_gen:** contiene archivos de configuración de los módulos y pines del microcontrolador; se actualizan desde la ventana de configuración, que veremos en la siguiente sección.
- **Archivos** fuente .c y cabeceras del programa .h.

Figura 5-24. Directorio del proyecto.

Los principales archivos fuente en los que se basa nuestro sintetizador son:

- **Midi_entry.c:** contiene las instrucciones del hilo principal, definidas en el capítulo 5.4.1.
- **Libreria_midi.c:** inicializa las variables del programa, y recoge todas la funciones de interrupción así como la función de cálculo de la onda.
- **Libreria_midi.h:** cabecera de la librería anterior, que se incluye en los ficheros de sendos hilos, pues recoge los prototipos de las funciones, las variables globales y los macros del programa.
- **Main_thread_entry.c:** contiene las funciones de configuración de la pantalla, de las comunicaciones puerto serie con el microcontrolador y de gestión de eventos táciles. Se incluye en el proyecto pero no es modificado por el usuario[9].
- **Guiapp_event_handlers.c:** contiene las funciones de control de las distintas ventanas de la pantalla LCD. Es la parte del código asociado a la LCD que programa el usuario.
- **Formas.c:** define la tabla de ondas como una matriz de 8x64, constituida por 8 vectores asociados a los distintos instrumentos del sintetizador, y 64 muestras por cada una de las ondas identificadas con ellos. Recoge también el vector de frecuencias vinculadas a las notas musicales.
- **Formas.h:** cabecera de la librería anterior, que debe incluirse en el hilo principal para poder acceder a las variables y tablas que contiene.

El resto de archivos que se observan en la carpeta src de la figura 5.8 son aquellos relacionados con el filtro, que no hemos llegado a integrar finalmente en el proyecto, pero dejamos constancia de ellos como semilla para posibles mejoras futuras.

[9] Código descargable en la galería de Renesas: Search | Renesas Electronics Corporation

6 RESULTADOS

Si buscas resultados distintos,

no hagas siempre lo mismo.

- Albert Einstein-

A pesar de que los resultados sonoros de nuestro sintetizador no pueden plasmarse sobre el texto, en este capítulo recogeremos como productos "tangibles" del proyecto las medidas de las ondas tomadas a la salida del convertidor DAC mediante el osciloscopio. Se mostrarán a continuación distinas configuraciones del sintetizador, presentando las opciones seleccionadas desde las ventanas de la LCD, junto a las capturas de las ondas producidas. Para la captura de los resultados en todo momento se estará generando una nota *Do* central, no siendo de interés la frecuencia de la onda dado que no afecta a las variaciones de su silueta.

Siguiendo el orden en el que se han presentado los distintos menús del programa, testearemos en primer lugar el módulo de los osciladores, DCO. Se muestran a continuación las 8 formas de onda básicas que pueden generarse:

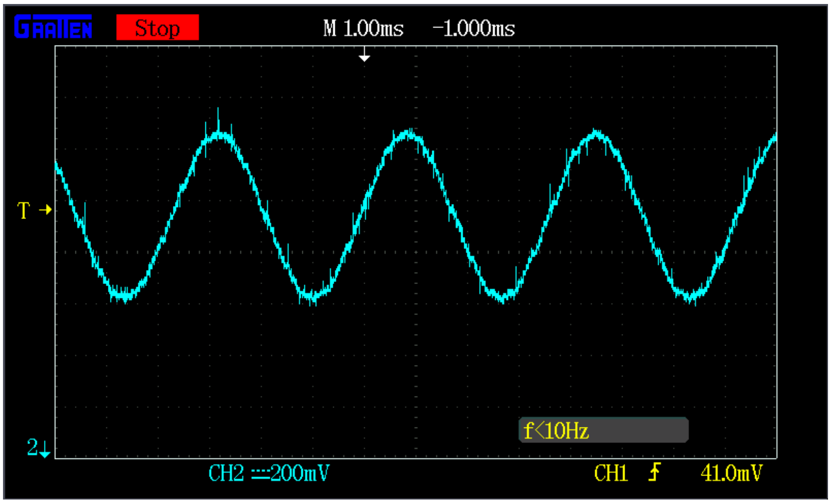


Figura 6-1. Onda senoidal.

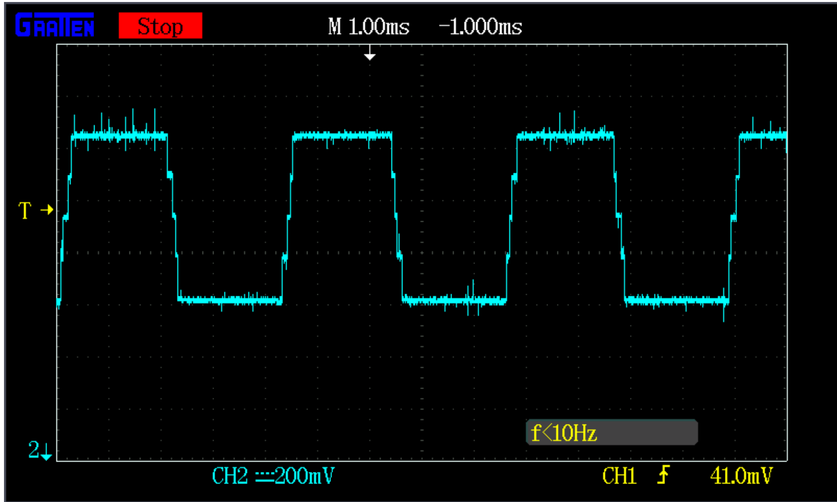


Figura 6-2. Onda cuadrada tipo 1.

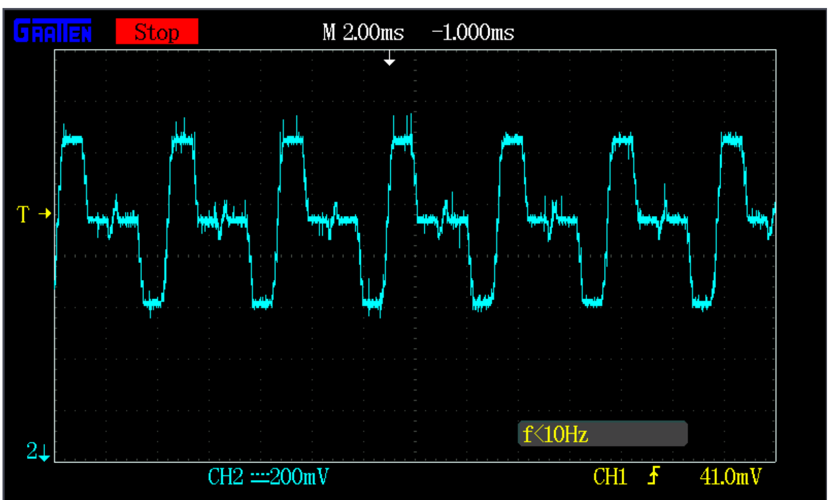


Figura 6-3. Onda cuadrada tipo 2.

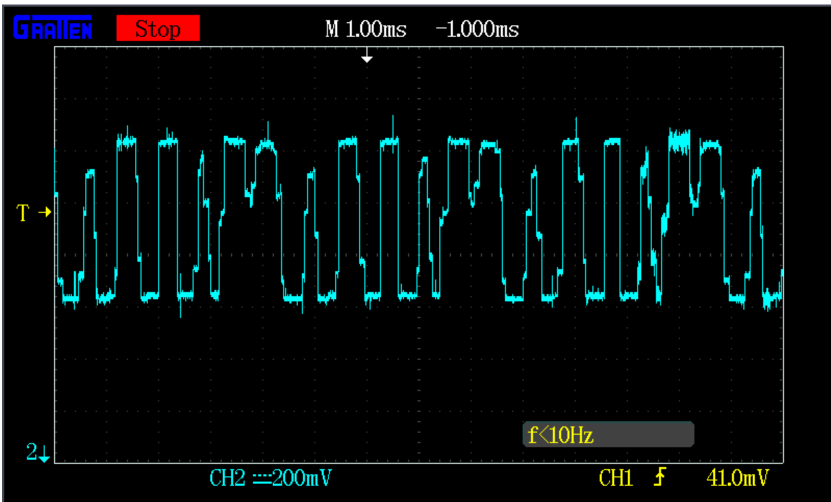


Figura 6-4. Onda tipo Guitarra.

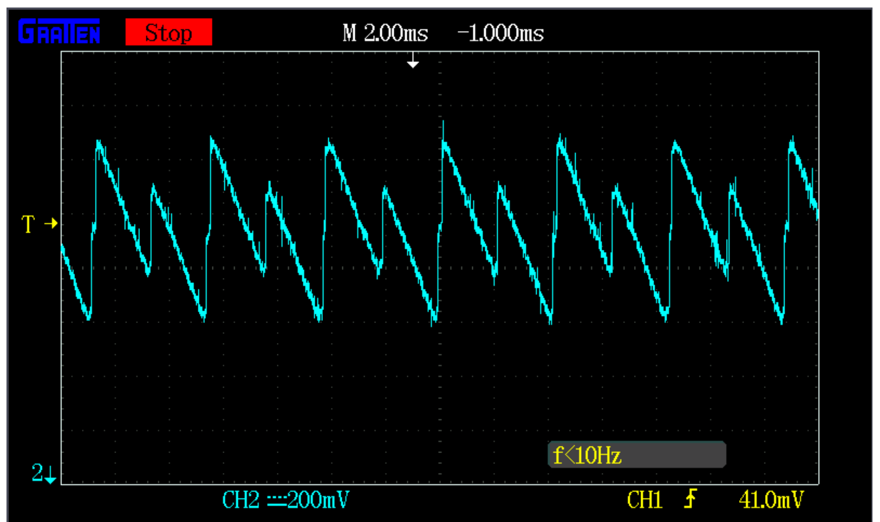


Figura 6-5. Onda tipo Órgano.

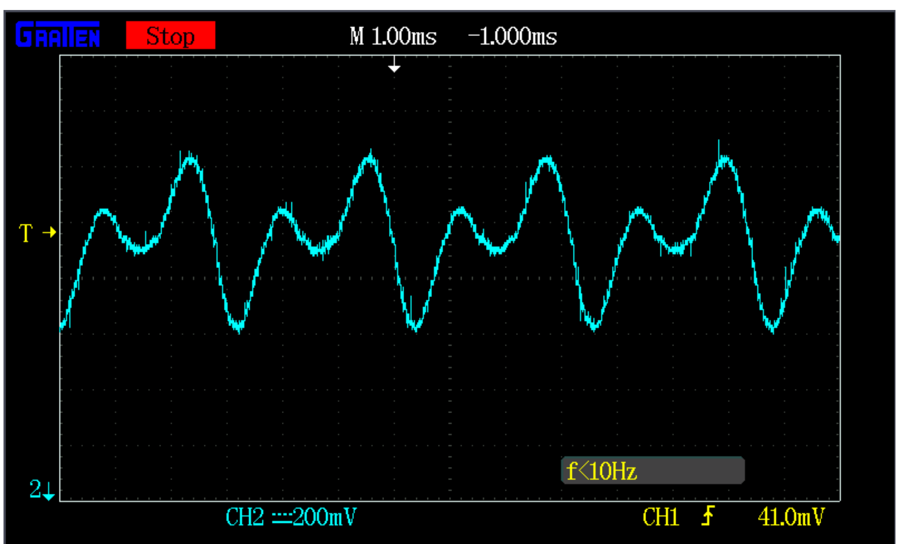


Figura 6-6. Onda tipo Oboe.

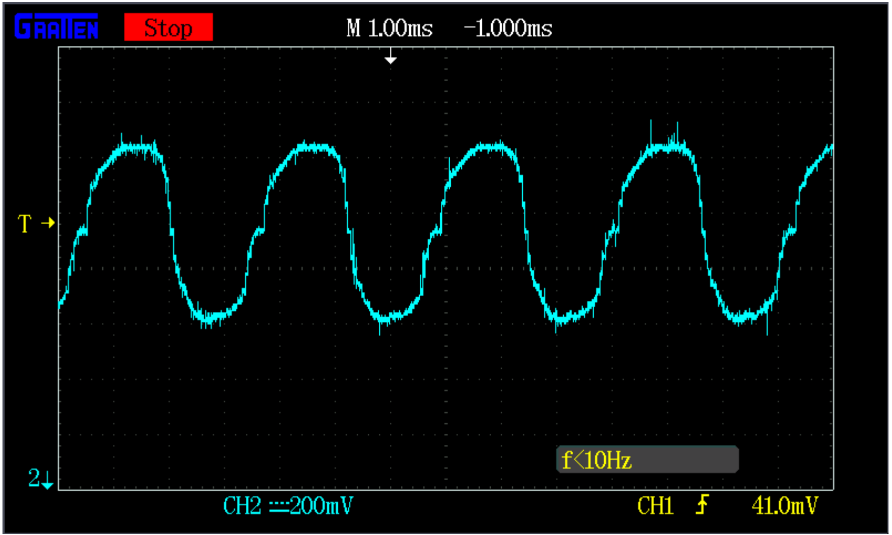


Figura 6-7. Onda tipo *Hand-drawn.*

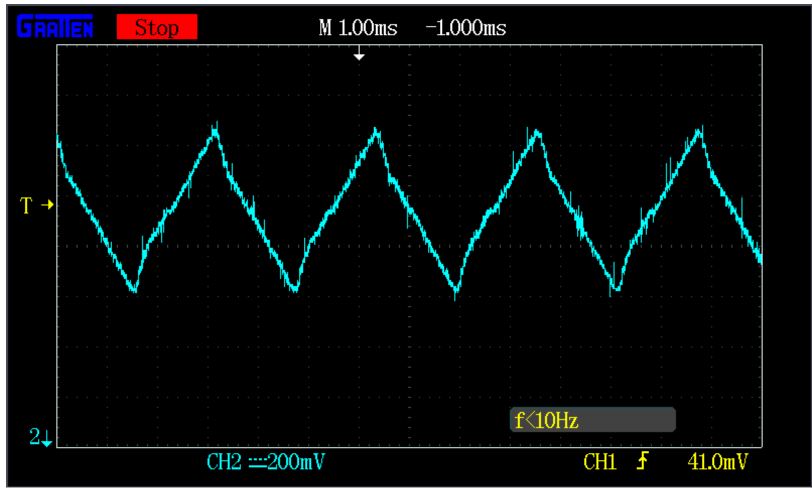


Figura 6-8. Onda tipo *Video Game.*

Seleccionando varios osciladores simultáneamente, se llevará a cabo la síntesis aditiva de la onda; se muestran a continuación varios ejemplos:

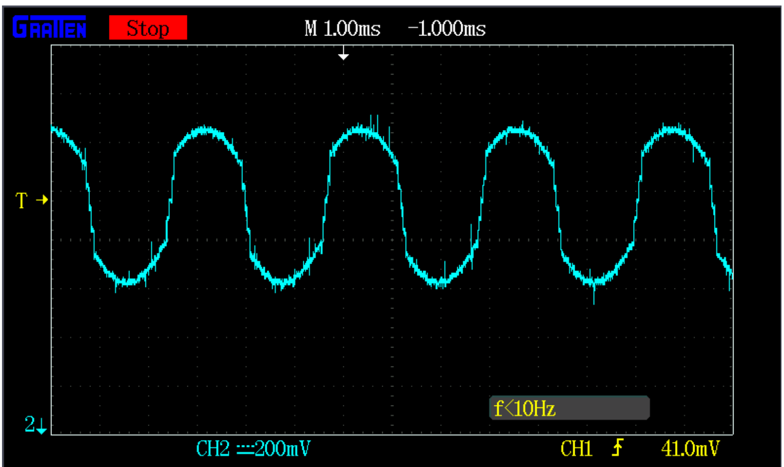


Figura 6-9. Onda senoidal + ona cuadrada tipo 1.

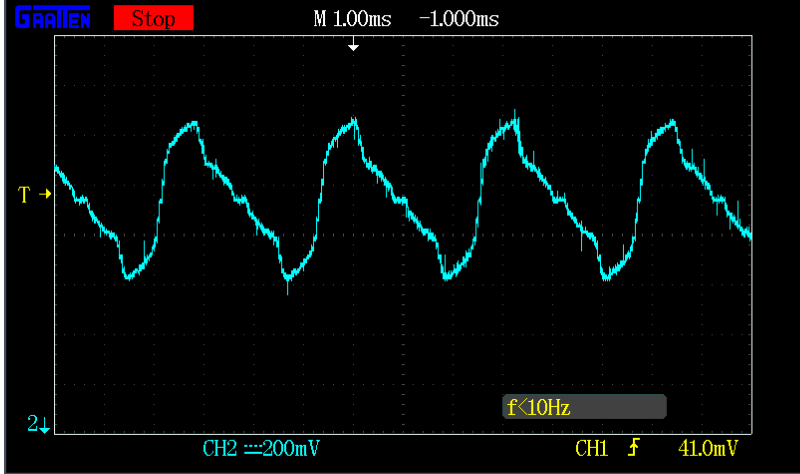


Figura 6-10. Onda senoidal + ona cuadrada tipo 2.

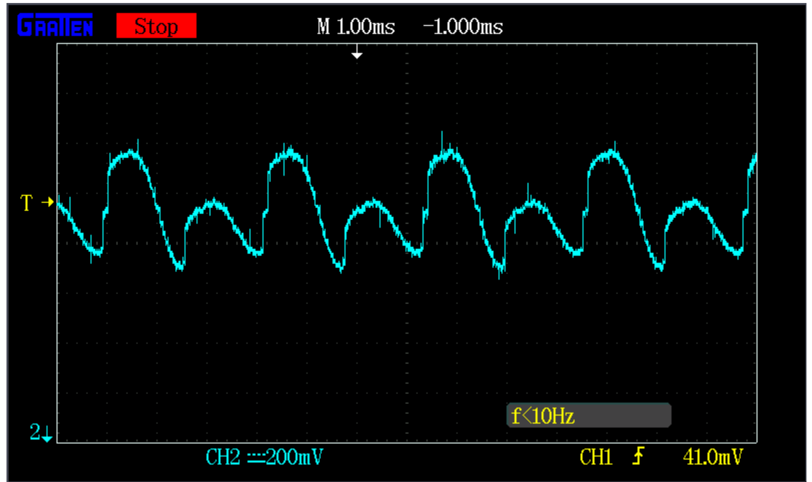


Figura 6-11. Onda de Órgano + Oboe

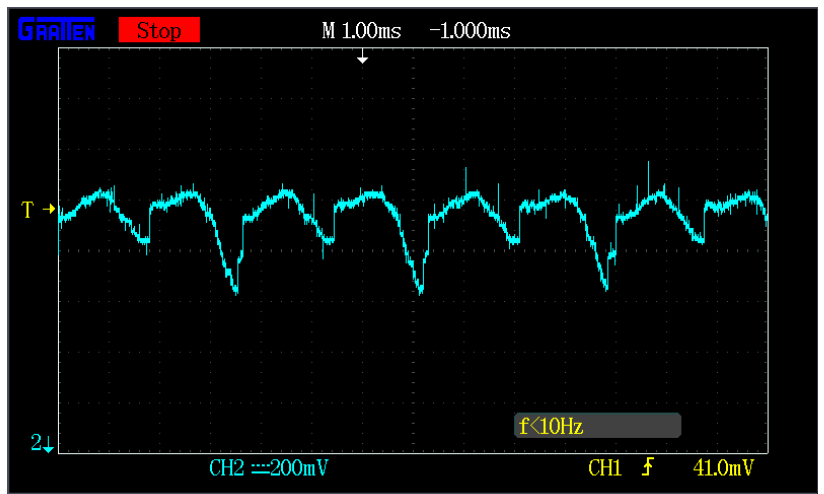


Figura 6-12. Ídem con desfase de valor 30.

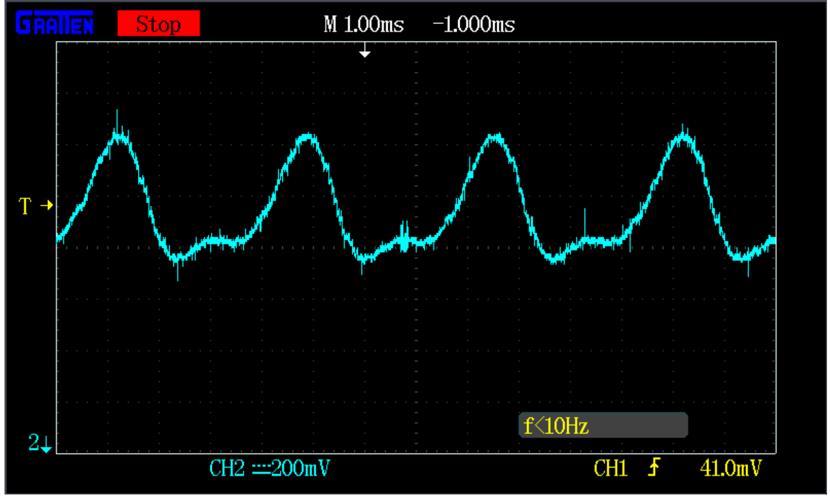
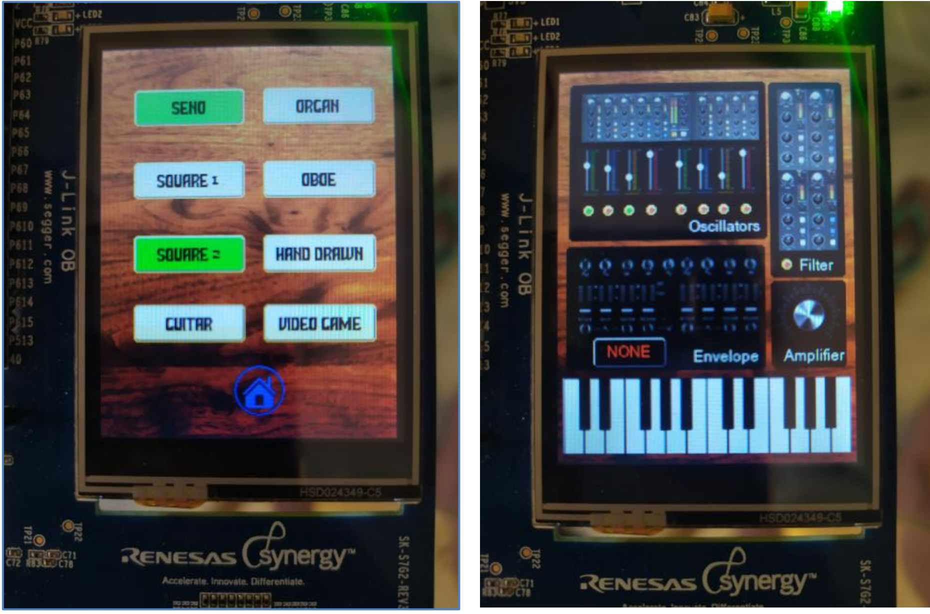


Figura 6-13. Onda senoidal + cuadrada tipo 1 con desfase 50.

Nótese que en el menú principal aparecen los indicadores luminosos de las ondas seleccionadas. En el último ejemplo mostrado (seno+cuadrada) las ventanas de la LCD mostrarán la apariencia siguiente:



A continuación se desmostrará el funcionamiento del módulo LFO, capturando una serie de ondas moduladas con las 4 envolventes posibles:

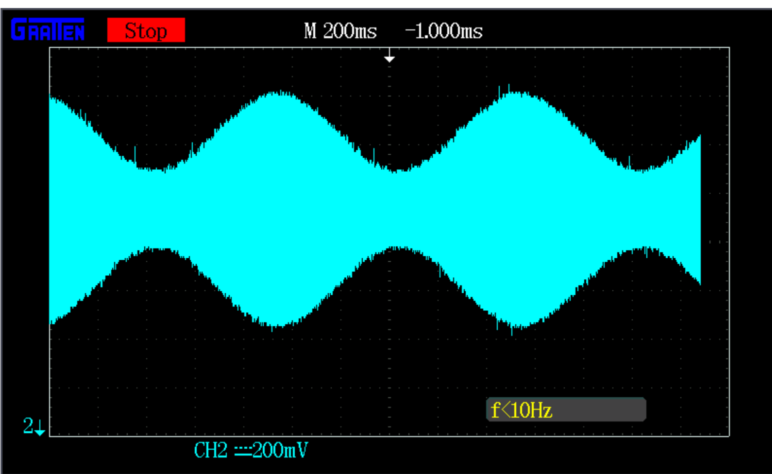


Figura 6-14. Onda senoidal con efecto trémolo. Envolvente senoidal.

Idem con cuadrada

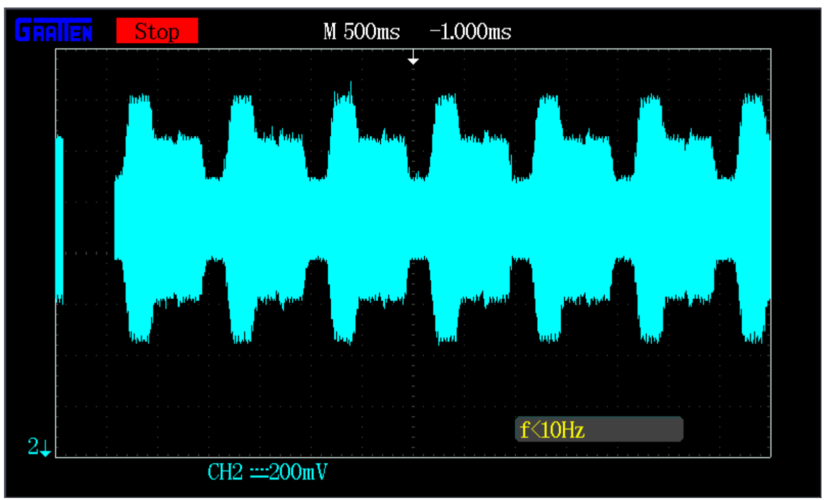


Figura 6-15. Onda senoidal con efecto trémolo. Envolvente cuadrada.

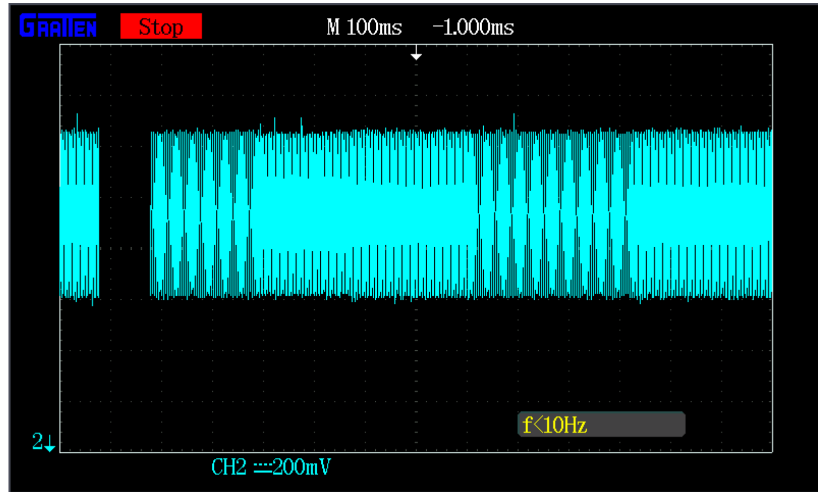


Figura 6-16. Onda senoidal con efecto vibrato. Envolvente tipo “órgano”.

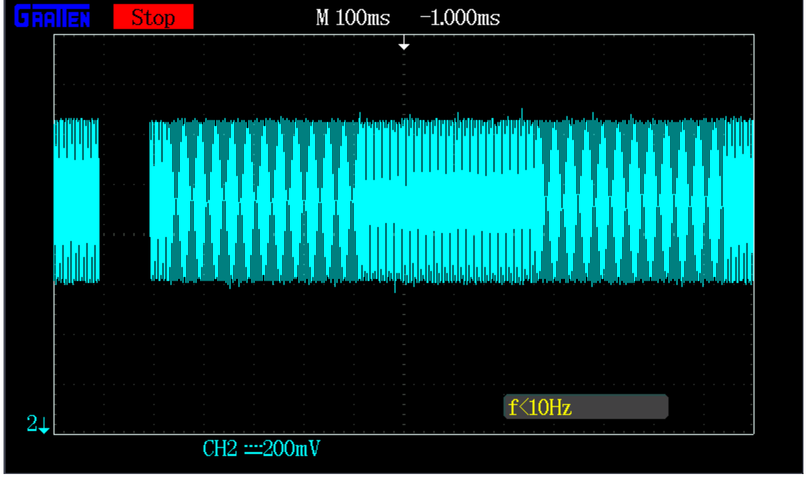


Figura 6-17. Onda senoidal con efecto vibrato. Envolvente cuadrada.

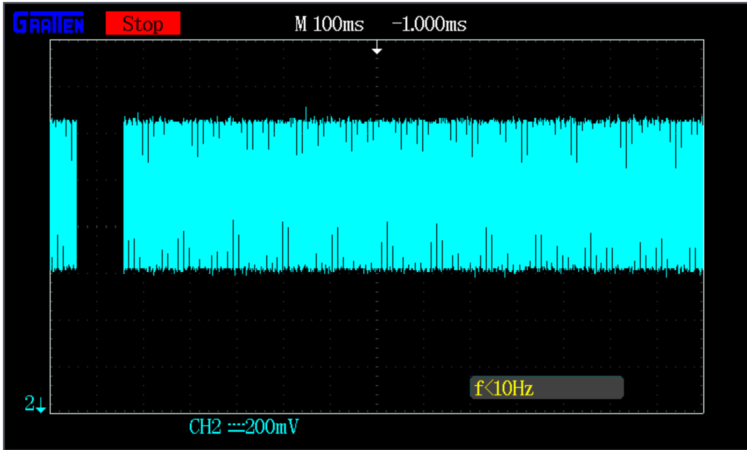


Figura 6-18. Onda senoidal con efecto FM. Envolvente senoidal.

Nótese en estos últimos dos efectos de modulación en frecuencia que las formas de onda no varían demasiado visualmente según se elija una envolvente u otra, dado que éstas no varían la amplitud sino solo la frecuencia, y la diferencia de este efecto entre distintas moduladoras es inapreciable.

Por último se muestra el efecto ADSR sobre una onda tipo Oboe, con tiempos de ataque y descenso de 50 milisegundos, tiempo de liberación de la nota de valor igual a 70 milisegundos, y con un *sustain* del 70%.

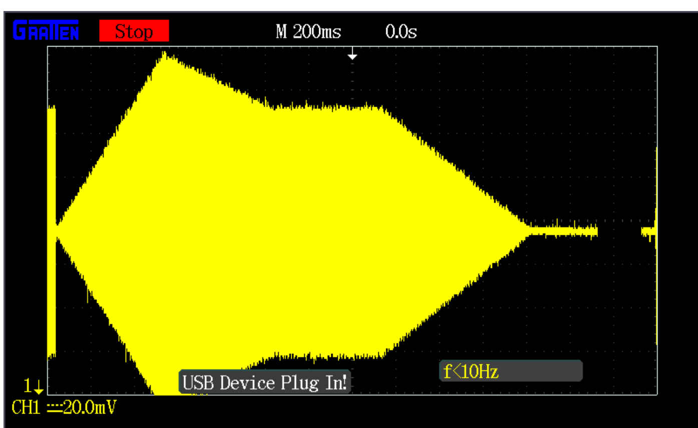


Figura 6-19. Onda senoidal con efecto ADSR.

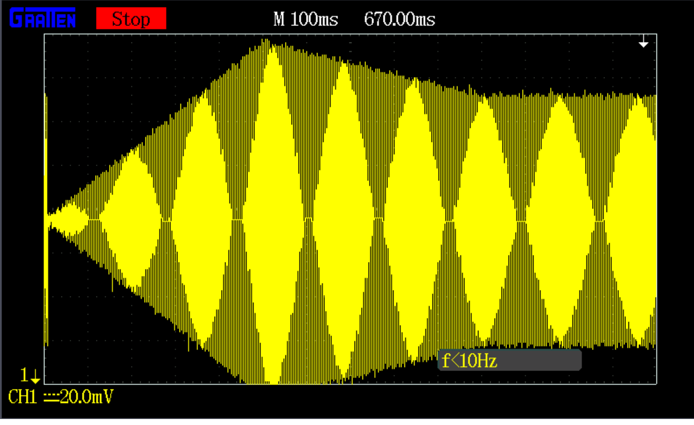


Figura 6-20. Detalle del efecto ADSR.

7 CONCLUSIONES

La música empieza donde se acaba el lenguaje.
E. T. A. Hoffmann

Si bien es cierto que hemos topado con diversos obstáculos durante el desarrollo del proyecto, toda piedra en el camino ha servido para aportarnos conocimiento: los problemas encontrados en la etapa inicial del proyecto, debido a las limitaciones presentadas por la placa WiPy 3.0, nos ha permitido empero familiarizarnos con el entorno de programación Micropython; de igual modo en nuestro intento de implementar el módulo de filtrado, hemos ahondado en diversos conceptos sobre procesamiento de señal y respuesta en frecuencia de sistemas.

Cabe decir que la implementación del sintetizador en la placa SK-S7G2 partiendo desde cero no ha sido sencilla, puesto que no teníamos experiencia alguna en el manejo de este microcontrolador, ni existían muchos trabajos previos sobre el mismo. Para lograr las especificaciones de funcionamiento del Midisynth en la nueva placa ha sido necesario un estudio minucioso de las posibilidades de cada uno de sus módulos, incorporándolos de forma progresiva, siguiendo un criterio de trabajo conservador. La programación de la pantalla táctil ha exigido bastantes horas de trabajo; el conocimiento sobre su manejo y la colección de funciones que hemos programado, recogidas en esta memoria, constituyen una base muy útil para futuros proyectos sobre este microcontrolador.

En definitiva, los avances aportados por Midisynth 3.0 no son baladíes: hemos logrado un dispositivo Midisynth más compacto, que integra microcontrolador y periféricos en una misma placa, una interfaz más intuitiva que las versiones anteriores, gracias a la pantalla LCD táctil a color, todo ello implementado sobre un microcontrolador con potencia de cálculo superior, que permitirá abrir líneas de trabajo futuro.

Como línea de mejora podría proponerse en primer lugar la programación del filtro pasabandas cuyo estudio se ha iniciado en este proyecto. Una idea más ambiciosa, una vez implementado el filtro, sería la creación de otro módulo LFO cuya envolvente afectara en este caso a los parámetros del filtro anterior. Esto supondría un mayor acercamiento a instrumentos clásicos, cuyos armónicos sabemos que varían temporalmente durante la ejecución de cada una de sus notas.

Propuestas adicionales podrían ser: la ampliación del número de puntos en la tabla de ondas para una mayor resolución del sintetizador; o la interpretación por parte del programa de mensajes MIDI complementarios, pues como se expuso en el capítulo segundo el protocolo MIDI permite la codificación de 8 tipos de mensajes, entre ellos los dos que hemos empleado (Note_ON/Note_OFF).

Confiamos haber despertado el interés del lector con este granito de arena aportado al campo de la síntesis musical; no obstante, las posibilidades de perfeccionamiento del Midisynth continúan siendo numerosas, tomando las palabras del griego Hipócrates: <<El arte es largo, la vida breve>>.

ANEXO I

En este apartado se incluyen los ficheros fuente de los códigos de prueba implementados para la placa Wipy 3.0.

- **prueba_botones.py**

```
globals().clear()                # Reseteo variables globales

from machine import Timer, DAC
from machine import Pin
import machine

import time
import math
import pycom
import array
import os

# Desactivar WTD de pybytes
# pybytes.set_config("connection_watchdog", value=False)

nota = False
mensaje = False

pycom.heartbeat(False)           # Pa q se quite la luz azul tan desagradable

pi = math.pi
t_paso = 1/20000                 # Cada cuanto tiempo muestreo (resolución de la onda)
-> 50us
frecuencia = 440                 # Nota LA4 - 440Hz
T = 1/frecuencia/(50*10**-6)
paso = 2*pi/T                    # Tamaño del paso ponderado para el seno

# Inicialización del DAC
dac = DAC('P22')                 # Crear un DAC object en el pin 21

global onda
onda = array.array('f', [0.0])

def button_ON_handler(arg):      # Función de interrupción para cuando se pulsa el
botón - prueba activar nota
    global nota
    global mensaje

    print("bandera1")

    if(button() == 0):                   # Botón pulsado
        mensaje = True
```

```
        nota = True
    else:
        mensaje = True
        nota = False
        print("bandera2")

# Botón S1
button = Pin('G17', mode = Pin.IN, pull=Pin.PULL_UP)   # Pull-up porque la
resistencia está a valor HIGH cuando el botón no está pulsado
button.callback(Pin.IRQ_FALLING | Pin.IRQ_RISING, button_ON_handler)

while True:
    if mensaje == True:                   # Mensaje indica que ha habido un flanco
en la variable nota
        #state = machine.disable_irq()
        #button.callback(handler=None)
        ### SECCIÓN CRÍTICA ###
        if nota == True:
            dac.init()
            pycom.rgbled(0xff00)          # Luz verde si hay nota
            #dco = DCO()         # Vuelvo a crear la alarma - NO FUNCIONA REASIGNAR
LA FUNCIÓN CALLBACK

        else:
            dac.deinit()                  # Detengo el DAC
            pycom.rgbled(0x7f0000)        # Luz ROJA si no hay nota

        mensaje = False
        ### SECCIÓN CRÍTICA ###
        #machine.enable_irq(state)
        #button = Pin('G17', mode = Pin.IN, pull=Pin.PULL_UP)   # Pull-up porque la
resistencia está a valor HIGH cuando el botón no está pulsado
        #button.callback(Pin.IRQ_FALLING | Pin.IRQ_RISING, button_ON_handler)
```

- **prueba_DCO.py**

```
globals().clear()               # Reseteo variables globales

from machine import Timer, DAC
from machine import SD
from machine import Pin
import machine

import time
import math
import pycom
import array
import os

# Desactivar WTD de pybytes
# pybytes.set_config("connection_watchdog", value=False)

nota = True
x = 0

pycom.heartbeat(False)          # Pa q se quite la luz azul tan desagradable
```

```
pi = math.pi
t_paso = 1/20000               # Cada cuanto tiempo muestreo (resolución de la onda)
-> 50us
frecuencia = 440               # Nota LA4 - 440Hz
T = 1/frecuencia/(50*10**-6)
paso = 2*pi/T                  # Tamaño del paso ponderado para el seno

# Inicialización del DAC
dac = DAC('P22')               # Crear un DAC object en el pin 21

global onda
onda = array.array('f', [0.0])

class DCO:

    def __init__(self):
        self.n = 0             # Variable - CONTADOR de minipasos en cada forma de
onda
        self.__alarm = Timer.Alarm(self._microseconds_handler,
us=int(t_paso*(10**6)), periodic=True)  # Convierto el paso a microsegundos y
convierto de float a int

    def _microseconds_handler(self, alarm):
        global nota
        global i


        #print("aqui estoy")
        if nota == True:
            #i+=1
            self.n += 1                            # Incremento el contador
            if self.n > round(T):                  # Si el contador supera al
valor de pasos de un periodo
                contador = self.n % round(T)       # Convierto el contador a la
primera forma de onda
            else:
                contador = self.n
            #print("y he pasado por aqui tambien y el contador vale ", contador)
            dac.write(onda[contador])
            #x = onda[contador]
            #print(x)
"""
        else:
            self.n = 0                             # Reseteo contador de la onda
            alarm.callback(None)                   # Dejar de interrumpir para
generar la onda
            i = 0 # Reseteo el contador
"""
# Bucle para calcular la forma de onda
for i in range(0, round(T)):
    #onda_i = round( (math.sin(paso*i)+1)/2, 3)                # Modelo el valor
entre 0 y 1
    onda_i = (math.sin(paso*i)+1)/2
    #if len(str(onda_i)) > 5:                                  # len() cuenta
tanto la cifra entera como el punto; quiero tener 3 decimales
    #    onda_i = round(onda_i,2)                              # Si no es
posible, redondeo a 2 decimales
    onda.append(onda_i)

dac.init()
```

```
pycom.rgbled(0xff00)            # Luz verde si hay nota
dco = DCO()
#dco.__alarm.callback(None)


i = 0 # Contador de prueba

while True:
    pass
"""
    if i==10**5:   ## Cuando pasan 5 segundos desactivo la nota (el bucle se ejecuta 
cada 50microsegundos)
        nota = False
        dac.deinit()                    # Detengo el DAC
        pycom.rgbled(0x7f0000)          # Luz ROJA si no hay nota

    if nota == False:
        i += 1
        if i == 10**3:
            nota = True
            #print("llegue aqui")
            pycom.rgbled(0xff00)            # Luz verde si hay nota
            dco = DCO()
"""
```

- **prueba_boton_y_DCO.py**

```
globals().clear()               # Reseteo variables globales

from machine import Timer, DAC
from machine import Pin
import machine

import time
import math
import pycom
import array
import os

# Desactivar WTD de pybytes
# pybytes.set_config("connection_watchdog", value=False)

nota = False
mensaje = False

pycom.heartbeat(False)          # Para desactivar la luz azul tan desagradable

pi = math.pi
t_paso = 1/20000                # Cada cuanto tiempo muestreo (resolución de la onda) 
-> 50us
frecuencia = 440                # Nota LA4 - 440Hz
T = 1/frecuencia/(50*10**-6)
paso = 2*pi/T                   # Tamaño del paso ponderado para el seno

# Inicialización del DAC
dac = DAC('P22')                # Crear un DAC object en el pin 21
```

```
global onda
onda = array.array('f', [0.0])

class DCO:

    def __init__(self):
        self.n = 0            # Variable - CONTADOR de minipasos en cada forma de
onda
        self.__alarm = Timer.Alarm(self._microseconds_handler,
us=int(t_paso*(10**6)), periodic=True)  # Convierto el paso a microsegundos y
convierto de float a int

    def _microseconds_handler(self, alarm):
        global nota

        print("DCO")

        if nota == True:
            self.n += 1                                  # Incremento el contador
            if self.n > round(T):                        # Si el contador supera al
valor de pasos de un periodo
                contador = self.n % round(T)             # Convierto el contador a la
primera forma de onda
            else:
                contador = self.n
            #print("y he pasado por aqui tambien y el contador vale ", contador)
            dac.write((onda[contador] + 1)/2)             # Modelo el valor entre 0 y
1
            x = (onda[contador] + 1)/2
            #print(x)
        else:
            self.n = 0                                   # Reseteo contador de la onda
            alarm.callback(None)                         # Dejar de interrumpir para
generar la onda


# Bucle para calcular la forma de onda
for i in range(0, round(T)):
    onda_i = math.sin(paso*i)
    onda.append(onda_i)

dco = DCO()
dco.__alarm.callback(None)

def button_ON_handler(arg):     # Función de interrupción para cuando se pulsa el
botón - prueba activar nota
    global nota
    global mensaje
    global dco

    dco.__alarm.callback(None)
    print("bandera1")

    if(button() == 0):                   # Botón pulsado
        mensaje = True
        nota = True
    else:
        mensaje = True
        nota = False
        print("bandera2")
```

```
# Botón S1
button = Pin('G17', mode = Pin.IN, pull=Pin.PULL_UP)   # Pull-up porque la
resistencia está a valor HIGH cuando el botón no está pulsado
button.callback(Pin.IRQ_FALLING | Pin.IRQ_RISING, button_ON_handler)

while True:
    if mensaje == True:                    # Mensaje indica que ha habido un flanco
en la variable nota
        print("main")
        dco.__alarm.callback(None)
        #button.callback(handler=None)
        ### SECCIÓN CRÍTICA ###
        if nota == True:
            dac.init()
            pycom.rgbled(0xff00)           # Luz verde si hay nota
            dco = DCO()         # Vuelvo a crear la alarma - NO FUNCIONA REASIGNAR LA
FUNCIÓN CALLBACK

        else:
            dac.deinit()                   # Detengo el DAC
            pycom.rgbled(0x7f0000)         # Luz ROJA si no hay nota

        mensaje = False
        ### SECCIÓN CRÍTICA ###
        #machine.enable_irq(state)
        #button = Pin('G17', mode = Pin.IN, pull=Pin.PULL_UP)   # Pull-up porque la
resistencia está a valor HIGH cuando el botón no está pulsado
        #button.callback(Pin.IRQ_FALLING | Pin.IRQ_RISING, button_ON_handler)
```

- **seno_con_globales.py**

```
globals().clear()               # Reseteo variables globales

from machine import Timer
from machine import DAC
import time
import math
import pycom
import array

# Desactivar WTD de pybytes: con desactivarlo una vez vale
#pybytes.set_config("connection_watchdog", value=False)

pycom.heartbeat(False)          # Pa q se quite la luz azul tan desagradable

pi = math.pi
t_paso = 1/20000                # Cada cuanto tiempo muestreo (resolución de la onda)
-> 50us
frecuencia = 440                # Nota LA4 - 440Hz
N = 1/frecuencia/(50*10**-6)
paso = 2*pi/N                   # Tamaño del paso ponderado para el seno

# Inicialización del DAC
#dac = DAC('P22')        # Crear un DAC object en el pin 22
```

```
#dac.write(0.5)         # prueba: El output va entre 0 y 1; hay que modelar la onda
posteriormente

global mi_vector
mi_vector = array.array('f', [0.0])

class Clock:

    def __init__(self):
        self.n = 0            # Variable - CONTADOR de minipasos en cada forma de
onda
        self.vector = array.array('f', [0.0])
        self.__alarm = Timer.Alarm(self._microseconds_handler,
us=int(t_paso*(10**6)), periodic=True)  # Convierto el paso a microsegundos y
convierto de float a int

    def _microseconds_handler(self, alarm):
        self.n += 1                                   # Incremento el contador
        onda = math.sin(paso*self.n)
        self.vector.append(onda)
        #dac.write((onda + 1)/2)                      # Modelo el valor entre 0 y 1

        # El valor de onda va entre [-1,1] -> es el rango de la función SENO

        if self.n == round(N):
            #dac.deinit()           # Paro el DAC
            print ("Número de pasos: %d" %self.n)
            for i in range(0, len(self.vector)):
                print(self.vector[i])
                mi_vector.append(self.vector[i])
                """
                # Send data  to Pybytes
                pybytes.send_signal(1, self.vector[i])
                print('sent signal {}'.format(i))

                """
            print(mi_vector)
            alarm.callback(None)     # stop counting after 10 seconds

clock = Clock()
```

- **seno_con_tabla_ext.py**

```
globals().clear()              # Reseteo variables globales

from machine import Timer
from machine import DAC
from machine import SD

import time
import math
import pycom
import array
import os

# Desactivar WTD de pybytes
#pybytes.set_config("connection_watchdog", value=False)
```

```
pycom.heartbeat(False)          # Pa q se quite la luz azul tan desagradable

pi = math.pi
t_paso = 1/20000                # Cada cuanto tiempo muestreo (resolución de la onda)
-> 50us
frecuencia = 440                # Nota LA4 - 440Hz
N = 1/frecuencia/(50*10**-6)
paso = 2*pi/N                   # Tamaño del paso ponderado para el seno

# Inicialización del DAC
dac = DAC('P22')                # Crear un DAC object en el pin 22

# Load SD card
sd = SD()
os.mount(sd, '/sd')         # Con hacerlo una vez vale, si no se ralla
os.listdir('/sd')           # Para ver lo que hay guardado ya en la SD

# Borro archivo preexistente
os.remove('/sd/onda.txt')

global onda
onda = array.array('f', [0.0])

f = open('/sd/onda.txt', 'a')

for i in range(0, 4*round(N)):
    onda_i = math.sin(paso*i)
    onda.append(onda_i)
    f.write("%f \n" %(onda_i))

# print(onda)
f.close()       # Cierro archivo txt de la SD



class Clock:

    def __init__(self):
        self.n = 0            # Variable - CONTADOR de minipasos en cada forma de
onda
        # self.vector = array.array('f', [0.0])
        self_valor = 0
        self.__alarm = Timer.Alarm(self._microseconds_handler,
us=int(t_paso*(10**6)), periodic=True)  # Convierto el paso a microsegundos y
convierto de float a int

    def _microseconds_handler(self, alarm):
        self.n += 1                                  # Incremento el contador

        if self.n != 4*round(N):
            self_valor = onda[self.n%round(N)]
            dac.write((self_valor + 1)/2)                # Modelo el valor entre 0 y
1

        else:
            print("FUNCIONA")
            dac.deinit()                             # Paro el DAC
            os.umount('/sd')
            alarm.callback(None)
```

```
        # El valor de onda va entre [-1,1] -> es el rango de la función SENO
        """
        if self.n == round(N):
            #dac.deinit()            # Paro el DAC
            print ("Número de pasos: %d" %self.n)

            for i in range(0, len(self.vector)):
                print(self.vector[i])
                mi_vector.append(self.vector[i])
        """

        """"
        # Send data to Pybytes
        pybytes.send_signal(1, self.vector[i])
        print('sent signal {}'.format(i))

        """

clock = Clock()
```

- **seno_indefinido.py**

```
globals().clear()                # Reseteo variables globales

from machine import Timer, DAC
from machine import SD
from machine import Pin
import machine

import time
import math
import pycom
import array
import os

# Desactivar WTD de pybytes
# pybytes.set_config("connection_watchdog", value=False)

nota = False
mensaje = True

pycom.heartbeat(False)           # Pa q se quite la luz azul tan desagradable

pi = math.pi
t_paso = 1/20000                 # Cada cuanto tiempo muestreo (resolución de la onda)
-> 50us
frecuencia = 440                 # Nota LA4 - 440Hz
T = 1/frecuencia/(50*10**-6)
paso = 2*pi/T                    # Tamaño del paso ponderado para el seno

# Inicialización del DAC
dac = DAC('P22')                 # Crear un DAC object en el pin 22

"""
# Load SD card
sd = SD()
```

```
os.mount(sd, '/sd')          # Con hacerlo una vez vale, si no se ralla
os.listdir('/sd')            # Para ver lo que hay guardado ya en la SD

# Borro archivo preexistente
os.remove('/sd/onda.txt')
"""

global onda
onda = array.array('f', [0.0])


def button_ON_handler(arg):     # Función de interrupción para cuando se pulsa el
botón - prueba activar nota
    global nota
    global mensaje
    print("bandera1")
    mensaje = True
    if(button() == 0):                  # Botón pulsado
        nota = True
    else:
        nota = False
        print("bandera2")


class DCO:

    def __init__(self):
        self.n = 0             # Variable - CONTADOR de minipasos en cada forma de
onda
        self.__alarm = Timer.Alarm(self._microseconds_handler,
us=int(t_paso*(10**6)), periodic=True)  # Convierto el paso a microsegundos y
convierto de float a int

    def _microseconds_handler(self, alarm):
        global nota
        global button
        button_local = button

        if nota == True:
            self.n += 1                                  # Incremento el contador
            print("y he pasado por aqui tambien y el contador vale ", contador)
            if self.n > round(T):                       # Si el contador supera al
valor de pasos de un periodo
                contador = self.n % round(T)            # Convierto el contador a la
primera forma de onda
            else:
                contador = self.n
            dac.write((onda[contador] + 1)/2)            # Modelo el valor entre 0 y
1
        else:
            self.n = 0                                  # Reseteo contador de la onda
            alarm.callback(None)                        # Dejar de interrumpir para
generar la onda

        # Como button_ON_handler no tiene prioridad, comprobamos si ha habido un
flanco durante la ejecución
        if button_local!=button:
            print("bandera3")
            nota = not nota
```

```
"""
f = open('/sd/onda.txt', 'a')
"""

# Bucle para calcular la forma de onda
for i in range(0, round(T)):
    onda_i = math.sin(paso*i)
    onda.append(onda_i)
    """
    f.write("%f \n" %(onda_i))
    """
# print(onda)
"""
f.close()       # Cierro archivo txt de la SD
"""

# Botón S1
button = Pin('G17', mode = Pin.IN, pull=Pin.PULL_UP)   # Pull-up porque la
resistencia está a valor HIGH cuando el botón no está pulsado
button.callback(Pin.IRQ_FALLING | Pin.IRQ_RISING, button_ON_handler)

#DCO()                                        # Habilito la interrupción
machine.enable_irq()

while True:
    if mensaje == True:                      # Mensaje indica que ha habido un flanco
en la variable nota
        state = machine.disable_irq()
        mensaje = False                      # Bajo la bandera
        print("he pasado por aqui")
        if nota == True:
            dac.init()
            print("Nota ON")
            pycom.rgbled(0xff00)             # Luz verde si hay nota
        else:
            dac.deinit()                     # Detengo el DAC
            print("Nota OFF")
            pycom.rgbled(0x7f0000)           # Luz ROJA si no hay nota
        machine.enable_irq(state)
```

- **seno_indefinido2.py**

```
globals().clear()               # Reseteo variables globales

from machine import Timer, DAC
from machine import SD
from machine import Pin
import machine

import time
import math
import pycom
import array
import os

# Desactivar WTD de pybytes
```

```
# pybytes.set_config("connection_watchdog", value=False)

nota = False
mensaje = False
mensaje_ant = False
x = 0

pycom.heartbeat(False)          # Pa q se quite la luz azul tan desagradable

pi = math.pi
t_paso = 1/20000                # Cada cuanto tiempo muestreo (resolución de la onda)
-> 50us
frecuencia = 440                # Nota LA4 - 440Hz
T = 1/frecuencia/(50*10**-6)
paso = 2*pi/T                   # Tamaño del paso ponderado para el seno

# Inicialización del DAC
dac = DAC('P22')                # Crear un DAC object en el pin 21

"""
# Load SD card
sd = SD()

os.mount(sd, '/sd')         # Con hacerlo una vez vale, si no se ralla
os.listdir('/sd')           # Para ver lo que hay guardado ya en la SD

# Borro archivo preexistente
os.remove('/sd/onda.txt')
"""

global onda
onda = array.array('f', [0.0])

def button_ON_handler(arg):     # Función de interrupción para cuando se pulsa el
botón - prueba activar nota
    global nota
    global mensaje

    print("bandera1")
    if mensaje_ant == False:
        if(button() == 0):              # Botón pulsado
            mensaje = True
            nota = True
        else:
            mensaje = True
            nota = False
            print("bandera2")


# Botón S1
button = Pin('G17', mode = Pin.IN, pull=Pin.PULL_UP)   # Pull-up porque la
resistencia está a valor HIGH cuando el botón no está pulsado
button.callback(Pin.IRQ_FALLING | Pin.IRQ_RISING, button_ON_handler)

class DCO:

    def __init__(self):
        self.n = 0              # Variable - CONTADOR de minipasos en cada forma de
onda
```

```
        self.__alarm = Timer.Alarm(self._microseconds_handler,
us=int(t_paso*(10**6)), periodic=True)  # Convierto el paso a microsegundos y
convierto de float a int

    def _microseconds_handler(self, alarm):
        global DCO_activo
        DCO_activo = True
        global nota
        global x
        if mensaje_ant == True:                             # Evitamos que haya un
mensaje pendiente
            alarm.callback(None)                            # Dejar de interrumpir
para generar la onda

        else:
            if nota == True:
                self.n += 1                                 # Incremento el contador
                if self.n > round(T):                       # Si el contador supera
al valor de pasos de un periodo
                    contador = self.n % round(T)            # Convierto el contador a
la primera forma de onda
                else:
                    contador = self.n
                #print("y he pasado por aqui tambien y el contador vale ", contador)
                dac.write((onda[contador] + 1)/2)            # Modelo el valor entre
0 y 1
                x = (onda[contador] + 1)/2
                #print(x)
            else:
                self.n = 0                                  # Reseteo contador de la
onda
                alarm.callback(None)                        # Dejar de interrumpir
para generar la onda

"""
f = open('/sd/onda.txt', 'a')
"""

# Bucle para calcular la forma de onda
for i in range(0, round(T)):
    onda_i = math.sin(paso*i)
    onda.append(onda_i)
    """
    f.write("%f \n" %(onda_i))
    """
# print(onda)
"""
f.close()       # Cierro archivo txt de la SD
"""
dco = DCO()
dco.__alarm.callback(None)

while True:
    mensaje_ant = mensaje
    if mensaje == True:                         # Mensaje indica que ha habido un flanco
en la variable nota
        #state = machine.disable_irq()
        button.callback(handler=None)
        ### SECCIÓN CRÍTICA ###
        print("he pasado por aqui")
```

```
        if nota == True:
            dac.init()
            pycom.rgbled(0xff00)            # Luz verde si hay nota
            dco = DCO()         # Vuelvo a crear la alarma - NO FUNCIONA REASIGNAR LA
FUNCIÓN CALLBACK
        else:
            dac.deinit()                    # Detengo el DAC
            pycom.rgbled(0x7f0000)          # Luz ROJA si no hay nota
        ### SECCIÓN CRÍTICA ###
        #machine.enable_irq(state)
        button = Pin('G17', mode = Pin.IN, pull=Pin.PULL_UP)   # Pull-up porque la
resistencia está a valor HIGH cuando el botón no está pulsado
        button.callback(Pin.IRQ_FALLING | Pin.IRQ_RISING, button_ON_handler)
        mensaje_ant = False                         # Bajo la bandera
    #else:
        #print("Soy MAIN y sigo vivo")
```

- **seno_20kHz.py**

```
from machine import Timer
from machine import DAC
import time
import math
import pycom
# Vectores de valores para plotear la onda
import array

pycom.heartbeat(False)      # Pa q se quite la luz azul tan desagradable

pi = math.pi
t_paso = 1/20000    # Cada cuanto tiempo muestreo (resolución de la onda) -> 50us
frecuencia = 440      # Nota LA4 - 440Hz
N = 1/frecuencia/(50*10**-6)
paso = 2*pi/N       # Tamaño del paso ponderado para el seno
# mi_vector = array.array('f', [0.0])

# Inicialización del DAC
dac = DAC('P22')        # Crear un DAC object en el pin 22
#dac.write(0.5)          # El output va entre 0 y 1; hay que modelar la onda
posteriormente


class Clock:

    def __init__(self):
        self.n = 0            # Variable - CONTADOR de minipasos en cada forma de
onda
        self.vector = array.array('f', [0.0])
        self.__alarm = Timer.Alarm(self._microseconds_handler,
us=int(t_paso*(10**6)), periodic=True)  # Convierto el paso a microsegundos y
convierto de float a int

    def _microseconds_handler(self, alarm):
        global mi_vector
        self.n += 1                             # Incremento el contador
        onda = math.sin(paso*self.n)
        self.vector.append(onda)
```

```
        dac.write((onda + 1)/2) # y dividir por dos                         #
Modelo el valor entre 0 y 1

        # print("Valor de la onda: %.2f " % onda)      # Imprimo el contador

        # El valor de onda va entre [-1,1] -> es el rango de la función SENO

        if self.n == round(N):
            dac.deinit()            # Paro el DAC
            print ("Número de pasos: %d" %self.n)
            for i in range(0, len(self.vector)):
                print(self.vector[i])
                # mi_vector[i] = self.vector[i]
                """"
                # Send data  to Pybytes
                pybytes.send_signal(1, self.vector[i])
                print('sent signal {}'.format(i))

                """

            alarm.callback(None)     # stop counting after 10 seconds

clock = Clock()
```

- **SPI.py**

```
globals().clear()                # Reseteo variables globales

import pycom
import machine
import time
import array
from machine import SPI
from machine import Pin

# configure the SPI master @ 20kHz
# This uses the SPI default pins for CLK, MOSI and MISO (``P10``, ``P11`` and
``P14``)
spi = SPI(0, mode=SPI.MASTER, baudrate=20000, polarity=0, phase=0, bits=16,
firstbit=SPI.MSB)
BUF = bytearray(2)
CS = Pin('P7', mode=Pin.OUT)        # Chip Select
CS.value(1)

"""
La conversión a binario es casi directa, cada dígito hexadecimal se puede sustituir
por cuatro bits,
el '0x0' por '0000', el '0x1' por '0001', hasta el '0xf', que equivale a '1111'.

"""

BUF[0:15] = (b'\x00\x07')        # Escribimos un 7
pycom.heartbeat(False)           # Pa q se quite la luz azul tan desagradable

# El DAC externo funciona entre 0 y 4095 (12bits)
```

```
while True:
    # The CS pin must be held low for the duration of a write comand.
    CS.value(0)
    sent = spi.write(BUF)              # Envía 2 bytes al bus the bus (8x2=16bits al 
DAC)
    CS.value(1)
```

- **triangulo.py**

```
globals().clear()               # Reseteo variables globales

import pycom
import machine
import time
import array

pycom.heartbeat(False)          # Pa q se quite la luz azul tan desagradable

# Cronómetro
chrono = machine.Timer.Chrono()


# El DAC funciona entre 0 y 3.3V  (valores entre 0 y 1)

x_i = 0
x = array.array('f', [0.0])     # Tabla de valores para los escalones del triángulo
dac = machine.DAC('P22')        # Creo un objeto DAC asociado al pin 22
contador = 0
#t_DAC = array.array('f', [0.0])
#t_DAC_i = 0

for i in range(0, 101):
    x_i += 0.01
    x.append(x_i)

while True:
    #chrono.start()
    contador += 1
    if contador > 101:
        contador = 0
    dac.write(x[contador])

    #t_DAC.append(t_DAC_i)

    #t_DAC_i = chrono.read_us()
    #chrono.reset()
```

- **triangulo_SPI.py**

```
globals().clear()               # Reseteo variables globales

import pycom
import machine
import time
import array
from machine import SPI
from machine import Pin
```

```
# configure the SPI master @ 20kHz
# this uses the SPI default pins for CLK, MOSI and MISO (``P10``, ``P11`` and
``P14``)
spi = SPI(0, mode=SPI.MASTER, baudrate=20000, polarity=0, phase=1, bits=16,
firstbit=SPI.MSB)

pycom.heartbeat(False)          # Pa q se quite la luz azul tan desagradable

# El DAC externo funciona entre 0 y 4095 (12bits)

x__i = 0
x = []                          # Inicializo una lista de valores binarios
x_aux = 0
contador = 0
BUF = bytearray(2)              # bytearray() es modificable, mientras que el método
bytes() NO lo es
CS = Pin('P7', mode=Pin.OUT)    # Chip Select
CS.value(1)

for i in range(0, 4096):        # Creo la tabla de enteros de 4096 valores
    x_i = bin(i)
    x_aux = x_i[2:len(x_i)]     # Extraigo la parte a la derecha de 0b
    x_i = '0b{:0>16}'.format(x_aux)
    x.append(x_i)

print("He terminado")

while True:

    CS.value(0)
    BUF = x[contador]
    spi.write(BUF)              # Envía 2 bytes al bus the bus (8x2=16bits al DAC)
    print(BUF)
    CS.value(1)

    contador += 1
    if contador > 4095:         # Cuando desborde, RESETEO el diente de sierra (4096
valores*2 = 8192)
        contador = 0
```

- **DAC_externo.py**

```
globals().clear()               # Reseteo variables globales

import pycom
import machine
import time
import array
from machine import SPI
from machine import Pin

# configure the SPI master @ 20kHz
# this uses the SPI default pins for CLK, MOSI and MISO (``P10``, ``P11`` and
``P14``)
spi = SPI(0, mode=SPI.MASTER, baudrate=20000, polarity=0, phase=0, bits=16,
firstbit=SPI.MSB)
```

```
pycom.heartbeat(False)          # Pa q se quite la luz azul tan desagradable

# El DAC externo funciona entre 0 y 4095 (12bits)

x_i = 0
x = array.array('i', [0])       # Tabla de valores para los escalones del triángulo
contador = 0
BUF = bytearray(2)              # bytearray() es modificable, mientras que el método
bytes() NO lo es
CS = Pin('P7', mode=Pin.OUT)    # Chip Select
CS.value(1)

for i in range(0, 4095):        # Creo la tabla
    x_i += 1
    x.append(x_i)

while True:

    bits = bin(x[contador])

    if len(bits) < 14:
        BUF_aux =
    for i in range(1, 1):
        BUF[i] = bits[i-2]

    """
    WRITE COMMAND REGISTER
    Upper half
    0 0 0 0 D1 D2 D3 D4
    Lower half
    D5 D6 D7 D8 D9 10D D11 D12

    """
    spi.write(BUF)              # Envía 2 bytes al bus the bus (8x2=16bits al DAC)

    contador += 1
    if contador > 4095:         # Cuando desborde, RESETEO el diente de sierra
        contador = 0
```

ANEXO II

En este apartado se incluyen los ficheros fuente del programa Midisynth 3.0 completo.

- **hal_entry.c**

```
/* HAL-only entry function */
#include "hal_data.h"
void hal_entry(void)
{
    /* TODO: add your own code here */
}
```

- **midi_entry.c**

```
/* HAL-only entry function */

#include "string.h"
#include <stdbool.h>
#include "midi.h"

#include "formas.h"
#include "libreria_midi.h"
#include "filtros.h"
#include "filtros2.h"
#include "filtros3.h"

int vacio[3];
int valor1[3]  = {1,2,3};
int *valor2;


void midi_entry(void)
{

    // Configuración entrada MIDI - CANAL 8 UART
    UART_midi_in.p_api->open(UART_midi_in.p_ctrl, UART_midi_in.p_cfg);  // Pines P104
(RX) y (P105) para entrada MIDI

    // Configuración del DAC
    g_dac0.p_api->open(g_dac0.p_ctrl, g_dac0.p_cfg);             // Abrimos módulo
DAC para el pin P014
    g_dac0.p_api->start(g_dac0.p_ctrl);

    // PROVISIONAL - DAC1 para ver salida del filtro por el canal2 del osciloscopio:

    g_dac1.p_api->open(g_dac1.p_ctrl, g_dac1.p_cfg);             // Abrimos módulo
DAC para el pin P015
    g_dac1.p_api->start(g_dac1.p_ctrl);

    // Inicialización ADSR
     tA=50; // 100ms
     tD=50; // 100ms
     tR_1=70;
     AmpSust=70; // 80% del total (80*128/100)
```

```
        Amp_a = (float)1000.0/tA;
        Amp_d = (float)(10*(100-AmpSust))/tD;
        Amp_r = (10*AmpSust)/(float)tR_1;

        // Inicializo modulación, por defecto Trémolo
        frecMod = (float)frecMod_ini_tremolo;

        // PROVISIONAL
        //calcula_filtro();
        //calcula_filtro2();
        //calcula_filtro3();

        // pRUEBA PUNTEROS
        valor2 = &valor1[0];
        int *puntero = vacio;
        memcpy(puntero, valor2, 3 * sizeof(int));


        // PROVISIONAL

        flag_Msg = 1;
        estado = 1;


        tx_thread_sleep (20); //10 clock ticks

        //Abrimos las interrupciones lo último para que de tiempo a arrancar
        // Configuración timer0 para salida DCO por el DAC
        Timer_DCO.p_api->open(Timer_DCO.p_ctrl, Timer_DCO.p_cfg);     // Timer0
configurado cada 50 microsegundos, pin P014 para salida DAC120

        // Configuración timer1 para salida LFO por el DAC
        Timer_LFO.p_api->open(Timer_LFO.p_ctrl, Timer_LFO.p_cfg);     // Timer0
configurado cada 50 microsegundos, pines 512 y 511 usados (?)

      while(1)
      {

          if (flag_Msg==1)                          // Msg indica flanco subida/bajada en
"nota"
          {
              flag_Msg = 0;                     // Bajamos la bandera
              if(estado == HAY_NOTA)
              {
                  nota_actual = altura - OFFSET_NOTA;
                  frec = (float)(Freq[nota_actual]);
                  fase = 0;

                  if (efecto == ADSR)
                  {
                      Amp_ADSR = 0.0;                       // Reseteo amplitud y contador
ADSR
                      t_nota_ADSR = 0;
                  }

              }
              else if (estado == ADSR_RELEASE)      // La modulación es tipo ADSR
              {
                  t_nota_off_ADSR = 0;
```

```
            }
        }

        if(flag_modulacion == 1)
        {
            flag_modulacion = 0;                // Bajamos la bandera

            if(efecto == TREMOLO)  frecMod = frecMod_ini_tremolo;
            if(efecto == VIBRATO)  frecMod = frecMod_ini_vibrato;
        }

        if(flag_instrumento == 1)               // Recalculamos la onda para el
cambio de instrumentos
        {
            flag_instrumento = 0;
            if (instrum_activos[instrum_selec] == 0)
             {
                instrum_activos[instrum_selec] = 1; // Activar instrumento en vector
suma
                /*
                if (efecto == ADSR)
                    //fase_ins[instrum] = relat;
                else
                {
                    if(relat >= 0)
                    fase_ins[instrum] = relat;
                    else
                    fase_ins[instrum] = (64-relat);
                }
                */
             }
             else
             {
                 instrum_activos[instrum_selec] = 0;
             }
             calcula_onda();
        }

        if(flag_filtro)
        {
            flag_filtro = 0;                    // Bajo la bandera
            //calcula_filtro2();
            calcula_filtro3();
        }
        tx_thread_sleep (10); //10 clock ticks
    }
}
```

- **libreria_midi.c**

```
/*
 * libreria_midi.c
 *
 *  Created on: 14 jun. 2022
 *      Author: pilar
 */
#include "midi.h"
#include "libreria_midi.h"
#include "filtros.h"
#include "filtros2.h"
#include "filtros3.h"
#include "formas.h"
#include <math.h>
#include "stdio.h"
#include "gui/guiapp_specifications.h"
#include "gui/guiapp_resources.h"
#include "guiapp_event_handlers.h"


/* ********************************* VARIABLES ********************************* */

// Variables para entrada MIDI
volatile float RX_Buffer;
int n;                              // Contador para interrupción MIDI - UART
volatile int flag_Msg = 0;          // Bandera para indicar llegada de mensajes MIDI
completos
volatile int estado = NO_HAY_NOTA;
volatile int altura=60;
volatile float fuerza=50;

// Variables para salida DAC
dac_size_t DAC_data12 = 0;                          // El UInt16 representa enteros
sin signo de 16 bits, pero tenemos que saturar en 12 bits que es la verdadera
reolución del DAC
//PROVISIONAL
dac_size_t DAC_data_filtro = 0;
int forma[64];                                      // Vector alturas: 64 puntos de
onda calculada
float DAC_aux;
int p;                                              // Contador
float calcula_offset[64];
float valor_max, valor_max_ciclo;
volatile int nota_actual = 0;
volatile float frec, fase;                          // Fase (w·t), es importante que
sean unsigned porque son punteros a tablas

// Variables para selección de instrumentos
int instrum_activos[NUMINST];                           // Vector de 0s/1s: indica
instrumentos activos
int instrum_selec = 1;                                  // Instrumento seleccionado
[1,8]
int N_instrum;                                          // No. de instrumentos
activos
int control_fase[NUMINST];
volatile float relat;                                              // Auxiliar para
fase relativa
```

```
int fase_ins[NUMINST];                                   // Vector fases relativas de
instrumentos

// Variables para efectos - envolvente LFO
int efecto = 0;
int control_ADSR = 0;
volatile float envolvente;
volatile float faseMod, frecMod, env_FM;
int instrumentMod = 0;
volatile int tA, tD, tR_1, AmpSust;
volatile float Amp_ADSR, Amp_final;
volatile int t_nota_ADSR, t_nota_off_ADSR;
volatile float Amp_a, Amp_d, Amp_r;
int indiceMod = 0;                                     // Índice de modulación para
efecto FM
double frecMod_ini_tremolo = 1 HzM;
double frecMod_ini_vibrato = 5 HzM;

// Para sincronización de la LCD con los controles del panel táctil
volatile int control_efecto;
volatile int ventana_control_ADSR;
volatile int ventana_control_filtro;
volatile char str_ADSR[80];
volatile char str_fase[80];
volatile char nombre_instrum[8][10] = {"Seno", "Square 1", "Square 2", "Guitar",
"Organ", "Oboe", "HandDrawn", "VideoGame"};

// Variables para filtros
volatile int filtro;
volatile int control_filtro = 0;

// Parámetros regulables
int f0 = 400;                              // Frecuencia de resonancia - 500Hz por
defecto
int W = 800;                               // Ancho de banda
int G = 20;                                // Ganancia (en dB)
float Q = 7;                               // Factor Q - Los valores comunes de Q
varían entre 0.2 y 10
float Fs = 20000;                          // Frecuencia de muestreo - valor FIJO,
son 20kHz

volatile double DenC2[FilterOrder+1];
volatile double NumC2[FilterOrder+1];

volatile double VectorE[N_muestras];
volatile double VectorS[N_muestras];

// Variables para panel táctil
volatile int boton = 0;
volatile int volumen_ON;
volatile float volumen = 0.5;
volatile int flag_modulacion = 0;
volatile int flag_instrumento = 0;
volatile int flag_filtro = 0;

// PROVISIONAL
float DAC_aux2;
```

```
/* ********************************* FUNCIONES PARA PERIFÉRICOS
******************************** */


void g_button_framework_user_callback(sf_touch_ctsu_button_callback_args_t *p_args)
{
    switch (p_args->event)
    {
        case TOUCH_BUTTON_STATE_RELEASED:
            flag_filtro = 1;                            // Subo la bandera

            if(p_args->id == 0) boton = BOTON_2;        // BOTÓN 1
            else if (p_args->id == 1) boton = BOTON_1;  // BOTÓN 2

            //calcula_filtro();
            break;

        default:
            break;
    }
    /*
    switch (p_args->event)
    {
        case TOUCH_BUTTON_STATE_RELEASED:
            flag_modulacion = 1;                          // Subo la bandera
            if(p_args->id == 0) boton = BOTON_2;        // BOTÓN 1
            else if (p_args->id == 1) boton = BOTON_1;  // BOTÓN 2

            break;

        default:
            break;
    }
    */
}


void g_slider_framework_user_callback(sf_touch_ctsu_slider_callback_args_t *p_args)
{
    if(volumen_ON)
    {
        if (p_args->id == 1 && (p_args->event == SF_TOUCH_CTSU_SLIDER_STATE_TOUCHED
|| p_args->event == SF_TOUCH_CTSU_SLIDER_STATE_HELD))
        {
                if(p_args->id == 1)
                    volumen = (float)(p_args->current_position/(float)500.0);    //
El volumen va entre 0 y 1
        }
    }

    else if(control_fase[instrum_selec] == 1)
    {
            if (p_args->id == 1 && (p_args->event ==
SF_TOUCH_CTSU_SLIDER_STATE_TOUCHED || p_args->event ==
SF_TOUCH_CTSU_SLIDER_STATE_HELD))
            {
                    if(p_args->id == 1)
                    {
                        relat = ((float)(p_args->current_position/(float)500.0)) -
0.5;   // La fase va entre -0,5 y +0,5
```

```
                        relat = relat*128.0;
// Escalamos entre -64 y +64
                    }
                    int aux_relat;
                    if (efecto == ADSR)
                    {
                        fase_ins[instrum_selec] = (int)relat;
                        aux_relat = fase_ins[instrum_selec];
                    }

                    else
                    {
                        if(relat >= 0)
                        {
                            fase_ins[instrum_selec] = (int)relat;
                            aux_relat = fase_ins[instrum_selec];
                        }
                        else
                        {
                            fase_ins[instrum_selec] = (int)(64-relat);
                            aux_relat = -fase_ins[instrum_selec];
                        }
                    }

                    sprintf(str_fase, "Fase %s = %d", nombre_instrum[instrum_selec],
aux_relat);
                    update_prompt_text((GX_WIDGET *) &_3_control_instrumentos,
ID_param_fase, str_fase);

                    calcula_onda();


            }
    }
    else if(control_efecto)
    {
        if (p_args->id == 1 && (p_args->event == SF_TOUCH_CTSU_SLIDER_STATE_TOUCHED
|| p_args->event == SF_TOUCH_CTSU_SLIDER_STATE_HELD))
        {
                if(p_args->id == 1)
                {
                    relat = ((float)(p_args->current_position/(float)500.0)) - 0.5;
// La fase va entre -0,5 y +0,5
                    relat = relat*24.0;
// La fase va entre -12 y +12
                }

                if(efecto == FM)
                {
                    indiceMod = relat;
                }
                else if (efecto == ADSR)
                {

                }
        }
    }

    else if(ventana_control_ADSR)
        {
```

```
            if (p_args->id == 1 && (p_args->event ==
SF_TOUCH_CTSU_SLIDER_STATE_TOUCHED || p_args->event ==
SF_TOUCH_CTSU_SLIDER_STATE_HELD))
            {
                if(p_args->id == 1)
                {
                    relat = ((float)((float)p_args->current_position/500.0));
// Relat va entre 0 y 1
                    relat = relat*100;
// Entre 0 y 100
                    if((int)relat==0)    relat = 1;
// Entre 1 y 100

                }

                if(control_ADSR == A_1)
                {
                    tA = (int)relat;
                    Amp_a = (float)(1000.0/tA);

                    sprintf(str_ADSR, "tA = %d", tA);
                    update_prompt_text((GX_WIDGET *) &_5_control_ADSR,
ID_param_ADSR, str_ADSR);

                }
                else if (control_ADSR == D)
                {
                    tD = relat;
                    Amp_d = (float)(10*(100-AmpSust))/tD;

                    sprintf(str_ADSR, "tD = %d", tD);
                    update_prompt_text((GX_WIDGET *) &_5_control_ADSR,
ID_param_ADSR, str_ADSR);
                }
                else if (control_ADSR == S)
                {
                    AmpSust = relat;
                    Amp_d = (float)(10*(100-AmpSust))/tD;

                    sprintf(str_ADSR, "SUSTAIN = %d %c", AmpSust,37);
                    update_prompt_text((GX_WIDGET *) &_5_control_ADSR,
ID_param_ADSR, str_ADSR);
                }
                else if (control_ADSR == R)
                {
                    tR_1 = relat;
                    Amp_r = (10*AmpSust)/(float)tR_1;

                    sprintf(str_ADSR, "tR = %d", tR_1);
                    update_prompt_text((GX_WIDGET *) &_5_control_ADSR,
ID_param_ADSR, str_ADSR);
                }
            }
        }
}

void calcula_onda(void)                              // Calcula la forma de onda (aditiva)
{
    int i,j;                                         // Contadores locales para recorrer
los bucles
```

```
    N_instrum = 0;                                  // Reseteo el nº de instrumentos
activos

    for(i=0; i<NUMINST; i++)                        // Para cada uno de los 8
instrumentos...
    {
        if(instrum_activos[i]==1)
        {
            N_instrum ++;                           // = No. de instrumentos sumados
(activos)
        }
        else
        {
            instrum_activos[i] = 0;
        }
    }

    for(i=0; i<64; i++)                             // 64 muestras almacenadas por
instrumento
    {
        forma[i] = 0;                               // Resetear ALTURA del punto
correspondiente
        for(j=0; j<NUMINST; j++)
        {
            if(instrum_activos[j]==1)               // Para cada instrumento ACTIVO
            {
             forma[i] += SAMPLES[j][(64+i+fase_ins[j]) & 0x3F];
            // fase+64 para compensar fases negativas
            // Producto (& 0x3F (=63)) para no tomar valores > 63
            }
        }
         if(N_instrum)      // If N_instrum !=0
         {
         forma[i] = forma[i]/N_instrum;             // Escalar la onda
         }
    }
}
/* ********************************* FUNCIONES DE INTERRUPCIÓN
******************************** */

// Función de interrupción UART para la entrada MIDI

void MIDI_callback(uart_callback_args_t *p_args)
{
    // Usamos el evento RX_CHAR para leer cada uno de los 3 mensajes MIDI

    if (p_args->channel == 8)       // Si llega algo por nuestra UART
    {
        switch (p_args->event)
        {
            case UART_EVENT_RX_CHAR:                                // Y recibimos un
caracter

                RX_Buffer = (float)(p_args->data);

                // MIDI ESTÁNDAR
                if ((int)RX_Buffer == 0x00)                              // LLega aviso
note OFF
                {
                    n = 0;                                       // Resetear contador
```

```
                    flag_Msg = 1;

                    if (efecto == ADSR)
                    {
                        estado = ADSR_RELEASE;
                    }
                    else

                    estado = NO_HAY_NOTA;
                }

                else if ((int)RX_Buffer == 0x90)                // Llega noteON
inicial
                {
                    n = 1;
                }
                else
                {
                    if (n == 1)                             // Llega mensaje de
Altura
                    {
                        altura = (int)RX_Buffer;
                        n++;
                    }
                    else if (n == 2)                        // Llega mensaje de
Velocidad
                    {
                        n = 0;
                        fuerza = RX_Buffer;                 // La fuerza es un entero
entre [0,127]
                        estado = HAY_NOTA;
                        flag_Msg = 1;                       // Mensaje note_ON
completo
                    }
                }
                break;
            default:                            // Si el callback llama por otra cosa
como por ej. un mensaje RX transmitido, no hacemos nada
                break;
        }
    }
}


// Función de interrupción DCO, cada 50 microsegundos

void DCO_callback(timer_callback_args_t *p_args)
{
    // DAC CON TABLA DE ONDAS
    /*
    DAC_data12 = 0;                 // Resetear
    DAC_data_filtro = 0;     //PROVISIONAL
*/

    DAC_aux = 0;
    DAC_aux2 = 0;                   // PROVISIONAL

    if(estado == HAY_NOTA || estado == ADSR_RELEASE)
// Hay NOTE-ON
    {
```

```
        DAC_aux = forma[(int)fase & 0x3f];                // Leer onda - los puntos
están el rango [-128,128]
        // Limitar el índice a 63 como máximo (0x3F = 63)

        if (efecto != ADSR)                               // Modular con la fuerza en
todos efectos menos ADSR
        {
            DAC_aux = (DAC_aux * fuerza)/128.0;

            // Dividir la fuerza entre 2^7 = 128, luego la fuerza pasa de ser [0,127]
a [0,1]
        }

        if (efecto == TREMOLO)
        {
             fase += frec;
             DAC_aux = (DAC_aux * envolvente)/64.0;                // Modular con LFO
- varía la amplitud
            // Dividir /64: envolvente-> [0,5;1,5]
            // Ahora los puntos DAC_aux están en el rango [-192,192]
         }
         else if (efecto == VIBRATO)
         {
             fase += frecMod;                                      // Modular con LFO
- varía frecuencia
         }
         else if (efecto == FM)
         {
            env_FM =
((128+(float)(SAMPLES[instrumentMod][(int)(faseMod)&0x3F]))/256.0);            //
faseMod/64;
            // FM = oscilador 2
            frecMod = frec + env_FM;
            fase += frecMod;
            faseMod += Freq[nota_actual+indiceMod];
         }
         else if (efecto == ADSR)
         {
             DAC_aux = (DAC_aux*Amp_final)/100.0;                     // Amp_int será
como máximo 1
             fase += frec;
         }
         else fase+=frec;
         }

        // Ahora la onda está en el rango (-128,+128)
        DAC_aux = DAC_aux*volumen;


        /* **************************************************************
         * ************************* FILTROS ****************************
         * **************************************************************
         */


        // Antes de desplazar la onda y escalarla para pasarla al DAC, LE APLICAMOS
EL FILTRO SI ESTÁ ACTIVO:

        for (long f = 0; f < (N_muestras-1); f++)       // Desplazamos los valores
anteriores en el vector E, cuidado de machacar primero el más antiguo E0
```

```
        {
            VectorE[f] = VectorE[f+1];
        }

        VectorE[N_muestras-1] = DAC_aux;                // ...y guardamos el valor
actual de entrada

        /*
        if(filtro == ON)
            DAC_aux = filtrobanda();        // Aplicamos el filtro a la salida

        DAC_aux2 = filtrobanda();           // Para señal auxiliar osciloscopio

        // Calculo el menor valor de DAC_aux
        p = (int)fase & 0x3f;                       // índice de la tabla de ondas

        if (p == 0 && (int)valor_max!=0)                 // Primer índice de la tabla
        {                                           // valor_max!=0 porque entra dos
veces en este if; los índices se actualizan cada dos pasadas
            valor_max_ciclo = valor_max;            // Actualizo el valor máximo
            valor_max = 0;
            calcula_offset[p] = DAC_aux2;
        }

        if (p>0 && p < 64)
        {
            calcula_offset[p] = DAC_aux2;
            if(calcula_offset[p]>valor_max)
                valor_max = calcula_offset[p];
        }

        */

/*
        if(filtro == ON)
        {
           filtro3();          // Aplicamos el filtro a la salida
           DAC_aux = VectorS[N_muestras-1];
        }
        */

        //filtro3();            // Aplicamos el filtro a la salida del segundo DAC
por defecto (pruebas osciloscopio)
        //DAC_aux2 = VectorS[N_muestras-1];

        if(DAC_aux>128)     DAC_aux = 128;
        if(DAC_aux<(-128))     DAC_aux = -128;

        if(DAC_aux2>128)     DAC_aux2 = 128;
        if(DAC_aux2<(-128))     DAC_aux2 = -128;

        DAC_data12 = (int)DAC_aux*16 + 2048;                   // Escalamos de
8bits(128) a 12 bits(4095)
        DAC_data_filtro = DAC_aux2*16.0+ 2048;

        /*
        DAC_data12 = DAC_aux + 2048;
        DAC_data_filtro = DAC_aux2 + 2048;
*/
    g_dac0.p_api->write(g_dac0.p_ctrl, DAC_data12);
```

```
    //Para observar en el osciloscopio la señal 2 del filtro
    g_dac1.p_api->write(g_dac1.p_ctrl, DAC_data_filtro);
}

void LFO_callback(timer_callback_args_t *p_args)
{
    if(estado)
    {
    if (efecto == TREMOLO)
    {
        envolvente = 64 +
((float)(SAMPLES[instrumentMod][(int)(faseMod)&0x3F])/(float)4);       // Env-> [32,
96]
        faseMod += frecMod;
    }
    else if (efecto == VIBRATO)     // Indice_mod = envolvente(variación frec) /
100Hz > 1   --> frec_mod > envolvente
    {
        envolvente =
(128+(float)(SAMPLES[instrumentMod][(int)(faseMod)&0x3F]))/(float)(256*10);   // Env-
> [32, 96]   // /64*100
        frecMod = frec + envolvente;
        faseMod += frecMod;
    }
    else if (efecto == FM)
        // La modulación FM se aplica cuando el vibrato es muy rápido: el modulador
ya no es un LFO sino otro oscilador con frecuencia del orden de la
        // onda carrier -> Indice_mod <= 1
    {

    }
    else if (efecto == ADSR)
    {
        if(estado == HAY_NOTA)
        {
                t_nota_ADSR++;

                if(t_nota_ADSR <= tA)               // Fase "Attack"
                {
                    Amp_ADSR += Amp_a;
                    if(Amp_ADSR>1000.0)
                        Amp_ADSR=1000.0;
                }
                else
                {
                    if(t_nota_ADSR < (tA+tD))       // Fase "Decay"
                    {
                        Amp_ADSR -= Amp_d;
                        if(Amp_ADSR<(10*AmpSust))
                            Amp_ADSR = 10*AmpSust;
                    }
                    else                            // Fase "Sustain"
                    {
                        t_nota_ADSR = tA+tD;
                        Amp_ADSR = 10*AmpSust;
                    }
                }
            }
            else if(estado == ADSR_RELEASE && flag_Msg==0)
            {
```

```
                t_nota_off_ADSR++;

                if(t_nota_off_ADSR < tR_1)
                {
                    Amp_ADSR -= Amp_r;
                    if(Amp_ADSR<0)
                        Amp_ADSR = 0;
                }
                else                              // Ha terminado la fase release del
ADSR
                {
                    estado = NO_HAY_NOTA;
                }
            }
            Amp_final = Amp_ADSR/(float)10;           // Amp_ADSR es, como máximo 100
        }
    }
}
```

- **libreria_midi.h**

```
/*
 * libreria_midi.h
 *
 *  Created on: 14 jun. 2022
 *      Author: pilar
 */

#ifndef LIBRERIA_MIDI_H_
#define LIBRERIA_MIDI_H_

#include "midi.h"

#define HzM *64L/100.0 // sizeof(sin)*256 / F(tmr2)

#define NO_HAY_NOTA 0
#define HAY_NOTA 1
#define ADSR_RELEASE 2

#define NUMINST 8

#define ON 1
#define OFF 0

#define TREMOLO 1
#define VIBRATO 2
#define FM 3
#define ADSR 4

#define A_1 1
#define D 2
#define S 3
#define R 4

#define c_f0 1
#define c_G 2
#define c_Q 3

#define BOTON_1 1
#define BOTON_2 2

// Filtros
#define FilterOrder 2
#define N_muestras 4                // Número de muestras de entrada y salida para el
filtro

/* ********************************* VARIABLES ********************************* */

// Variables para entrada MIDI
extern volatile float RX_Buffer;
extern int n;                               // Contadores
extern volatile int flag_Msg;               // Bandera para indicar llegada de
mensajes MIDI completos
extern volatile int estado;
extern volatile int altura;
extern volatile float fuerza;
```

```
// Variables para salida DAC
extern dac_size_t DAC_data12;            // El UInt16 representa enteros sin signo
de 16 bits, pero tenemos que saturar en 12 bits que es la verdadera reolución del DAC
extern dac_size_t DAC_data_filtro;
extern int forma[64];                    // Vector alturas: 64 puntos de onda
calculada
extern float DAC_aux;
extern volatile int nota_actual;
extern volatile float frec, fase;        // Fase (w·t), es importante que sean
unsigned porque son punteros a tablas

// Variables para selección de instrumentos
extern int instrum_activos[NUMINST];               // Vector de 0s/1s: indica
instrumentos activos
extern int instrum_selec;                          // Instrumento seleccionado
[1,8]
extern int N_instrum;                              // No. de instrumentos activos
extern int control_fase[NUMINST];
extern volatile float relat;
extern int fase_ins[NUMINST];                      // Vector fases relativas de
instrumentos

// Variables para efectos - envolvente LFO
extern int efecto;
extern int control_ADSR;
extern volatile float envolvente;
extern volatile float faseMod, frecMod, env_FM;
extern int instrumentMod;
extern volatile int tA, tD, tR_1, AmpSust;
extern volatile float Amp_ADSR;
extern volatile int t_nota_ADSR, t_nota_off_ADSR;
extern volatile float Amp_a, Amp_d, Amp_r;
extern int indiceMod;                                     // Índice de modulación
para efecto FM
extern double frecMod_ini_tremolo;
extern double frecMod_ini_vibrato;
extern volatile int flag_modulacion;

// Para sincronización de la LCD con los controles del panel táctil
extern volatile int control_efecto;
extern volatile int ventana_control_ADSR;
extern volatile int ventana_control_filtro;
extern volatile char str_ADSR[80];
extern volatile char str_fase[80];
extern volatile char nombre_instrum[8][10];

// Variables para filtros
extern volatile int filtro;
extern volatile int control_filtro;

extern int f0;                                        // Frecuencia de resonancia
- 500Hz por defecto
extern int W;                                         // Ancho de banda
extern int G;                                         // Ganancia (en dB)
extern float Q;                                         // Factor Q - Los valores
comunes de Q varían entre 0.2 y 10
extern float Fs;                                        // Frecuencia de muestreo
- valor FIJO, son 20kHz
```

```
extern volatile double DenC2[FilterOrder+1];
extern volatile double NumC2[FilterOrder+1];

extern volatile double VectorE[N_muestras];
extern volatile double VectorS[N_muestras];

// Variables para panel táctil
extern volatile int boton;
extern volatile float volumen;
extern volatile int volumen_ON;
extern volatile int flag_modulacion;
extern volatile int flag_instrumento;
extern volatile int flag_filtro;

/* ************************ FUNCIONES ************************ */

void MIDI_callback(uart_callback_args_t *p_args);
void DCO_callback(timer_callback_args_t *p_args);
void LFO_callback(timer_callback_args_t *p_args);
void g_button_framework_user_callback(sf_touch_ctsu_button_callback_args_t *p_args);
void g_slider_framework_user_callback(sf_touch_ctsu_slider_callback_args_t *p_args);
void calcula_onda(void);

#endif /* LIBRERIA_MIDI_H_ */
```

- **formas.c**

```c
/*
 * formas.c
 *
 *  Created on: 14 jun. 2022
 *      Author: pilar
 */

#include "formas.h"


const int SAMPLES[NUMINST][64] =
{
    // senoidal
    {0,13,25,37,49,60,71,81,91,99,106,113,118,122,126,127,128,127,126,122,118,113,106,99,91,81,
     71,60,49,37,25,13,0,-13,-25,-37,-49,-60,-71,-81,-91,-99,-106,-113,-118,-122,-126,-127,-128,
     -127,-126,-122,-118,-113,-106,-99,-91,-81,-71,-60,-49,-37,-25,-13},
    // square 1
    {-128,-64,0,64,127,127,127,127,127,127,127,127,127,127,127,127,127,127,127,127,127,127,127,
    127,127,127,127,127,127,127,127,127,64,0,-64,-128,-128,-128,-128,-128,-128,-128,-128,-128,
    -128,-128,-128,-128,-128,-128,-128,-128,-128,-128,-128,-128,-128,-128,-128,-128,-128,-128,
    -128,-128},
    // square 2
    {0,55,100,118,127,127,127,127,127,127,127,127,127,127,127,100,63,27,0,0,0,0,0,0,0,0,0,0,0,-8,
    -27,-19,0,18,27,9,0,0,0,0,0,0,0,0,0,0,-26,-63,-100,-128,-128,-128,-128,-128,-128,-128,-128,
    -128,-128,-128,-119,-101,-57},
    // Electric_guitar
    {11,123,122,121,121,120,120,39,21,55,106,117,116,115,115,106,38,-101,-127,-127,-126,-126,
    -125,-91,11,70,72,-27,-124,-123,-123,-121,-120,-87,120,120,120,119,118,63,-67,-123,-122,-121,
    -120,-118,119,120,119,118,118,-124,-122,-122,-122,-120,-79,72,95,30,-63,-120,-119,-9},
    // E_organ
    {7,120,128,126,121,114,107,100,93,86,79,72,64,57,50,43,36,29,21,14,7,-0,-7,-14,-21,-29,-36,
    -43,-50,-57,-62,-55,54,62,57,50,43,36,29,22,15,7,0,-7,-14,-21,-28,-35,-43,-50,-57,-64,-71,
    -78,-86,-93,-100,-107,-114,-120,-126,-128,-120,-14},
    // oboe 5
    {1,10,22,34,46,59,71,82,92,101,108,113,115,114,108,100,89,75,59,38,14,-10,-36,-62,-83,-101,
    -114,-122,-127,-128,-125,-118,-107,-94,-79,-65,-49,-32,-18,
    -6,7,19,31,39,45,45,43,40,34,26,18,10,4,-1,-7,-13,-16,-18,-18,-18,-17,-14,-9, 0},
    // hdrawns 05
    {0,43,63,68,78,83,90,94,100,106,110,114,116,118,119,120,121,121,121,121,121,121,120,119,117,
     113,110,105,90,73,38,-7,-42,-64,-80,-87,-96,-105,-111,-120,-123,-125,-128,-128,-128,-128,-127,
    -127,-126,-125,-125,-122,-118,-115,-112,-106,-102,-91,-63,-39,-20,-9,-5,-1},
    // vgame 03
    {0,8,15,22,30,37,44,51,59,66,73,81,88,95,102,110,117,124,126,120,105,88,73,61,53,47,41,34,27,
    19,12,4,-3,-11,-18,-25,-33,-40,-47,-55,-62,-69,-76,-84,-91,-98,-105,-113,-120,-127,-125,-114,
    -97,-81,-67,-58,-50,-44,-38,-31,-24,-16,-8,-1}
};


const double Freq[] = {
                        ( 65.4064f Hz), //DO0
                        ( 69.2957f Hz),
                        ( 73.4162f Hz),
                        ( 77.7817f Hz),
                        ( 82.4069f Hz),
                        ( 87.3071f Hz),
                        ( 92.4986f Hz),
                        ( 97.9989f Hz),
                        (103.8262f Hz),
                        (110.0000f Hz),
                        (116.5409f Hz),
                        (123.4708f Hz),
                        (130.81278f Hz), //DO1
                        (138.59132f Hz),
                        (146.83238f Hz),
                        (155.56349f Hz),
                        (164.81378f Hz),
                        (174.61412f Hz),
                        (184.99721f Hz),
                        (195.99772f Hz),
                        (207.65235f Hz),
                        (220.00000f Hz),
                        (233.08188f Hz),
```

```
    (246.94165f Hz),
    (261.62557f Hz), //DO2
    (277.18263f Hz),
    (293.66477f Hz),
    (311.12698f Hz),
    (329.62756f Hz),
    (349.22823f Hz),
    (369.99442f Hz),
    (391.99544f Hz),
    (415.30470f Hz),
    (440.00000f Hz),
    (466.16376f Hz),
    (493.88330f Hz),
    (523.25113f Hz), //DO3
    (554.36526f Hz),
    (587.32954f Hz),
    (622.25397f Hz),
    (659.25511f Hz),
    (698.45646f Hz),
    (739.98885f Hz),
    (783.99087f Hz),
    (830.60940f Hz),
    (880.00000f Hz),
    (932.32752f Hz),
    (987.76660f Hz),
    (1046.50226f Hz), //DO4
    (1108.73052f Hz),
    (1174.65907f Hz),
    (1244.50793f Hz),
    (1318.51023f Hz),
    (1396.91293f Hz),
    (1479.97769f Hz),
    (1567.98174f Hz),
    (1661.21879f Hz) //LA4
};
```

- **formas.h**

```
#ifndef FORMAS_H_
#define FORMAS_H_

#define NUMINST 8
#define Hz *64L/20000.0              // Normalizado de frecuencias
#define OFFSET_NOTA 36               // Primera nota en la tabla: MIDI 36


extern const double Freq[];
extern const int SAMPLES[NUMINST][64];


#endif /* FORMAS_H_ */
```

- **main_thread_entry.c**

```
/* Main Thread entry function */
#include <main_thread.h>
#include "bsp_api.h"
#include "gx_api.h"
#include "gui/guiapp_specifications.h"
#include "gui/guiapp_resources.h"
#include "gx_user_heap.h"

#if defined(BSP_BOARD_S7G2_SK)
#include "hardware/lcd.h"
#endif

/*************************************************************************************
***********************************
    Private function prototypes

*************************************************************************************
**********************************/
static bool ssp_touch_to_guix(sf_touch_panel_payload_t * p_touch_payload, GX_EVENT *
g_gx_event);
void main_thread_entry(void);

#if defined(BSP_BOARD_S7G2_SK)
void g_lcd_spi_callback(spi_callback_args_t * p_args);
#endif

/*************************************************************************************
***********************************
    Private global variables

*************************************************************************************
**********************************/
static GX_EVENT g_gx_event;


GX_WINDOW_ROOT * p_window_root;
extern GX_CONST GX_STUDIO_WIDGET *guiapp_widget_table[];

/*************************************************************************************
******************************//**
    @brief  Primary logic for handling events generated by the various sub-systems.

*************************************************************************************
**********************************/


static UINT status_flag;    // Variable para recoger el estado de la función que
realiza la espera no bloqueante del flag
ULONG actual_events_flag;   // Variable donde se va a volcar el valor del grupo de
flags. Puede albergar hasta 32 banderas (una por bit, unsigned long). Por simplicidad
usaremos una bandera por grupo.


void main_thread_entry(void) {
      ssp_err_t        err;
```

```
        sf_message_header_t * p_message = NULL;
        UINT       status = TX_SUCCESS;

    /* Initializes GUIX. */
    status = gx_system_initialize();
    if(TX_SUCCESS != status)
    {
        while(1);
    }

    /* Initializes GUIX drivers. */
    err = g_sf_el_gx.p_api->open (g_sf_el_gx.p_ctrl, g_sf_el_gx.p_cfg);
    if(SSP_SUCCESS != err)
    {
        while(1);
    }

    gx_studio_display_configure ( DISPLAY_1,
                                  g_sf_el_gx.p_api->setup,
                                  LANGUAGE_ENGLISH,
                                  DISPLAY_1_THEME_1,
                                  &p_window_root );

    err = g_sf_el_gx.p_api->canvasInit(g_sf_el_gx.p_ctrl, p_window_root);
    if(SSP_SUCCESS != err)
    {
        while(1);
    }

    // Create the widgets we have defined with the GUIX data structures and
resources.
    GX_CONST GX_STUDIO_WIDGET ** pp_studio_widget = &guiapp_widget_table[0];
    GX_WIDGET * p_first_screen = NULL;


    while (GX_NULL != *pp_studio_widget)
    {
        // We must first create the widgets according the data generated in GUIX
Studio.

        // Once we are working on the widget we want to see first, save the pointer
for later.
        if (0 == strcmp("_1_inicio", (char*)(*pp_studio_widget)->widget_name))
        {
            gx_studio_named_widget_create((*pp_studio_widget)->widget_name,
(GX_WIDGET *)p_window_root, GX_NULL);
        } else {
            gx_studio_named_widget_create((*pp_studio_widget)->widget_name, GX_NULL,
GX_NULL);
        }
        // Move to next top-level widget
        pp_studio_widget++;
    }
    // Attach the first screen to the root so we can see it when the root is shown
   //gx_widget_attach(p_window_root, p_first_screen);


    if(TX_SUCCESS != status)
    {
            while(1);
```

```
    }

    /* Shows the root window to make it and patients screen visible. */
    status = gx_widget_show(p_window_root);
    if(TX_SUCCESS != status)
    {
        while(1);
    }

    /* Lets GUIX run. */
    status = gx_system_start();
    if(TX_SUCCESS != status)
    {
        while(1);
    }

#if defined(BSP_BOARD_S7G2_SK)
    /** Open the SPI driver to initialize the LCD (SK-S7G2) **/
    err = g_spi_lcdc.p_api->open(g_spi_lcdc.p_ctrl, (spi_cfg_t *)g_spi_lcdc.p_cfg);
    if (err)
    {
        while(1);
    }
    /** Setup the ILI9341V (SK-S7G2) **/
    ILI9341V_Init();
#endif

    /* Controls the GPIO pin for LCD ON (DK-S7G2, PE-HMI1) */
#if defined(BSP_BOARD_S7G2_DK)
    err = g_ioport.p_api->pinWrite(IOPORT_PORT_07_PIN_10, IOPORT_LEVEL_HIGH);
    if (err)
    {
        while(1);
    }
#elif defined(BSP_BOARD_S7G2_PE_HMI1)
    err = g_ioport.p_api->pinWrite(IOPORT_PORT_10_PIN_03, IOPORT_LEVEL_HIGH);
    if (err)
    {
        while(1);
    }
#endif

    /* Opens PWM driver and controls the TFT panel back light (DK-S7G2, PE-HMI1) */
#if defined(BSP_BOARD_S7G2_DK) || defined(BSP_BOARD_S7G2_PE_HMI1)
    err = g_pwm_backlight.p_api->open(g_pwm_backlight.p_ctrl, g_pwm_backlight.p_cfg);
    if (err)
    {
        while(1);
    }
#endif

    while (1)
    {
        bool new_gui_event = false; //reseteamos el valor tras el manejo de cada
evento

        err = g_sf_message0.p_api->pend(g_sf_message0.p_ctrl,
&main_thread_message_queue, (sf_message_header_t **) &p_message, TX_WAIT_FOREVER);
//cola de mensajes bloqueante, espera un evento en la pantalla.
        if (err)
```

```
        {
            /** TODO: Handle error. */
        }

        switch (p_message->event_b.class_code)
        {
        case SF_MESSAGE_EVENT_CLASS_TOUCH:              //si el evento de es de la
clase tactil
        {
            switch (p_message->event_b.code)
            {
            case SF_MESSAGE_EVENT_NEW_DATA:             //y se ha recibido un nuevo
dato
            {
                /** Translate an SSP touch event into a GUIX event */
                new_gui_event =
ssp_touch_to_guix((sf_touch_panel_payload_t*)p_message, &g_gx_event);   //enviamos
los datos recibido a la función que generará el evento GUIX en función del evento en
la pantalla.

//le pasamos el payload y el puntero a la estructura a rellenar en función del
payload
            }
            default:
                break;
            }
            break;
        }
        default:
            break;
        }

        /** Message is processed, so release buffer. */
        err = g_sf_message0.p_api->bufferRelease(g_sf_message0.p_ctrl,
(sf_message_header_t *) p_message, SF_MESSAGE_RELEASE_OPTION_FORCED_RELEASE);
//borramos el buffer por el que hemos leido los datos de la cola.

        if (err)
        {
            /** TODO: Handle error. */
        }

        /** Post message. */
        if (new_gui_event) {
            gx_system_event_send(&g_gx_event);  //si hay un nuevo evento guix, lo
enviamos
        }

    }
}
/*********************************************************************************
**********************************
 End of function main_thread_entry

*********************************************************************************
********************************/


static bool ssp_touch_to_guix(sf_touch_panel_payload_t * p_touch_payload, GX_EVENT *
gx_event)
```

```
{
    bool send_event = true;

    switch (p_touch_payload->event_type)                //clasificamos el evento
    {
    case SF_TOUCH_PANEL_EVENT_DOWN:                     //se ha pulsado la pantalla
        gx_event->gx_event_type = GX_EVENT_PEN_DOWN;
        break;
    case SF_TOUCH_PANEL_EVENT_UP:
        gx_event->gx_event_type = GX_EVENT_PEN_UP;      //se ha dejado de pulsar
        break;
    case SF_TOUCH_PANEL_EVENT_HOLD:                     //se mantiene la pulsación
    case SF_TOUCH_PANEL_EVENT_MOVE:                     //se mueve la pulsación
        gx_event->gx_event_type = GX_EVENT_PEN_DRAG;
        break;
    case SF_TOUCH_PANEL_EVENT_INVALID:                  //es invalido
        send_event = false;                             //no devolvemos datos
        break;
    default:
        break;
    }

    if (send_event)
    {
        /** Send event to GUI */
        gx_event->gx_event_sender = GX_ID_NONE;
        gx_event->gx_event_target = 0;
        gx_event->gx_event_display_handle = 0;

        gx_event->gx_event_payload.gx_event_pointdata.gx_point_x = p_touch_payload-
>x;  //guardamos las coordenadas para enviarlas

#if defined(BSP_BOARD_S7G2_SK)
        gx_event->gx_event_payload.gx_event_pointdata.gx_point_y = (GX_VALUE)(320 -
p_touch_payload->y);  // SK-S7G2
#else
        gx_event->gx_event_payload.gx_event_pointdata.gx_point_y = p_touch_payload-
>y;  // DK-S7G2, PE-HMI1
#endif

        status_flag = tx_event_flags_get(&g_touch_event_flags, (ULONG) 1, TX_OR,
&actual_events_flag, TX_NO_WAIT);        //espera no bloqueante del flag w5, si
estamos en la pantalla 5, actuamos sobre la misma, mostrando por pantalla los valores
de las coordenadas x e y.
    }

    return send_event;
}


#if defined(BSP_BOARD_S7G2_SK)
void g_lcd_spi_callback(spi_callback_args_t * p_args)
{
    (void)p_args;
    tx_semaphore_ceiling_put(&g_main_semaphore_lcdc, 1);
}
#endif
```

- **guiapp_event_handlers.c**

```
#include <main_thread.h>
#include "gui/guiapp_resources.h"
#include "gui/guiapp_specifications.h"
#include "libreria_midi.h"
#include "guiapp_event_handlers.h"

#include "tx_api.h"             // Header de threadx. permite uso de aquellos
elementos inherentes a un RTOS (mutex, colas, etc.)

// Instrumentos seleccionados por los botones de la lcd
#define SENO 0
#define SQUARE_1 1
#define SQUARE_2 2
#define ELECTRIC_GUITAR 3
#define ORGAN 4
#define OBOE 5
#define HAND_DRAWN 6
#define VIDEO_GAME 7

GX_RESOURCE_ID ID_efecto;
GX_RESOURCE_ID ID_controlADSR;
GX_RESOURCE_ID ID_controlfiltro;

volatile int modulacion;


extern GX_WINDOW_ROOT * p_window_root;

static UINT show_window(GX_WINDOW * p_new, GX_WIDGET * p_widget, bool detach_old);

static void update_button_color_id(GX_WIDGET * p_widget, GX_RESOURCE_ID id, int
enable)
{
    GX_TEXT_BUTTON * p_button = NULL;

    ssp_err_t err = (ssp_err_t)gx_widget_find(p_widget, (USHORT)id,
GX_SEARCH_DEPTH_INFINITE, (GX_WIDGET**)&p_button);

    if (TX_SUCCESS == err)
    {
        if(enable == 0)
        {
            gx_widget_fill_color_set(p_button, GX_COLOR_ID_BUTTON_UPPER,
GX_COLOR_ID_BUTTON_LOWER);
        }
        else if (enable == 1)
        {
            gx_widget_fill_color_set(p_button,
GX_COLOR_ID_BUTTON_SELECTED,GX_COLOR_ID_BUTTON_SELECTED);
        }
    }
}
```

```
static void update_button_color_id2(GX_WIDGET * p_widget, GX_RESOURCE_ID id, int
enable)
{
    GX_TEXT_BUTTON * p_button = NULL;

    ssp_err_t err = (ssp_err_t)gx_widget_find(p_widget, (USHORT)id,
GX_SEARCH_DEPTH_INFINITE, (GX_WIDGET**)&p_button);

    if (TX_SUCCESS == err)
    {
        if(enable == 0)
        {
            gx_widget_fill_color_set(p_button, GX_COLOR_ID_BUTTON_TEXT,
GX_COLOR_ID_BUTTON_TEXT);
        }
        else if (enable == 1)
        {
            gx_widget_fill_color_set(p_button,
GX_COLOR_ID_BUTTON_SELECTED,GX_COLOR_ID_BUTTON_SELECTED);
        }
    }
}

static void update_control_color_id(GX_WIDGET * p_widget, GX_RESOURCE_ID id, int
enable)
{
    GX_TEXT_BUTTON * p_button = NULL;

    ssp_err_t err = (ssp_err_t)gx_widget_find(p_widget, (USHORT)id,
GX_SEARCH_DEPTH_INFINITE, (GX_WIDGET**)&p_button);

    if (TX_SUCCESS == err)
    {
        if(enable == 0)
        {
            gx_widget_fill_color_set(p_button, GX_COLOR_ID_SLIDER_NEEDLE_LINE2,
GX_COLOR_ID_SLIDER_NEEDLE_LINE2);
        }
        else if (enable == 1)
        {
            gx_widget_fill_color_set(p_button,
GX_COLOR_ID_BUTTON_SELECTED,GX_COLOR_ID_BUTTON_SELECTED);
        }
    }
}

static void update_text_color_id(GX_WIDGET * p_widget, GX_RESOURCE_ID id, int enable)
{
    GX_TEXT_BUTTON * p_button = NULL;

    ssp_err_t err = (ssp_err_t)gx_widget_find(p_widget, (USHORT)id,
GX_SEARCH_DEPTH_INFINITE, (GX_WIDGET**)&p_button);

    if (TX_SUCCESS == err)
    {
        if(enable == 0)
        {
            gx_prompt_text_color_set(p_button, GX_COLOR_ID_READONLY_TEXT,
GX_COLOR_ID_READONLY_TEXT);
        }
```

```
        else if (enable == 1)
        {
            gx_prompt_text_color_set(p_button,
GX_COLOR_ID_BUTTON_SELECTED,GX_COLOR_ID_BUTTON_SELECTED);
        }
    }
}

void update_prompt_text_id(GX_WIDGET * p_widget, GX_RESOURCE_ID id, UINT string_id,
int enable)
{
    GX_PROMPT * p_prompt = NULL;

    ssp_err_t err = (ssp_err_t)gx_widget_find(p_widget, (USHORT)id,
GX_SEARCH_DEPTH_INFINITE, (GX_WIDGET**)&p_prompt);
    if (TX_SUCCESS == err)
    {
        gx_prompt_text_id_set(p_prompt, string_id);

        if(enable == 1)
            gx_prompt_text_color_set(p_prompt,GX_COLOR_ID_BUTTON_SELECTED,
GX_COLOR_ID_BUTTON_SELECTED);
        else
            gx_prompt_text_color_set(p_prompt,GX_COLOR_ID_SLIDER_NEEDLE_LINE2,
GX_COLOR_ID_SLIDER_NEEDLE_LINE2);
    }
}

void update_prompt_text(GX_WIDGET * p_widget, GX_RESOURCE_ID id, char * p_text)
{
    GX_PROMPT * p_prompt = NULL;

    UINT err = gx_system_widget_find((USHORT)id, GX_SEARCH_DEPTH_INFINITE, (GX_WIDGET
**) &p_prompt);
    if (GX_SUCCESS == err)
    {
       //gx_prompt_text_set(p_prompt, (GX_CHAR*)"HOLA");  //Prueba
       gx_prompt_text_set(p_prompt, p_text);

       gx_system_dirty_mark((GX_WIDGET *) p_prompt);
       gx_system_canvas_refresh();
    }
}

static void update_instrumento(int instrum, GX_WINDOW *widget, GX_RESOURCE_ID ID,
GX_RESOURCE_ID ID_control)
{
    int f;
    flag_instrumento = 1;
    instrum_selec = instrum;
    if(instrum_activos[instrum] == 0)               // Estaba apagado
    {
        update_button_color_id(widget->gx_widget_parent, ID,1);
// Activamos los botones del Menú
        update_control_color_id((GX_WIDGET *) &_2_moog_window, ID_control,1);
// Actualizamos los indicadores del menú Moog

        for(f=0;f<8;f++)
        {
```

```
            control_fase[f] = 0;                           // Primero reseteo para
activar el slider solo en el último pulsado
        }
        volumen_ON = 0;                                    // Desactivo por si estaba
activo
        control_fase[instrum] = ON;
    }
    else
    {
        update_button_color_id(widget->gx_widget_parent, ID,0);
        update_control_color_id((GX_WIDGET *) &_2_moog_window, ID_control,0);    //
Actualizamos los indicadores del menú Moog
        control_fase[instrum] = OFF;
    }
}

static void update_efecto(int nuevo_efecto, GX_RESOURCE_ID ID_nuevo_efecto, GX_WINDOW
*widget, GX_RESOURCE_ID ID_string)
{
    if ((efecto!=nuevo_efecto) && (modulacion == 1))       // Que la modulación esté
activa y que no esté ya seleccionado ese efecto
    {
        if(efecto!=0)
        {
            update_button_color_id(widget->gx_widget_parent, ID_efecto,0);      // Lo
apago
        }
        efecto = nuevo_efecto;
        ID_efecto = ID_nuevo_efecto;
        update_button_color_id(widget->gx_widget_parent, ID_efecto,1);

        update_prompt_text_id((GX_WIDGET *) &_2_moog_window, ID_mod_activa,ID_string,
1);

        control_efecto = ON;
        volumen_ON = 0;
        control_fase[instrum_selec] = 0;
        flag_modulacion = 1;
    }
    else if(efecto == nuevo_efecto)
    {
        efecto = 0;
        update_button_color_id(widget->gx_widget_parent, ID_efecto,0);
        ID_efecto = 0;
    }
}

static void update_ADSR(int nuevo, GX_RESOURCE_ID ID_nuevo, GX_WINDOW *widget)
{
    if (control_ADSR!=nuevo)       // Que no esté ya seleccionado ese control
    {
        if(control_ADSR!=0)        // Si ya hay otro control activo
        {
            update_button_color_id2(widget->gx_widget_parent, ID_controlADSR,0);
// Lo apago
        }
        control_ADSR = nuevo;
        ID_controlADSR = ID_nuevo;
        update_button_color_id2(widget->gx_widget_parent, ID_controlADSR,1);
```

```
        ventana_control_ADSR = ON;
        // Desactivo el resto de controles del slider
        volumen_ON = 0;
        control_fase[instrum_selec] = 0;
        control_efecto = 0;
    }
    else if(control_ADSR == nuevo)
    {
        control_ADSR = 0;
        update_button_color_id2(widget->gx_widget_parent, ID_controlADSR,0);
        ID_controlADSR = 0;
    }
}

static void update_filtro(int nuevo, GX_RESOURCE_ID ID_nuevo, GX_WINDOW *widget)
{
    if (control_filtro!=nuevo)       // Que no esté ya seleccionado ese control
    {
        if(control_filtro!=0)        // Si ya hay otro control activo
        {
            update_button_color_id2(widget->gx_widget_parent, ID_controlfiltro,0);
// Lo apago
        }
        control_filtro = nuevo;
        ID_controlfiltro = ID_nuevo;
        update_button_color_id2(widget->gx_widget_parent, ID_controlfiltro,1);

        ventana_control_filtro = ON;
    }
    else if(control_filtro == nuevo)
    {
        control_filtro = 0;
        update_button_color_id2(widget->gx_widget_parent, ID_controlfiltro,0);
        ID_controlfiltro = 0;
    }
}

ULONG event;

UINT control_instrumentos_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    UINT result = gx_window_event_process(widget, event_ptr);
    event = event_ptr->gx_event_type;

    switch (event_ptr->gx_event_type)
    {
        case GX_SIGNAL(ID_SENO, GX_EVENT_CLICKED):
            update_instrumento(SENO, widget, ID_SENO, ID_ins_0);
            sprintf(str_fase, "Fase %s = %d", nombre_instrum[SENO], fase_ins[0]);
            update_prompt_text((GX_WIDGET *) &_3_control_instrumentos, ID_param_fase,
str_fase);
        break;

        case GX_SIGNAL(ID_SQUARE1, GX_EVENT_CLICKED):
            update_instrumento(SQUARE_1,widget, ID_SQUARE1, ID_ins_1);
            sprintf(str_fase, "Fase %s = %d", nombre_instrum[SQUARE_1], fase_ins[1]);
            update_prompt_text((GX_WIDGET *) &_3_control_instrumentos, ID_param_fase,
str_fase);
        break;
```

```
        case GX_SIGNAL(ID_SQUARE2, GX_EVENT_CLICKED):
            update_instrumento(SQUARE_2,widget, ID_SQUARE2, ID_ins_2);
            sprintf(str_fase, "Fase %s = %d", nombre_instrum[SQUARE_2], fase_ins[2]);
            update_prompt_text((GX_WIDGET *) &_3_control_instrumentos, ID_param_fase,
str_fase);
        break;

        case GX_SIGNAL(ID_GUITAR, GX_EVENT_CLICKED):
            update_instrumento(ELECTRIC_GUITAR,widget, ID_GUITAR, ID_ins_3);
            sprintf(str_fase, "Fase %s = %d", nombre_instrum[ELECTRIC_GUITAR],
fase_ins[3]);
            update_prompt_text((GX_WIDGET *) &_3_control_instrumentos, ID_param_fase,
str_fase);
        break;

        case GX_SIGNAL(ID_ORGAN, GX_EVENT_CLICKED):
            update_instrumento(ORGAN,widget, ID_ORGAN, ID_ins_4);
            sprintf(str_fase, "Fase %s = %d", nombre_instrum[ORGAN], fase_ins[4]);
            update_prompt_text((GX_WIDGET *) &_3_control_instrumentos, ID_param_fase,
str_fase);
        break;

        case GX_SIGNAL(ID_OBOE, GX_EVENT_CLICKED):
            update_instrumento(OBOE,widget, ID_OBOE, ID_ins_5);
            sprintf(str_fase, "Fase %s = %d", nombre_instrum[OBOE], fase_ins[5]);
            update_prompt_text((GX_WIDGET *) &_3_control_instrumentos, ID_param_fase,
str_fase);
        break;

        case GX_SIGNAL(ID_HAND, GX_EVENT_CLICKED):
            update_instrumento(HAND_DRAWN,widget, ID_HAND, ID_ins_6);
            sprintf(str_fase, "Fase %s = %d", nombre_instrum[HAND_DRAWN],
fase_ins[6]);
            update_prompt_text((GX_WIDGET *) &_3_control_instrumentos, ID_param_fase,
str_fase);
        break;

        case GX_SIGNAL(ID_GAME, GX_EVENT_CLICKED):
            update_instrumento(VIDEO_GAME,widget, ID_GAME, ID_ins_7);
            sprintf(str_fase, "Fase %s = %d", nombre_instrum[VIDEO_GAME],
fase_ins[7]);
            update_prompt_text((GX_WIDGET *) &_3_control_instrumentos, ID_param_fase,
str_fase);
        break;

        default:
            gx_window_event_process(widget, event_ptr);
        break;
    }

    return result;
}

UINT control_modulacion_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    UINT result = gx_window_event_process(widget, event_ptr);
    event = event_ptr->gx_event_type;

    switch (event_ptr->gx_event_type)
    {
```

```
        case GX_SIGNAL(ID_ON_OFF, GX_EVENT_TOGGLE_ON):
            modulacion = ON;
            break;

        case GX_SIGNAL(ID_ON_OFF, GX_EVENT_TOGGLE_OFF):
            modulacion = OFF;
            efecto = 0;
            update_button_color_id(widget, ID_efecto,0);      // Deselecciono efecto
activo en la pantalla
            update_prompt_text_id((GX_WIDGET *) &_2_moog_window,
ID_mod_activa,GX_STRING_ID_ID_NONE, 0);
            ID_efecto = 0;                                    // Importante resetear
después!
            break;

        case GX_SIGNAL(ID_TREMOLO, GX_EVENT_CLICKED):
            update_efecto(TREMOLO, ID_TREMOLO, widget,GX_STRING_ID_TREMOLO_STR);
            break;

        case GX_SIGNAL(ID_VIBRATO, GX_EVENT_CLICKED):
            update_efecto(VIBRATO, ID_VIBRATO, widget,GX_STRING_ID_VIBRATO_STR);
            break;

        case GX_SIGNAL(ID_FM, GX_EVENT_CLICKED):
            update_efecto(FM, ID_FM, widget,GX_STRING_ID_FM_STR);
            break;

        case GX_SIGNAL(ID_ADSR, GX_EVENT_CLICKED):
            update_efecto(ADSR, ID_ADSR, widget, GX_STRING_ID_ADSR_STR);
            break;

        case GX_SIGNAL(ID_control_ADSR, GX_EVENT_CLICKED):
            if(efecto == ADSR)  show_window((GX_WINDOW*)&_5_control_ADSR,
(GX_WIDGET*)widget, true);
            break;

        default:
            gx_window_event_process(widget, event_ptr);
            break;
    }

    return result;
}

UINT control_ADSR_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    UINT result = gx_window_event_process(widget, event_ptr);
    event = event_ptr->gx_event_type;

    switch (event_ptr->gx_event_type)
    {
        case GX_SIGNAL(ID_tA, GX_EVENT_CLICKED):
            update_ADSR(A_1, ID_tA, widget);
            sprintf(str_ADSR, "tA = %d", tA);
            update_prompt_text((GX_WIDGET *) &_5_control_ADSR, ID_param_ADSR,
str_ADSR);
            break;

        case GX_SIGNAL(ID_tD, GX_EVENT_CLICKED):
            update_ADSR(D, ID_tD, widget);
```

```
            sprintf(str_ADSR, "tD = %d", tD);
            update_prompt_text((GX_WIDGET *) &_5_control_ADSR, ID_param_ADSR,
str_ADSR);
            break;

        case GX_SIGNAL(ID_tS, GX_EVENT_CLICKED):
            update_ADSR(S, ID_tS, widget);
            sprintf(str_ADSR, "SUST = %d %c", AmpSust, 37);
            update_prompt_text((GX_WIDGET *) &_5_control_ADSR, ID_param_ADSR,
str_ADSR);
            break;

        case GX_SIGNAL(ID_tR, GX_EVENT_CLICKED):
            update_ADSR(R, ID_tR, widget);
            sprintf(str_ADSR, "tR = %d", tR_1);
            update_prompt_text((GX_WIDGET *) &_5_control_ADSR, ID_param_ADSR,
str_ADSR);
            break;

        default:
            gx_window_event_process(widget, event_ptr);
            break;
    }

    return result;
}

UINT control_filtro_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    UINT result = gx_window_event_process(widget, event_ptr);
    event = event_ptr->gx_event_type;

    switch (event_ptr->gx_event_type)
    {
        case GX_SIGNAL(ID_ON_OFF, GX_EVENT_TOGGLE_ON):
            filtro = ON;
            update_control_color_id((GX_WIDGET *)&_2_moog_window, ID_filtro_ON,1);
// Actualizamos los indicadores del menú Moog
            break;

        case GX_SIGNAL(ID_ON_OFF, GX_EVENT_TOGGLE_OFF):
            filtro = OFF;
            update_control_color_id((GX_WIDGET *)&_2_moog_window, ID_filtro_ON,0);
// Actualizamos los indicadores del menú Moog
            break;

        case GX_SIGNAL(ID_f0, GX_EVENT_CLICKED):
            update_filtro(c_f0, ID_f0, widget);
            break;

        case GX_SIGNAL(ID_G, GX_EVENT_CLICKED):
            update_filtro(c_G, ID_G, widget);
            break;

        case GX_SIGNAL(ID_Q, GX_EVENT_CLICKED):
            update_filtro(c_Q, ID_Q, widget);
            break;

        default:
            gx_window_event_process(widget, event_ptr);
```

```
            break;
    }

    return result;
}

UINT control_volumen_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)

{
    UINT result = gx_window_event_process(widget, event_ptr);
    event = event_ptr->gx_event_type;

    switch (event_ptr->gx_event_type)
        {
            case GX_SIGNAL(ID_icon_volumen, GX_EVENT_CLICKED):

                if(volumen_ON == 0)
                {
                    update_text_color_id(widget->gx_widget_parent, ID_amplifier,1);
                    volumen_ON = 1;

                }
                else
                {
                    update_text_color_id(widget->gx_widget_parent, ID_amplifier,0);
                    volumen_ON = 0;
                }

                break;

            default:
                gx_window_event_process(widget, event_ptr);
                break;
        }

    return result;
}

static UINT show_window(GX_WINDOW * p_new, GX_WIDGET * p_widget, bool detach_old)
{
    UINT err = GX_SUCCESS;

    if (!p_new->gx_widget_parent)
    {
        err = gx_widget_attach(p_window_root, p_new);
    }
    else
    {
        err = gx_widget_show(p_new);
    }

    gx_system_focus_claim(p_new);

    GX_WIDGET * p_old = p_widget;
    if (p_old && detach_old)
    {
        if (p_old != (GX_WIDGET*)p_new)
        {
            gx_widget_detach(p_old);
        }
```

```
    }

    return err;
}

/*
static void update_button_text_id(GX_WIDGET * p_widget, GX_RESOURCE_ID id, UINT
string_id)
{
    GX_TEXT_BUTTON * p_button = NULL;

    ssp_err_t err = (ssp_err_t)gx_widget_find(p_widget, (USHORT)id,
GX_SEARCH_DEPTH_INFINITE, (GX_WIDGET**)&p_button);
    if (TX_SUCCESS == err)
    {
        gx_text_button_text_id_set(p_button, string_id);
    }
}
*/
```

- **guiapp_event_handlers.h**

```
/*
 * guiapp_event_handlers.h
 *
 *  Created on: 8 jul. 2022
 *      Author: pilar
 */

#ifndef GUIAPP_EVENT_HANDLERS_H_
#define GUIAPP_EVENT_HANDLERS_H_

void update_prompt_text(GX_WIDGET * p_widget, GX_RESOURCE_ID id, char * p_text);

#endif /* GUIAPP_EVENT_HANDLERS_H_ */
```

- **iir.c**

Este es un repositorio que recoge las funciones auxiliares para el cálculo de los coeficientes del filtro pasabanda de Butterwort [13]:

```
/*
 * iir.c

 *
 *  Created on: 5 jul. 2022
 *      Author: pilar
 */


/*
 *                            COPYRIGHT
 *
 *  liir - Recursive digital filter functions
 *  Copyright (C) 2007 Exstrom Laboratories LLC
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  A copy of the GNU General Public License is available on the internet at:
 *
 *  http://www.gnu.org/copyleft/gpl.html
 *
 *  or you can write to:
 *
 *  The Free Software Foundation, Inc.
 *  675 Mass Ave
 *  Cambridge, MA 02139, USA
 *
 *  You can contact Exstrom Laboratories LLC via Email at:
 *
 *  stefan(AT)exstrom.com
 *
 *  or you can write to:
 *
 *  Exstrom Laboratories LLC
 *  P.O. Box 7651
 *  Longmont, CO 80501, USA
 *
 */

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

```
#include "iir.h"

#define PI 3.14159265358979323846

/**********************************************************************
  binomial_mult - multiplies a series of binomials together and returns
  the coefficients of the resulting polynomial.

  The multiplication has the following form:

  (x+p[0])*(x+p[1])*...*(x+p[n-1])

  The p[i] coefficients are assumed to be complex and are passed to the
  function as a pointer to an array of doubles of length 2n.

  The resulting polynomial has the following form:

  x^n + a[0]*x^n-1 + a[1]*x^n-2 + ... +a[n-2]*x + a[n-1]

  The a[i] coefficients can in general be complex but should in most
  cases turn out to be real. The a[i] coefficients are returned by the
  function as a pointer to an array of doubles of length 2n. Storage
  for the array is allocated by the function and should be freed by the
  calling program when no longer needed.

  Function arguments:

  n  -  The number of binomials to multiply
  p  -  Pointer to an array of doubles where p[2i] (i=0...n-1) is
        assumed to be the real part of the coefficient of the ith binomial
        and p[2i+1] is assumed to be the imaginary part. The overall size
        of the array is then 2n.
*/

double *binomial_mult( int n, double *p )
{
    int i, j;
    double *a;

    a = (double *)calloc( 2 * n, sizeof(double) );
    if( a == NULL ) return( NULL );

    for( i = 0; i < n; ++i )
    {
    for( j = i; j > 0; --j )
    {
        a[2*j] += p[2*i] * a[2*(j-1)] - p[2*i+1] * a[2*(j-1)+1];
        a[2*j+1] += p[2*i] * a[2*(j-1)+1] + p[2*i+1] * a[2*(j-1)];
    }
    a[0] += p[2*i];
    a[1] += p[2*i+1];
    }
    return( a );
}


/**********************************************************************
  trinomial_mult - multiplies a series of trinomials together and returns
  the coefficients of the resulting polynomial.
```

```
  The multiplication has the following form:

  (x^2 + b[0]x + c[0])*(x^2 + b[1]x + c[1])*...*(x^2 + b[n-1]x + c[n-1])

  The b[i] and c[i] coefficients are assumed to be complex and are passed
  to the function as a pointers to arrays of doubles of length 2n. The real
  part of the coefficients are stored in the even numbered elements of the
  array and the imaginary parts are stored in the odd numbered elements.

  The resulting polynomial has the following form:

  x^2n + a[0]*x^2n-1 + a[1]*x^2n-2 + ... +a[2n-2]*x + a[2n-1]

  The a[i] coefficients can in general be complex but should in most cases
  turn out to be real. The a[i] coefficients are returned by the function as
  a pointer to an array of doubles of length 4n. The real and imaginary
  parts are stored, respectively, in the even and odd elements of the array.
  Storage for the array is allocated by the function and should be freed by
  the calling program when no longer needed.

  Function arguments:

  n  -  The number of trinomials to multiply
  b  -  Pointer to an array of doubles of length 2n.
  c  -  Pointer to an array of doubles of length 2n.
*/

double *trinomial_mult( int n, double *b, double *c )
{
    int i, j;
    double *a;

    a = (double *)calloc( 4 * n, sizeof(double) );
    if( a == NULL ) return( NULL );

    a[2] = c[0];
    a[3] = c[1];
    a[0] = b[0];
    a[1] = b[1];

    for( i = 1; i < n; ++i )
    {
    a[2*(2*i+1)]   += c[2*i]*a[2*(2*i-1)]   - c[2*i+1]*a[2*(2*i-1)+1];
    a[2*(2*i+1)+1] += c[2*i]*a[2*(2*i-1)+1] + c[2*i+1]*a[2*(2*i-1)];

    for( j = 2*i; j > 1; --j )
    {
        a[2*j]   += b[2*i] * a[2*(j-1)]   - b[2*i+1] * a[2*(j-1)+1] +
        c[2*i] * a[2*(j-2)]   - c[2*i+1] * a[2*(j-2)+1];
        a[2*j+1] += b[2*i] * a[2*(j-1)+1] + b[2*i+1] * a[2*(j-1)] +
        c[2*i] * a[2*(j-2)+1] + c[2*i+1] * a[2*(j-2)];
    }

    a[2] += b[2*i] * a[0] - b[2*i+1] * a[1] + c[2*i];
    a[3] += b[2*i] * a[1] + b[2*i+1] * a[0] + c[2*i+1];
    a[0] += b[2*i];
    a[1] += b[2*i+1];
    }

    return( a );
```

```
}


/**********************************************************************
  dcof_bwlp - calculates the d coefficients for a butterworth lowpass
  filter. The coefficients are returned as an array of doubles.

*/

double *dcof_bwlp( int n, double fcf )
{
    int k;            // loop variables
    double theta;     // PI * fcf / 2.0
    double st;        // sine of theta
    double ct;        // cosine of theta
    double parg;      // pole angle
    double sparg;     // sine of the pole angle
    double cparg;     // cosine of the pole angle
    double a;         // workspace variable
    double *rcof;     // binomial coefficients
    double *dcof;     // dk coefficients

    rcof = (double *)calloc( 2 * n, sizeof(double) );
    if( rcof == NULL ) return( NULL );

    theta = PI * fcf;
    st = sin(theta);
    ct = cos(theta);

    for( k = 0; k < n; ++k )
    {
    parg = PI * (double)(2*k+1)/(double)(2*n);
    sparg = sin(parg);
    cparg = cos(parg);
    a = 1.0 + st*sparg;
    rcof[2*k] = -ct/a;
    rcof[2*k+1] = -st*cparg/a;
    }

    dcof = binomial_mult( n, rcof );
    free( rcof );

    dcof[1] = dcof[0];
    dcof[0] = 1.0;
    for( k = 3; k <= n; ++k )
        dcof[k] = dcof[2*k-2];
    return( dcof );
}

/**********************************************************************
  dcof_bwhp - calculates the d coefficients for a butterworth highpass
  filter. The coefficients are returned as an array of doubles.

*/

double *dcof_bwhp( int n, double fcf )
{
    return( dcof_bwlp( n, fcf ) );
}
```

```
/*********************************************************************
  dcof_bwbp - calculates the d coefficients for a butterworth bandpass
  filter. The coefficients are returned as an array of doubles.

*/

double *dcof_bwbp( int n, double f1f, double f2f )
{
    int k;            // loop variables
    double theta;     // PI * (f2f - f1f) / 2.0
    double cp;        // cosine of phi
    double st;        // sine of theta
    double ct;        // cosine of theta
    double s2t;       // sine of 2*theta
    double c2t;       // cosine 0f 2*theta
    double *rcof;     // z^-2 coefficients
    double *tcof;     // z^-1 coefficients
    double *dcof;     // dk coefficients
    double parg;      // pole angle
    double sparg;     // sine of pole angle
    double cparg;     // cosine of pole angle
    double a;         // workspace variables

    cp = cos(PI * (f2f + f1f) / 2.0);
    theta = PI * (f2f - f1f) / 2.0;
    st = sin(theta);
    ct = cos(theta);
    s2t = 2.0*st*ct;        // sine of 2*theta
    c2t = 2.0*ct*ct - 1.0;  // cosine of 2*theta

    rcof = (double *)calloc( 2 * n, sizeof(double) );
    tcof = (double *)calloc( 2 * n, sizeof(double) );

    for( k = 0; k < n; ++k )
    {
    parg = PI * (double)(2*k+1)/(double)(2*n);
    sparg = sin(parg);
    cparg = cos(parg);
    a = 1.0 + s2t*sparg;
    rcof[2*k] = c2t/a;
    rcof[2*k+1] = s2t*cparg/a;
    tcof[2*k] = -2.0*cp*(ct+st*sparg)/a;
    tcof[2*k+1] = -2.0*cp*st*cparg/a;
    }

    dcof = trinomial_mult( n, tcof, rcof );
    free( tcof );
    free( rcof );

    dcof[1] = dcof[0];
    dcof[0] = 1.0;
    for( k = 3; k <= 2*n; ++k )
        dcof[k] = dcof[2*k-2];
    return( dcof );
}

/*********************************************************************
  dcof_bwbs - calculates the d coefficients for a butterworth bandstop
  filter. The coefficients are returned as an array of doubles.
```

```
*/

double *dcof_bwbs( int n, double f1f, double f2f )
{
    int k;              // loop variables
    double theta;       // PI * (f2f - f1f) / 2.0
    double cp;          // cosine of phi
    double st;          // sine of theta
    double ct;          // cosine of theta
    double s2t;         // sine of 2*theta
    double c2t;         // cosine 0f 2*theta
    double *rcof;       // z^-2 coefficients
    double *tcof;       // z^-1 coefficients
    double *dcof;       // dk coefficients
    double parg;        // pole angle
    double sparg;       // sine of pole angle
    double cparg;       // cosine of pole angle
    double a;           // workspace variables

    cp = cos(PI * (f2f + f1f) / 2.0);
    theta = PI * (f2f - f1f) / 2.0;
    st = sin(theta);
    ct = cos(theta);
    s2t = 2.0*st*ct;          // sine of 2*theta
    c2t = 2.0*ct*ct - 1.0;   // cosine 0f 2*theta

    rcof = (double *)calloc( 2 * n, sizeof(double) );
    tcof = (double *)calloc( 2 * n, sizeof(double) );

    for( k = 0; k < n; ++k )
    {
    parg = PI * (double)(2*k+1)/(double)(2*n);
    sparg = sin(parg);
    cparg = cos(parg);
    a = 1.0 + s2t*sparg;
    rcof[2*k] = c2t/a;
    rcof[2*k+1] = -s2t*cparg/a;
    tcof[2*k] = -2.0*cp*(ct+st*sparg)/a;
    tcof[2*k+1] = 2.0*cp*st*cparg/a;
    }

    dcof = trinomial_mult( n, tcof, rcof );
    free( tcof );
    free( rcof );

    dcof[1] = dcof[0];
    dcof[0] = 1.0;
    for( k = 3; k <= 2*n; ++k )
        dcof[k] = dcof[2*k-2];
    return( dcof );
}

/**********************************************************************
  ccof_bwlp - calculates the c coefficients for a butterworth lowpass
  filter. The coefficients are returned as an array of integers.

*/

int *ccof_bwlp( int n )
```

```
{
    int *ccof;
    int m;
    int i;

    ccof = (int *)calloc( n+1, sizeof(int) );
    if( ccof == NULL ) return( NULL );

    ccof[0] = 1;
    ccof[1] = n;
    m = n/2;
    for( i=2; i <= m; ++i)
    {
        ccof[i] = (n-i+1)*ccof[i-1]/i;
        ccof[n-i]= ccof[i];
    }
    ccof[n-1] = n;
    ccof[n] = 1;

    return( ccof );
}

/**********************************************************************
  ccof_bwhp - calculates the c coefficients for a butterworth highpass
  filter. The coefficients are returned as an array of integers.

*/

int *ccof_bwhp( int n )
{
    int *ccof;
    int i;

    ccof = ccof_bwlp( n );
    if( ccof == NULL ) return( NULL );

    for( i = 0; i <= n; ++i)
        if( i % 2 ) ccof[i] = -ccof[i];

    return( ccof );
}

/**********************************************************************
  ccof_bwbp - calculates the c coefficients for a butterworth bandpass
  filter. The coefficients are returned as an array of integers.

*/

int *ccof_bwbp( int n )
{
    int *tcof;
    int *ccof;
    int i;

    ccof = (int *)calloc( 2*n+1, sizeof(int) );
    if( ccof == NULL ) return( NULL );

    tcof = ccof_bwhp(n);
    if( tcof == NULL ) return( NULL );
```

```
    for( i = 0; i < n; ++i)
    {
        ccof[2*i] = tcof[i];
        ccof[2*i+1] = 0.0;
    }
    ccof[2*n] = tcof[n];

    free( tcof );
    return( ccof );
}

/**********************************************************************
  ccof_bwbs - calculates the c coefficients for a butterworth bandstop
  filter. The coefficients are returned as an array of integers.

*/

double *ccof_bwbs( int n, double f1f, double f2f )
{
    double alpha;
    double *ccof;
    int i, j;

    alpha = -2.0 * cos(PI * (f2f + f1f) / 2.0) / cos(PI * (f2f - f1f) / 2.0);

    ccof = (double *)calloc( 2*n+1, sizeof(double) );

    ccof[0] = 1.0;

    ccof[2] = 1.0;
    ccof[1] = alpha;

    for( i = 1; i < n; ++i )
    {
    ccof[2*i+2] += ccof[2*i];
    for( j = 2*i; j > 1; --j )
        ccof[j+1] += alpha * ccof[j] + ccof[j-1];

    ccof[2] += alpha * ccof[1] + 1.0;
    ccof[1] += alpha;
    }

    return( ccof );
}

/**********************************************************************
  sf_bwlp - calculates the scaling factor for a butterworth lowpass filter.
  The scaling factor is what the c coefficients must be multiplied by so
  that the filter response has a maximum value of 1.

*/

double sf_bwlp( int n, double fcf )
{
    int m, k;          // loop variables
    double omega;      // PI * fcf
    double fomega;     // function of omega
    double parg0;      // zeroth pole angle
    double sf;         // scaling factor
```

```
    omega = PI * fcf;
    fomega = sin(omega);
    parg0 = PI / (double)(2*n);

    m = n / 2;
    sf = 1.0;
    for( k = 0; k < n/2; ++k )
        sf *= 1.0 + fomega * sin((double)(2*k+1)*parg0);

    fomega = sin(omega / 2.0);

    if( n % 2 ) sf *= fomega + cos(omega / 2.0);
    sf = pow( fomega, n ) / sf;

    return(sf);
}

/**********************************************************************
  sf_bwhp - calculates the scaling factor for a butterworth highpass filter.
  The scaling factor is what the c coefficients must be multiplied by so
  that the filter response has a maximum value of 1.

*/

double sf_bwhp( int n, double fcf )
{
    int m, k;           // loop variables
    double omega;       // PI * fcf
    double fomega;      // function of omega
    double parg0;       // zeroth pole angle
    double sf;          // scaling factor

    omega = PI * fcf;
    fomega = sin(omega);
    parg0 = PI / (double)(2*n);

    m = n / 2;
    sf = 1.0;
    for( k = 0; k < n/2; ++k )
        sf *= 1.0 + fomega * sin((double)(2*k+1)*parg0);

    fomega = cos(omega / 2.0);

    if( n % 2 ) sf *= fomega + sin(omega / 2.0);
    sf = pow( fomega, n ) / sf;

    return(sf);
}

/**********************************************************************
  sf_bwbp - calculates the scaling factor for a butterworth bandpass filter.
  The scaling factor is what the c coefficients must be multiplied by so
  that the filter response has a maximum value of 1.

*/

double sf_bwbp( int n, double f1f, double f2f )
{
    int k;              // loop variables
    double ctt;         // cotangent of theta
```

```
    double sfr, sfi;  // real and imaginary parts of the scaling factor
    double parg;      // pole angle
    double sparg;     // sine of pole angle
    double cparg;     // cosine of pole angle
    double a, b, c;   // workspace variables

    ctt = 1.0 / tan(PI * (f2f - f1f) / 2.0);
    sfr = 1.0;
    sfi = 0.0;

    for( k = 0; k < n; ++k )
    {
    parg = PI * (double)(2*k+1)/(double)(2*n);
    sparg = ctt + sin(parg);
    cparg = cos(parg);
    a = (sfr + sfi)*(sparg - cparg);
    b = sfr * sparg;
    c = -sfi * cparg;
    sfr = b - c;
    sfi = a - b - c;
    }

    return( 1.0 / sfr );
}

/**********************************************************************
  sf_bwbs - calculates the scaling factor for a butterworth bandstop filter.
  The scaling factor is what the c coefficients must be multiplied by so
  that the filter response has a maximum value of 1.

*/

double sf_bwbs( int n, double f1f, double f2f )
{
    int k;            // loop variables
    double tt;        // tangent of theta
    double sfr, sfi;  // real and imaginary parts of the scaling factor
    double parg;      // pole angle
    double sparg;     // sine of pole angle
    double cparg;     // cosine of pole angle
    double a, b, c;   // workspace variables

    tt = tan(PI * (f2f - f1f) / 2.0);
    sfr = 1.0;
    sfi = 0.0;

    for( k = 0; k < n; ++k )
    {
    parg = PI * (double)(2*k+1)/(double)(2*n);
    sparg = tt + sin(parg);
    cparg = cos(parg);
    a = (sfr + sfi)*(sparg - cparg);
    b = sfr * sparg;
    c = -sfi * cparg;
    sfr = b - c;
    sfi = a - b - c;
    }
    return( 1.0 / sfr );
}
```

- **iir.h**

```c
/*
 * iir.h
 *
 *  Created on: 5 jul. 2022
 *      Author: pilar
 */

#ifndef IIR_H_
#define IIR_H_

double *binomial_mult( int n, double *p );
double *trinomial_mult( int n, double *b, double *c );

double *dcof_bwlp( int n, double fcf );
double *dcof_bwhp( int n, double fcf );
double *dcof_bwbp( int n, double f1f, double f2f );
double *dcof_bwbs( int n, double f1f, double f2f );

int *ccof_bwlp( int n );
int *ccof_bwhp( int n );
int *ccof_bwbp( int n );
double *ccof_bwbs( int n, double f1f, double f2f );

double sf_bwlp( int n, double fcf );
double sf_bwhp( int n, double fcf );
double sf_bwbp( int n, double f1f, double f2f );
double sf_bwbs( int n, double f1f, double f2f );

#endif /* IIR_H_ */
```

A continuación se recogen los códigos de prueba de los filtros, que finalmente no se han incluido en el programa.

- **Filtros.c**

```
/*
 * filtros.c
 *
 *  Created on: 10 jun. 2022
 *      Author: pilar
 */

#include "math.h"
#include "filtros.h"
#include "libreria_midi.h"


double Pi = M_PI;

volatile float A, alpha, a0, a1, a2, b0, b1, b2;
volatile double w0;


void calcula_filtro(void)
{
        switch (boton)
        {
            case BOTON_1:
                switch (control_filtro)
                {
                    case c_f0:
                        f0 = f0+50;
                    break;

                    case c_G:
                        G = G+1;
                    break;

                    case c_Q:
                        Q = Q+(float)0.5;
                    break;

                    default:
                       break;
                }
                break;

            case BOTON_2:
                switch (control_filtro)
                {
                    case c_f0:
                        f0 = f0-50;
                    break;

                    case c_G:
                         G = G-1;
                    break;
```

```
                    case c_Q:
                        Q = Q-(float)0.5;
                    break;

                    default:
                      break;
                }
            break;

            default:
                break;
        }
    // Calcula parámetros intermedios

        A = (float) (pow (10, G/40.0));
        w0 = (double) (2.0 * Pi * f0 / Fs);
        alpha = (float) (sin(w0)/(2*Q));

    // Calcula coeficientes

        b0 =  1 + alpha * A;
        b1 = (float) (-2.0 * (float)cos(w0));
        b2 =  1 - alpha * A;

        a0 =  1 + (float)(alpha / A);
        a1 = (float) (-2.0 * (float)cos(w0));
        a2 =  1 - (float)(alpha / A);
}

float filtrobanda(void)
{
    // NOTA: (NF-1) en este caso =2, es el valor actual, y el más antiguo guardado es
el 0 (NF-NF)

    /*
    VectorS[0] = (float)(b0/a0)*VectorE[0];
    VectorS[1] = (float)(b0/a0)*VectorE[1] + (float)(b1/a0)*VectorE[0]
      *                                                  - (float)(a1/a0)*VectorS[0];
//OJO
      *                                                  */

    VectorS[0] = b0*VectorE[0]/a0;
    VectorS[1] = b0*VectorE[1]/a0 + b1*VectorE[0]/a0
                                                - a1*VectorS[0]/a0;  //OJO

    for (long f = 2; f < N_muestras; f++)
    {
        /*
        VectorS[f] = (float)(b0/a0)*VectorE[f] + (float)(b1/a0)*VectorE[f-1] +
(float)(b2/a0)*VectorE[f-2]
                                        - (float)(a1/a0)*VectorS[f-1] -
(float)(a2/a0)*VectorS[f-2];
                                        */
        VectorS[f] = b0*VectorE[f]/a0 + b1*VectorE[f-1]/a0 + b2*VectorE[f-2]/a0
                                                - a1*VectorS[f-1]/a0 - a2*VectorS[f-
2]/a0;
    }

    return VectorS[N_muestras-1];
}
```

- **Filtros.h**

```
/*
 * filtros.h
 *
 *  Created on: 23 jun. 2022
 *      Author: pilar
 */

#ifndef FILTROS_H_
#define FILTROS_H_

#define M_PI        3.14159265358979323846


/* ********************************* VARIABLES ********************************* */
extern double Pi;

extern volatile float A, alpha, a0, a1, a2, b0, b1, b2;
extern volatile double w0;


/* ********************************* FUNCIONES ********************************* */
void calcula_filtro(void);
float filtrobanda(void);

#endif /* FILTROS_H_ */
```

- **Filtro2.c**

```
/*
 * filtro2.c
 *
 *  Created on: 4 jul. 2022
 *      Author: pilar
 */
//#include <iostream>
#include <stdio.h>
//#include <vector>
#include <math.h>
#include <stdlib.h>
#include <complex.h>

#include "filtros2.h"
#include "libreria_midi.h"

//using namespace std;

#define N 10                                //The number of images which construct a time
series for each pixel
#define PI 3.14159

double Bw2, Wn2;

double *ComputeLP()
{
    double *NumCoeffs;
    int m;
    int i;

    NumCoeffs = (double *)calloc( FilterOrder+1, sizeof(double) );
    if( NumCoeffs == NULL ) return( NULL );

    NumCoeffs[0] = 1;
    NumCoeffs[1] = FilterOrder;
    m = FilterOrder/2;

    for( i=2; i <= m; ++i)
    {
        NumCoeffs[i] =(double) (FilterOrder-i+1)*NumCoeffs[i-1]/i;
        NumCoeffs[FilterOrder-i]= NumCoeffs[i];
    }
    NumCoeffs[FilterOrder-1] = FilterOrder;
    NumCoeffs[FilterOrder] = 1;

    return NumCoeffs;
}

double *ComputeHP()
{
    double *NumCoeffs;
    int i;

    NumCoeffs = ComputeLP();
```

```
    if(NumCoeffs == NULL ) return( NULL );

    for( i = 0; i <= FilterOrder; ++i)
        if( i % 2 ) NumCoeffs[i] = -NumCoeffs[i];

    return NumCoeffs;
}

double *TrinomialMultiply(double *b, double *c )
{
    int i, j;
    double *RetVal;

    RetVal = (double *)calloc( 4 * FilterOrder, sizeof(double) );

    if( RetVal == NULL ) return( NULL );

    RetVal[2] = c[0];
    RetVal[3] = c[1];
    RetVal[0] = b[0];
    RetVal[1] = b[1];

    for( i = 1; i < FilterOrder; ++i )
    {
        RetVal[2*(2*i+1)]   += c[2*i] * RetVal[2*(2*i-1)]   - c[2*i+1] *
RetVal[2*(2*i-1)+1];
        RetVal[2*(2*i+1)+1] += c[2*i] * RetVal[2*(2*i-1)+1] + c[2*i+1] *
RetVal[2*(2*i-1)];

        for( j = 2*i; j > 1; --j )
        {
            RetVal[2*j]   += b[2*i] * RetVal[2*(j-1)]   - b[2*i+1] * RetVal[2*(j-
1)+1] +
                c[2*i] * RetVal[2*(j-2)]   - c[2*i+1] * RetVal[2*(j-2)+1];
            RetVal[2*j+1] += b[2*i] * RetVal[2*(j-1)+1] + b[2*i+1] * RetVal[2*(j-1)]
+
                c[2*i] * RetVal[2*(j-2)+1] + c[2*i+1] * RetVal[2*(j-2)];
        }

        RetVal[2] += b[2*i] * RetVal[0] - b[2*i+1] * RetVal[1] + c[2*i];
        RetVal[3] += b[2*i] * RetVal[1] + b[2*i+1] * RetVal[0] + c[2*i+1];
        RetVal[0] += b[2*i];
        RetVal[1] += b[2*i+1];
    }

    return RetVal;
}

/*
double *ComputeNumCoeffs(int FilterOrder)             // Calcular coeficientes del
numerador
{
    double *TCoeffs;
    double *NumCoeffs;
    int i;

    NumCoeffs = (double *)calloc( 2*FilterOrder+1, sizeof(double) );
    if( NumCoeffs == NULL ) return( NULL );

    TCoeffs = ComputeHP(FilterOrder);
```

```
    if( TCoeffs == NULL ) return( NULL );

    for( i = 0; i < FilterOrder; ++i)
    {
        NumCoeffs[2*i] = TCoeffs[i];
        NumCoeffs[2*i+1] = 0.0;
    }
    NumCoeffs[2*FilterOrder] = TCoeffs[FilterOrder];

    free(TCoeffs);

    return NumCoeffs;
}
*/

// Con factor de escala

double *ComputeNumCoeffs(double Lcutoff, double Ucutoff, double *DenC)
{
    double *TCoeffs;
    double *NumCoeffs;
    double complex *NormalizedKernel;       // Con Re = 0, Im = 0
    double Numbers[2*FilterOrder+1];
    int i;

    for (i = 0; i<(2*FilterOrder); i++)
    {
        Numbers[i]= i;
    }

    NumCoeffs = (double *)calloc( 2*FilterOrder+1, sizeof(double) );
    if( NumCoeffs == NULL ) return( NULL );

    NormalizedKernel = (complex*)calloc( 2*FilterOrder+1, sizeof(complex));
    if( NormalizedKernel == NULL ) return( NULL );

    TCoeffs = ComputeHP();
    if( TCoeffs == NULL ) return( NULL );

    for( i = 0; i < FilterOrder; ++i)
    {
        NumCoeffs[2*i] = TCoeffs[i];
        NumCoeffs[2*i+1] = 0.0;
    }
    NumCoeffs[2*FilterOrder] = TCoeffs[FilterOrder];

    double cp[2];
    double Bw, Wn;
    cp[0] = 2*2.0*tan(PI * Lcutoff/ 2.0);
    cp[1] = 2*2.0*tan(PI * Ucutoff / 2.0);

    Bw = cp[1] - cp[0];                     // Ancho de banda

    //Center frequency
    Wn = sqrt(cp[0]*cp[1]);
    Wn = 2*atan2(Wn,4);

    // Para observar variables globales
    Wn2 = Wn;
    Bw2 = Bw;
```

```c
    double complex result = -1.0*I;

    for(int k = 0; k<(2*FilterOrder); k++)
    {
        NormalizedKernel[k] = cexp(result*Wn*Numbers[k]);
    }
    double b=0;
    double den=0;

    for(int d = 0; d<(2*FilterOrder); d++)
    {
        b+=creal(NormalizedKernel[d]*NumCoeffs[d]);
        den+=creal(NormalizedKernel[d]*DenC[d]);
    }
    for(int c = 0; c<(2*FilterOrder); c++)
    {
        NumCoeffs[c]=(NumCoeffs[c]*den)/b;
    }

    free(TCoeffs);
    return NumCoeffs;
}

double *ComputeDenCoeffs(double Lcutoff, double Ucutoff )     // Calcular
coeficientes del denominador
{
    int k;            // loop variables
    double theta;     // PI * (Ucutoff - Lcutoff) / 2.0

    double cp;        // Coseno de phi
    double st;        // Seno de theta
    double ct;        // Coseno de theta
    double s2t;       // Seno de 2*theta
    double c2t;       // Coseno de 2*theta

    double *RCoeffs;        // z^-2 coefficients
    double *TCoeffs;        // z^-1 coefficients
    double *DenomCoeffs;    // dk coefficients
    double PoleAngle;       // Ángulo del polo
    double SinPoleAngle;    // Seno del ángulo del polo
    double CosPoleAngle;    // Coseno del ángulo del polo
    double a;               // workspace variables

    cp = cos(PI * (Ucutoff + Lcutoff) / 2.0);
    theta = PI * (Ucutoff - Lcutoff) / 2.0;
    st = sin(theta);
    ct = cos(theta);
    s2t = 2.0*st*ct;        // Seno de 2*theta
    c2t = 2.0*ct*ct - 1.0;  // Coseno de 2*theta

    RCoeffs = (double *)calloc( 2 * FilterOrder +1, sizeof(double) );
    TCoeffs = (double *)calloc( 2 * FilterOrder +1, sizeof(double) );

    for( k = 0; k < FilterOrder; ++k )
    {
        PoleAngle = PI * (double)(2*k+1)/(double)(2*FilterOrder);
        SinPoleAngle = sin(PoleAngle);
        CosPoleAngle = cos(PoleAngle);
        a = 1.0 + s2t*SinPoleAngle;
```

```
        RCoeffs[2*k] = c2t/a;
        RCoeffs[2*k+1] = s2t*CosPoleAngle/a;
        TCoeffs[2*k] = -2.0*cp*(ct+st*SinPoleAngle)/a;
        TCoeffs[2*k+1] = -2.0*cp*st*CosPoleAngle/a;
    }


    DenomCoeffs = TrinomialMultiply(TCoeffs, RCoeffs);
    free(TCoeffs);
    free(RCoeffs);

    DenomCoeffs[1] = DenomCoeffs[0];
    DenomCoeffs[0] = 1.0;
    for( k = 3; k <= 2*FilterOrder; ++k )
        DenomCoeffs[k] = DenomCoeffs[2*k-2];

    return DenomCoeffs;
}

void filtro2(void)
{
    int i,j;                        // Contadores

    VectorS[0]=NumC2[0] * VectorE[0];       // Primera salida y

    for (i=1; i<(FilterOrder+1); i++)
    {
        VectorS[i]=0.0;
        for (j=0;j<i+1;j++)
            VectorS[i] = VectorS[i]+NumC2[j]*VectorE[i-j];

        for (j=0;j<i;j++)                       // NO es repetición, es que reutiliza
el "y" calculado en el bucle anterior!
            VectorS[i] = VectorS[i]-DenC2[j+1]*VectorS[i-j-1];
    }

    for (i=(FilterOrder+1); i<N_muestras+1; i++)
    {
        VectorS[i]=0.0;

        for (j=0; j<FilterOrder+1; j++)
            VectorS[i] = VectorS[i]+NumC2[j]*VectorE[i-j];

        for (j=0; j<FilterOrder; j++)
            VectorS[i] = VectorS[i]-DenC2[j+1]*VectorS[i-j-1];
    }
}


void calcula_filtro2(void)
{
    /*Frequency bands is a vector of values - Lower Frequency Band and Higher
Frequency Band:
    -> First value is lower cutoff and second value is higher cutoff
    -> These values are as a ratio of f/fs, where fs is sampling rate, and f is
cutoff frequency
    -> Therefore they should lie in the range [0 1]  */

    int LF = f0 - W/2;
    int HF = f0 + W/2;
```

```
    double FrequencyBands[2] = {LF/Fs, HF/Fs};

    double *DenC = ComputeDenCoeffs(FrequencyBands[0], FrequencyBands[1]);
    double *NumC = ComputeNumCoeffs(LF, HF, DenC);

    // ojo: DenC que generan las funciones sn punteros y queremos copiarlas en el
vector global DenC2:
    double* ptr = NumC2;     // Puntero que contiene la dirección de la variable
global NumC2
    double* ptr2 = DenC2;    // Puntero que contiene la dirección de la variable
global DenC2

    memcpy(ptr, NumC, (FilterOrder*2 + 1) * sizeof(double));
    memcpy(ptr2, DenC, (FilterOrder*2 + 1) * sizeof(double));
}
```

- **Filtros2.h**

```
/*
 * filtros2.h
 *
 *  Created on: 4 jul. 2022
 *      Author: pilar
 */

#ifndef FILTROS2_H_
#define FILTROS2_H_

double *ComputeLP();
double *ComputeHP();
double *TrinomialMultiply(double *b, double *c);
double *ComputeNumCoeffs(double Lcutoff, double Ucutoff, double *DenC);
double *ComputeDenCoeffs(double Lcutoff, double Ucutoff );
void filtro2(void);
void calcula_filtro2(void);

#endif /* FILTROS2_H_ */
```

- **Filtros3.c**

```
/*
 * filtros3.c
 *
 *  Created on: 5 jul. 2022
 *      Author: pilar
 */


//#include <iostream>
#include <stdio.h>
//#include <cstdlib>
#include <stdlib.h>
//#include <cstdio>
#include <stdio.h>
//#include <vector.h>
#include <string.h>
//#include <cmath>
#include <math.h>
#include "iir.h"

#include "libreria_midi.h"
#include "filtros3.h"

//using namespace std;
//typedef std::vector<double> vectord;

double vectord;

#define M_PI        3.14159265358979323846


int calcula_filtro3(void)
{
    switch (boton)
    {
        case BOTON_1:
            switch (control_filtro)
            {
                case c_f0:
                    f0 = f0+50;
                break;

                case c_G:
                    G = G+1;
                break;

                case c_Q:
                    Q = Q+(float)0.5;
                break;

                default:
                   break;
            }
            break;

        case BOTON_2:
            switch (control_filtro)
            {
```

```
                case c_f0:
                    f0 = f0-50;
                break;

                case c_G:
                     G = G-1;
                break;

                case c_Q:
                    Q = Q-(float)0.5;
                break;

                default:
                   break;
            }
        break;

        default:
            break;
    }


    //int n = 4;                 // Filter order = 4
    int sff = 1;                 // Normalization - a 0 para no escalar, a 1 para
escalar
    //double fcf = 180;            // Cutoff frequency

    int LF = f0 - W/2;
    int HF = f0 + W/2;


    double cp[2];
    double Bw, Wn;
    cp[0] = 2*2.0*tan(M_PI * LF/ 2.0);
    cp[1] = 2*2.0*tan(M_PI * HF / 2.0);

    Bw = cp[1] - cp[0];
    //center frequency
    Wn = sqrt(cp[0]*cp[1]);
    Wn = 2*atan2(Wn,4);

    double fcf = Wn;


    double sf;
    double *dcof;
    int *ccof;
    double *BB;
    double *AA;

    /*
    BB = new double[n+1];
    AA = new double[n+1];
    */

    BB = (double *)calloc( FilterOrder+1, sizeof(double) );
    AA = (double *)calloc( FilterOrder+1, sizeof(double) );

    /* calculate the d coefficients */
    dcof = dcof_bwhp(FilterOrder, fcf );
```

```
    if( dcof == NULL )
    {
        return -1;
    }

    /* calculate the c coefficients */
    ccof = ccof_bwhp(FilterOrder);
    if( ccof == NULL )
    {
        return -1;
    }

    sf = sf_bwhp(FilterOrder, fcf );                  /* scaling factor for the c
coefficients */

    for(int i = 0;i <= FilterOrder; ++i)
    {
        BB[i] = (double)ccof[i] * sf;
        AA[i] = dcof[i];
    }
    //These two vectors are only for calculating filtfilt function in another
repository
    /*
    vector a_coeff(AA,AA + n + 1);
    vector b_coeff(BB,BB + n + 1);
    */

    /*
    delete[] AA;
    delete[] BB;
    */

    // ojo: DenC que generan las funciones sn punteros y queremos copiarlas en el
vector global DenC2:
    double* ptr = NumC2;     // Puntero que contiene la dirección de la variable
global NumC2
    double* ptr2 = DenC2;    // Puntero que contiene la dirección de la variable
global DenC2

    memcpy(ptr, BB, (FilterOrder*2 + 1) * sizeof(double));
    memcpy(ptr2, AA, (FilterOrder*2 + 1) * sizeof(double));


    return 1;
}

void filtro3(void)              // Es igual que el 2 pero calculamos los coeficientes
de otra forma
{
    int i,j;                    // Contadores

    VectorS[0]=NumC2[0] * VectorE[0];       // Primera salida y

    for (i=1; i<(FilterOrder+1); i++)
    {
        VectorS[i]=0.0;
        for (j=0;j<i+1;j++)
            VectorS[i] = VectorS[i]+NumC2[j]*VectorE[i-j];
```

```
        for (j=0;j<i;j++)                       // NO es repetición, es que reutiliza
el "y" calculado en el bucle anterior!
            VectorS[i] = VectorS[i]-DenC2[j+1]*VectorS[i-j-1];
    }

    for (i=(FilterOrder+1); i<N_muestras+1; i++)
    {
        VectorS[i]=0.0;

        for (j=0; j<FilterOrder+1; j++)
            VectorS[i] = VectorS[i]+NumC2[j]*VectorE[i-j];

        for (j=0; j<FilterOrder; j++)
            VectorS[i] = VectorS[i]-DenC2[j+1]*VectorS[i-j-1];
    }

}
```

- **Filtros3.h**

```
/*
 * filtros3.h
 *
 *  Created on: 5 jul. 2022
 *      Author: pilar
 */

#ifndef FILTROS3_H_
#define FILTROS3_H_

int calcula_filtro3(void);
void filtro3(void);


#endif /* FILTROS3_H_ */
```

REFERENCIAS

[1] MicroPython, [En línea]. Available: https://micropython.org/.

[2] M. d. P. Martínez Serrano, «Programación de un sintetizador MIDI usando el microcontrolador MSP430FR2355,» 2020. [En línea]. Available: https://idus.us.es/bitstream/handle/11441/94524/TFG-2752-MARTINEZ%20SERRANO.pdf?sequence=1&isAllowed=y.

[3] Real Academia Española, «RAE,» 2019. [En línea]. Available: https://dle.rae.es/.. [Último acceso: Septiembre 2019].

[4] SFU, «Frequency Modulation,» [En línea]. Available: https://www.sfu.ca/~truax/Frequency_Modulation.html.

[5] B. Bianchini, «Protocolo MIDI,» [En línea]. Available: https://proyectoidis.org/protocolo-midi/. [Último acceso: 2022].

[6] Universidad Politécnica de Valencia, «Efectos de sonido, Filtrado y Ecualización,» [En línea]. Available: http://www.disca.upv.es/adomenec/IASPA/tema4/ES-Filtros.html.

[7] Pycom, «WiPy 3.0 Datasheet,» [En línea]. Available: https://docs.pycom.io/gitbook/assets/specsheets/Pycom_002_Specsheets_WiPy3.0_v2.pdf. [Último acceso: Diciembre 2021].

[8] Pycom, «Expansion 3.0 Datasheet,» [En línea]. Available: https://docs.pycom.io/gitbook/assets/expansion3-specsheet-1.pdf. [Último acceso: Diciembre 2021].

[9] MICROCHIP, *MCP4921/4922, 12-Bit DAC with SPI™ Interface.*

[10] A. Camarillo, «Introducción a los SO en tiempo real: ¿Qué es un RTOS?,» Marzo 2021. [En línea]. Available: https://blog.330ohms.com/2021/03/29/introduccion-a-los-so-en-tiempo-real-que-es-un-rtos/.

[11] Renesas Synergy, *S7G2 Microcontroller Group.*

[12] Revista Española de Electrónica, «La Plataforma Synergy de Renesas,» 29 Junio 2016. [En línea]. Available: https://www.redeweb.com/articulos/componentes/la-plataforma-synergy.

[13] La Vanguardia, «Un siglo de instrumentos electrónicos,» Marzo 2019. [En línea]. Available: https://www.lavanguardia.com/tecnologia/20190323/461165138005/siglo-instrumentos-electronicos-futuro-mas-autonomo.html.

[14] Renesas Synergy, *Starter Kit SK-S7G2 User Manual.*

[15] charles-typ, «Butterworth_filter_coefficients-MATLAB-in-C,» [En línea]. Available: https://github.com/wustyuyi/Butterworth_filter_coefficients-MATLAB-in-C.