FACULTAD DE INGENIERÍAS Y CIENCIAS APLICADAS

"DESARROLLO DE UN MÓDULO DE PROCESAMIENTO DE AUDIO DIGITAL DE AMPLIFICACIÓN Y EFECTOS PARA SEÑALES MONOFÓNICAS Y ESTEREOFÓNICAS"

AUTOR

José Sebastián Aguilar Silva

AÑO

2020

FACULTAD DE INGENIERÍAS Y CIENCIAS APLICADAS

DESARROLLO DE UN MÓDULO DE PROCESAMIENTO DE AUDIO DIGITAL DE AMPLIFICACIÓN Y EFECTOS PARA SEÑALES MONOFÓNICAS Y ESTEREOFÓNICAS

Trabajo de titulación presentado en conformidad con los requisitos establecidos para optar por el título de Ingeniero en Sonido y Acústica

Profesor Guía

Msc. Paul Adrián Cabezas Yánez

Autor

José Sebastián Aguilar Silva

Año

2020

DECLARACIÓN DEL PROFESOR GUÍA

Declaro haber dirigido el trabajo, Desarrollo de un módulo de procesamiento de audio digital de amplificación y efectos para señales monofónicas y estereofónicas, a través de reuniones periódicas con el estudiante José Sebastián Aguilar Silva, en el semestre 202010, orientando sus conocimientos y competencias para un eficiente desarrollo del tema escogido y dando cumplimiento a todas las disposiciones vigentes que regulan los Trabajos de Titulación.

Paúl Adrián Cabezas Yánez

Master en Industrias Creativas en Música y Sonido

CI: 1719189548

DECLARACIÓN DE PROFESOR CORRECTOR

Declaro haber revisado este trabajo, Desarrollo de un módulo de procesamiento de audio digital de amplificación y efectos para señales monofónicas y estereofónicas, del estudiante José Sebastián Aguilar Silva, en el semestre 202010, dando cumplimiento a todas las disposiciones vigentes que regulan los Trabajos de Titulación".

Daniel Núñez Solano

Daniel Alejandro Núñez Solano

Master of science of acoustic engineering

CI: 1716430911

DECLARACIÓN DE AUTORÍA DEL ESTUDIANTE

Declaro que este trabajo es original, de mi autoría, que se han citado las fuentes correspondientes y que en su ejecución se respetaron las disposiciones legales que protegen los derechos de autor vigentes.

Jose Sebastián Aguilar Silva

CI: 1723031439

AGRADECIMIENTOS

A Dios por darme todas las oportunidades para alcanzar mis metas. A mi familia por ayudarme siempre a seguir adelante sin desmayar. A los docentes que han impartido sus conocimientos para que yo pueda ser un profesional altamente capacitado.

DEDICATORIA

A Dios y a mi familia por apoyarme en todos los caminos que emprendo y guiarme siempre por el camino del bien. A mi perrita Stacey que me da la fuerza para seguir adelante y me llena la vida de alegría

RESUMEN

El trabajo correspondiente al proyecto de titulación presentado en este documento consiste en el diseño, desarrollo e implementación de un módulo para señales digitales de audio, divido en 4 efectos o *plugins* individuales, para los cuales se consiguió compatibilidad con los sistemas operativos Windows y MacOS, así como con los principales formatos de extensiones disponibles actualmente en la industria del software de audio. El desarrollo de dichos módulos se ha realizado en el lenguaje de programación C++, utilizando un marco de referencia optimizado para desarrollo de software de audio (*JUCE*). Dentro de los módulos desarrollados se encuentran los siguientes efectos de audio: Ecualizador, Compresor, *Delay* o Eco y Reverberación. Posteriormente en este documento se presentan las distintas maneras utilizadas para la optimización de *plugins* implementados, así como los problemas de codificación y compatibilidad enfrentados en su desarrollo.

Los resultados que se presentan se han obtenido mediante pruebas realizadas en las estaciones de trabajo de audio digital (*DAW*s) que trabajan con los formatos de los *plugins* desarrollados. En la sección de anexos se presentan los códigos utilizados para el desarrollo de los módulos de este proyecto.

Palabras clave: *plugins*, efectos, módulos, algoritmo, *DAW*, procesamiento.

ABSTRACT

The work corresponding to the degree project presented in this document consists in the design, development and implementation of one module for digital audio signals, divided in four individual effects, which are compatible with the Windows and MacOS operating systems, as well as with the main formats of extensions currently available in the audio software industry. The development of these modules has been carried out in the C ++ programming language, using a reference framework optimized for the development of audio software (JUCE). Within the modules are the following audio effects: Equalizer, Compressor, Delay or Delay and Reverb. Later in this document the different alternatives used for the optimization of implemented complements are presented, as well as the coding and compatibility problems faced in its development.

The results presented have been obtained through tests carried out in the Smaart7 program and in the digital audio workstations (DAW) that work with the formats of the developed accessories. The codes used for the development of the modules of this project are presented in the annexes section.

Keywords: *plugins*, effects, algorithm, *DAW*, processing.

ÍNDICE

1. CAPÍTULO I: INTRODUCCIÓN

1.1 Antecedentes

Las señales analógicas de audio se definen porque dependen de una o más variables continuas, que en el caso del audio es el tiempo. En cambio, para las señales digitales, como tienen que ser computadas por un ordenador, las variables de las cuales dependen deben ser discretizadas para que sea posible su conversión, del dominio analógico al digital (ADC). Al procedimiento de discretización se lo conoce como muestreo, y consiste en tomar datos de la señal analógica en períodos idénticos de tiempo, generando un número de muestras distribuidas en el tiempo total del proceso realizado para de esta manera obtener la señal digitalizada (Oppenheim, Willsky y Young, 1990).

Según la ley Nyquist, para muestrear señales de audio, es necesario que el período de muestreo corresponda por lo menos al doble de la frecuencia más alta que se desea muestrear, para no perder información de la señal original. Tomando en cuenta que el oído humano puede percibir las frecuencias de sonido hasta los 20 kHz, entonces, la frecuencia de muestreo mínima tiene que ser a partir 40 kHz. Mientras mayor sea la frecuencia de muestreo, mejor sea la calidad de la señal digitalizada, sin embargo, también será mayor el procesamiento y espacio de memoria requeridos para reproducirla y almacenarla (Oppenheim, Willsky y Young, 1990).

Otra manera de obtener señales de audio digitales es a través de la síntesis, mediante la cual se generan tonos con osciladores digitales que manipulan diversos parámetros que pueden ser modificados para alterar el sonido resultante de la señal. Una ventaja en este tipo de sonidos es que no hay necesidad de una conversión analógica digital para que la señal esté discretizada, ya que, desde su generación ya pertenece al dominio digital. Sin embargo, para poder escucharla. sí será necesario realizar una conversión digital-analógica. A las señales generadas mediante síntesis, también se les puede aplicar efectos como distorsión, ecualización, compresión, así como modificar parámetros de manera lineal o no lineal. A pesar de ello, es importante tomar en cuenta que, si bien, de manera teórica dichos efectos afectarán de forma idéntica a una señal analógica digitalizada mediante muestreo que a una señal generada mediante osciladores digitales, es posible que en la práctica las modificaciones no se escuchen iguales, especialmente si se utilizan métodos no lineales en la modificación de los parámetros de dichas señales (Lazzarini y Timoney, 2013).

Una vez que se conoce de dónde viene la generación o digitalización de las señales de audio, es necesario mencionar ciertos parámetros fundamentales que permiten manipularlas para generar amplificación y efectos, lo cual se desea para el desarrollo del módulo descrito en este documento. Algunos de los parámetros más básicos son: el número de muestras, que corresponde al tiempo en el dominio analógico, la amplitud, que corresponde al voltaje y a la presión en el dominio eléctrico y acústico respectivamente, así como también un parámetro muy importante, el cual es el de la frecuencia, ya que, este contiene el espectro de la señal. Estos parámetros pueden manejarse de manera individual o como conjunto dependiendo de lo que se desee lograr. En algunas ocasiones, al modificar alguna de estas variables, también se pueden notar cambios en las demás, sin embargo, en otros casos, cada parámetro se puede modificar de manera independiente a los demás (Oppenheim, Willsky y Young, 1990).

A las técnicas de manipular los distintos parámetros de una señal muestreada se denomina procesamiento digital de señales (Digital Signal Processing o DSP en inglés). El DSP se puede aplicar no solamente a contenido sonoro sino también a otros tipos como por ejemplo tratamiento de imágenes o incluso de datos estadísticos. Una de las herramientas más útiles para el procesamiento digital de señales es el uso de las matemáticas, por ejemplo, al aplicar transformadas de Fourier a la información de la señal tratada, se puede pasar del dominio del tiempo (o muestras) al dominio frecuencial y viceversa. Además, con el aprovechamiento de operadores matemáticos complejos como la convolución se puede modelar el comportamiento de muchos dispositivos analógicos para procesar señales digitales con una muy alta calidad (Pirkle, 2017a).

Cuando se tiene señales que han sufrido daños al atravesar el proceso de muestra para su digitalización o que simplemente han perdido parte de su información por razones desconocidas, se puede aplicar un tipo de procesamiento de reconstrucción, en el cual se intenta recuperar la información pérdida, ya sea mediante el análisis de histogramas e interpolación predictiva de datos faltantes o por el reinicio del proceso de muestreo para constatar si la información muestreada en un inicio se ha digitalizado con fallas o simplemente originalmente, desde el dominio analógico no tenía información suficiente para ser muestreada. En general, los procesos de reconstrucción de audio pueden ser muy complejos, pero al realizarse de manera correcta se pueden obtener muy buenos resultados (Stankovic y Brajovic, 2018).

Dentro de los procesos de reconstrucción de señales, se puede incluir los casos en los que la señal no se encuentra dañada ni ha perdido información al ser digitalizada, simplemente dentro de su contenido sonoro tiene algo indeseado que el usuario procura eliminar, por ejemplo, un audio puede tener demasiada reverberación, eco o demasiado ruido de fondo que pueden causar que ciertos sonidos de este no se entiendan y, por lo tanto, estos excesos requieran ser removidos. Estos procesos son inclusive más complejos que el tipo de reconstrucción mencionada anteriormente, ya que en estos casos se utilizan procedimientos matemáticos como ecuaciones de diferencias o mínimos cuadrados, entre otras, los cuales son aproximaciones que ayudarán a acercarse a un resultado deseado, según como se los aplique, pero nunca se podrá eliminar por completo el exceso de contenido indeseado de la señal tratada (Lim, Zhang, Habets y Naylor, 2014).

Por otro lado, existe otro tipo de procesamiento que se enfoca en añadir características a los parámetros existentes de la señal tratada. Este tipo de procesamiento puede ser aplicado para el espectro, la amplitud o cualquier otro parámetro manipulable de la señal digital. Uno de los factores que más se suele manipular es el espectro o el contenido frecuencial del audio, para esto, se acostumbra a utilizar filtros, los cuales pueden ser diseñados de maneras distintas, como por ejemplo iterativos, bicuadráticos, monopolares, bipolares, entre otros. Dependiendo del tipo de filtro que se utilice, se pueden obtener resultados que pueden ser muy similares o pueden variar, tanto en la modificación del sonido en sí como en la utilización de recursos computacionales requeridos para realizar dicho procesamiento (Ramo, Valimaki y Bank, 2014).

En otros casos se utilizan filtros para ecualización que se basan en equipos analógicos que se utilizaron en legendarias grabaciones del siglo XX, y que han alcanzado fama por sus características sonoras. Algunos de estos tipos de filtros son los de escalera, de retroalimentación, de control previo, etc. Se puede obtener este tipo de filtros de dos maneras principales, la primera es realizar una serie de mediciones que reflejen en comportamiento del equipo analógico que se desea modelar e ingresar los datos al ordenador para que el filtro diseñado procese las señales según la información obtenida en las mediciones. La segunda manera, es diseñar un modelo digital con base en una representación teórica del circuito electrónico del equipo analógico que se desee replicar, sin necesidad de realizar medición alguna. El grado de semejanza de lo que se ha diseñado digitalmente en comparación al comportamiento analógico del equipo modelado, siempre tendrá un componente relativo a la percepción de quien lo analice, por lo cual será muy difícil de valorarlo de manera objetiva (D'angelo y Valimaki, 2014).

Con respecto a la dinámica de la señal manipulada, un tipo de procesamiento muy comúnmente utilizado es la compresión, que consiste en modificar el rango dinámico del audio para que esté más balanceado y uniforme. Para este propósito se pueden utilizar algunas de las herramientas matemáticas mencionadas anteriormente como la convolución. También se puede combinar este tipo de procesamiento con el de modificación del espectro sonoro, obteniendo por ejemplo una compresión multibanda, similar a la antes mencionada pero enfocada a distintas bandas de frecuencias (Bessell, 2013).

Existen muchos otros resultados que se pueden obtener al procesar digitalmente una señal de audio, las cuales dependerán del sonido que se desee conseguir a través de dicho procesamiento. Sin importar si se busca diseñar efectos, amplificadores u otro tipo de procesadores, en la mayoría de los casos se obtendrá, adicionalmente al procesamiento planteado, cierto contenido frecuencial correspondiente a la aparición de armónicos de la señal original, lo cual se conoce como ruido digital. Este fenómeno puede ser beneficioso o perjudicial para el sonido buscado, por lo cual se debe tomar muy en cuenta cuánto de este “ruido digital” puede generarse para poder maximizarlo o disminuirlo según se requiera (Zivanovic, 2015).

Para que la manipulación de la señal se pueda realizar con facilidad, es necesario que las modificaciones que se realicen en la misma se escuchen mientras se ejecuten, es decir en tiempo real. Para esto se puede generar un buffer en el ordenador, que consiste en asignar un espacio de memoria temporal a la información de la señal para que se realice el procesamiento de esta por partes, permitiendo al usuario escuchar los cambios que efectúa en la señal. Dependiendo del tamaño del buffer que se le asigne al procesamiento de la señal, así como de si esta es monofónica (1 sonido de manera simultánea) o polifónica (más de un sonido simultáneamente), se generará un retardo de esta que es conocido como latencia. Para aplicaciones de uso de módulo de audio en vivo, es recomendable que la latencia no supere los 10 milisegundos (Baelde, Biernacki, y Greff, 2019).

Un post proceso al DSP es el de codificar la señal, es decir obtener un nuevo audio procesado con las modificaciones realizadas. Codificar el audio consiste en colocarlo en un objeto digital, en una variable de un determinado número de bits, ubicado por dispersión de matriz o vector interno de datos, entre muchos otras opciones, y se le encapsula en un formato establecido por distintos protocolos de desarrolladores de audio con una extensión establecida (.Wav, .mp3, .AIFF, etc.). La calidad y el espacio de memoria que ocupe el audio resultante dependerá del tipo de codificación que se le asigne (Oppenheim, Willsky y Young, 1990).

Desde los años 60 la industria discográfica se ha utilizado el procesamiento digital de señales con el objetivo de mejorar la manipulación, tratamiento, y reproducción del audio. Sin embargo, no fue sino hasta mediados de la década de los 80, con el surgimiento del disco compacto “CD”, que se empezaron a comercializar dispositivos asequibles con procesadores digitales de señales integrados que fueron poco a poco desplazando a los mecanismos analógicos y sentando las bases para nuevos y complejos estándares del tratamiento de señales de audio (Pirkle, 2017b).

A partir de la década de los 90, se empezaron a comercializar módulos de audio de procesamiento digital, los cuales generaron la necesidad de que las compañías de audio contraten y capaciten a profesionales para que desarrollen habilidades de programación y encriptación avanzada. Inicialmente las compañías de audio contrataron personal con experiencia en diseño de equipos médicos, tecnologías de vigilancia o incluso desarrolladores del equipo que habían trabajado en la NASA (Tanev y Božinovski, 2014).

Con el pasar de los años, las distintas compañías de dispositivos de audio se vieron forzadas a desarrollar módulos digitales innovadores, que tuvieran un desempeño igual o incluso mejor que el de los equipos analógicos de la época.

De esta manera, el mercado experimentó una masificación, que dio lugar al desarrollo de la industria del desarrollo de software de audio, eliminando la necesidad de disponer de grandes infraestructuras y permitiendo generar estaciones de trabajo de audio digital de alto desempeño. (Archila, 2016).

Hoy en día existen algunas empresas alrededor del mundo que se dedican al desarrollo de software de audio, sin embargo, en Latinoamérica y en el Ecuador, esta industria es prácticamente nula a pesar de existir un amplio mercado de usuarios que consumen los productos de otros países y continentes. Esto muestra un gran potencial para el establecimiento de esta industria en el país y la región sin la existencia de ninguna competencia local.

1.2 Objetivos

1.2.1 Objetivo General

Desarrollar un módulo de audio digital de amplificación y efectos para señales monofónicas y estereofónicas, codificado en lenguaje de programación C++, a través del uso del marco de referencia de código abierto “JUCE”, para que pueda ser soportado por distintas estaciones de trabajo (DAW).

1.2.2 Objetivos específicos

Identificar el tipo de procesamiento más eficiente para el módulo a desarrollar, mediante la comparación de distintas maneras de implementación del algoritmo generado, para obtener una latencia menor a 7,5 milisegundos.

Generar una intuitiva interfaz de usuario a través de las plantillas prediseñadas en el marco de referencia “JUCE”, para que el uso del módulo de audio a desarrollar sea muy fácil y fluido.

Implementar un analizador de espectro en tiempo real, mediante el uso de transformadas de Fourier, para que el usuario tenga una idea más clara del contenido frecuencial de la señal que se esté manipulando.

1.3 Alcance

Este proyecto busca desarrollar un módulo de audio digital (*plugin*) que sea capaz de procesar señales de audio monofónicas y estereofónicas para poder amplificar o atenuar su amplitud y también para añadirle efectos como distorsión, ecualización, compresión, Delay y reverberación en tiempo real. Para este propósito se utilizarán técnicas de procesamiento conocidas, es decir no se buscará generar nuevas maneras de manipular las señales de audio.

El módulo no se basará en el funcionamiento de ningún equipo analógico de procesamiento de audio, ya que, no se dispone del tiempo necesario para realizar las mediciones y análisis correspondientes que llevarían al modelo de un dispositivo analógico de estas características.

El *plugin* será compatible con las principales estaciones de trabajo de audio digital que acepten las siguientes extensiones de procesadores de audio: AAX, VST, VST3 y Audio y Unit.

1.4 Justificación

Con el creciente desarrollo de la tecnología, se ha incrementado exponencialmente el uso de programación de algoritmos para realizar operaciones que antes eran ejecutadas mediante el uso de equipos analógicos, provocando que en la actualidad casi todos los sistemas se basen en sistemas digitales para su funcionamiento. El caso del audio no es la excepción: muchas de las legendarias compañías de dispositivos analógicos de audio han tenido que migrar a diversas plataformas de codificación para que sus sistemas incluyan procesos digitales generando una enorme industria de desarrollo de software de audio alrededor del mundo.

Sin embargo, en el Ecuador, esta industria es prácticamente nula, a pesar de que existir usuarios que consumen productos de este tipo desarrollados por empresas extranjeras. Esto genera un gran potencial para impulsar dicha industria de manera local ya que no existen competidores en el país. A pesar de ello, es importante considerar que, en el mercado global, si aparecen competidores muy fuertes y experimentados en el campo, lo cual implica que para poder posicionarse en la industria, el software desarrollado y los algoritmos implementados necesitan ser innovadores y de alta calidad, lo que a su vez conlleva altas capacidades de programación y codificación de distintos lenguajes de programación.

Adicionalmente a lo presentado anteriormente, el proyecto propuesto contribuirá a promover e impulsar el desarrollo de software de audio por estudiantes de la Universidad de las Américas. De esta manera fomentar la interrelación integral entre varias disciplinas para obtener un resultado de alta calidad y poder medirse con otros competidores a nivel mundial.

2 CAPÍTULO II: MARCO TEÓRICO

2.2 Señales digitales de audio

2.2.1 Conversión analógica digital

Desde hace aproximadamente 80 años, la necesidad de captar, cuantificar y procesar las variables de señales sonoras ha sido un tema muy complejo de investigación de varias ramas de la ingeniería como la eléctrica y acústica, y ha generado un gran avance tecnológico, dando como resultado la creación de muchos dispositivos construidos para este fin. A partir del aparecimiento de la digitalización de la información, se han intentado combinar estos sistemas para que sean capaces de actuar a manera de una interfaz entre el mundo analógico y digital. Sin embargo, para la captación de dichas señales siempre será necesario al menos un elemento analógico que generalmente es mediante sensores. Dentro del audio, estos sensores son llamados transductores, los cuales trabajan como conversores acústico- mecánico – eléctricos, y una vez con la señal en niveles eléctricos se procede a realizar un proceso llamado muestreo, que es el inicio de la conversión analógica digital (Oppenheim, Willsky y Young, 1990).

A finales de los años 20, el físico Harry Nyquist formuló el teorema del muestreo, siendo este demostrado por Claude Shannon 21 años más tarde. Este teorema es fundamental para la teoría de la información y establece que para que no se pierda información de una señal analógica periódica al momento de reconstruirla, se debe tomar muestras en un intervalo de tiempo al menos 2 veces mayor a la frecuencia máxima que contenga dicha señal. Entonces como el rango frecuencial de escucha humano es desde 20 Hz hasta 20000 Hz, el intervalo mínimo de tiempo para tomar muestras de una señal de audio de acuerdo con este teorema sería de 25 nanosegundos, lo cual corresponde a 40 kHz. Con el avance del audio digital se han creado distintos valores estandarizados que van desde 44100 kHz a 192 kHz, con los cuales se puede reconstruir con diferente calidad las señales muestreadas en su reconstrucción. El hecho de “tomar” una muestra consiste en obtener toda la información disponible de la señal cada vez que transcurra el intervalo de tiempo establecido, obteniendo el número de valores de la frecuencia de muestreo cada segundo. En la actualidad el muestreo de audio puede ser utilizado no solamente para reconstruir señales sonoras sino también para procesamiento de imágenes o registro de eventos de cualquier índole, como es el caso del desarrollo de métodos de muestreo disperso de audio por fotogramas (Ashraf et al, 2015).

Una vez muestreada la señal en función de lo antes mencionado, el siguiente paso es cuantificar los valores de dichas muestras para representar toda su información recopilada en estas. Este proceso dependerá de muchas variables como por ejemplo el número de bits de profundidad con el que se disponga para realizarlo o el tipo de cuantificación que se desee efectuar. A una mayor profundidad de bits se tendrá una mejor resolución de la señal, pero con un costo en el consumo de procesamiento, pero independientemente de esta profundidad. El primer bit generalmente se utiliza para representar el signo de la muestra, ya que corresponde a una señal de corriente alterna. Después, un segmento determinado de bits se encarga de los valores enteros de la señal y otro de los decimales o flotantes. Hay sistemas que permiten el uso de una coma flotante en donde se extiende el margen de precisión de la cuantificación de manera considerable. En estos, se procesan los valores de amplitud de una señal entre 0 y 1 y si existen sobrecargas de nivel les asigna el máximo valor posible. Con el avance en el desarrollo de procesos cuantificadores, hoy en día es posible minimizar o maximizar la linealidad o no linealidad con la que estos se ejecutan. Entre los más importantes tipos de cuantificación utilizados a nivel profesional se encuentra los de aproximaciones sucesivas, flash y delta sigma. Estos realizan operaciones matemáticas con la ayuda de chips de procesamiento que ayudan a optimizar el proceso de cuantificación (Adimulam et al, 2010).

El último paso en el proceso de conversión analógica digital de señales de audio es la codificación, la cual consiste en empaquetar la información del audio en un formato específico. Hasta la fecha existan algunos formatos de audio y se siguen creando más, pero en algunos casos, en esta etapa se requerirá realizar una especie de compresión de los datos para que el archivo del formato que los contenga ocupe un mayor o menor espacio de memoria digital según se requiera (Adimulam et al, 2010).

2.2.2 Corrección de errores

Como en todo sistema, al realizar la conversión analógica digital de una señal se pueden producir errores que produzcan que la reconstrucción digitalizada de dicha señal tenga información que no corresponda con la correspondiente en el dominio analógico. Cuando esto sucede, es necesario realizar una corrección de estos errores si es posible. Si se trabaja con señales que puedan volver a ser reproducidas, la mejor manera de corregir los posibles errores es volviendo a realizar la conversión y todos sus pasos nuevamente, sin embargo, si se trabaja con señal generadas y trabajadas en tiempo real, las opciones son limitadas y si es posible deben ser preventivas antes que correctivas (Oppenheim, Willsky y Young, 1990).

Entre los problemas principales se tiene al posible solapamiento de las frecuencias de las señales, haciendo que estas se vuelvan indistinguibles entre sí y produciendo que sea imposible la reconstrucción correcta de dicha señal muestreada. Otro problema puede ser el *Jitter* que es un error temporal en la toma de cada muestra que puede ser medido en término de la relación nivel señal ruido. Un tercer problema que puede presentarse es la pérdida de nivel a frecuencias altas, provocando que la amplitud de estas no coincida con sus correspondientes en el dominio analógico. Estas 3 problemáticas ocurren inevitablemente cuando no se respeta lo establecido por el teorema del muestreo, sin embargo, aunque es raro, también se pueden producir aun cuando se haya respetado dicho teorema. Para lo cual, existen varias soluciones posibles. En cuanto al solapamiento o la pérdida de nivel a frecuencias altas, suelen implementarse filtros para evitar que se den estos fenómenos o para compensar el efecto de los mismo. En cambio, para el problema del *Jitter*, lo más recomendado es aumentar la frecuencia de muestreo, pudiendo de esta manera reducirlo hasta ser prácticamente indetectable (Oppenheim, Willsky y Young, 1990).

Existen otros errores posibles producidos por causas indeterminadas en cuyos casos se deben realizar aproximaciones matemáticas de interpolación o regresión lineal con las muestras válidas que se tengan. Existen métodos de ajuste de audio que solucionan estas dificultades de manera efectiva utilizando modelos Gaussianos de correlación de coeficientes que han logrado resultados más cercanos a la realidad que los métodos convencionales (Sanjay, Yu y Yongjian, 2018).

Por último, el problema se puede presentar no en la toma de muestra de datos sino en la reconstrucción de estos y se puede evitar creando matrices de almacenamiento de datos, los cuales sean verificados al momento que dicha reconstrucción tome lugar. Un enfoque novedoso, se maneja a través de la detección compresiva, la cual permite que, desde la adquisición de la señal, esta se realice de manera eficiente aun trabajando por debajo de la frecuencia establecida por Nyquist manteniendo una prometedora y confiable recuperación de los datos (Guoxian y Lei, 2017).

2.3 Procesamiento digital de señales

Consiste en manipular parte o toda la data disponible de señales digitales para un fin específico, el cual puede ser aumentar su amplitud, modificar su frecuencia, entre muchas otras aplicaciones, que no solo se limitan al mundo del audio, sino también pueden implementarse en el tratamiento de imágenes y en general de toda data digital disponible. Si bien existe una innumerable cantidad de maneras de realizar procesamiento digital de señales y de optimizarlo, a continuación, se detallan las que tienen relación con el contexto de desarrollo de los módulos construidos en este proyecto con sus principales propiedades y ventajas de uso.

2.3.1 Respuesta al impulso

Como su nombre, lo indica se trata de la manera en la que un sistema responde a una excitación de tipo impulsiva. En el campo acústico, el sistema puede ser un recinto o edificación, en el mundo eléctrico, puede ser un circuito y en el mundo digital el sistema es el objeto de audio y toda su data. Esta respuesta al impulso puede describir prácticamente todas sus propiedades y características intrínsecas debido a que mediante la superposición de ondas se puede conseguir representar todas las señales y su comportamiento en el sistema efectivamente. Es por esta razón que sus aplicaciones son muy amplias, entre las más comunes está la implementación de filtros de varios tipos para uso en ecualizadores, efectos de eco o sistemas reverberantes (Oppenheim, Willsky y Young, 1990).

Dentro de los filtros posibles se suele utilizar dos tipos principales, los de respuesta finita (FIR) e infinita (IIR). En los de respuesta finita se tiene un número finito de valores con los que se trabaja en su entrada. La principal ventaja de estos filtros es que, en sus propiedades, existe simetría, permitiendo de esta manera que estos filtros puedan ser de fase lineal, pero esto puede significar un costo considerable en el consumo de recursos computacionales debido a que se obtienen partir de operaciones que implican el uso de la convolución, la cual consume bastante procesamiento. Su memoria requerida es finita y son muy utilizados en la representación de analizadores de espectro, siendo de 6 tipos para esta aplicación: rectangular, Barlet, Hann, Hamming, Blackman y Kaiser. Este tipo de filtros no funciona de manera recursiva. Cada una de estos tiene diferentes niveles de precisión en la resolución de la escala frecuencial y de la escala de nivel que puede representar, pero los más utilizadas por su adecuada resolución son los de Hamming y los de Hann (Srivatsan y Venkatesan, 2019).

Por su parte los filtros de respuesta infinita nunca regresan al reposo, es decir que tiene un número infinito de términos a utilizar, siendo esto posible, mediante la retroalimentación de la salida de muestras pasadas combinada con la entrada muestras actuales y anteriores, permitiendo, además, que estos filtros sean recursivos, es decir que funcionan de manera iterativa. De manera teórica este tipo de filtros requiere de memoria infinita para su funcionamiento, pero en la práctica, se utiliza memoria dinámica para un funcionamiento óptimo. Además, para dichos filtros (IIR), su obtención se puede realizar mediante ecuaciones de diferencias. Por lo tanto, estos filtros son más eficientes que los de respuesta finita. Existen dos formas comunes de implementar este tipo de filtros. La primera es la forma directa, en la cual se pueden realizar impulsos invariantes, transformadas bilineales y aproximaciones de derivadas, los cuales se realizan a partir de modelos analógicos. En cambio, para la forma indirecta, se pueden realizar operaciones de aproximación de Padé y de mínimos cuadrados. A pesar de que por definición estos filtros no tienen la capacidad de ser de fase lineal, pueden aplicarse de manera bidireccional para intentar simular este efecto (Widmark, 2018).

2.3.2 Función de transferencia

Se trata de una operación matemática que considera la relación entre la salida y la entrada de un sistema en el dominio de Laplace. Entonces es importante considerar la transformada de Laplace, es decir una integral que toma la variable real del tiempo en el dominio analógico o de muestras en el dominio digital y la transforma a la variable compleja que contiene los componentes frecuenciales de la señal. Generalmente se utiliza como una representación bidimensional que permite determinar lo que el sistema hace y poder identificar lo que el sistema realiza en cuanto a modificación de la señal tratada. Es muy útil también para reconocer eventos indeterminados en la información tratada para poder identificar sus características y cuantificarlas (Oppenheim, Willsky y Young, 1990).

Las funciones de transferencia tienen funcionamiento casi siempre correcto mayormente para los sistemas lineales invariantes en el tiempo, de los cuales las señales de audio forman parte. A diferencia del mundo analógico en donde esta función se puede ver afectada por las condiciones en las que se midan los datos de entrada y salida del sistema, En el mundo digital, en algunos casos, se pueden obtener resultados más reales ya que, se trabaja sobre las condiciones del sistema en sí y no de las mediciones de su entrada y salida. Sin embargo, esto solamente es posible si se toman en cuenta todos los parámetros, variables y criterios que afecten el comportamiento de dicho sistema, de otro modo los resultados a obtener no serán confiables, por esto, existen nuevos enfoques que utilizan diversos tipos modelos de expansión de más de dos dimensiones para considerar la mayor cantidad de variables para que los datos obtenidos sea lo más cercano posible a la realidad (Samarasinghe, Abhayapala, Poletti y Betlehem, 2015).

2.3.3 Transformada discreta de Fourier

Es 1822, el científico francés Joseph Fourier, determinó que cualquier señal determinada en el dominio temporal, puede ser divida en las frecuencias que la componen a través de series trigonométricas. En sí la transformada de Fourier es una operación matemática relativamente compleja que es de gran ayuda en el mundo del audio ya que permite trabajar en el dominio del tiempo y de la frecuencia y modificar variables como la amplitud, la fase y muchos otros en ambos dominios (Oppenheim, Willsky y Young, 1990).

De manera teórica consiste en una integral infinita de la ecuación característica de la señal en función del tiempo por un término de conversión hacia el dominio de la frecuencia. Sin embargo, al trabajar en el mundo de la data digital, la variable temporal se convierte en el número de muestras disponibles y ya no puede tratarse más de una sumatoria infinita sino de una sumatoria limitada de datos que da como resultado una secuencia de valores complejos (Torchinsky, 2011).

A esta se le llama transformada discreta de Fourier y se calcula según la siguiente ecuación:

$$X_f = \sum_{n=0}^{N-1} X_n * \; e^{-\frac{2\pi}{N}*f*n} \qquad (1)$$

Ecuación 1: Transformada de *Fourier*

Donde:

n: índice de la muestra que se está analizando.

N: Número total de muestras tomadas en cuenta para realizar la operación.

f: Valor en frecuencia.

Xn: Amplitud de la muestra actual.

Xn: Amplitud de la frecuencia f.

Es muy común que se confundan el término de transformada discreta con el de transformada rápida (*FFT*), la cual consiste en distintas maneras de implementar algoritmos que realicen la operación en un computador o procesador de manera eficiente. Existen muchas formas conocidas de realizar las FFT que vienen por defecto en muchos de los programas de procesamiento de señales y con el avance de la tecnología continúan surgiendo nuevas maneras de realizar estos procesos de manera aún más efectiva (Jiang y Xu, 2014).

2.3.4 Ecualización

Es uno de los efectos más comúnmente usados en el mundo del audio tanto en el dominio analógico como digital. Es muy útil debido a que permite modificar el nivel de las bandas de frecuencia de una señal de audio según se requiera. La ecualización en sí se puede definir como la implementación de uno o más tipos de filtros a una señal sonora (Pirkle, 2017a).

Estos filtros pueden ser: de paso alto o bajo, *shelf* o *shelving*, *peak*, entre muchos otros que tienen distintas características. Los filtros de paso sirven para deshacerse del contenido frecuencial que no se desee tener en la señal de audio y pueden tener un orden determinado. Este orden determina la caída de la amplitud de las frecuencias que quieren atenuar a partir de la frecuencia de corte. Esta caída aumenta 6 decibelios por banda de octava al aumentar el orden. También se puede trabajar con una variable llamada factor de calidad (Q) que, en el caso de los filtros de paso, causará una resonancia de entre -6 y +6 dB en la frecuencia de corte según el valor que este tenga. Los Filtros tipo *shelf* o *shelving* aplican ganancia positiva o negativa a partir de una frecuencia de corte, de manera similar que los filtros de paso, ya que también pueden ser de corte frecuencial bajo o alto, pero con la diferencia de que en estos no existe una caída de la amplitud de las frecuencias dependiendo de un orden de los filtros, sino que la misma ganancia es aplicada a todas las frecuencias que a la de corte. La acción del factor de calidad en estos filtros es a manera de suavizado de la aplicación de la ganancia en la frecuencia de corte (Sabin y Pardo, 2009).

Los filtros de tipo *peak*, en cambio, actúan solamente en una banda de frecuencia y se determinan por tres parámetros: la frecuencia central de la banda, el factor de calidad, que a su vez establece el ancho de la banda seleccionada y la ganancia en sí que se aplica en el filtro, la cual puede ser positiva o negativa. Algunos autores dividen los filtros con ganancia positiva de los de ganancia negativa, o por las diferencias en la selección y aplicación de la banda de frecuencia, pero todos estos son muy similares y pueden estar considerados dentro de la misma categoría. Considerando este último tipo de filtro, un ecualizador puede dividirse en categorías principales: paramétricos, semi paramétricos y gráficos. Los paramétricos permiten el control tanto de la frecuencia central de la banda ecualizada, del factor de calidad, y por ende del ancho de banda de esta y de la ganancia. Los semi paramétricos tienen fijo el factor de calidad y los gráficos solo permiten que se modifique la ganancia con sus otros dos parámetros fijos. En el mundo actual, existen muchos módulos de procesamiento de audio digital que han sido modelados en función de legendarios dispositivos de hardware analógico con resultados con una similitud relativamente alta a los equipos originales (Hodgson, 2010).

2.3.5 Compresión

Al igual que el efecto de ecualización, este también es uno de los más utilizados en el procesamiento de señales y su principal propósito es modificar el rango dinámico de una señal de audio. Su funcionamiento involucra una sección de detección como una de modificación de la señal y entre estas dos se aplica una etapa intermedia que implica la velocidad en la que se implementa o deja de implementar el efecto (Pirkle, 2017a).

La primera sección en actuar en este efecto es la de detección, la cual puede constar de una o dos variables: el umbral de nivel y el suavizado o *knee* de este. El primer parámetro, consiste en un valor de nivel, generalmente en decibelios de escala completa para módulos de procesamiento digital, a partir del cual, si la señal de entrada supera el nivel, será comprimida de acuerdo con otros parámetros que se mencionan más adelante. Existe la posibilidad de implementar un parámetro de suavizado conocido como “rodilla” o *Knee*, el cual provoca que la compresión se realice al superar un nivel un poco inferior al del umbral de manera atenuada. Existen módulos en los que este parámetro es fijo o toma un valor de 0, en cuyo caso solamente se considera el umbral para implementar la compresión (Hamada, 2012).

Una vez detectada la señal se aplica una especie de envolvente de dos parámetros: el tiempo de ataque y el de relajación. En dispositivos analógicos generalmente el tiempo de ataque se refiere al lapso que transcurre entre el momento en el que se detecta un nivel de señal superior al del umbral establecido y el momento en el que se ha aplicado el 90% de la compresión final a dicha señal. En otros dispositivos el porcentaje usado es de 60%, pero en la mayoría de los módulos digitales o *plugins* este dependerá enteramente del fabricante. El tiempo de relajación funciona de manera similar, pero para dejar de comprimir, es decir, si el efecto está siendo implementado sobre la señal y se detecta un nivel de entrada menor al umbral seleccionado, este parámetro determinará cuanto se demora la señal en recuperar su nivel de acuerdo con los mismos porcentajes aplicados para el tiempo de ataque. En cuanto a la implementación de la compresión en sí, se efectúa de acuerdo con la variable de relación de compresión. Entonces, si la relación de compresión tiene un valor de 2 a 1, se refiere a que por cada 2 dB que la señal de entrada supere el umbral establecido, pasará solamente 1 hacia la señal de salida. Existen compresores que incluyen una variable de ganancia de compensación para elevar o bajar de la señal cuando ya ha sido comprimida (Oppenheim, Willsky y Young, 1990).

En el mundo analógico existen diferentes tipos de compresores que actúan a diferentes velocidades de acuerdo con sus elementos de construcción y principios de funcionamiento, los cuales pueden ser modelados en el mundo digital con gran fidelidad y calidad. Existen otros que incluyen una función llamada “mirar hacia el futuro” que detectan el comportamiento futuro de la señal de audio y según eso ajustan sus parámetros de funcionamiento, sin embargo, estos solamente son posibles en el dominio digital y con señales de audio que ya han sido grabadas, es decir no funcionan en tiempo real. Por último, en los últimos años han ganado gran popularidad los compresores multibanda y los ecualizadores dinámicos, que de cierta manera combinan las funcionalidades de estos dos efectos con resultados muy impresionantes (Hodgson, 2010).

2.3.6 *Delay* o eco

Es un efecto de audio en el que se aplica utilizando un retraso o retardo de un tiempo determinado de la señal de entrada y lo implementa en la señal de salida. Antes de que este efecto se pudiera implementar en dispositivos analógicos, lo que se hacía era reproducir la señales a las que se quería aplicar el retardo en espacios reverberantes lo suficientemente grandes para que se produzcan ecos de acuerdo con sus propiedades acústicas. Sin embargo, como era de esperarse, era muy difícil adecuar dichos recintos para los ecos generados cumplan con los tiempos requeridos, por lo tanto, se empezaron a desarrollar dispositivos de implementación de este efecto. En primera instancia estos dispositivos fueron desarrollados con funcionamiento a base de cinta con un mecanismo para ajustar el tiempo aplicado de retardo en la señal. Después, con el avance de la electrónica, se fueron utilizando otros elementos para conseguir la implementación del efecto (Oppenheim, Willsky y Young, 1990).

En general existen muchas maneras de construir un dispositivo analógico del efecto de *Delay*, y en casi todos los casos, su implementación requiere de complejos circuitos de capacitores, resistencias y muchos otros elementos. Sin embargo, en el dominio digital, la implementación de este es mucho más sencilla. Como en este dominio se trabaja con muestras de audio, el tiempo de *Delay* dependerá de la frecuencia de muestreo con la que la señal haya sido transformada desde su correspondencia analógica. Con el problema de la determinación del tiempo de retardo resuelto, quedan otros dos parámetros que configurar, los cuales son la retroalimentación y la cantidad de efecto para implementar en la señal original denominado mezcla. El parámetro de retroalimentación consiste simplemente en la amplitud que toma el valor retrasado de la señal, generalmente se utiliza una escala porcentual para trabajar de manera más fácil. En el caso del parámetro de mezcla solo se trata de cuanta señal procesada con el efecto va a salir y cuanta señal original, de igual manera en este también se suele aplicar una escala porcentual. Muchas veces se acostumbra a trabajar con buffers de procesamiento en paralelo para manipular la señal procesada por separado de la original. En algunos casos se aplica otros que son la velocidad y profundidad de modulación que añaden cierta espacialidad y modificación de tono a la señal procesada (Pirkle, 2017 a).

2.3.7 Reverberación

Este es un fenómeno acústico real presente en todos los recintos acústicos que se produce por las reflexiones en las superficies de estos, las cuales dependen de muchos parámetros acústicos como la absorción, la difracción, el volumen, entre otras. Al igual que en los inicios del *Delay*, para este efecto también se utilizaba recintos reverberantes con propiedades específicas. Poco a poco se empezaron a desarrollar dispositivos muy complejos que podían recrear espacialidad de un recinto en específico. Creándose de esta manera, diversos tipos de reverberaciones, cada una de ellas con funcionamiento base similar, pero con características sonoras distintas. (Goussios, Tsinikas y Kitsiou, 2015).

Son muchas las características que pueden definir el comportamiento de la reverberación en una señal sonora y dentro de los módulos digitales que la aplican existe una gran variedad en la que se puede efectuar, causando que en ocasiones dos *plugins* de este efecto no son comparables entre sí de manera cuantificable. Sin embargo, de manera general se pueden definir ciertos parámetros básicos con los que muchas veces se trabaja. Uno de esos parámetros viene a ser el pre-retardo o pre *Delay*¸ que simplemente indica un lapso que va a transcurrir antes de aplicar el efecto en cuestión a la señal original. Otro parámetro fundamental es el tiempo de reverberación, que establece la definición acústica de este fenómeno, es decir el tiempo que le toma a la señal decaer 60 dB de su amplitud en estado estacionario cuando se apaga la fuente. A pesar de que este tiempo es incluido en muchos módulos de implementación del efecto, existen otros que no lo manejan de manera temporal sino a través de un parámetro que usualmente se refiere al tamaño del recinto hipotético en el que la señal es reverberada. Otro criterio que suele considerarse en ocasiones es el amortiguamiento y la difracción, que de cierta manera modifican la respuesta frecuencial en la implementación del efecto. Una de las formas más comunes de aplicar la reverberación es mediante la respuesta al impulso, utilizado lo mencionado anteriormente, pudiendo de esta manera simular espacios de lugares muy afamados digitalmente (Oppenheim, Willsky y Young, 1990).

2.4 Lenguaje de programación C++

2.4.1 Concepto y principios

Los lenguajes de programación tienen reglas y principios formalmente establecidos que permiten codificar instrucciones y algoritmos para que una máquina realice una acción. De manera general, pueden ser compilados o directamente lenguajes de máquina. De acuerdo con cuan específico sea un lenguaje de programación en relación de la arquitectura computacional perteneciente al sistema con el que se esté trabajando, se pueden clasificar en de bajo y alto nivel. A pesar de lo que se puede llegar a pensar en un inicio, los lenguajes de bajo nivel son más complejos y pueden llegar a ser diferentes según la estructura del sistema, por el contrario, los de alto nivel suelen ser lenguajes de compilación más sencillos que son interpretados a lenguaje de máquina por un compilador, sin tener cambios significativos según la arquitectura del sistema (Goytia, 2014).

En el caso de C++, se trata de un lenguaje diseñado por el científico de la computación Bjarne Stroustrup como una extensión del lenguaje c, orientada a objetos, heredando su sintaxis de este. Sin embargo, logró conservar sus propiedades correspondientes a la programación estructural convirtiéndose en un lenguaje multiparadigma y hoy es considerado como un híbrido. Es muy utilizado en el desarrollo de aplicaciones de escritorio, así como en la programación de microcontroladores. Algunos de los software creados y desarrollados en el lenguaje en cuestión: *Adobe Inc.*, *Google Chrome, uTorrent*, entre otros. Además, la mayor cantidad de software de audio es desarrollado en C++. (Goytia, 2014).

Una de las principales ventajas de C++ es que permite la agrupación de instrucciones en clases, las cuales pueden ser implementadas mediante la creación de objetos. Además, es bastante didáctico, portable y compatible con prácticamente todos los sistemas operativos y plataformas. Incluso permite separar un mismo proyecto o programa en varios módulos para compilarlos de manera independiente y realizar un análisis más exhaustivo sobre su desarrollo (Parga, 2014).

2.4.2 Tipos de datos

Los datos que se manejan en C++ pueden ser: caracteres, enteros, decimales o de coma flotante y booleanos. Los caracteres por defecto utilizan un espacio de 8 bits y se utilizan para información de texto, sin embargo, en casos en los que se requiera de un mayor almacenamiento como al utilizar símbolos Unicode, se les puede asignar hasta 32 bits. Para los enteros se puede asignar entre 16 para variables cortas y 32 bits para variables largas o *int*. En cambio, para los números decimales se puede asignar un espacio de 32 bits para variables *float* y 64 bits para variables *double*. Con estos tamaños se puede almacenar datos con una precisión muy alta y manejarlos de manera fluida en este lenguaje (Schildt, 2009).

2.4.3 Declaración e implementación de proyectos

Para una mayor efectividad, como ya se mencionó antes, C++, permite separar un programa en módulos conocidos como *Threads*, los cuales pueden ser compilados de manera individual. Esto también faculta separar el procesamiento de cada uno de estos módulos e inclusive procesarlos desde distintas partes del procesador, dando prioridad a ciertas partes del código que necesiten ser ejecutadas sin interrupciones o con la menor cantidad posible de estas. Es posible realizar esto, mediante archivos de declaración de variables y clases y archivos de implementación. Estas posibilidades ayudan mucho a optimizar el desarrollo de un software o programa y también a mejorar la detección de errores o fallos durante sus procesos (Parga, 2014).

2.5 JUCE

2.5.1 Origen y funcionamiento

Es un marco de referencia de código abierto, multiplataforma, para lenguaje C++, que está optimizado para desarrollo de software de audio. Fue creado en 2004 por el desarrollador Julian Storer y sus siglas responden a "*Jules Utility Class Extension*" o Extensión de clases de utilidad de Jules, lo cual muestra de manera general en que consiste dicho marco de referencia. Fue creado para facilitar el desarrollo de software de procesamiento de señales de audio y funciona a través de una herramienta llamada *Projucer* que ayuda a gestionar y guiar en la creación y desarrollo del tipo de software que se desee construir. En general con la ayuda de JUCE se podría crear cualquier tipo de aplicación que no solamente esté enfocada al audio, pero es el fin para el que más se lo utiliza. Existen un gran número de compañías que utilizan este marco de referencia para el desarrollo de sus productos y programas entre las cuales están: *Cycling 74´, Korg, M-Audio, Tracktion, PreSonus* y muchos otros más. El caso de la compañía *PreSonus* es muy interesante ya que esta ha desarrollado la estación de trabajo de audio digital (*DAW*) *Studio One*. Lo que muestra que JUCE es un marco de referencia de nivel profesional y que con su ayuda se pueden crear programas muy sofisticados (Butcher, 2014).

Para utilizar *Projucer* por primera, este debe compilarse para que todas sus funcionalidades trabajen correctamente. Es compatible con la mayoría de los entornos de desarrollo de todos los sistemas operativos, desde los de código abierto como *Code Blocks* y *Xcode,* hasta los de paga como Visual Studio Enterprise. Una vez compilado, este trabaja a manera de aplicación ejecutable en donde se puede seleccionar el tipo de proyecto que se requiera desarrollar, accediendo a guías, recomendaciones, tutoriales, e incluso a una plataforma de comunidad donde se pueden resolver muchas dudas que se tenga sobre el funcionamiento de JUCE y del desarrollo de software en general. Si bien está diseñado para trabajar con C++, es compatible con librerías de interpretación de otros lenguajes como *Python* y *Java* (Butcher, 2014).

2.5.2 Ventajas

La principal y mayor ventaja de JUCE, es que se encarga de la mayor parte de programación de bajo nivel, permitiendo que una persona con conocimientos intermedios en programación pueda crear proyectos y desarrollar softwares profesionales de audio, enfocándose en los procesos relacionados con la calidad del audio más que con los que tengan que ver con la programación. En comparación con otros marcos de referencia que se utilizan para aplicaciones de audio como RackAFX, JUCE tiene un sistema más amigable con el usuario y permite que todos los proyectos del tipo que sean se puedan crear desde una misma aplicación (Butcher, 2014).

Aunque *Projucer* no es realmente un entorno de desarrollo integrado, incluye un motor de compilación para aplicaciones sencillas que puede ser muy útil en el diseño de interfaces de usuario, ya que, al activar este motor los cambios realizados en el código que se realice se ven reflejados en tiempo real en su implementación sin necesidad de estar compilando a cada momento. Una ventaja adicional que presenta JUCE es que los archivos de todas sus clases y algoritmos están disponibles de manera completamente libre y son editables para poder personalizarlos e incluso mejorarlos, sin embargo, esto no es recomendable si no se tiene mucha experiencia, debido a que se puede dejar sin funcionamiento o con uno erróneo a algunas clases importantes. Si se realizan este tipo de modificaciones, JUCE recomienda volver a compilar la aplicación *Projucer* para evitar cualquier tipo de problemas. Además de ser un marco de referencia de código abierto, también se puede acceder a versiones pagadas que incluyen soporte técnico y asesoramiento según se requiera (Butcher, 2014).

2.5.3 Audio *plugins* en *JUCE*

Una ayuda muy grande que proporciona *JUCE* es el desarrollo de *plugins* de audio, para lo cual tiene una estructura establecida permitiendo construir versiones en los principales formatos disponibles en el mercado de la industria de software de audio digital. Por defecto se generan 4 archivos de codificación al crear un *plugin* con *JUCE*. Dos de estos archivos corresponden a la declaración de variables, funciones y objetos del procesador y su implementación, y los otros dos se refieren a la interfaz de usuario. Con esto se puede seleccionar que el procesamiento para cada sección se realice en distintas partes del procesador del sistema en el que se vaya a ejecutar el software (Butcher, 2014).

En la sección del procesador por defecto vienen las funciones básicas como el constructor y destructor del *plugin*, la configuración de canales, la creación de la interfaz y de la instancia del software, pero principalmente se crean dos funciones principales en las cuales se da lugar a todas las acciones a ejecutarse. La primera función se llama *preparetoPlay* y es en donde se configuran los datos de frecuencia de muestreo y se asignan los valores de las variables a los parámetros de audio en cuanto se crea una instancia del *plugin*. Al ejecutar un módulo de audio, aunque se esté reproduciendo una señal que vaya a atravesar a este, al principio hay un pequeño momento de autoconfiguración que es donde *preparetoPlay* se ejecuta. La segunda función denominada *Process Block* o bloque de procesamiento es donde realmente se realiza todo el procesamiento de señales, sea de ecualización, ganancia, compresión, corrección de tono, etc. Esta función se va ejecutando cada vez que entra un nuevo bloque de datos cuyo tamaño está determinado por el tamaño seleccionado del buffer de procesamiento. Esto es lo que permite que el *plugin* funcione en tiempo real (Butcher, 2014).

En cuanto a la interfaz de audio, por defecto también implementa un constructor y un destructor y varias otras funciones en donde se puede configurar los colores, palabras y demás elementos visuales que pertenecen a esta. A diferencia del procesador, la interfaz no se dibuja continuamente cada vez que entra un nuevo paquete de datos de información de audio, a menos que se la programe de esta manera, sino que esta, por defecto solo se actualiza, cuando existe un cambio en uno de los elementos que contenga, optimizando una considerable cantidad de recursos computacionales y dando prioridad a la sección del procesador (Butcher, 2014).

2.5.4 Futuro

El 21 de noviembre de 2018, Jules Storer anunció el lanzamiento de un nuevo lenguaje e infraestructura de desarrollo de software de audio llamada *SOUL* (*Sound Library*), el cual, aseguró que será mucho más óptimo que C++ ya que permitirá acceder a los chips de procesamiento digital de señales que prácticamente todas las máquinas actuales poseen pero que extrañamente no son utilizadas en la mayoría de software de audio desarrollados para este fin. Todavía no se ha publicado una versión oficial estable de este nuevo lenguaje, pero se espera que suceda en el transcurso del año 2020.

2.6 Módulos de procesamiento de audio digital (*Plugins)*

2.6.1 Definición y funcionamiento

Se trata de programas desarrollados para realizar modificaciones a la información de audio digital, generalmente en tiempo real. Se les asignó el término *plugin*, debido a que en sus principios se crearon como complementos de otros programas sin poder funcionar por si solos, sin embargo, en la actualidad, existen versiones de *plugins* que pueden trabajar a modo de aplicaciones ejecutables, constituyéndose así en piezas completas de software. Cuando no pueden funcionar de manera aislada, lo hacen a manera de complementos en otros programas denominados estaciones de trabajo de audio digital (*DAW*), los cuales hacen las veces de anfitriones de estos *plugins* entregándoles la información no solamente del audio, sino del proyecto en el que se esté trabajando. Dentro de esta información puede estar la frecuencia de muestreo del proyecto, el tempo de la sesión de trabajo, la profundidad de bits, el tamaño del buffer de procesamiento, entre otras. Al trabajar como complementos, pueden crearse muchas instancias de un mismo *plugin* sin que esto cause problemas, siempre y cuando este haya sido programado para responder de manera estable aun en casos extremos (Silva, Mattos y Junior, 2019).

Existen tres categorías principales de clasificación de plugins. Los primeros son los que transforman las señales que reciben a su entrada, también se tiene los que pueden generar señales sonoras a través de los distintos tipos conocidos de la síntesis y por último se tiene a los que no realizan ninguna modificación en la señal de entrada, pero pueden ejecutar análisis y cálculos de varios tipos con la información de esta. El funcionamiento de los que si realizan una transformación de la señal recibida funcionan obteniendo los datos de las muestras del audio en su entrada que son entregados por el software anfitrión, después se realizan todas las acciones que se hayan considerado en sus algoritmos durante la etapa de codificación y desarrollo, modificando la información que han recibido. Por último, sale una señal que ha sido digitalmente procesada (Pirkle, 2017 b).

2.6.2 Principales extensiones o formatos y compatibilidad

Existen más de 10 formatos de módulos de audio digital, los cuales han sido diseñados o adoptados por los softwares utilizados como anfitriones. En teoría un *plugin* diseñado y desarrollado correctamente debería funcionar de manera casi idéntica independientemente del formato en el cual se lo compile, pero en la práctica esto no suele ser así. Debido a la arquitectura de cada uno de los formatos disponibles, así como a la compatibilidad de estos con sus anfitriones pueden existir casos en los que se tenga un mejor o peor funcionamiento (Pirkle, 2017 b).

En este proyecto se han considerado tres formatos para desarrollar módulos de audio digital, los cuales son: VST, VST3, Audio Unit (MacOS). En el caso de los *plugins* VST y VST3, responden las siglas de *Virtual Studio Technology*. Fue desarrollado por la compañía Alemana *Steinberg*, también creadora de la *DAW* Cubase. Este formato de *plugin* funciona a manera de interfaz de procesador para conectar los correspondientes *plugins* con las distintas otras herramientas de la *DAW*. La primera versión de VST fue publicada en 1999 y a partir de ella nuevas han sido desarrolladas, siendo la que hoy se conoce con VST, realmente la versión VST 2.4, que ya es una versión de legado. Su arquitectura es bastante estable, siendo una de las más simples, sin embargo, en esta versión ya se puede alcanzar una precisión de hasta 64 bit. Dentro de las limitaciones de VST 2.4 se tiene que la precisión de los parámetros en cuanto a la actualización de sus valores puede tener errores porque no se realiza muestra a muestra y la configuración de canales es limitada. De acuerdo con la compañía *Steinberg,* este formato de *plugin* ya es obsoleto, sin embargo, muchas personas lo siguen utilizando debido a su gran compatibilidad con muchos softwares de audio, especialmente con los más antiguos (Tanev y Božinovski, 2014).

En el caso de VST3, se trata de una versión mucho más estable y compleja que la anterior, pero más que nada su seguridad de ejecución es mayor. Funciona de una manera en la que la información viaja en una sola dirección en cada evento, ya sea del anfitrión al *plugin* o viceversa, consiguiendo que no se pierda información o se generen interrupciones por intentar recibir y solicitar información al mismo tiempo. Sin embargo, tiene la capacidad de guardar la información de los eventos que se dan simultáneamente para que la actualización de los valores de los parámetros sea mucho más precisa. La configuración de canales es extensa y permite trabajar con buses o canales externos y en formatos de audio envolvente. A partir de su creación en 2008, se han ido corrigiendo ciertas fallas y limitaciones que tenía siendo hoy una de las versiones de *plugins* más estable. Para desarrollar estos dos tipos de *plugins*, *Steinberg* ha publicado de manera libre las herramientas de desarrollo de software (SDK) para estos formatos. Las cuales constan de más de 70 000 líneas de código que ayudan a configurar las características estandarizadas de dichos formatos de *plugin* (Pirkle, 2017 a).

En el caso de la versión en Audio Unit, la cual solo funciona para el sistema operativo *MacOS,* se tiene muchas de las mismas características que en VST3, pero con la ventaja de que se pueden crear buses dinámicamente, haciendo que la configuración de canales sea ilimitada. En general los tres formatos de *plugins* descritos dan muestra de un gran trabajo de desarrollo por detrás, pero hay que considerar que tienen diferencias que pueden causar un posible mejor funcionamiento en uno u otro formato (Pirkle, 2017 b).

3 CAPÍTULO III: METODOLOGÍA

3.2 Consideraciones generales de diseño

Dentro del marco de referencia utilizado (*JUCE*), existen ciertas plantillas, objetos y clases creadas por defecto que fueron utilizadas en el desarrollo del módulo, especialmente en la parte del diseño de la interfaz de usuario. La idea en principio fue desarrollar un solo módulo con el cual se procesarían señales digitales de audio, aplicando cuatro de los efectos más utilizados dentro del mundo de la mezcla de audio, los cuales son: ecualización, compresión, retardo o Delay y reverberación. Durante el desarrollo del proyecto se decidió separar al módulo en *plugins* individuales por razones que se explicarán posteriormente en este documento. En general se puede definir la metodología implementada en el desarrollo de este proyecto como cuantitativa, debido a que todas las variables utilizadas son cuantificables numéricamente.

Como cada efecto finalmente fue separado en un módulo individual, algunos parámetros como los de ganancia y polaridad tuvieron que ser aplicados de manera similar o idéntica a cada uno de ellos. La estructura de cada efecto, tanto para señales monofónicas como estereofónicas fue la siguiente: la señal entra al módulo y el primer proceso que se aplica es la inversión de polaridad en caso de que esta esté activada, posteriormente se aplica la ganancia de entrada y luego la señal es procesada con el efecto correspondiente (Ecualización, Compresión, *Delay* o Reverberación). Finalmente se aplica una ganancia de salida como último proceso. El flujo de la señal implementada en cada *plugin* desarrollado se puede entender de manera más clara según el siguiente diagrama:

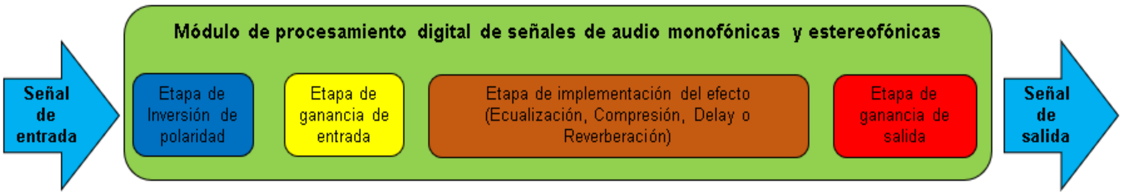


Figura 1. Diagrama de flujo de los módulos desarrollados.

Los *pulgins* fueron diseñados para señales monofónicas y estereofónicas, para los sistemas operativos *Windows* y *MacOS*. La compatibilidad de los sistemas operativos fue posible, en primer lugar, por el uso del lenguaje de programación *C*++, que es en general utilizado para desarrollar programas y aplicaciones multiplataforma. Además, *JUCE* permite evitar posibles errores de compatibilidad al migrar los códigos entre los diferentes sistemas operativos. Por estas razones, no fue necesario realizar codificaciones por separado, sino que los mismos algoritmos desarrollados fueron compilados en los diferentes sistemas operativos utilizados.

El proceso general de diseño de cada uno de los módulos de audio fue de la siguiente manera: se escogió la creación de un proyecto de tipo “*Plugin* de audio” en el software de *JUCE*, posteriormente se seleccionó las plataformas de compatibilidad del módulo. Es importante mencionar que el marco de referencia utilizado permite también desarrollar software de audio para otros sistemas operativos como *Linux*, *Android*, *IOS*, entre otros. Sin embargo, se decidió desarrollar los módulos descritos en este documento solamente para *Windows* y *MacOS* debido a que la mayoría de las estaciones de trabajo de audio digital (*DAWs*), funcionan principalmente en dichos sistemas.

Cada *plugin* fue desarrollado con ”*threads*” o “hilos” separados para el procesamiento del *plugin* y la interfaz de este. La razón de esta separación fue para dividir el trabajo computacional del procesador de audio del módulo y de la interfaz de usuario. Todo el procesamiento digital de señales (*DSP*) de cada módulo desarrollado se dirigió a la unidad central de procesamiento del ordenador (*CPU*) y el de la interfaz gráfica se dirigió a la unidad de procesamiento gráfico (*GPU*). Estas divisiones de cada parte de los *plugins*, ayuda a dar prioridad al procesador de audio sobre la interfaz para que, en caso de errores por sobrecarga de procesamiento o interrupciones de origen desconocido, la primera parte en fallar sea la interfaz y no el audio, siendo este, el último en fallar, únicamente frente a un colapso total del sistema que lo esté ejecutando.

Este tipo de ordenamientos del flujo del procesamiento de las señales de los módulos y sus componentes, así como de los sistemas que los ejecuten, vienen preparadas por defecto o disponibles con simples configuraciones al usar *JUCE*, lo cual permite enfocar el desarrollo hacia la calidad y optimización del procesamiento del audio. Cada módulo fue compilado en los formatos *VST* y *VST3* (*Virtual Studio Technology*) tanto para *Windows* como para *MacOS*. Además, para el sistema de Apple también fueron compilados en el formato *Audio Unit* (*AU*). Para esto fue necesario obtener las herramientas de desarrollo de software o *SDKs* (*software development kit*) de cada formato de los módulos. En el caso de las versiones en *VST* y *VST3*, al ser estos formatos creados por la compañía *Steinberg*, los *SDKs* fueron descargados del sitio web oficial de esta empresa. En el caso de la versión *Audio Unit*, no fue necesario descargar ningún *SDK*, debido a que todos los ordenadores con el sistema operativo *MacOS* tienen esta herramienta incluida por defecto. Para las versiones de *MacOS*, los módulos fueron compilados en el entorno de desarrollo integrado (*IDE*) *Xcode*, versión 12, en un ordenador *IMac* 2015 *Intel Core* I5 2.8 GHz, memoria RAM de 8 GB 1867 MHz DDR3. Asimismo, para *Windows*, estos *plugins* fueron compilados en el *IDE Visual Studio Enterprise 2017*, en un computador con procesador *Intel Core I7 4510U CPU 2.00 GHz – 2,60 GHz*, memoria *RAM* de 8,00 *GB*, 64 *bits*, con sistema operativo *Windows* 10 *Pro*-2019. Al iniciar el proyecto, se comenzó utilizando la versión 5.4.3 del marco de referencia, sin embargo, durante el desarrollo de este, se publicaron dos versiones nuevas y mejoradas, por lo que al final se terminó utilizando la versión 5.4.5. A continuación, se presenta el desarrollo y los diseños de cada una de las etapas de los módulos desarrollados.

3.3 Diseño de la etapa de inversión de polaridad

Esta es la primera etapa de procesamiento en cada plugin desarrollado. En sí, la inversión de polaridad de una señal digital de audio consiste en multiplicar cualquiera que sea su amplitud en una escala lineal (no logarítmica) por -1. Es muy importante recalcar que esta etapa no consiste en modificar la fase (tiempo de retardo) de la señal procesada sino invertirla para que todos los valores que correspondían a valores de amplitud negativa en el dominio analógico (de corriente alterna), correspondan ahora a valores positivos y viceversa. La razón por la que esta fue la primera etapa implementada en los módulos es debido a que con el procesamiento de las etapas siguientes de aplicación de ganancia y del efecto de cada *plugin*, se pueden producir desfases, y, por lo tanto, realizar una inversión de polaridad posterior a estas otras etapas resultaría en invertir la señal una vez que ya ha sido desfasada, produciendo potenciales problemas de sincronización temporal de las señales procesadas. En esta etapa se utilizaron parámetros booleanos (verdadero/falso) para controlar la aplicación de la inversión de polaridad. A continuación, se presenta un ejemplo de la parte del algoritmo de declaración e implementación de esta etapa:

```
auto PhaseLMode = std::make_unique<AudioParameterBool>("phaselmode", "Polarity", false);
PolarityL = *ReverbParams.getRawParameterValue("phaselmode") ? -1.0f : 1.0f;
```

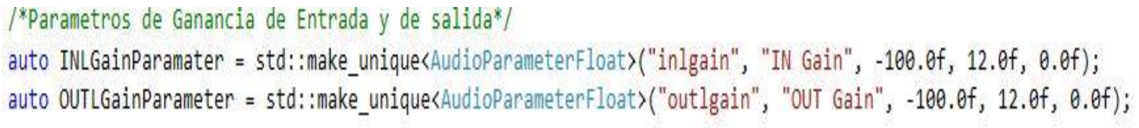
Figura 2. Declaración e implementación de variable de inversión de polaridad en el módulo de reverberación para señales monofónicas.

En la figura 2 mostrada, se tiene, en la primera línea, la declaración de la variable booleana, en la cual se coloca el identificador de la variable, su nombre y su estado por defecto. En la segunda línea, se muestra la implementación de la inversión o no inversión de polaridad si la variable tiene valor verdadero o falso, al multiplicar la señal por -1 o por 1, respectivamente. Como se puede notar, de acuerdo con lo mostrado, esta etapa es muy sencilla de implementar y fue aplicada en todos los *plugins* desarrollados de manera idéntica.

3.4 Diseño de las etapas de ganancia de entrada y salida

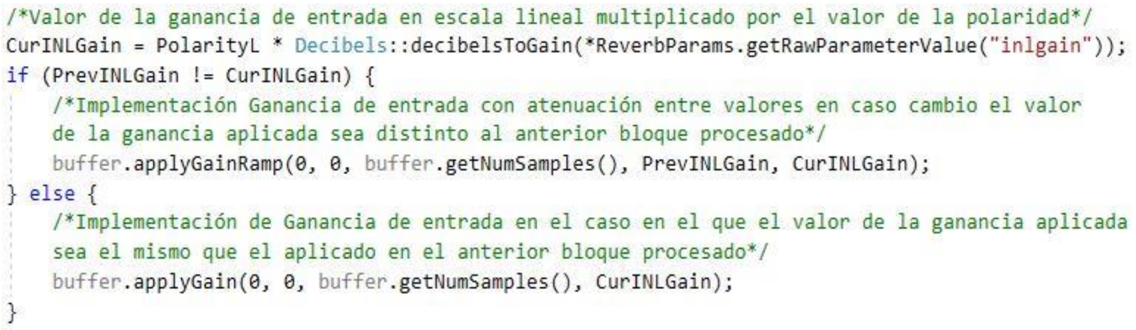
En la implementación de estas etapas se utilizaron funciones de algunas clases que vienen dadas con el marco de referencia utilizado. La primera clase tiene que ver con la expresión de las variables de ganancia de entrada y salida. Como se explicó en la sección de marco teórico, al realizar el muestreo una señal de audio analógica y convertirla en datos digitales, esta toma valores binarios que representan la amplitud de la señal entre 0 y 1, los cuales son utilizados para cualquier procesamiento de dicha señal. Sin embargo, debido a que la escucha del ser humano se aproxima más a una escala logarítmica que a una escala lineal, se decidió trabajar con la escala en decibelios. Para este propósito se utilizó una clase de JUCE llamada *“Decibels*”, la cual es capaz de transformar valores lineales a logarítmicos y viceversa de una manera mucho más óptima. Para la ganancia de entrada y para la de salida, se utilizaron valores desde -100 dB a +12dB, que equivale a multiplicar la amplitud de la señal entre 0,00001 a 4 veces, siendo este, un rango bastante amplio de valores de ganancia para aplicar al audio procesado.

Una vez con los valores de ganancia establecidos, para aplicarlos a las señales de audio se utilizaron dos funciones pertenecientes al buffer del bloque de procesamiento (*Process Block*) de cada módulo. La primera función utilizada considera el caso en el que los valores de ganancia aplicados sean modificados entre un bloque de datos y el siguiente. En este caso, se pueden producir discontinuidades y ruidos no deseados en las señales de audio. Esta función se utiliza con el fin de evitar este problema, ya que divide la variación de ganancia aplicada entre ambos bloques de audio para el número de muestras que tiene el bloque. De manera simplificada, se podría entender como una atenuación o una “rampa” a la velocidad de cambio de la ganancia implementada. La segunda función es mucho más sencilla y considera el caso en el que la ganancia a aplicar no se modifica entre dos bloques de audio, siendo inexistente el riesgo de discontinuidades ni ruidos problemáticos y, por lo tanto, aplicando la ganancia directamente. Es importante recordar que las etapas de ganancia fueron implementadas según el diagrama mostrado en la figura 1, es decir, la ganancia de entrada se aplicó después de la etapa de inversión de polaridad y la ganancia de salida después de la etapa de procesamiento del efecto de cada módulo. Al igual que la etapa de inversión de polaridad, las de ganancia también fueron implementadas de manera idéntica en cada *plugin*.



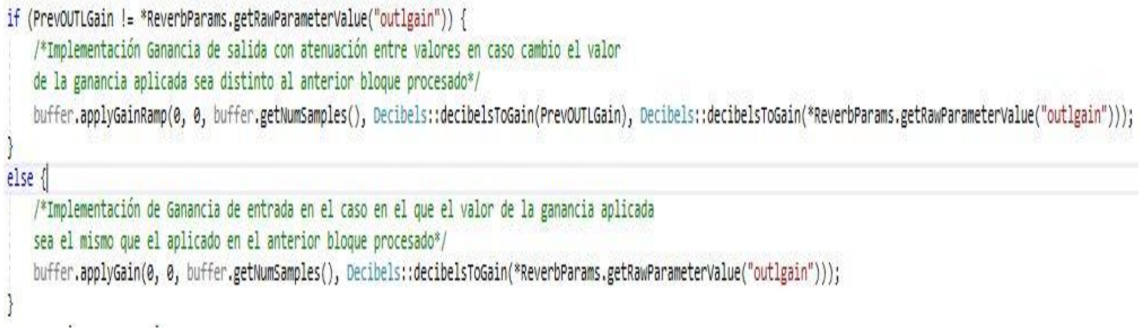
```
/*Parametros de Ganancia de Entrada y de salida*/
auto INLGainParamater = std::make_unique<AudioParameterFloat>("inlgain", "IN Gain", -100.0f, 12.0f, 0.0f);
auto OUTLGainParameter = std::make_unique<AudioParameterFloat>("outlgain", "OUT Gain", -100.0f, 12.0f, 0.0f);
```

Figura 3. Variables de ganancia de entrada y salida.



```
/*Valor de la ganancia de entrada en escala lineal multiplicado por el valor de la polaridad*/
CurINLGain = PolarityL * Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("inlgain"));
if (PrevINLGain != CurINLGain) {
    /*Implementación Ganancia de entrada con atenuación entre valores en caso cambio el valor
    de la ganancia aplicada sea distinto al anterior bloque procesado*/
    buffer.applyGainRamp(0, 0, buffer.getNumSamples(), PrevINLGain, CurINLGain);
} else {
    /*Implementación de Ganancia de entrada en el caso en el que el valor de la ganancia aplicada
    sea el mismo que el aplicado en el anterior bloque procesado*/
    buffer.applyGain(0, 0, buffer.getNumSamples(), CurINLGain);
}
```

Figura 4. Implementación de la ganancia de entrada.



```
if (PrevOUTLGain != *ReverbParams.getRawParameterValue("outlgain")) {
    /*Implementación Ganancia de salida con atenuación entre valores en caso cambio el valor
    de la ganancia aplicada sea distinto al anterior bloque procesado*/
    buffer.applyGainRamp(0, 0, buffer.getNumSamples(), Decibels::decibelsToGain(PrevOUTLGain), Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("outlgain")));
}
else {
    /*Implementación de Ganancia de entrada en el caso en el que el valor de la ganancia aplicada
    sea el mismo que el aplicado en el anterior bloque procesado*/
    buffer.applyGain(0, 0, buffer.getNumSamples(), Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("outlgain")));
}
```

Figura 5. Implementación de la ganancia de salida.

3.5 Diseño y desarrollo del efecto de ecualización

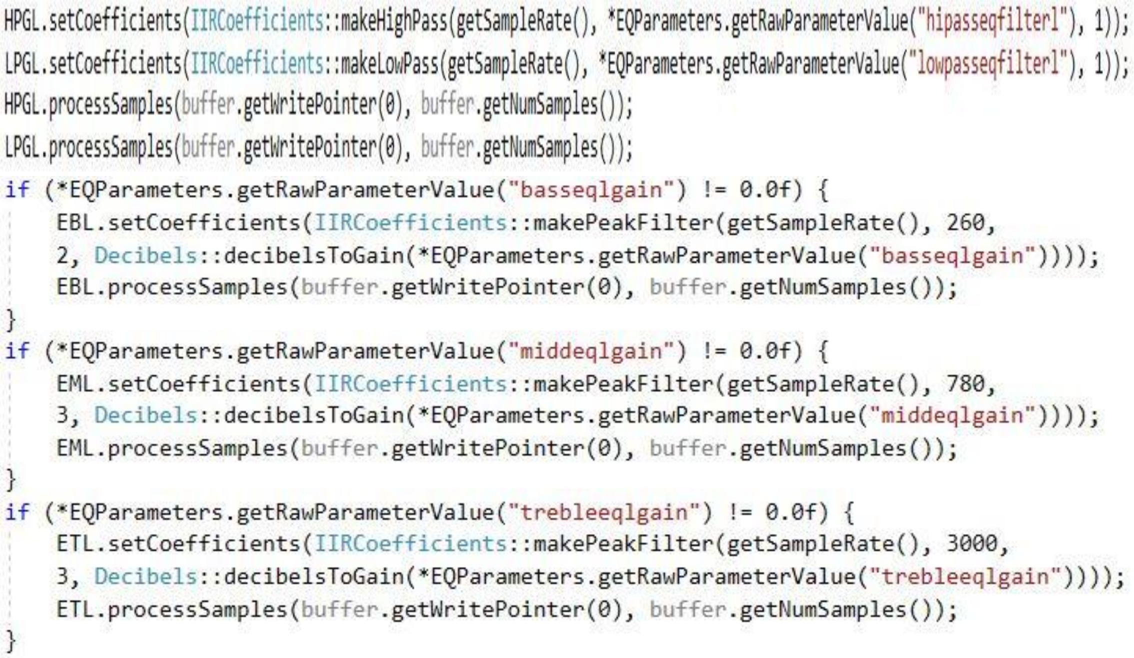
Este efecto fue uno de los más simples de implementar, ya que se utilizó una clase que viene configurada en JUCE, la cual crea filtros de respuesta infinita al impulso y tiene varias configuraciones o tipos de filtros para utilizar. Todo el procesamiento realizado para este efecto fue en serie, es decir, que se ejecutó dentro del buffer principal de audio sin necesidad de crear buffers secundarios de procesamiento. La elección de filtros de respuesta infinita al impulso fue debido a que este tipo de filtros son muy eficientes y precisos en el ámbito digital. En total se utilizaron 5 filtros, para señales monofónicas, uno de paso alto, uno de paso bajo y tres de tipo *peak* para bajas, medias y altas frecuencias. Cabe mencionar que, para los módulos diseñados para señales en estéreo, el número de filtros utilizados fue de 10, lo cual permitió tener el control de cada canal por separado, es decir, en las señales estereofónicas procesadas, no se duplicó el procesamiento de uno de los canales hacia el otro sino de manera individual permitiendo de esta manera filtrar de forma distinta cada canal, de ser requerido. El orden de implementación de los filtros en este efecto se desarrolló según se muestra en el siguiente diagrama:

Figura 6. Diagrama de flujo de la señal de audio en el ecualizador.

Los filtros de paso alto y bajo se construyeron con un factor de calidad Q de 1. Posteriormente en la sección de resultados se mostrará a que orden de filtro corresponde dicho factor de calidad. Tanto el filtro de paso alto como el de paso bajo se diseñaron con el rango de frecuencias del espectro humano audible (20 Hz – 20 kHz). Para los filtros tipo *peak*, se establecieron frecuencias de 125 Hz, 700, 4000 Hz, con factores de calidad de 3 respectivamente. Los valores de frecuencias, así como los de factor de calidad, fueron se seleccionaron para que cada banda de estos tenga el menor solapamiento posible con una ganancia de entre -18 y +18 dB (figura 7, 8).

```
IIRFilter HPGL, LPGL, EBL, EML, ETL; //Creación de filtros IIR con ayuda de la clase del Framework JUCE IIRFilter
HPGL.reset(); LPGL.reset(); EBL.reset(); EML.reset(); ETL.reset();/*Reinicia los filtros para limpiar de cualquier ruido indeseado
antes de reproducir el audio*/
auto HipassEQFilterLParameter = std::make_unique<AudioParameterFloat>("hipasseqfilterl", "HP Filter", 20.f, 20000.f, 20.f);
auto LowPassEQFilterLParameter = std::make_unique<AudioParameterFloat>("lowpasseqfilterl", "LP Filter", 20.f, 20000.f, 20000.f);
auto BassEQLParameter = std::make_unique<AudioParameterFloat >("basseqlgain", "Bass EQ", -18.0f, 18.0f, 0.0f);
auto MiddEQLParameter = std::make_unique<AudioParameterFloat >("middeqlgain", "Midd EQ", -18.0f, 18.0f, 0.0f);
auto TrebleEQLParameter = std::make_unique<AudioParameterFloat >("trebleeqlgain", "Treble EQ", -18.0f, 18.0f, 0.0f);
```

Figura 7. Creación y reinicio de filtros y declaración de sus variables dentro del ecualizador.



```
HPGL.setCoefficients(IIRCoefficients::makeHighPass(getSampleRate(), *EQParameters.getRawParameterValue("hipasseqfilterl"), 1));
LPGL.setCoefficients(IIRCoefficients::makeLowPass(getSampleRate(), *EQParameters.getRawParameterValue("lowpasseqfilterl"), 1));
HPGL.processSamples(buffer.getWritePointer(0), buffer.getNumSamples());
LPGL.processSamples(buffer.getWritePointer(0), buffer.getNumSamples());
if (*EQParameters.getRawParameterValue("basseqlgain") != 0.0f) {
    EBL.setCoefficients(IIRCoefficients::makePeakFilter(getSampleRate(), 260,
    2, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("basseqlgain"))));
    EBL.processSamples(buffer.getWritePointer(0), buffer.getNumSamples());
}
if (*EQParameters.getRawParameterValue("middeqlgain") != 0.0f) {
    EML.setCoefficients(IIRCoefficients::makePeakFilter(getSampleRate(), 780,
    3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqlgain"))));
    EML.processSamples(buffer.getWritePointer(0), buffer.getNumSamples());
}
if (*EQParameters.getRawParameterValue("trebleeqlgain") != 0.0f) {
    ETL.setCoefficients(IIRCoefficients::makePeakFilter(getSampleRate(), 3000,
    3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqlgain"))));
    ETL.processSamples(buffer.getWritePointer(0), buffer.getNumSamples());
}
```

Figura 8. Implementación de filtros tipo *peak* para bajas, medias y altas frecuencias.

Como se puede observar en la figura 8, la implementación de cada uno de los filtros tipo *peak*, se realiza solamente en los casos en los que la ganancia aplicada a cada banda de frecuencia sea distinta a 0,0 dB. Este tipo de diseño se desarrolló de esta manera con el fin de optimizar el procesamiento del *plugin* y poder minimizarlo.

3.6 Limitaciones de diseño en el ecualizador

Desde su diseño, el ecualizador desarrollado tiene dos limitaciones principales. La primera tiene que ver con los filtros de paso alto y bajo. En primer lugar, estos se diseñaron con un factor de calidad de 1, lo cual no indica un orden de filtro en específico. Sin embargo, como se mostrará más adelante en este documento, en las pruebas realizadas a los módulos, se estimó a que orden corresponde dicho factor de calidad. La limitación en este sentido viene dada en tanto que el valor del factor de calidad de estos filtros es fijo, lo cual implica que el orden de los mismo también lo sea, sin permitir que el usuario lo modifique según requiera, como en otros *plugins* disponibles en el mercado de compañías como *Waves* o *Universal Audio*.

La segunda limitación de diseño del ecualizador tiene que ver con los filtros tipo *peak*, que fueron diseñados de manera similar a los de un ecualizador gráfico, es decir, su único parámetro modificable es la ganancia de cada filtro, siendo fijos los parámetros correspondientes a la frecuencia y factor de calidad. Esto representa una restricción frente a otros tipos de ecualizadores como los paramétricos o semi paramétricos.

3.7 Diseño y desarrollo del efecto de compresión

Para el desarrollo de este efecto se utilizó la propiedad del lenguaje C++ de crear clases para implementarlas en objetos con atributos regidos por dichas clases. Se crearon dos clases para este efecto en particular. La primera se trata de una especie de envolvente que trabaja con los parámetros temporales del efecto y la segunda es la que describe el funcionamiento del compresor en sí con el resto de sus parámetros.

3.7.1 Envolvente del compresor

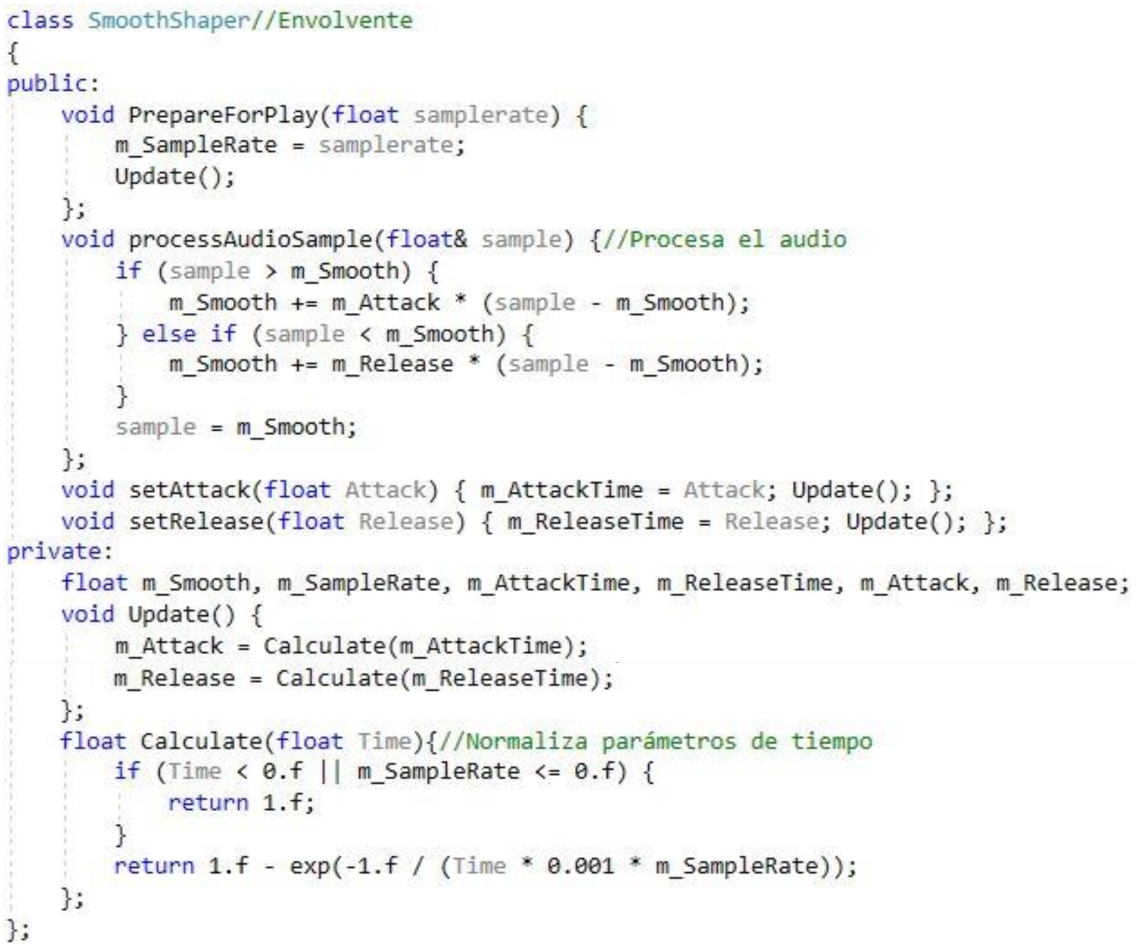
Se creó una clase que corresponde a la envolvente del efecto a aplicar, esto quiere decir, la manera en que la amplitud de la señal evoluciona en función del tiempo. Es muy común encontrar envolventes de audio en sintetizadores y suelen tener varios parámetros de tiempo y nivel. Sin embargo, para un compresor, solo se requiere de dos parámetros: el tiempo de ataque, que dictamina cuanto se demora el efecto en ser implementado en la señal una vez que se ha activado. El tiempo de relajación, el cual se refiere a cuanto se demora el efecto en dejar de actuar después de haber sido implementado.

Estos dos parámetros temporales deben ser normalizados en una especie de valor alfa, el cual representará el 100 % del tiempo de cada uno de estos parámetros. Esto se realiza para que sin importar cuál sea el valor, se aplique de manera porcentual al implementar el efecto de compresión. Como se está trabajando en el dominio digital se debe incluir la frecuencia de muestreo para que los valores temporales de los parámetros se relaciones con las muestras procesadas. La normalización de estos parámetros se calcula mediante la siguiente ecuación:

$$\alpha = 1 - e^{\left(-\frac{1}{Tiempo\ en\ segundos\ * frecuencia\ de\ muestreo}\right)} \quad (2)$$

Ecuación 2: Tiempo de ataque/relajación normalizado.

Existen otras maneras de trabajar con los parámetros de ataque y relajación de un compresor que contemplan uso de contadores temporales de un número de muestras o cálculos de diseño de transientes de audio. Sin embargo, la utilizada en esta envolvente es una de las formas más eficientes, ya que no requiere de un alto consumo de procesamiento del ordenador en el que se ejecute el módulo. El algoritmo de construcción de la clase de la envolvente del compresor se presenta a continuación:



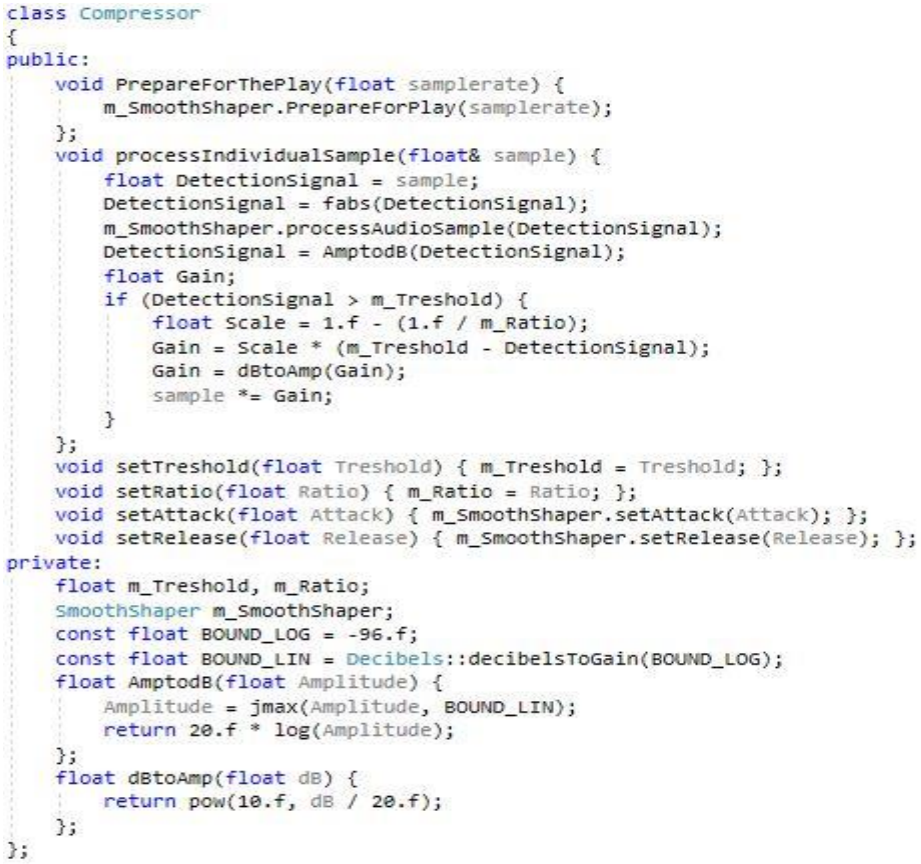
```
class SmoothShaper//Envolvente
{
public:
    void PrepareForPlay(float samplerate) {
        m_SampleRate = samplerate;
        Update();
    };
    void processAudioSample(float& sample) {//Procesa el audio
        if (sample > m_Smooth) {
            m_Smooth += m_Attack * (sample - m_Smooth);
        } else if (sample < m_Smooth) {
            m_Smooth += m_Release * (sample - m_Smooth);
        }
        sample = m_Smooth;
    };
    void setAttack(float Attack) { m_AttackTime = Attack; Update(); };
    void setRelease(float Release) { m_ReleaseTime = Release; Update(); };
private:
    float m_Smooth, m_SampleRate, m_AttackTime, m_ReleaseTime, m_Attack, m_Release;
    void Update() {
        m_Attack = Calculate(m_AttackTime);
        m_Release = Calculate(m_ReleaseTime);
    };
    float Calculate(float Time){//Normaliza parámetros de tiempo
        if (Time < 0.f || m_SampleRate <= 0.f) {
            return 1.f;
        }
        return 1.f - exp(-1.f / (Time * 0.001 * m_SampleRate));
    };
};
```

Figura 9. Clase envolvente para el efecto de compresión.

Como se puede observar en la figura 9, en esta clase es en donde se procesa la señal de audio muestra por muestra. Para poder realizar dicho procesamiento tiene que formar parte de otra clase que es la que se encarga de realizar los cálculos de compresión de la señal de audio.

3.7.2 Clase de compresión

En esta clase se trabaja con los demás parámetros de un compresor, los cuales son el punto de nivel umbral desde el cual se aplica el efecto (*Treshold*), la relación de compresión (*Ratio*) y el suavizado (*Knee*). En esta clase se estableció el funcionamiento del compresor, sin embargo, para que los parámetros temporales fuesen aplicados, se creó un objeto perteneciente a la envolvente dentro de esta clase para que al procesar la información de audio en la clase de compresión se realice el procesamiento mediante la envolvente. El algoritmo de construcción de esta clase se muestra a continuación:



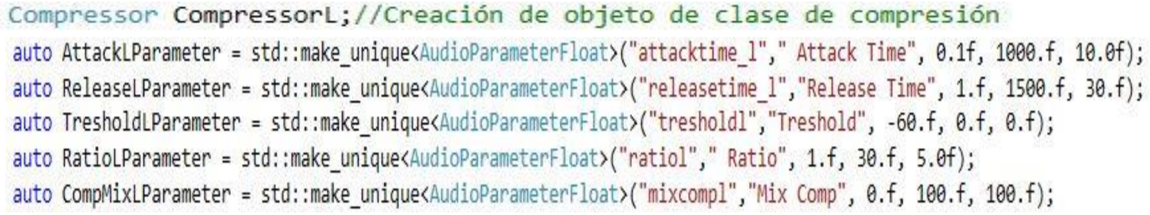
```
class Compressor
{
public:
    void PrepareForThePlay(float samplerate) {
        m_SmoothShaper.PrepareForPlay(samplerate);
    };
    void processIndividualSample(float& sample) {
        float DetectionSignal = sample;
        DetectionSignal = fabs(DetectionSignal);
        m_SmoothShaper.processAudioSample(DetectionSignal);
        DetectionSignal = AmptodB(DetectionSignal);
        float Gain;
        if (DetectionSignal > m_Treshold) {
            float Scale = 1.f - (1.f / m_Ratio);
            Gain = Scale * (m_Treshold - DetectionSignal);
            Gain = dBtoAmp(Gain);
            sample *= Gain;
        }
    };
    void setTreshold(float Treshold) { m_Treshold = Treshold; };
    void setRatio(float Ratio) { m_Ratio = Ratio; };
    void setAttack(float Attack) { m_SmoothShaper.setAttack(Attack); };
    void setRelease(float Release) { m_SmoothShaper.setRelease(Release); };
private:
    float m_Treshold, m_Ratio;
    SmoothShaper m_SmoothShaper;
    const float BOUND_LOG = -96.f;
    const float BOUND_LIN = Decibels::decibelsToGain(BOUND_LOG);
    float AmptodB(float Amplitude) {
        Amplitude = jmax(Amplitude, BOUND_LIN);
        return 20.f * log(Amplitude);
    };
    float dBtoAmp(float dB) {
        return pow(10.f, dB / 20.f);
    };
};
```

Figura 10. Clase de compresión.

Como se puede observar en la figura 10, dentro de esta clase se consideran funciones de conversión de los valores de amplitud de las muestras de audio de valores lineales a decibelios y viceversa. Esto se realizó debido a que cada muestra ingresa al compresor con su amplitud en una escala lineal, pero para su procesamiento es necesario transformarla a un valor logarítmico y, posteriormente, convertirla nuevamente a un valor lineal para que salga del efecto a la siguiente etapa. Es importante mencionar que tanto en la clase de la envolvente de audio como en la de compresión se establece una función llamada “preparefortheplay” (preparar para reproducir), que sirve para configurar el parámetro de la frecuencia de muestreo que tiene el audio procesado antes de que este se reproduzca con el fin de que no existan problemas de que este parámetro sea distinto para el audio y para el compresor. Esta función también es similar a la función que cada módulo crea por defecto llamada “preparetoplay”, la cual fue explicada en la sección del marco teórico. La utilidad de crear estas clases permitió que la implementación del efecto dentro del módulo en sí sea mucho más sencilla que realizando directamente en el bloque de procesamiento general.

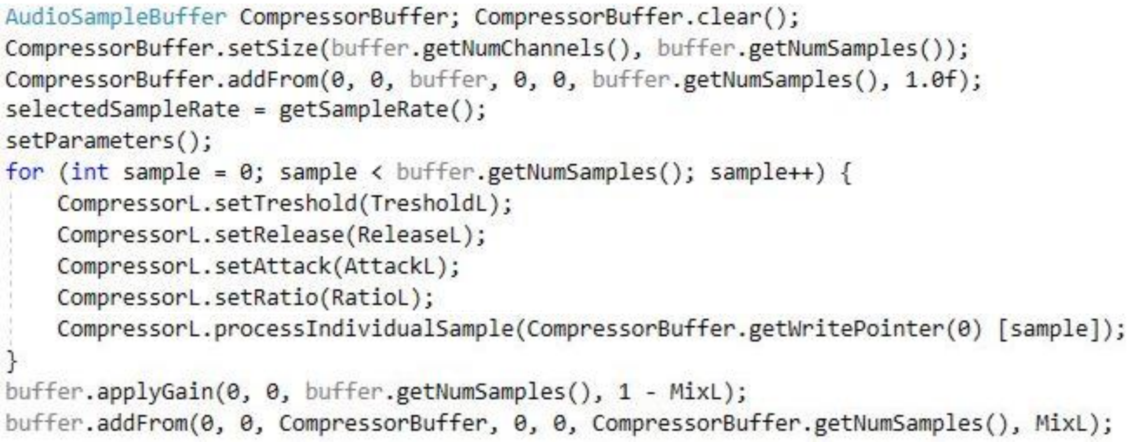
3.7.3 Implementación del efecto de compresión

La implementación de este efecto dentro del módulo correspondiente fue relativamente sencilla ya que, al crear un objeto correspondiente a la clase de compresión, que a su vez engloba un objeto interno correspondiente a la clase envolvente de audio, el trabajo de implementación consiste solamente en actualizar los parámetros de los objetos creados en el bloque de procesamiento general. A continuación, se muestra la implementación del efecto de este *plugin*:



```
Compressor CompressorL;//Creación de objeto de clase de compresión
 auto AttackLParameter = std::make_unique<AudioParameterFloat>("attacktime_l"," Attack Time", 0.1f, 1000.f, 10.0f);
 auto ReleaseLParameter = std::make_unique<AudioParameterFloat>("releasetime_l","Release Time", 1.f, 1500.f, 30.f);
 auto TresholdLParameter = std::make_unique<AudioParameterFloat>("tresholdl","Treshold", -60.f, 0.f, 0.f);
 auto RatioLParameter = std::make_unique<AudioParameterFloat>("ratiol"," Ratio", 1.f, 30.f, 5.0f);
 auto CompMixLParameter = std::make_unique<AudioParameterFloat>("mixcompl","Mix Comp", 0.f, 100.f, 100.f);
```

Figura 11. Creación del objeto de compresión y declaración de variables del efecto.



```
AudioSampleBuffer CompressorBuffer; CompressorBuffer.clear();
CompressorBuffer.setSize(buffer.getNumChannels(), buffer.getNumSamples());
CompressorBuffer.addFrom(0, 0, buffer, 0, 0, buffer.getNumSamples(), 1.0f);
selectedSampleRate = getSampleRate();
setParameters();
for (int sample = 0; sample < buffer.getNumSamples(); sample++) {
    CompressorL.setTreshold(TresholdL);
    CompressorL.setRelease(ReleaseL);
    CompressorL.setAttack(AttackL);
    CompressorL.setRatio(RatioL);
    CompressorL.processIndividualSample(CompressorBuffer.getWritePointer(0) [sample]);
}
buffer.applyGain(0, 0, buffer.getNumSamples(), 1 - MixL);
buffer.addFrom(0, 0, CompressorBuffer, 0, 0, CompressorBuffer.getNumSamples(), MixL);
```

Figura 12. Implementación de la compresión en el módulo de audio digital.

Al observar la figura 12, se puede notar que el procesamiento del efecto se realizó en un buffer de audio separado del principal, esto se desarrolló de esta manera para que la compresión pueda ser aplicada de manera paralela y poder controlar la cantidad aplicada de la señal comprimida sobre la señal original. Los valores de los parámetros de ataque y relajación se establecieron en una escala de milisegundos para tener una mayor precisión en el control de estos. Además, es importante tomar en cuenta que los parámetros son actualizados muestra por muestra, lo cual es esencial cuando se trabaja con variables de tiempo.

3.8 Limitaciones de diseño en el compresor

A pesar de que el compresor diseñado es bastante completo, una de las limitaciones más importantes consiste en que este efecto no fue desarrollado para compresión en cadena, la cual se activa de acuerdo con el nivel de la señal de un canal de audio adicional. Este tipo de compresión viene disponible en algunos *plugins* presentes en la industria del software de audio, sin embargo, en el desarrollo de este módulo en particular, se decidió no incluirla. La otra limitación importante es que se diseñó un parámetro de control del suavizado o

Knee de la compresión, sino que está fue implementada con un valor de dicho suavizado de 0.

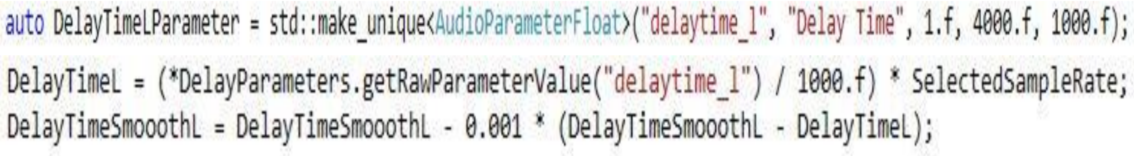
3.9 Diseño y desarrollo del efecto de retardo o *Delay*

La construcción se realizó considerando 3 parámetros: el tiempo de retardo, la retroalimentación y la cantidad de efecto aplicados a la señal. Al igual que el efecto de compresión, este también fue aplicado de manera paralela para poder controlar el porcentaje de la señal original y de la señal procesada que forman la señal de salida. En general, un efecto de *Delay* es de implementación poco complicada en el dominio digital, pero existen varias maneras de configurar los parámetros que lo conforman que pueden que su desarrollo implique una complejidad considerable.

3.9.1 Obtención y configuración del tiempo de retardo

Siendo este uno de los parámetros más importantes del efecto de *Delay*, desde su diseño se planteó configurar el tiempo de retardo con dos posibles opciones: libre o sincronizada con el tiempo dado por el software anfitrión que este ejecutando al módulo. Para los dos casos se utilizó un buffer con un tamaño de 4 segundos. La razón para utilizar el mismo buffer para las dos opciones de tiempo de retardo fue el ahorro de procesamiento del ordenador.

Para la opción del tiempo en modo libre simplemente se añadió un parámetro con valores en el rango del buffer y calcular su tiempo correspondiente en muestras de audio de acuerdo con la frecuencia de muestreo con la que se trabaje. En general, este caso fue sencillo de codificar y para una mayor precisión en el control del parámetro de tiempo para esta opción del *Delay* se utilizó una escala temporal en milisegundos. A continuación, se muestra el código correspondiente:



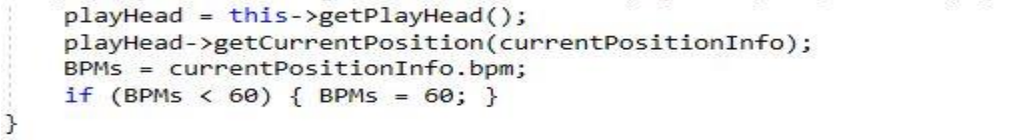
```
auto DelayTimeLParameter = std::make_unique<AudioParameterFloat>("delaytime_l", "Delay Time", 1.f, 4000.f, 1000.f);
DelayTimeL = (*DelayParameters.getRawParameterValue("delaytime_l") / 1000.f) * SelectedSampleRate;
DelayTimeSmooothL = DelayTimeSmooothL - 0.001 * (DelayTimeSmooothL - DelayTimeL);
```

Figura 13. Declaración de variable y configuración de parámetros del tiempo de *Delay* en el modo libre.

Como se puede observar en la figura 13, se añadió un parámetro de atenuación del tiempo (*DelayTimeSmooth*) para reducir e incluso eliminar posibles interrupciones y discontinuidades producidas en el audio procesado por la velocidad de cambio del parámetro del tiempo para la opción libre.

Para el segundo caso, la configuración del retardo temporal fue más compleja. En general, existen tres maneras principales para obtener la información del tempo o pulsos por minuto (*bpm*) del software anfitrión que ejecuta al módulo de procesamiento de audio digital. La primera manera, la cual es la menos eficiente, es construir un algoritmo de detección de tiempo que evalúe las transientes de la señal que tengan mayor amplitud y calcule el espacio temporal entre dichas transientes para obtener de esta manera una información aproximada sobre los *bpm*s de dicho audio. Sin embargo, dicho algoritmo necesitaría un determinado número mínimo de muestras para realizar la detección, lo cual puede llegar a consumir mucho procesamiento del ordenador y generar una latencia considerable que no permita un procesamiento en tiempo real, sin mencionar que, la detección puede tener fallas indetectables al ser codificadas, resultando en cálculos erróneos de retardo temporal. Por estas razones, esta manera de configurar el tiempo de *Delay* fue descartada desde un inicio. La segunda manera consiste en entrar a las propiedades del sistema general del ordenador, en las cuales se esté almacenando toda la información acerca del software anfitrión, sin embargo, esto puede llevar una codificación muy extensa y al existir demasiada información que el anfitrión está enviando al sistema, la cual no tiene nada que ver con el efecto a implementar, el encontrar los datos correctos para calcular el retardo temporal puede llegar a consumir demasiado procesamiento. Se intentó implementar esta forma de obtención del tiempo de *Delay*, sin embargo, en las pruebas iniciales de los primeros prototipos del módulo, el procesamiento fue muy alto y, por lo tanto, fue necesario buscar otra manera de conseguir dicha información.

La tercera forma, la cual fue finalmente implementada en el módulo en cuestión, consiste en utilizar una clase que viene dada en el marco de referencia utilizado, la cual solicita la información del tempo directamente al software anfitrión, sin necesidad de un algoritmo de detección de tiempo, ni de ingresos al sistema operativo en busca de dicha información. Como se trata de una clase que ha sido diseñada por los desarrolladores de JUCE, está optimizada y para su implementación se necesitan de pocas líneas de código tanto en la declaración de variables como en la implementación de estas para la obtención del tempo. La clase utilizada tiene el nombre de "*AudioPlayHead*", por lo cual solamente fue necesario crear un objeto de la misma para obtener la información requerida del anfitrión. Además, como se trata de un *plugin* de procesamiento en tiempo real, también fue necesario crear un objeto correspondiente a una clase derivada de la antes mencionada, para que la solicitud de información se realice en tiempo real. A continuación, se muestra el algoritmo correspondiente:



```
AudioPlayHead* playHead;
AudioPlayHead::CurrentPositionInfo currentPositionInfo;
auto TempoSyncBindingL = std::make_unique<AudioParameterBool>("synctempol", "Sync Tempo", false);
if (*DelayParameters.getRawParameterValue("synctempol") == true) {
    playHead = this->getPlayHead();
    playHead->getCurrentPosition(currentPositionInfo);
    BPMs = currentPositionInfo.bpm;
    if (BPMs < 60) { BPMs = 60; }
}
```

Figura 14. Solicitud del tempo al software anfitrión del módulo de *Delay*.

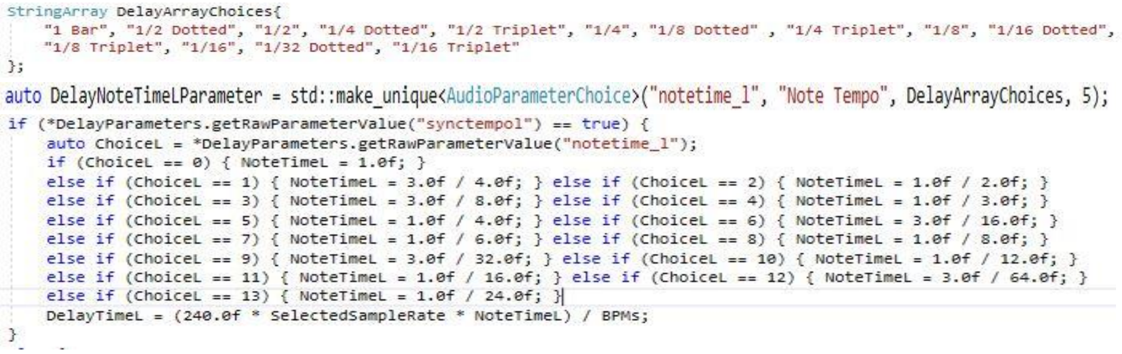
Como se muestra en la figura 14, se estableció un límite inferior para los pulsos por minuto que entrega el anfitrión, el cual fue de 60 *bpm*. Esto se realizó debido al tamaño del buffer de retardo, permitiendo que en el mismo ingrese información de audio correspondiente hasta 4 figuras musicales negras en dicho *bpm*. También se muestra en esta figura, un parámetro booleano creado para cambiar el tempo de retardo entre el modo libre, anteriormente explicado, y el modo sincronizado.

Una vez que se tiene acceso a la información de tempo entregada por el software anfitrión, se realizó una transformación del valor entregado a una escala temporal según la siguiente ecuación:

$$T\ (muestras) = \ sample\ Rate * \ figura\ musical\ del\ tempo * \frac{240}{Tempo\ (bpm)} \quad (3)$$

Ecuación 3: Tiempo de retardo en muestras digitales de audio.

En total se consideraron 14 posibilidades de figuras musicales en función del tempo con el que se esté trabajando, desde 1 compás de 4/4 hasta un tresillo de semicorchea (1/24). El algoritmo de cálculo y conversión del tempo obtenido del anfitrión a tiempo de retardo en muestras se muestra a continuación:



```
StringArray DelayArrayChoices{
    "1 Bar", "1/2 Dotted", "1/2", "1/4 Dotted", "1/2 Triplet", "1/4", "1/8 Dotted" , "1/4 Triplet", "1/8", "1/16 Dotted",
    "1/8 Triplet", "1/16", "1/32 Dotted", "1/16 Triplet"
};
auto DelayNoteTimeLParameter = std::make_unique<AudioParameterChoice>("notetime_l", "Note Tempo", DelayArrayChoices, 5);
if (*DelayParameters.getRawParameterValue("synctempol") == true) {
    auto ChoiceL = *DelayParameters.getRawParameterValue("notetime_l");
    if (ChoiceL == 0) { NoteTimeL = 1.0f; }
    else if (ChoiceL == 1) { NoteTimeL = 3.0f / 4.0f; } else if (ChoiceL == 2) { NoteTimeL = 1.0f / 2.0f; }
    else if (ChoiceL == 3) { NoteTimeL = 3.0f / 8.0f; } else if (ChoiceL == 4) { NoteTimeL = 1.0f / 3.0f; }
    else if (ChoiceL == 5) { NoteTimeL = 1.0f / 4.0f; } else if (ChoiceL == 6) { NoteTimeL = 3.0f / 16.0f; }
    else if (ChoiceL == 7) { NoteTimeL = 1.0f / 6.0f; } else if (ChoiceL == 8) { NoteTimeL = 1.0f / 8.0f; }
    else if (ChoiceL == 9) { NoteTimeL = 3.0f / 32.0f; } else if (ChoiceL == 10) { NoteTimeL = 1.0f / 12.0f; }
    else if (ChoiceL == 11) { NoteTimeL = 1.0f / 16.0f; } else if (ChoiceL == 12) { NoteTimeL = 3.0f / 64.0f; }
    else if (ChoiceL == 13) { NoteTimeL = 1.0f / 24.0f; }
    DelayTimeL = (240.0f * SelectedSampleRate * NoteTimeL) / BPMs;
}
```

Figura 15. Cálculo de tiempo de retardo en muestras en función del tempo del anfitrión.

3.9.2 Interpolación de valores entre muestras

Para evitar posibles ruidos indeseados o discontinuidades en el audio producidas por datos no conocidos de valores de la señal entre muestras cuando el tiempo de retardo no fuese correspondiente a un índice entero de muestra de audio, se implementó interpolación lineal, la cual responde a la siguiente ecuación:

$$A\,(j) = A\,(i) * (1 - j - i) + A\,(i+1) * (j - i) \qquad (4)$$

Ecuación 4: Interpolación lineal.

Donde:

j: Índice no entero de un dato para el cual se requiere obtener la amplitud interpolada.

i: Máximo índice entero menor a j.

A (i): Amplitud conocida de la muestra de índice i.

A (i + 1): Amplitud conocida de la muestra de índice i + 1.

A (j): Amplitud Interpolada de índice j.

Cabe mencionar que esta es una de las formas más sencillas de interpolar datos, pero también es una de las que menos procesamiento consume. Se intentó implementar otros tipos de interpolación más precisos como la polinómica y la de splines cúbicos, sin embargo, el procesamiento de estos cálculos era muy elevado, por lo cual se optó por la interpolación de la ecuación mostrada.

3.9.3 Implementación del efecto en el módulo

Con el tiempo de *Delay* calculado, la implementación del eco fue mucho más sencilla. En el buffer paralelo, se procesó la señal multiplicando las muestras de retardo por el valor porcentual del parámetro de retroalimentación. Para la salida del efecto hacia la siguiente etapa, se multiplicó la señal del buffer con efecto por el valor porcentual por el parámetro de mezcla o cantidad de efecto aplicado y la señal del buffer original (sin efecto) por el residuo porcentual de dicho parámetro. A continuación, se muestra el código de implementación de este efecto de retardo:

```
/*Prepare to play*/
mCircularBufferLength= sampleRate * 4;
if (mCircularBufferLeft == nullptr)
{
    mCircularBufferLeft=new float [mCircularBufferLength];
}
zeromem(mCircularBufferLeft, mCircularBufferLength*sizeof(float));
mCirculkarBufferWriteHead=0;
/*Process Block*/
for (int sample=0;sample<buffer.getNumSamples();sample++)
{
    setDelayTime();
    mCircularBufferLeft[DelayWriteHeadL]= FeedbackL !=0.f ? buffer.getWritePointer(0) [sample] + mFeedbackLeft
        : buffer.getWritePointer(0) [sample];
    mDelayReadHeadL= *DelayParameters.getRawParameterValue("synctempol") == true ? DelayWriteHeadL-DelayTimeL
        : DelayWriteHeadL-DelayTimeSmooothL;
    if (mDelayReadHeadL<0) {
        mDelayReadHeadL += mCircularBufferLength;//Tamaño lectura señal con Delay
    }
    float delay_sample_left;
    if ((mDelayReadHeadL - int(mDelayReadHeadL)) == 0) {
        delay_sample_left = mCircularBufferLeft[(int)mDelayReadHeadL];
    } else {
        int ReadHeadL = int(mDelayReadHeadL);
        int ReadHeadLpp = ReadHeadL++;
        if (ReadHeadLpp >= mCircularBufferLength) {
            ReadHeadLpp -= mCircularBufferLength;
        }
        float ReadHeadfloatL = mDelayReadHeadL - ReadHeadL;
        delay_sample_left = LinearInterpolation(mCircularBufferLeft[ReadHeadL],
            mCircularBufferLeft[ReadHeadLpp], ReadHeadfloatL);
    }
    int ReadHeadL = int(mDelayReadHeadL);
    int ReadHeadLpp = ReadHeadL++;
    if (ReadHeadLpp >= mCircularBufferLength) {
        ReadHeadLpp -= mCircularBufferLength;
    }
    mFeedbackLeft = delay_sample_left * FeedbackL;
    DelayWriteHeadL++;
    buffer.setSample(0, sample, (buffer.getSample(0, sample) * (1- MixL)) + (delay_sample_left* MixL));
    if (DelayWriteHeadL>=mCircularBufferLength) {
        DelayWriteHeadL=0;//Circular Buffer
    }
}
```

Figura 16. Implementación del efecto de *Delay* en el módulo de audio.

Como se muestra en la figura 16, el parámetro de tiempo de retardo se actualiza en cada muestra de audio para tener una precisión muy alta en caso de que la información del tempo entregada por el software anfitrión no sea estática. Sin embargo, con esto, puede subir el procesamiento del módulo en general. En el caso de la interpolación, anteriormente explicada, se aplica solamente cuando el tiempo de retardo corresponde a un índice no entero de las muestras del bloque de procesamiento de audio. Durante las pruebas realizadas en este módulo, se determinó que el parámetro de retroalimentación (*Feedback*) influye en las unidades de volumen de escala completa según la siguiente ecuación:

$$LUFS = 0{,}2143Feedback^2 + \ 0{,}2143\ Feedback - 14{,}4 \quad (5)$$

Ecuación 5. Relación del nivel de volumen y el parámetro de retroalimentación del módulo de *Delay*.

3.10 Limitaciones de diseño en el efecto de *Delay*

En general, el módulo de *Delay* desarrollado es muy similar a muchos *plugins* disponibles en el mercado, sin embargo, la que es probablemente la principal limitación, tiene que ver con la interpolación, la cual, como se mencionó anteriormente, es muy simple. Es posible que, debido a la limitada precisión de este proceso, se pueden producir cambios inesperados en la señal de audio al modificar los valores del parámetro del tiempo de retardo. De todas maneras, estos cambios no serán discontinuidades, sobrecargas de amplitud o indeterminaciones en los datos, ya que, con la interpolación siempre se tendrá un valor de salida, ya sea exacto o aproximado.

3.11 Diseño y desarrollo del efecto de reverberación

La construcción de este efecto fue muy sencilla, ya que, al igual que en el módulo de ecualización, se utilizó una clase que viene dada en JUCE. Esta clase tiene el mismo nombre del efecto a implementar y por defecto tiene ciertos parámetros configurables que con simples ajustes quedan listos para emplearse. A diferencia de muchos *plugins* de reverberación, en el caso de este módulo, no se consideraron parámetros temporales como el pre-retardo o el tiempo de reverberación. En lugar de esto, se hizo uso de los parámetros acústicos que vienen dados en la clase utilizada. Al igual que en los módulos de compresión y *Delay*, en este también se procesó la señal en un buffer paralelo, y adicionalmente, se implementaron filtros de paso alto y bajo idénticos a los del módulo de ecualización. Estos filtros se aplicaron solamente a la señal con efecto y no a la señal original. La idea detrás de esto considera que muchas veces el efecto de reverberación se requiere aplicar a un rango determinado de frecuencias, y generalmente, las frecuencias bajas no se procesan con este efecto. Al filtrar la señal se permite al usuario controlar el rango del espectro con el cual procesar la señal.

3.11.1 Parámetros acústicos dentro del módulo de reverberación e implementación del efecto en el módulo

La clase utilizada para la implementación del efecto en cuestión contempla 3 parámetros acústicos, el tamaño, la anchura y el amortiguamiento. El primer parámetro se refiere, como su nombre lo indica, al tamaño de un recinto acústico hipotético del cual la señal obtiene la reverberación. El segundo parámetro a pesar de que su nombre se relaciona con el audio, en realidad tiene que ver con la extensión del campo sonoro, es decir, una especie de profundidad de este, esto quiere decir que, al maximizar este parámetro, las señales estereofónicas se abrirán totalmente a cada lado del panorama sonoro, y las señales monofónicas se extienden por completo a través del mismo. Finalmente, el último parámetro de amortiguamiento se relaciona con la absorción del recinto acústico hipotético, así como la difracción de este. También se relaciona con la diafonía o "*crosstalk*". De manera simplificada, este parámetro controla, de cierta forma, la cantidad de "reflexiones acústicas" que se producirían en un recinto reverberante. Los tres parámetros por defecto están normalizados entre los valores en el rango de 0 a 1, lo cual permite aplicarlos de manera porcentual. El algoritmo de declaración e implementación para este efecto se muestra a continuación:

```
Reverb ReverbL; // Objeto de reverberación
Reverb::Parameters ReverbParamsL; // Objeto de parámetros de reverberación
IIRFilter HPRL, LPRL;// Filtros de paso alto y bajo
/*Parámetros de Filtros*/
auto HipassVerbFilterLParameter = std::make_unique<AudioParameterFloat>("hipassverbfilterl", "HP Reverb Filter",
    20.f, 20000.f, 20.f);
auto LowPassVerbFilterLParameter = std::make_unique<AudioParameterFloat>("lowpassverbfilterl", "LP Reverb Filter",
    20.f, 20000.f, 20000.f);
/*Parámetros para el efecto de Reverb*/
auto ReverbSizeL = std::make_unique<AudioParameterFloat>("sizel", "Size", 0.001f, 1.0f, 0.5f);//
auto ReverbWidthL = std::make_unique<AudioParameterFloat>("widthl", "Width", 0.001f, 1.0f, 0.5f);//
auto ReverbWetLParameter = std::make_unique<AudioParameterFloat>("reverbmixl", "Wet", 0.0f, 100.0f, 50.0f);
auto ReverbDryLParameter = std::make_unique<AudioParameterFloat>("reverbdryl", "Dry", 0.f, 100.f, 50.0f);
auto DampingL = std::make_unique<AudioParameterFloat>("dampingl", "Damping L", 0.001f, 1.0f, 0.5f);//
```

Figura 17. Declaración de objetos y variables para el módulo de reverberación.

```
AudioSampleBuffer RVerbBuffer; RVerbBuffer.clear(); RVerbBuffer.setSize(buffer.getNumChannels, buffer.getNumSamples());
RVerbBuffer.addFrom(0, 0, buffer, 0, 0, buffer.getNumSamples());
DryL = *ReverbParams.getRawParameterValue("reverbdryl") / 100.f;
buffer.applyGain(0, 0, buffer.getNumSamples(), DryL);
if (*ReverbParams.getRawParameterValue("reverbmixl") != 0) {
    HPRL.setCoefficients(IIRCoefficients::makeHighPass(getSampleRate(), *ReverbParams.getRawParameterValue("hipassverbfilterl"), 1));
    LPRL.setCoefficients(IIRCoefficients::makeLowPass(getSampleRate(), *ReverbParams.getRawParameterValue("lowpassverbfilterl"), 1));
    HPRL.processSamples(RVerbBuffer.getWritePointer(0), RVerbBuffer.getNumSamples());
    LPRL.processSamples(RVerbBuffer.getWritePointer(0), RVerbBuffer.getNumSamples());
    ReverbParamsL.roomSize = *ReverbParams.getRawParameterValue("sizel");
    ReverbParamsL.width = *ReverbParams.getRawParameterValue("widthl");
    ReverbL.setParameters(ReverbParamsL);
    ReverbL.processMono(RVerbBuffer.getWritePointer(0), RVerbBuffer.getNumSamples());
    MixL = *ReverbParams.getRawParameterValue("reverbmixl") / 100.0f;
    buffer.addFrom(0, 0, RVerbBuffer, 0, 0, RVerbBuffer.getNumSamples(), MixL);
}
RVerbBuffer.clear();
```

Figura 18. Implementación del efecto de reverberación en el módulo correspondiente.

Como se puede observar en la figura 18, se aplicó una ganancia adicional a la señal original para que esta no quede demasiado reducida al momento de añadir la señal procesada en la salida del efecto. También es importante recalcar que el buffer de procesamiento en paralelo se crea en cada bloque de datos de audio, no en la declaración de objetos y variables, y se limpia este buffer después de que las señales de cada paquete de datos han sido procesadas. Esto se realizó de esta manera para impedir que el dicho buffer se llene de ruido basura y fue posible realizarlo así porque no se trabajó con parámetros temporales sino acústicos como ya se explicó anteriormente.

3.12 Limitaciones de diseño en el efecto de reverberación

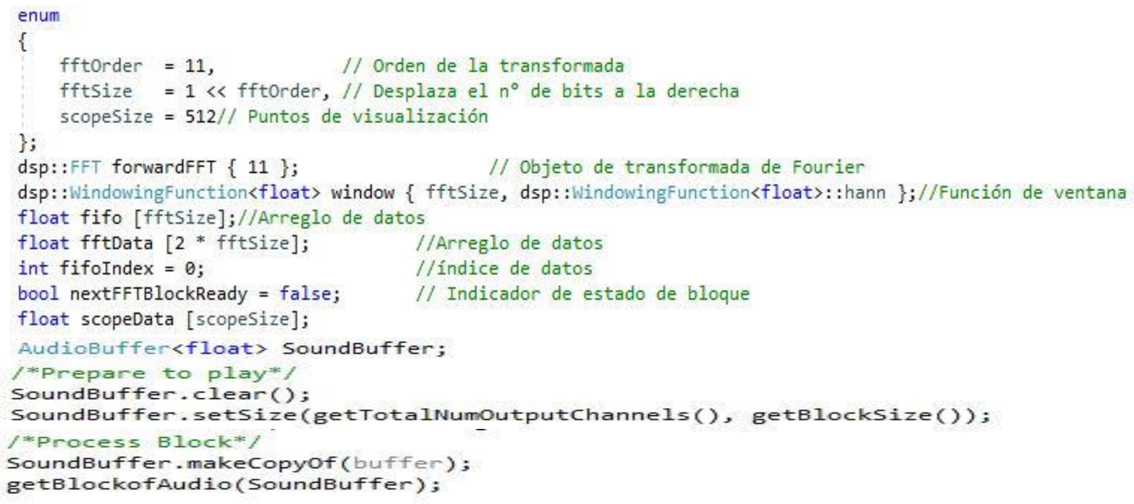
El hecho de no trabajar con valores temporales en este módulo puede parecer un poco fuera de lo común para este tipo de efectos, sin embargo, esto no implica necesariamente una desventaja en este *plugin*, sino una manera relativamente innovadora de implementar este efecto, ya que, existen varios módulos de reverberación disponibles en la industria del software de audio digital, los cuales, incluyen parámetros acústicos, como en este. El hecho de que el nivel de la señal procesada y de la señal original se controlen con diferentes parámetros puede llegar a ser una gran ventaja si se utiliza responsablemente, no obstante, si no se controla de manera adecuada puede producir niveles totales que superen los máximos del dominio digitales, resultando en saturaciones de amplitud debido a la amplitud adicional que aporta la señal procesada sobre la original.

3.13 Diseño y desarrollo del analizador de espectro en tiempo real

Este módulo no consiste en ningún efecto ni procesamiento de la señal de audio en sí, sino de una herramienta muy general de visualización del nivel del espectro frecuencial de una señal de audio. En el procesador no se modifica en ninguna manera la señal de audio que ingresa al módulo, sino que se crea una copia del buffer de audio y este es el que maneja los datos muestra por muestra en un arreglo del doble del tamaño del bloque del buffer del software anfitrión. De manera simultánea se activa un temporizador que solicita información del audio cada 33,3 milisegundos (30 cuadros por segundo). Cada vez que este llega a cero, se realiza una transformada de Fourier correspondiente. Como se trata de un módulo donde la parte de visualización es muy importante, mucha parte del algoritmo del procesador depende del algoritmo de la interfaz de usuario y viceversa.

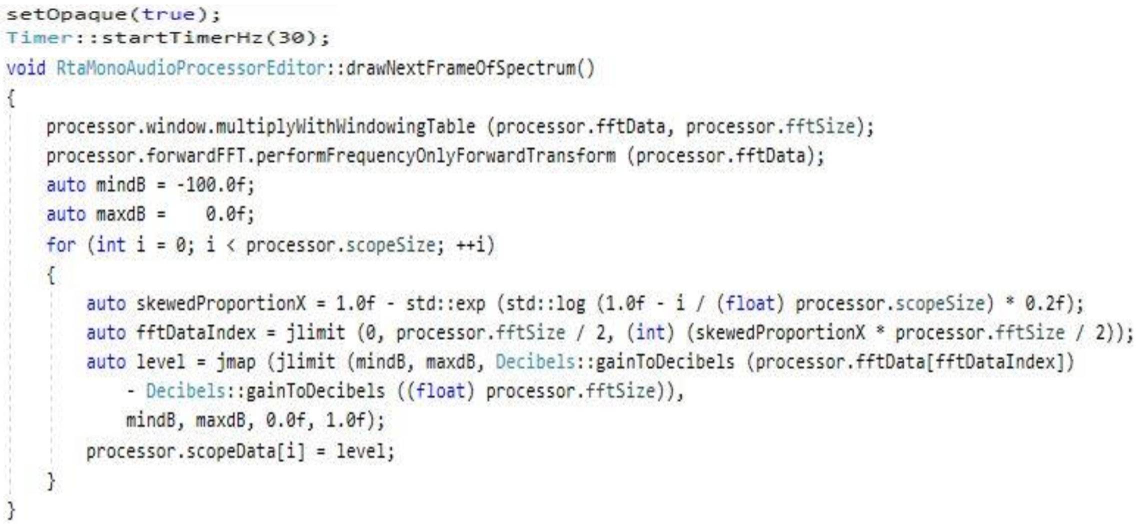
Para realizar la transformada de Fourier, se utilizó un paradigma de *JUCE* llamado “*DSP”* por sus siglas en inglés “*Digital Signal Processing*”, el cual tiene muchas funciones de procesamiento entre las cuales se consideran varias funciones como las transformadas de Fourier. Al ser un plugin en el cual no se modifica la señal de audio, solamente se utilizan los valores reales de la transformada, para lo cual, se utilizó una función de “*DSP*” que realiza esta acción optimizando procesamiento al no calcular valores complejos. Antes de realizar la transformada se vacía el último valor utilizado y se le asigna un índice de 0 al arreglo de datos y se coloca la muestra de audio actual en el índice siguiente. Es ahí donde se realiza la transformada requerida. La transformada calculada fue de orden 11, lo cual permite un total de 2048 muestras para realizarse, siendo este número el doble del máximo tamaño del buffer disponible en la mayoría de los softwares de audio. Se aplicaron funciones de ventanas en la señal antes de realizar las transformadas de Fourier para evitar la fuga espectral. La función de ventana utilizada fue la de Hann, con una resolución decente y un rango dinámico bastante bueno. El número de puntos de visualización del espectro fue de 512, con lo cual se tiene una buena representación sin consumir mucho procesamiento.

En la interfaz de usuario se escaló toda la longitud vertical desde -100 dBFS (decibelios de fondo de escala) a 0 dBFS, es decir, se representan niveles absolutos en lugar de relativos, como lo realizan otros módulos con analizadores de espectro. Para la visualización se ubicó una escala que muestra diferentes niveles en el eje x y en el eje x para mostrar las frecuencias centrales de las bandas de octava de 125 Hz a 8 kHz. El llamado del temporizador se implementó en la parte de la interfaz para que se dibuje nuevamente el espectro de la señal cada vez que este suceda. En *plugins* de procesamiento de audio es poco recomendables asignar memoria del procesador a la interfaz debido a que los datos de dicha memoria se pueden bloquear y causar problemas de acceso, sin embargo, como este módulo no es de procesamiento sino solamente de visualización, este criterio no representa un problema. A continuación, se muestran los algoritmos correspondientes al desarrollo de este módulo:



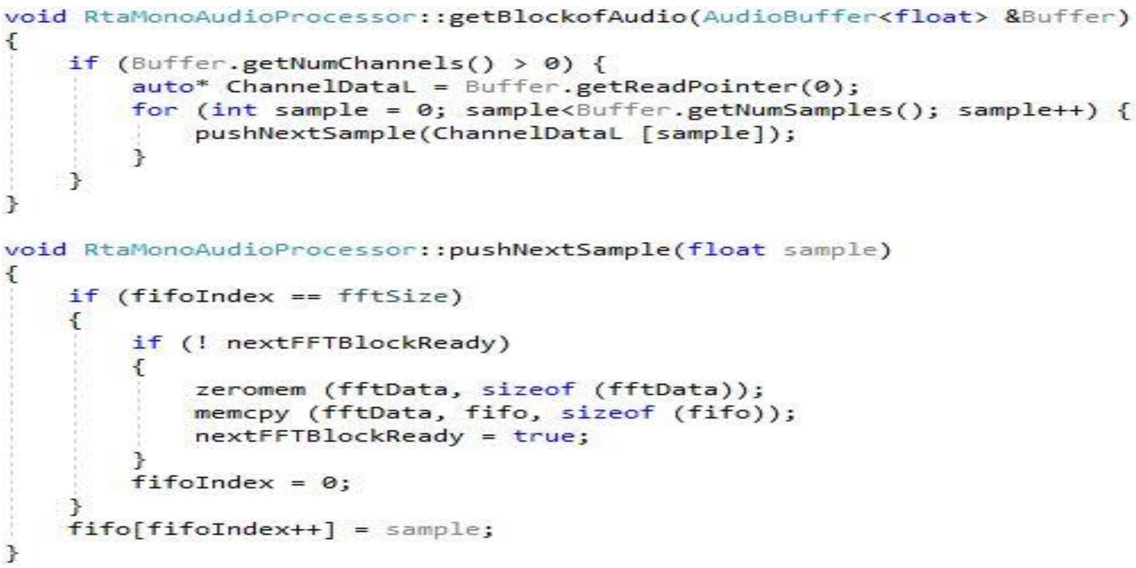
```
enum
{
    fftOrder  = 11,           // Orden de la transformada
    fftSize   = 1 << fftOrder, // Desplaza el n° de bits a la derecha
    scopeSize = 512// Puntos de visualización
};
dsp::FFT forwardFFT { 11 };                 // Objeto de transformada de Fourier
dsp::WindowingFunction<float> window { fftSize, dsp::WindowingFunction<float>::hann };//Función de ventana
float fifo [fftSize];//Arreglo de datos
float fftData [2 * fftSize];          //Arreglo de datos
int fifoIndex = 0;                    //índice de datos
bool nextFFTBlockReady = false;       // Indicador de estado de bloque
float scopeData [scopeSize];
AudioBuffer<float> SoundBuffer;
/*Prepare to play*/
SoundBuffer.clear();
SoundBuffer.setSize(getTotalNumOutputChannels(), getBlockSize());
/*Process Block*/
SoundBuffer.makeCopyOf(buffer);
getBlockofAudio(SoundBuffer);
```

Figura 19. Declaración de variables, objetos, y copia del buffer de audio en el procesador del analizador de espectro



```
setOpaque(true);
Timer::startTimerHz(30);
void RtaMonoAudioProcessorEditor::drawNextFrameOfSpectrum()
{
    processor.window.multiplyWithWindowingTable (processor.fftData, processor.fftSize);
    processor.forwardFFT.performFrequencyOnlyForwardTransform (processor.fftData);
    auto mindB = -100.0f;
    auto maxdB =    0.0f;
    for (int i = 0; i < processor.scopeSize; ++i)
    {
        auto skewedProportionX = 1.0f - std::exp (std::log (1.0f - i / (float) processor.scopeSize) * 0.2f);
        auto fftDataIndex = jlimit (0, processor.fftSize / 2, (int) (skewedProportionX * processor.fftSize / 2));
        auto level = jmap (jlimit (mindB, maxdB, Decibels::gainToDecibels (processor.fftData[fftDataIndex])
            - Decibels::gainToDecibels ((float) processor.fftSize)),
            mindB, maxdB, 0.0f, 1.0f);
        processor.scopeData[i] = level;
    }
}
```

Figura 20. Algoritmo correspondiente al procesador del módulo analizador de espectro que prepara la señal para realizar la transformada de Fourier.



```
void RtaMonoAudioProcessor::getBlockofAudio(AudioBuffer<float> &Buffer)
{
    if (Buffer.getNumChannels() > 0) {
        auto* ChannelDataL = Buffer.getReadPointer(0);
        for (int sample = 0; sample<Buffer.getNumSamples(); sample++) {
            pushNextSample(ChannelDataL [sample]);
        }
    }
}

void RtaMonoAudioProcessor::pushNextSample(float sample)
{
    if (fifoIndex == fftSize)
    {
        if (! nextFFTBlockReady)
        {
            zeromem (fftData, sizeof (fftData));
            memcpy (fftData, fifo, sizeof (fifo));
            nextFFTBlockReady = true;
        }
        fifoIndex = 0;
    }
    fifo[fifoIndex++] = sample;
}
```

Figura 21. Algoritmo de la interfaz del módulo analizador de espectro.

Para las señales estereofónicas se estableció parámetros para seleccionar si se desea representar el espectro de cada canal o de los dos canales unidos. La razón para realizar una copia del buffer de audio en lugar de procesar los datos directamente de este es para evitar que ingresen ruidos y datos indeseados al buffer de la señal original a causa de dicho procesamiento generado por los cálculos de las transformadas realizadas.

3.14 Limitaciones de diseño en el analizador

Cabe resaltar que este módulo, no constituye una herramienta de precisión exacta de medición de nivel de audio, ya que no incluye algoritmo de detección de “verdaderos picos” de la señal, sino que muestra los niveles de espectro cada vez que el temporizador llega a cero y solicita la información de audio. Esto significa que la precisión de este módulo es de 33,3 milisegundos, y cualquier valor de la amplitud del espectro que varíe durante un espacio de tiempo menor puede llegar a no considerarse dentro de los cálculos de las transformadas.

Dentro de la etapa de diseño, se probó con un tiempo del temporizador de 16,6 milisegundos (60 cuadros por segundo), con lo cual se esperaba una mejor visualización de los datos, sin embargo, en las primeras pruebas de prototipos se observó que no existía una considerable diferencia en comparación con la versión del módulo que trabaja con el temporizador a 33,3 milisegundos por lo cual se optó por implementar uno de 30 cuadros por segundo. Una limitación importante puede ser que este módulo solamente muestra los valores que recibe del procesador al realizar la transformada de Fourier, pero no incluye opciones para mostrar valores RMS, ni otros que se rigen con estándares de ciertas normas internacionales de medición del nivel digital de audio (ITU, EBU, ATSC), siendo esta una función característica en algunos *plugins* disponibles en el mercado, como el *WLM Plus Meter Loudness Meter* de la compañía de *Waves Ltd*.

3.15 Optimización y limitaciones generales de todos los módulos

La primera manera en la que se optimizó el procesamiento fue al separar los efectos, que, en primera instancia, se diseñaron para ser procesados en un solo módulo. La principal razón para separarlos fue que, al estar en conjunto, los efectos consumían demasiado procesamiento causando que el módulo que los contenía no fuese funcional, ya que, al ejecutarlo en distintos softwares de audio, el agotamiento de recursos computacionales era mayor al 100 % produciendo que dichos programas colapsaran e incluso se cerraran automáticamente. Al separar los efectos en diferentes módulos, se redujo el procesamiento de forma mínima por lo que se tuvo que realizar un análisis de la codificación de estos para encontrar una manera de realmente optimizarlos. Con los efectos separados en diferentes módulos, se intentó reducir el consumo de procesamiento al aumentar la latencia individual por procesamiento de cada uno de estos. Se probaron valores de latencia de hasta 2 segundos, sin embargo, el procesamiento no tuvo una reducción notable.

Después de una larga revisión de los algoritmos de cada módulo, se determinó que el problema de procesamiento se producía debido a que los parámetros de cada *plugin* se estaban actualizando muestra por muestra incluso cuando no existían cambios en estos. Por esta razón se decidió que los parámetros se actualicen cada vez que ingrese un paquete de datos al bloque de procesamiento de cada *plugin*. Esto significa que la precisión de los módulos está limitada al tamaño del buffer determinado por el software anfitrión que los esté ejecutando. Al realizar este ajuste se redujo el procesamiento a valores aceptables para que los módulos sean funcionales.

Con los cambios realizados, si bien el consumo de procesamiento fue óptimo, los módulos que incluyen parámetros temporales perdieron precisión ya que, al actualizarlos cada vez en cada bloque de audio, los valores de tiempo que indican sus parámetros no son correctos ni exactos. Por esta razón se decidió que los parámetros temporales como el tiempo de retardo en el módulo de *Delay* y los tiempos de ataque y relajación del compresor se actualicen muestra a muestra, como lo hacían en el diseño original. A pesar de que esto aumentó ligeramente el consumo de procesamiento de estos *plugins*, estos siguieron estables y funcionales, además, se incrementó la precisión de estos parámetros temporales para que los valores mostrados por estos sean exactos y su error sea máximo de una muestra de audio. Al eliminar la latencia anteriormente aplicada, el consumo de recursos computacionales no aumentó, por lo que se decidió dejar a todos los módulos con una latencia de 0 muestras relativas, resultando así en *plugins* optimizados funcionales sin latencia, listos para la fase de pruebas.

Se decidió mantener los efectos separados en cada módulo debido a que además de consumir menos procesamiento que cuando estaban unidos en un solo módulo, también se determinó que al estar separados se tenía un mayor y mejor control de cada efecto, ya que, es posible controlar una ganancia de entrada y salida independiente de cada efecto, así como invertir su polaridad.

Además, se puede utilizar solamente los módulos de los efectos que se deseen de acuerdo con las necesidades requeridas por el usuario.

3.16 Diseño y desarrollo de las interfaces de usuario

Se creó un modelo de interfaz general para todos los *plugins* desarrollados, el cual se basa en un fondo de color negro, los controles deslizantes o "*sliders*" de control de ganancia se dispusieron a manera de *faders* verticales en una escala logarítmica que comprende un rango desde -100 dB hasta 12 dB. Para los módulos de señales estereofónicas, en la sección de la ganancia de entrada se colocó un control a manera de botón para sincronizar los *faders* de entrada o los de salida. Al activar la sincronización de los *sliders* de la ganancia de entrada, se configuraron los parámetros correspondientes para que también se sincronicen los controles de inversión de polaridad de cada canal para que al activar o desactivar dicha inversión en un canal muestran la sincronización de la ganancia de entrada esté encendida, también se active o desactive la inversión de polaridad del otro canal. En la sección central de la interfaz de usuario se insertaron sliders horizontales con los parámetros de cada uno de los efectos correspondientes a cada módulo. Al igual que en la sección de ganancia de entrada y salida, se colocaron controles para sincronizar los canales cuando se trabaja con señales estereofónicas, pero también se añadieron controles para seleccionar el canal con el que se desee trabajar y así poder modificar los parámetros del efecto implementado en cada canal individualmente.

En cuanto al tamaño de la interfaz, al igual que en la mayoría de *plugins* disponibles en el mercado, esta no fue diseñada para ser escalable, es decir, para modificar su tamaño, sin embargo, se establecieron dos tamaños estándar que varían de acuerdo con el tamaño de la pantalla del ordenador que los ejecute. Se tomaron como referencia las dimensiones de los computadores en los cuales se desarrollaron los módulos. Esto se muestra a continuación:

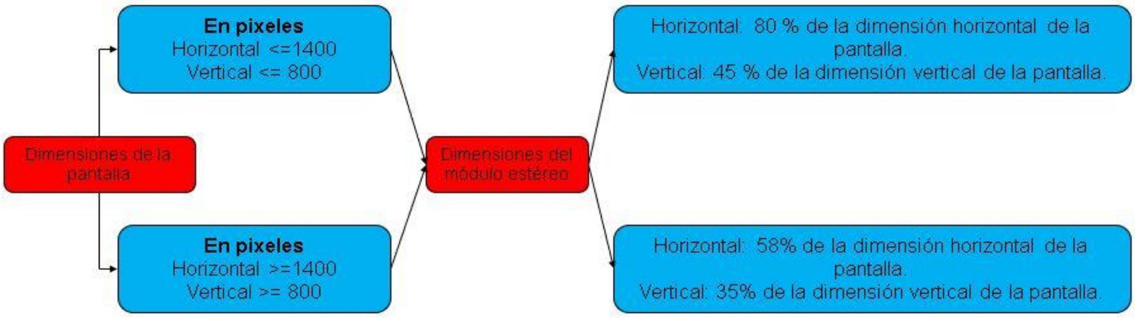


Figura 22. Dimensiones de la interfaz de usuario para los módulos para señales estereofónicas.

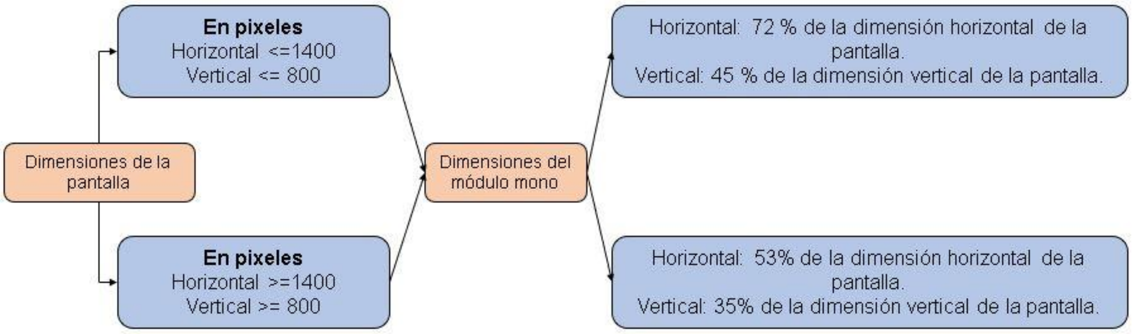


Figura 23. Dimensiones de la interfaz de usuario para los módulos para señales monofónicas.

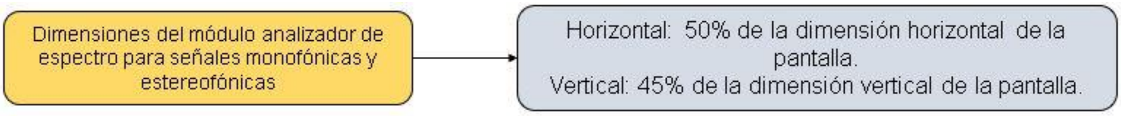


Figura 24. Dimensiones de la interfaz de usuario del analizador de espectro.

Como se puede observar en las figuras 21, 22 y 23, se utilizaron distintas dimensiones para los módulos de señales estereofónicas, monofónicas y para el analizador de espectro. En todos los módulos se añadieron etiquetas para los *sliders* y elementos de la interfaz con una fuente llamada “*Snell Roundhand*”.

Esta fuente viene cargada dentro del sistema operativo *MacOS*, sin embargo, para *Windows*, no está disponible de fábrica, pudiendo descargarla libremente de internet e instalarla de manera sencilla. Esta fuente, desde su diseño es cursiva y fue seleccionada para mostrar un estilo elegante en la interfaz de los *plugins* desarrollados. Las etiquetas fueron colocadas en inglés debido a que es uno de los lenguajes más universales que existen y la mayoría de las compañías de desarrollo de software de audio integran a este como el idioma único o principal de sus productos.

A cada módulo se le asignaron distintos colores para sus elementos con el fin de que las interfaces sean más llamativas y las palabras incluidas en las mismas sean legibles y entendibles. Para controlar el encendido y apagado de cada módulo se utilizó un botón que controla un parámetro booleano. En el estado apagado de todos los *plugins* sus elementos toman colores en distintos matices de gris para mostrar de manera visual que están apagados. En los módulos en estéreo, se diseñó la interfaz para poder controlar los parámetros de cada canal (LR) de manera individual o conjunto, dando la posibilidad de usos comunes (mezcla, masterización), así como para otras aplicaciones como el diseño sonoro.

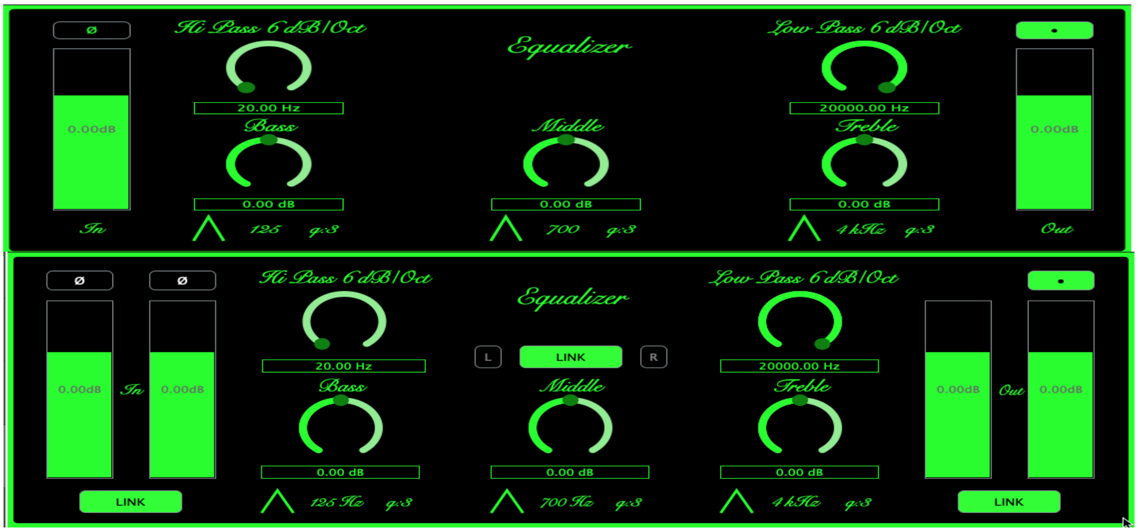


Figura 25. Interfaces de usuario de los módulos de ecualización para señales mono y estéreo.

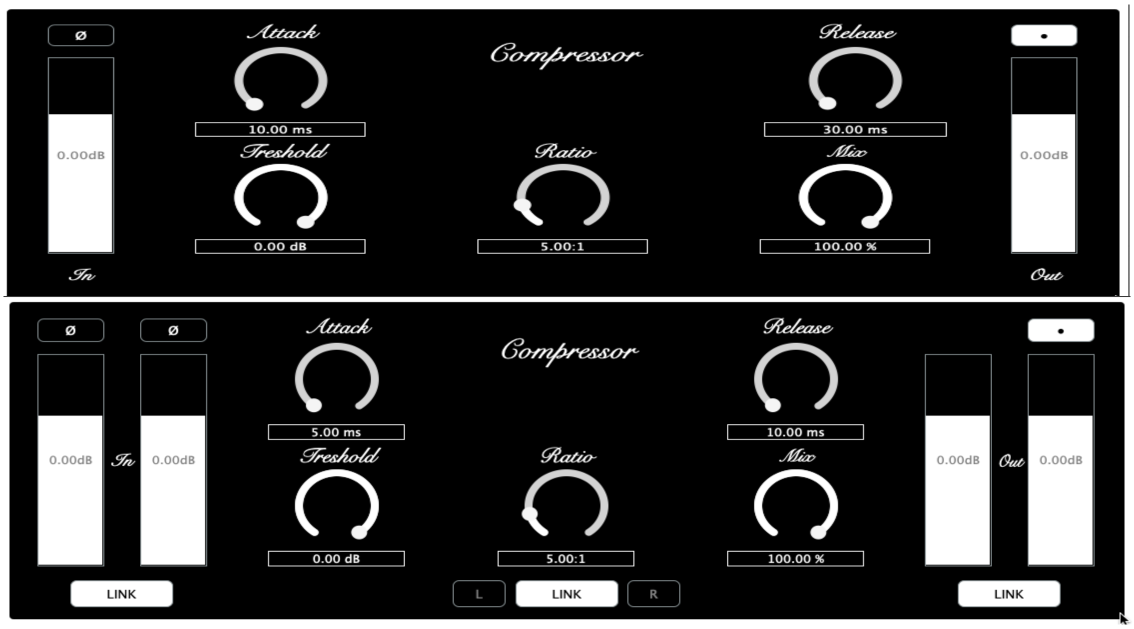


Figura 26. Interfaces de usuario de los módulos de compresión para señales mono y estéreo.

Figura 27. Interfaces de usuario de los módulos de retardo o *Delay* para señales mono y estéreo.

Figura 28. Interfaces de usuario de los módulos de reverberación para señales mono y estéreo.

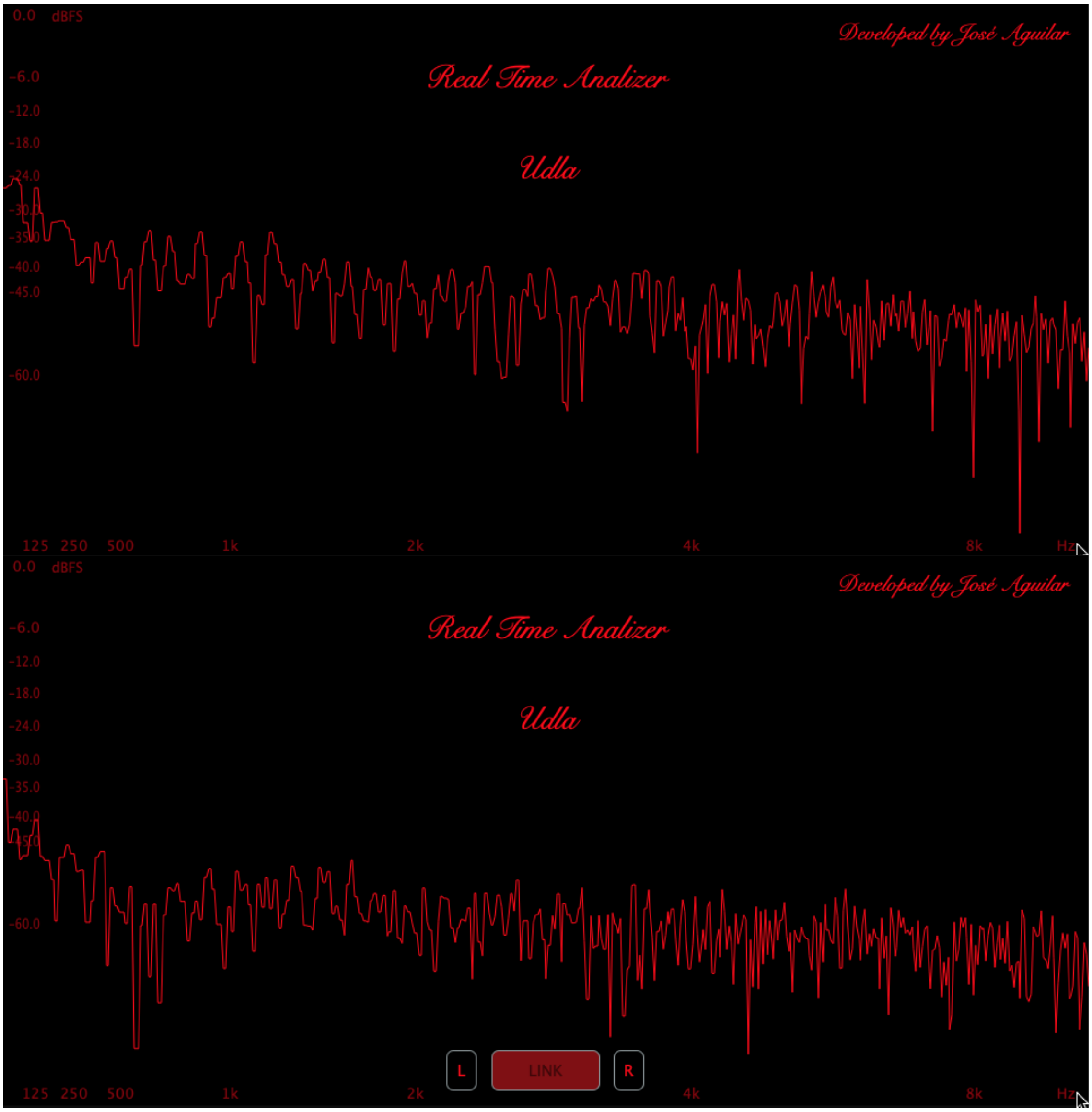


Figura 29. Interfaces de usuario de los módulos analizadores de espectro para señales mono y estéreo.

3.16.1 Sincronización del procesador con interfaces de usuario

Como ya se mencionó en la sección del diseño y desarrollo del analizador de espectro, no es recomendable asignar la memoria del procesador de un *plugin* a la de la interfaz, por esta razón en los módulos que si realizan procesamiento de las señales de audio se utilizaron varias clases del marco de referencia utilizado para sincronizar los parámetros del procesador con los distintos *faders*, *sliders* y botones de las interfaces.

La principal y más importante clase utilizada para este fin tiene el nombre de "*Audio Processor Value Tree State*", cuya traducción vendría a significar como el estado del árbol de valores del procesador de audio. Entonces, como su nombre lo indica, esta clase ayuda a pasar solamente el estado de los parámetros del procesador hacia la interfaz sin asignar la memoria desde el procesador. Esta clase permite crear los parámetros del procesador mediante la construcción de un vector de rango, en el que se pueden ingresar todos los tipos de parámetros del procesador, como booleanos, flotantes, dobles y enteros. Para evitar que, sea la interfaz la que acceda directamente a la memoria del procesador para solicitar los valores de sus parámetros, en este árbol de estado se crean punteros automáticamente, los cuales solamente hacen referencia a los valores de dichos parámetros, es decir, esta clase ayuda a crear una especie de puente de conexión segura y estable entre el procesador y la interfaz de cada módulo. El funcionamiento correspondiente se detalla de manera más clara a continuación:

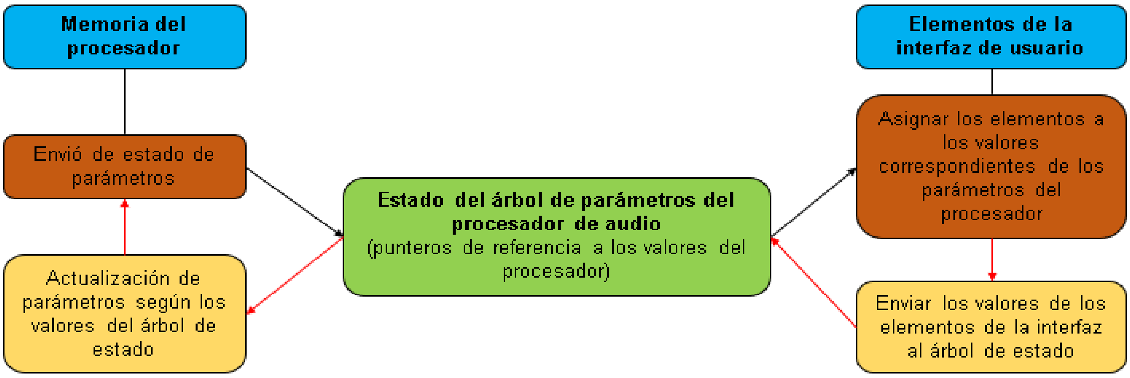


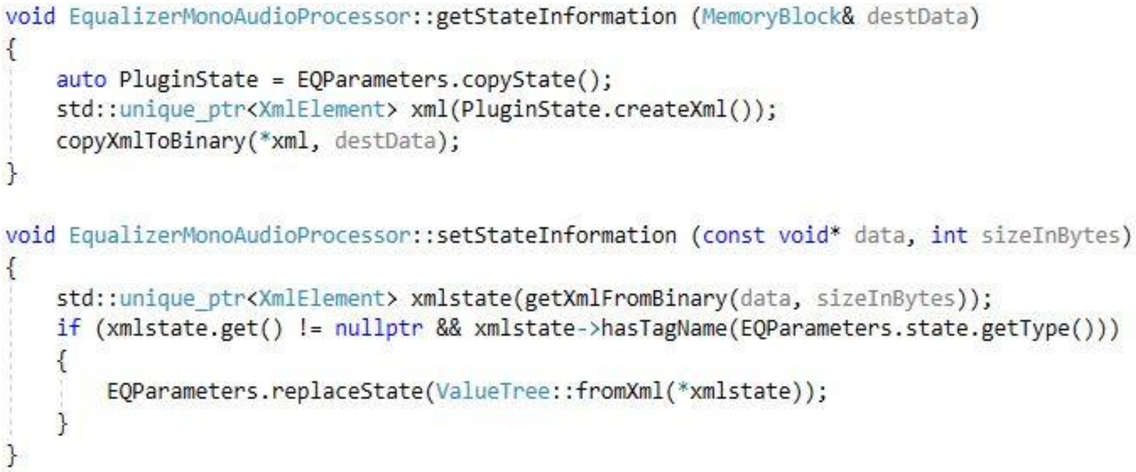
Figura 30. Funcionamiento de la clase de árbol de estado de parámetros.

El árbol de estado mencionado se actualiza cada vez que se modifican los valores correspondientes a sus parámetros, ya sea desde el procesador o desde la interfaz, lo cual permite que, si se modifica algún parámetro del procesador, la interfaz se actualizará inmediatamente y viceversa. Gracias a esto, se logró tener una gran estabilidad en el funcionamiento de los módulos desarrollados, además que permitió que los parámetros del procesador sean automatizables. Es importante mencionar, que, si bien el diseño visual de la interfaz fue original, este tuvo ciertas bases en los principios de simetría, así como en la psicología del color, pero fundamentalmente, se cimentó en el análisis de muchas interfaces de módulos de terceros para lograr desarrollar un modelo de interfaz que sea lo más intuitiva posible para un uso fácil y fluido.

3.17 Obtención del estado de los módulos

Con todo lo mencionado el funcionamiento de los módulos está completo, sin embargo, hay una consideración adicional que realizar. Es muy común utilizar *plugins* de audio en sesiones del software anfitrión que los esté ejecutando, y muchas veces, esos programas se cierran para continuar su trabajo en otro momento. Para esto, es necesario que el módulo entregue información de su estado en el momento que este se cierre, para que al abrir la sesión o el programa nuevamente, el estado del *plugin* sea el cual con el que se cerró el software. Lamentablemente no se puede realizar con el árbol de estado de parámetros, ya que, este solamente actualiza la información de estado dentro del módulo y no fuera de este.

Para enviar la información de estado hacia el software anfitrión, se creó un archivo *xml*, el cual guarda la meta data de los valores de los parámetros del procesador. Mediante la función “*getStateInformation*”, la cual viene por defecto al crear un módulo de audio con *JUCE*, se crea el archivo *xml* y se envía su información al anfitrión cuando la sesión que esté abierta se cierra o cuando se guarda dicha sesión. Al momento de reabrir el programa anfitrión el módulo solicita la información del archivo para actualizar sus parámetros mediante la función “*setStateInformation*”. Este intercambio de datos entre el anfitrión y el procesador viene a ser similar al del árbol de estado de parámetros, solamente que, en lugar de servir de conexión entre el software procesador y la interfaz, lo hace entre el módulo y el software anfitrión. Con estos ajustes también se crea la posibilidad de guardar y cargar bancos o “*presets*” según lo requiera el usuario. La implementación de esta parte del desarrollo de los módulos se ejemplifica en la siguiente figura:



```
void EqualizerMonoAudioProcessor::getStateInformation (MemoryBlock& destData)
{
    auto PluginState = EQParameters.copyState();
    std::unique_ptr<XmlElement> xml(PluginState.createXml());
    copyXmlToBinary(*xml, destData);
}

void EqualizerMonoAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    std::unique_ptr<XmlElement> xmlstate(getXmlFromBinary(data, sizeInBytes));
    if (xmlstate.get() != nullptr && xmlstate->hasTagName(EQParameters.state.getType()))
    {
        EQParameters.replaceState(ValueTree::fromXml(*xmlstate));
    }
}
```

Figura 31. Función de establecimiento y obtención del estado del procesador hacia el software anfitrión.

3.18 Consideraciones finales de diseño

Cada módulo fue compilado para ser ejecutado en procesadores de 64 bits únicamente, tanto para el sistema operativo *Windows* como para *MacOS*. En la siguiente sección se detallan las pruebas realizadas, junto con sus resultados y análisis correspondientes. La razón por la cual no se desarrolló una versión de los módulos en formato AAX (*Avid Audio Extension*) o *Audio Suite*, fue porque la compañía que desarrolló este formato no entrega sus herramientas de desarrollo de software sin una licencia específica, la cual, no se otorga para fines académicos, sino solamente para fines comerciales, siendo posible acceder a ella solamente en representación de una empresa de desarrollo de software de audio. Sin embargo, existen herramientas disponibles en la industria del audio que permiten abrir *plugins* en formato *VST* o *VST3* en el programa “*Pro Tools*”, el cual es el único programa que solamente trabaja con módulos en el formato antes mencionado. Un ejemplo de una herramienta que realiza esta acción es un *plugin* llamado “*Blue cat patch work*”, desarrollado por la compañía *Blue Cat Audio*. En la sección de Anexos se adjuntan todos los algoritmos completos de cada uno de los módulos desarrollados.

4 CAPÍTULO IV: RESULTADOS Y ANÁLISIS

Dentro de la etapa de diseño y desarrollo de cada uno de los módulos, se realizaron pruebas de prototipo que permitieron depurar los fallos iniciales que se tenían en estos, así como optimizarlos para que sean estables y funcionales. Sin embargo, con las versiones definitivas de cada *plugin* se realizaron dos tipos de pruebas reales, las cuales fueron: validez, compatibilidad y consumo de procesamiento en estaciones de trabajo de audio digital (*DAW*s) y de medición de características de funcionamiento. Cada una de estas pruebas se realizó en los dos sistemas operativos para los cuales fueron diseñados los módulos y su detalle se presenta posteriormente. Todas las pruebas se realizaron en los ordenadores en los que se desarrollaron y compilaron todas las versiones de los módulos con sus formatos respectivos, y en su ejecución, se utilizaron dos interfaces de audio tanto para el sistema operativo *Windows* como para MacOS, las cuales fueron una *Universal Audio Apollo Twin MKII* de dos canales, y una *M Audio M-Track Eight* de ocho canales. Estás interfaces fueron utilizadas con el fin de poder alcanzar valores altos de frecuencia de muestreo al realizar las distintas pruebas en los módulos desarrollados. La conexión de la interfaz de *Universal Audio* es a través de un cable *Thunderbolt*, por lo tanto, para trabajar con ella en el sistema operativo el ordenador con sistema operativo *Windows* fue necesario utilizar un conector de conversión de *Thunderbolt* a *HDMI*. En cambio, en la otra interfaz usada, la conexión se realiza mediante cable *USB* 3.0, para la cual no se necesitó ningún tipo de conector.

4.2 Pruebas de validez, compatibilidad y consumo de procesamiento en estaciones de trabajo de audio digital (*DAW*s)

Para verificar que la versión correspondiente a cada uno de los formatos en los que se diseñó cada módulo fuese válida, se probó su ejecución en 4 *DAW*s para el sistema operativo *MacOS* y en 3 para el sistema operativo de *Windows*. En general, estas pruebas consistieron en verificar que cada uno de los módulos funcione de manera correcta en las distintas estaciones de trabajo de audio digital, y en cuantificar el procesamiento que estos consumían. En cada uno de los programas utilizados como anfitriones se probaron las versiones de los *plugins* que fueran compatibles con estos como se muestra en la tabla 1.

Tabla 1.

DAWs utilizados para las pruebas de funcionamiento y compatibilidad.

Windows 10 Pro-2019			**MacOS Mojave 10.14.6**			
Ableton Live Suite 10.0.6	**Reason 10.4d3**	**Cubase Elements 8.1.2**	**Ableton Live Suite 10.1.6**	**Reason 10.0d72**	**Logic Pro 10.4.1**	**Cubase Elements 8.0.4**
VST	VST	VST3	VST	VST	Audio Unit	VST3
Formato de plugin probado						

En todos estos programas mostrados en la tabla 1, las pruebas se realizaron de manera idéntica, y consistieron en utilizar varias señales de audio en cada módulo, como ruido rosa, tonos puros y archivos de audio musicales, para determinar los cambios en el procesamiento de distintos tipos de señales sonoras, sin embargo, no se notaron diferencias importantes al variar las señales sonoras en ninguno de los programas utilizados ni en ninguno de los módulos. Como los valores del porcentaje de procesamiento son fluctuantes, se tomaron los máximos que arrojaron las pruebas, sin importar el tipo de señal, evaluando de esta manera los peores escenarios posibles en los que los *plugins* desarrollados pueden ser utilizados.

Primeramente, se evaluaron los valores de procesamiento del CPU que consumían los programas anfitriones sin ejecutar ninguna instancia de los módulos en cuestión para poder determinar el consumo de las acciones intrínsecas de los procesos realizados por los anfitriones. Cabe mencionar que tampoco se ejecutó ningún otro programa además del software anfitrión al realizar esta evaluación inicial, obteniendo valores distintos para cada *DAW* y para cada sistema operativo utilizado (*Tabla 2*).

Tabla 2.

Procesamiento de las funciones intrínsecas de los programas anfitriones sin la ejecución de ninguno de los plugins desarrollados.

Windows 10 Pro-2019			**MacOS Mojave 10.14.6**			
Ableton Live 10	**Reason**	**Cubase**	**Ableton Live 10**	**Reason**	**Logic Pro**	**Cubase**
4,7%	4,0%	2,1%	4,5%	3,8%	3,1%	2,3%

De acuerdo con la tabla 2, el consumo de procesamiento intrínseco de cada *DAW* está por debajo del 5% para los dos sistemas operativos utilizados.

En todas las pruebas realizadas que tienen que ver con consumo de procesamiento, la interfaz de usuario de cada módulo estuvo abierta para que se pudiera considerar los escenarios más exigentes en cuando a los recursos computacionales utilizados. Con respecto al funcionamiento de los módulos desarrollados, en primera instancia se realizaron dos pruebas generales. La primera consistió en colocar una señal sonora en un canal de audio del software anfitrión y ejecutar los 5 módulos en serie para verificar su funcionalidad y la cantidad de procesamiento que estos consumen. Es importante mencionar que se analizó el procesamiento consumido tanto de manera interna en cada *DAW* utilizado, como el general del procesador (CPU y GPU) del ordenador. Esta prueba se realizó con distintas combinaciones de frecuencia de muestreo (44100 – 192000) y tamaños del buffer de procesamiento (128 - 2048), tanto para señales monofónicas como estereofónicas (*Figura 32 - 35*).

Figura 32. Consumo de procesamiento interno en cada *DAW* al ejecutar los 5 *plugins* mono en serie en un mismo canal con una señal de audio monofónica.

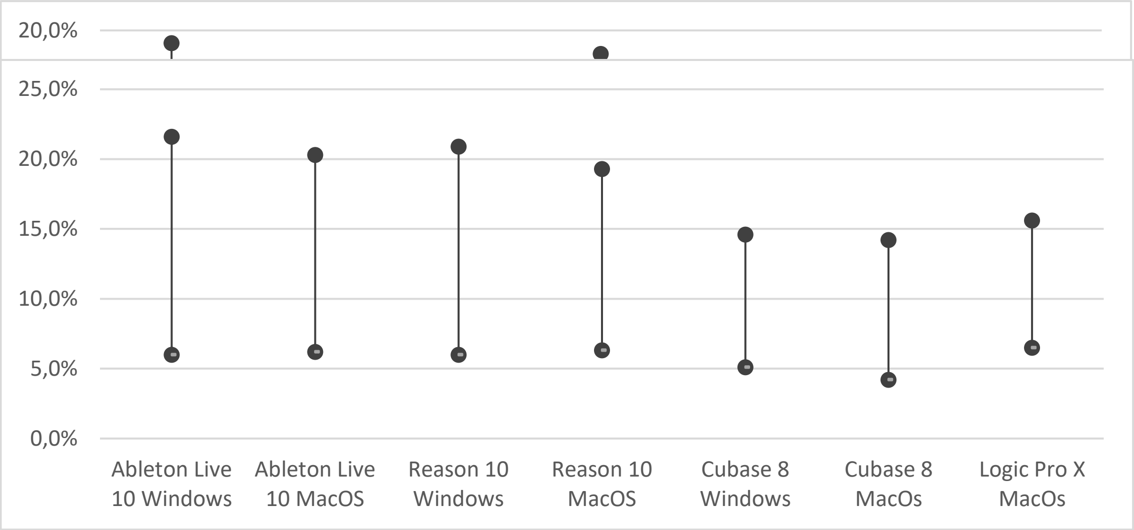


Figura 33. Consumo de procesamiento interno en cada *DAW* al ejecutar los 5 *plugins* estéreo en serie en un mismo canal con una señal de audio estereofónica.

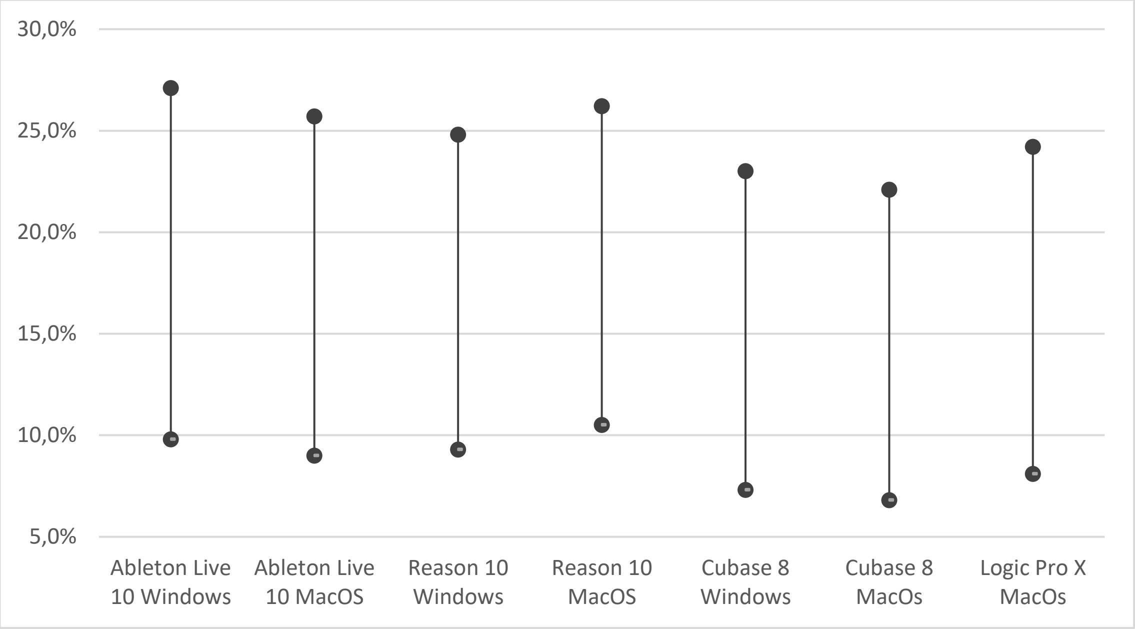


Figura 34. Consumo de procesamiento general del procesador del ordenador al ejecutar los 5 *plugins* mono en serie en un mismo canal con una señal de audio monofónica en cada uno de los *DAW*s utilizados.

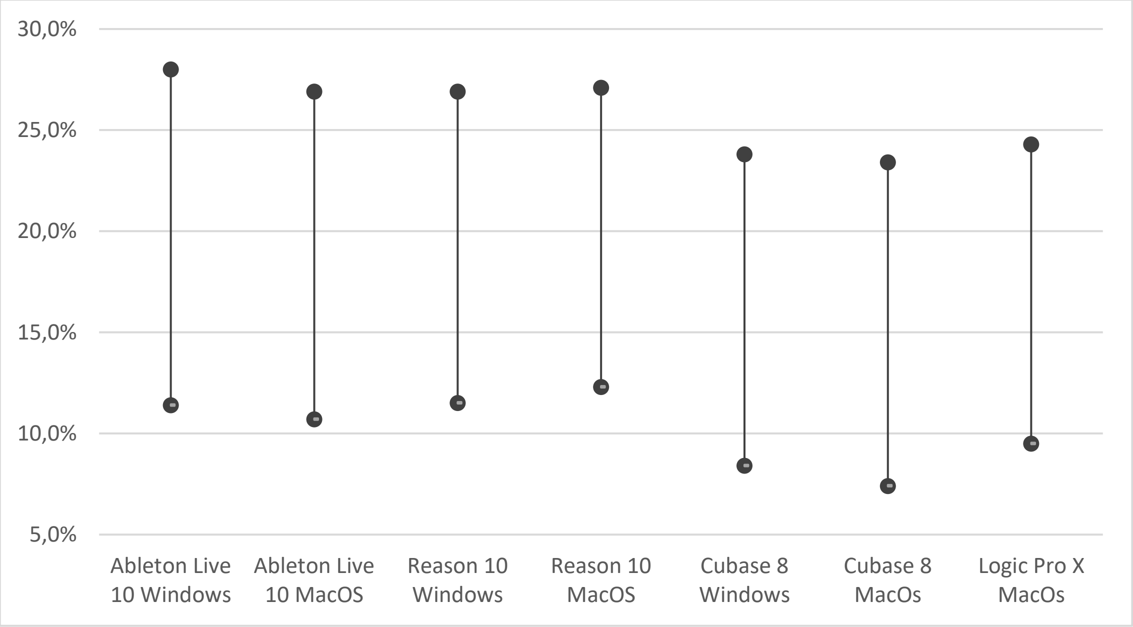


Figura 35. Consumo de procesamiento general del procesador del ordenador al ejecutar los 5 *plugins* estéreo en serie en un mismo canal con una señal de audio estereofónica en cada uno de los *DAW*s utilizados.

En general, los valores de porcentaje del procesamiento consumido para esta prueba en todos los anfitriones utilizados son similares tanto de manera interna para cada *DAW* como en general para el procesador del ordenador, pero existen ciertas diferencias que se pueden analizar.

Como se puede observar en las figuras 32 – 35, n el software Ableton Live 10, los valores de procesamiento de esta prueba son muy similares tanto internamente como como en el CPU, siendo el sistema operativo *MacOS* ligeramente más óptimo que *Windows* para este caso. En cambio, en Reason 10, el sistema operativo de *Windows* presenta valores de procesamiento ligeramente menores que el sistema operativo *MacOS*, sin embargo, los dos son bastantes similares. Nuevamente, en Cubase 8, el sistema operativo *MacOS* presenta valores inferiores a los de *Windows*, e incluso, en este anfitrión, en general los valores son menores que en los otros 2 programas. Esto puede deberse a que en Cubase utiliza los formatos VST3 de los *plugins*, los cuales son los más actuales de los implementados en los módulos desarrollados, y por lo tanto están más optimizados por defecto. Los valores de esta prueba realizada en Logic Pro X (solo *MacOS*), fueron superiores a los obtenidos en Cubase, pero inferiores a los de Ableton Live 10 y Reason 10, lo cual muestra la eficiencia de cada uno de los formatos de los módulos desarrollos, lo cual no tiene nada que ver con el diseño en sí de estos, sino que son propiedades que han sido establecidas por los desarrolladores y dueños de cada formato de *plugin* mencionado. También es importante mencionar que el consumo de procesamiento en el procesador del ordenador (CPU Y GPU) es mayor al interno de cada *DAW*, lo cual se puede deber a que en el procesador general se toma en cuenta los valores intrínsecos que consume cada anfitrión además del procesamiento de los *plugins* probados (cf. *Tabla 2.*), mientras que los valores internos solamente muestran el procesamiento real de los módulos. En esta primera prueba todos los valores de procesamiento fueron menores a 30 %, mostrando un bajo consumo de recursos bajo las condiciones de la misma.

La segunda prueba general consistió en simular uno de los escenarios reales en los que se pueden utilizar los módulos, una sesión de trabajo de post producción. Para esto se crearon 15 canales de audio en cada uno de ellos se colocaron 5 instancias los 5 *plugins* desarrollados, es decir, un total de 75 instancias de estos módulos*.* Esto se realizó con el fin de llevar al límite a los sistemas, verificar la estabilidad de los módulos y evaluar escenario de trabajo muy exigentes. De igual manera que la primera prueba, esta también se realizó para señales monofónicas y estereofónicas en cada anfitrión utilizado con combinaciones de frecuencia de muestreo (44100 – 192000) y tamaño del búfer de procesamiento (256 - 2048).

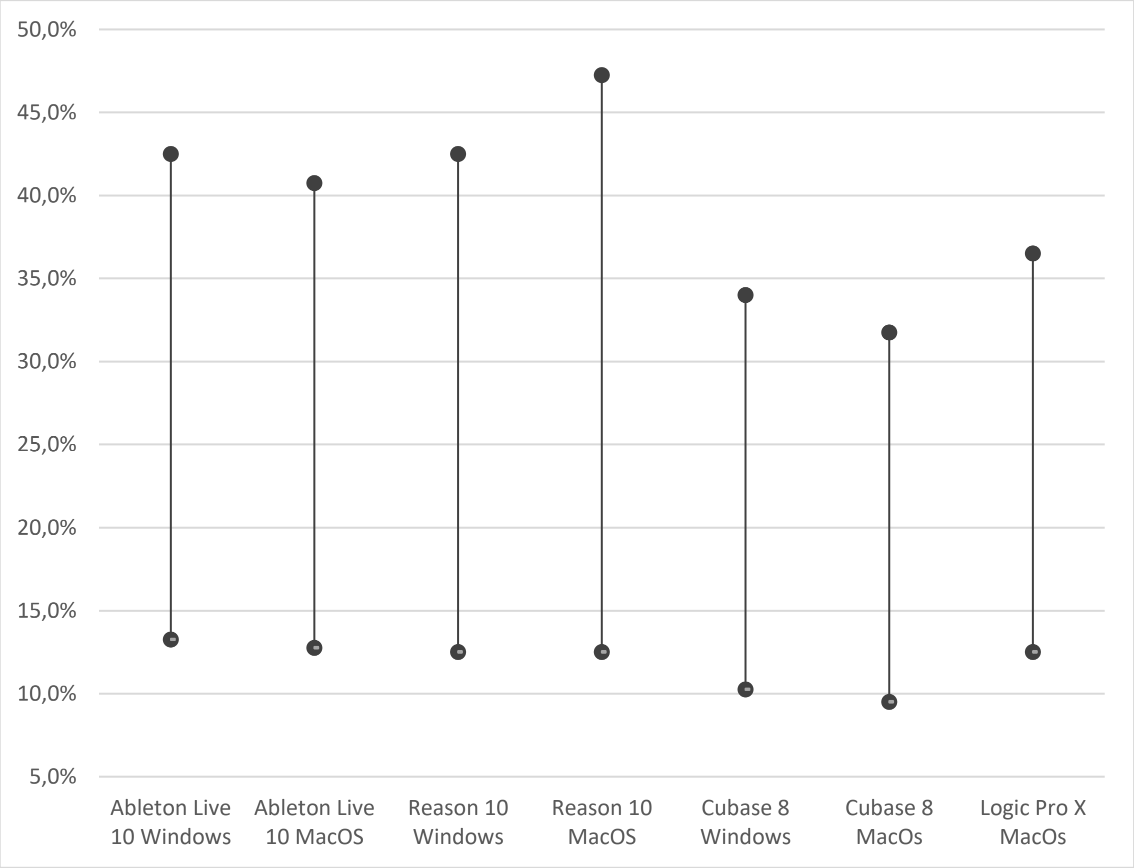


Figura 36. Consumo de procesamiento interno en cada *DAW* al ejecutar los 5 *plugins* mono en serie en 15 canales con señales de audio monofónicas (75 instancias de *plugins* en total).

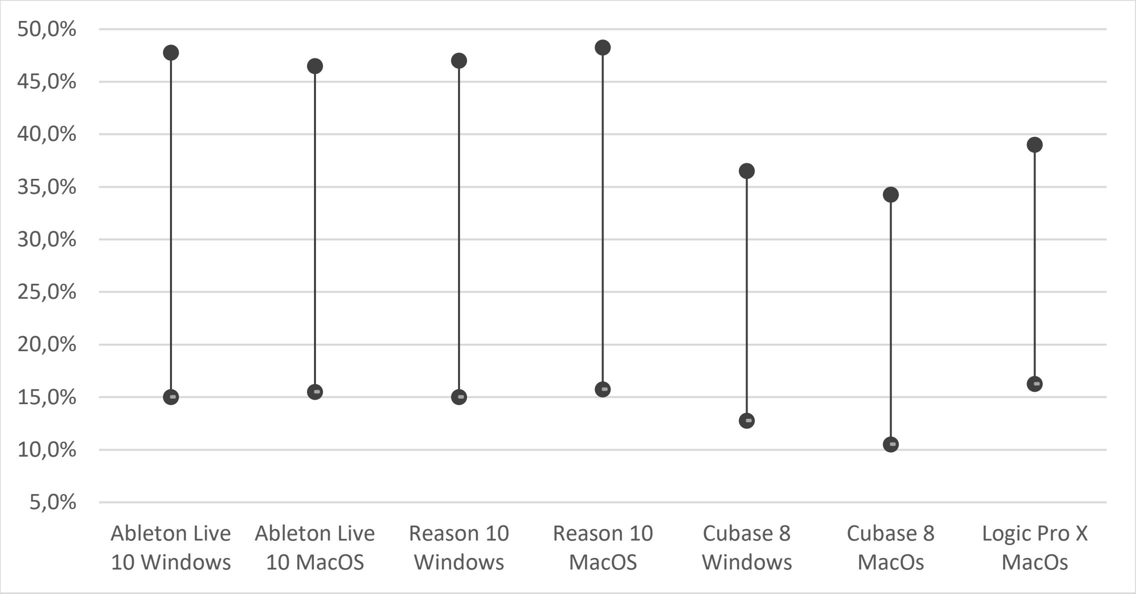


Figura 37. Consumo de procesamiento interno en cada *DAW* al ejecutar los 5 *plugins* estéreo en serie en 15 canales con señales de audio estereofónicas (75 instancias de *plugins* en total).

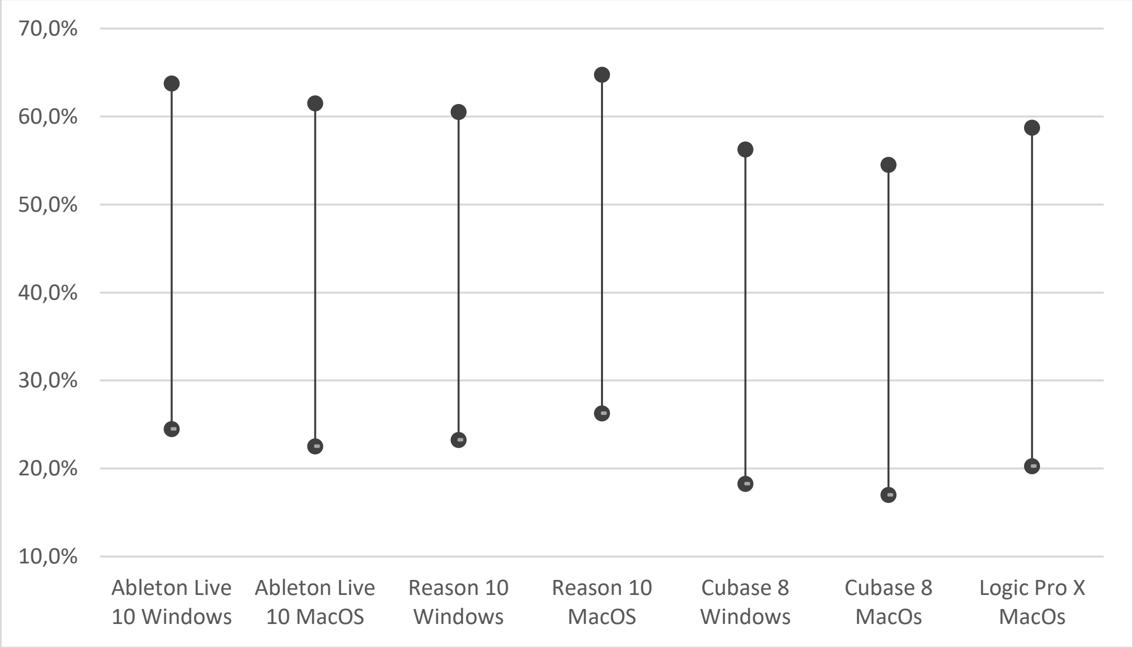


Figura 38. Consumo de procesamiento general del procesador del ordenador al ejecutar los 5 *plugins* mono en serie en 15 canales con señales de audio monofónicas en cada uno de los *DAWs* utilizados.

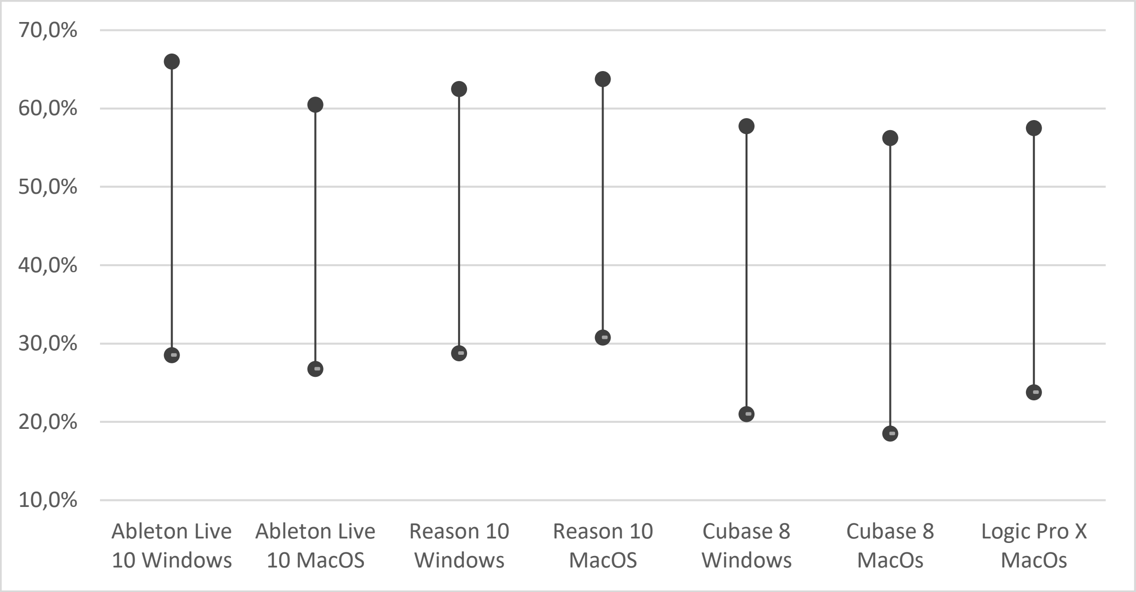


Figura 39. Consumo de procesamiento general del procesador del ordenador al ejecutar los 5 *plugins* estéreo en serie en 15 canales con señales de audio estereofónicas en cada uno de los *DAW*s utilizados.

Como se puede observar en las figuras 36 – 39, para esta prueba, el programa Ableton Live 10, da menores valores de procesamiento en el sistema operativo *MacOS*, sin embargo, se puede observar valores que superan el 60 % de consumo del CPU al utilizar frecuencias de muestreo altas y tamaños de buffer de procesamiento bajos. No obstante, se debe considerar que durante estas pruebas se estaban ejecutando 75 instancias de los módulos desarrollados, lo cual, de cierta manera, justifica el alto consumo de recursos computacionales. También se puede notar que se repite el patrón de la primera prueba general, ya que, en esta también se obtienen valores inferiores de procesamiento en el sistema operativo de *Windows* al utilizar el software anfitrión Reason 10. El software anfitrión Cubase se mantiene como el que menos procesamiento ha arrojado en todas las pruebas generales, en los dos sistemas operativos en los que se ha trabajado. El programa Logic Pro-X se mantiene con un rendimiento aceptable, presentando valores de procesamiento intermedios en comparación a los demás anfitriones utilizados. En general, el rendimiento de los módulos en las dos pruebas generales es muy aceptable y consume valores de procesamiento similares en todos los programas anfitriones utilizados, siendo compatibles, funcionales y estables en todos estos. Sin embargo, cabe realizar un resumen general a manera de comparación del procesamiento entre los distintos anfitriones en las dos pruebas generales, lo cual se muestra en las figuras 40 y 41.

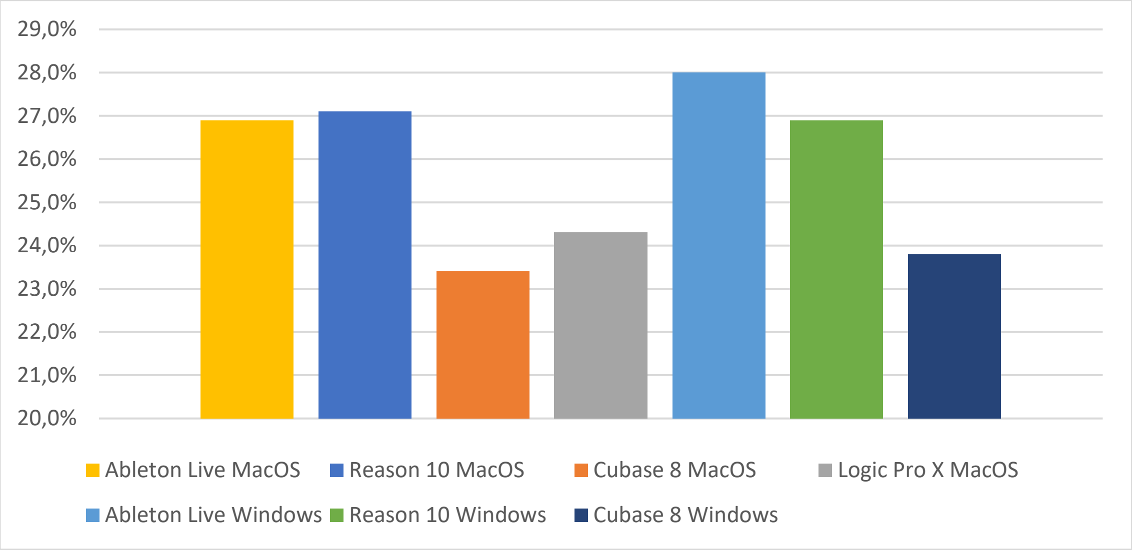


Figura 40. Valores de procesamiento máximos consumidos por los distintos anfitriones utilizados en la primera prueba general de los módulos desarrollados.

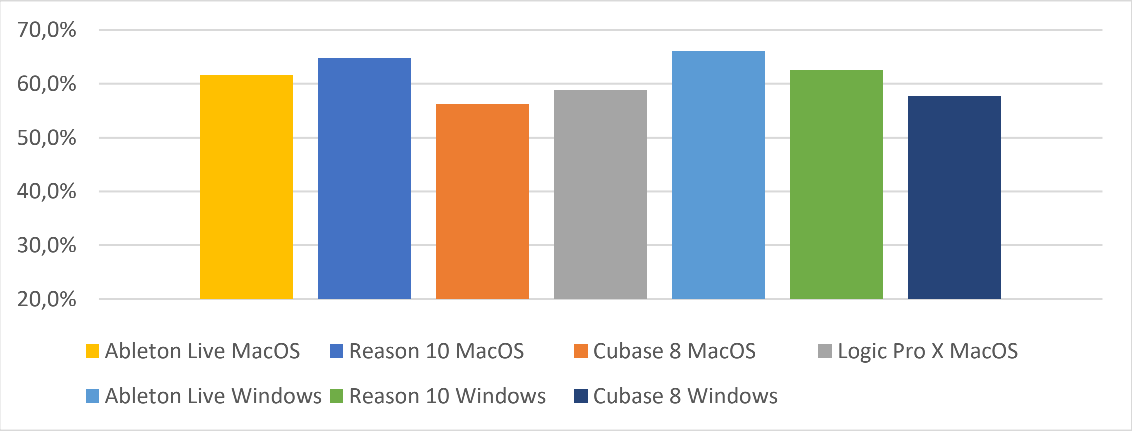


Figura 41. Valores de procesamiento máximos consumidos por los distintos anfitriones utilizados en la segunda prueba general de los módulos desarrollados.

Como ya se evidenció en las figuras mostradas (32-41), el programa Cubase 8 es el que entrega valores más bajos de procesamiento en ambos sistemas operativos en donde se realizaron las pruebas. Esto indica una eficiencia mayor de este programa; además, Que el formato probado en el mismo consume menos recursos que los demás formatos. Con esto claro, es posible que una de las razones para la eficiencia de procesos en el software Cubase tenga que ver con que este programa es desarrollado por la misma compañía que creó los formatos de módulos VST y VST3, y debido a esto, este software probablemente tenga sus algoritmos mucho más optimizados para estos formatos que los demás programas utilizados como anfitriones. Esto no quiere decir que los otros formatos compilados de los módulos sean defectuosos, sino solamente que su consumo de recursos computacionales es ligeramente más elevado.

4.2.1 Funcionalidad de cada módulo de manera individual con variables de tamaño de buffer y frecuencia de muestreo

Una vez realizadas las pruebas generales, en la cuales se verificó la validez, compatibilidad y consumo de procesamiento de cada uno de los *plugins* en cuestión con cada uno de los programas anfitriones, se procedió a realizar pruebas individuales en cada módulo.

Si bien se mostró la validez de los módulos en las pruebas generales, también fue necesario realizar pruebas individuales en cada *plugin*, para cuantificar el procesamiento característico y las condiciones en las que cada uno de estos es funcional para utilizarse. La primera prueba de este tipo consistió en verificar las combinaciones de tamaño de buffer y frecuencia de muestreo con las cuales cada módulo es funcional. Se consideran funcionales a los módulos cuando estos además de procesar las señales de audio sin sobrecargas y de manera correcta, no presentaron alteraciones impredecibles indeterminados como discontinuidades.

Tabla 3.

Funcionalidad de los módulos de ecualización y reverberación.

Ecualizador Mono / Reverb Mono							
Tamaño del buffer/ Frecuencia de muestreo	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
44100	Sí	Sí	Sí	Sí	Sí	Sí	Sí
48000	Sí	Sí	Sí	Sí	Sí	Sí	Sí
88200	Sí	Sí	Sí	Sí	Sí	Sí	Sí
96000	Sí	Sí	Sí	Sí	Sí	Sí	Sí
192000	No	Sí	Sí	Sí	Sí	Sí	Sí
Ecualizador Estéreo / Reverb Estéreo							
44100	Sí	Sí	Sí	Sí	Sí	Sí	Sí
48000	Sí	Sí	Sí	Sí	Sí	Sí	Sí
88200	Sí	Sí	Sí	Sí	Sí	Sí	Sí
96000	No	Sí	Sí	Sí	Sí	Sí	Sí
192000	No	Sí	Sí	Sí	Sí	Sí	Sí

Tabla 4.

Funcionalidad de los módulos de compresión y Delay.

Compresor Mono / Delay Mono							
Tamaño del buffer/ Frecuencia de muestreo	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
44100	Sí	Sí	Sí	Sí	Sí	Sí	Sí
48000	Sí	Sí	Sí	Sí	Sí	Sí	Sí
88200	Sí	Sí	Sí	Sí	Sí	Sí	Sí
96000	No	Sí	Sí	Sí	Sí	Sí	Sí
192000	No	Sí	Sí	Sí	Sí	Sí	Sí
Compresor Estéreo / Delay Estéreo							
44100	Sí	Sí	Sí	Sí	Sí	Sí	Sí
48000	Sí	Sí	Sí	Sí	Sí	Sí	Sí
88200	Sí	Sí	Sí	Sí	Sí	Sí	Sí
96000	No	Sí	Sí	Sí	Sí	Sí	Sí
192000	No	No	Sí	Sí	Sí	Sí	Sí

Como se puede observar en las tablas 3 y 4, la funcionalidad de los módulos fue la misma para la ecualización y reverberación, siendo estos válidos en todas las combinaciones de frecuencia de muestreo y tamaño del buffer, excepto para 32 muestras en 96000 Hz y 192000 Hz. De igual manera, la funcionalidad de los módulos de compresión y de *Delay* fue la misma, siendo válida para todas las combinaciones de frecuencia de muestreo y tamaño de buffer excepto para 32 muestras en 96000 Hz y 192 000 Hz, y adicionalmente para 64 muestras en 192 000 Hz de frecuencia de muestreo. En el caso del analizador de espectro, tanto para señales monofónicas como estereofónicas, fue funcional para todas las combinaciones de frecuencia de muestreo y tamaño de buffer de procesamiento por lo tanto no se lo incluyó en las tablas mostradas (3, 4). Los valores obtenidos, no son concluyentes a la funcionalidad absoluta de los módulos desarrollados, ya que, probablemente si se los ejecuta en un ordenador con un procesador más potente es posible que sean funcionales en los casos para los cuales no fueron válidos en las pruebas individuales realizadas.

Una vez verificada la funcionalidad de cada módulo con las variables antes mencionadas, se evaluó el procesamiento individual de los *plugins,* dando como resultado un rango de valores de procesamiento para cada módulo en cada anfitrión utilizado (*figuras 42 - 51*). Al igual que en las demás pruebas de procesamiento se utilizó distintas combinaciones de frecuencias de muestreo (44100 – 192000) y tamaños de buffer (32 - 2048).

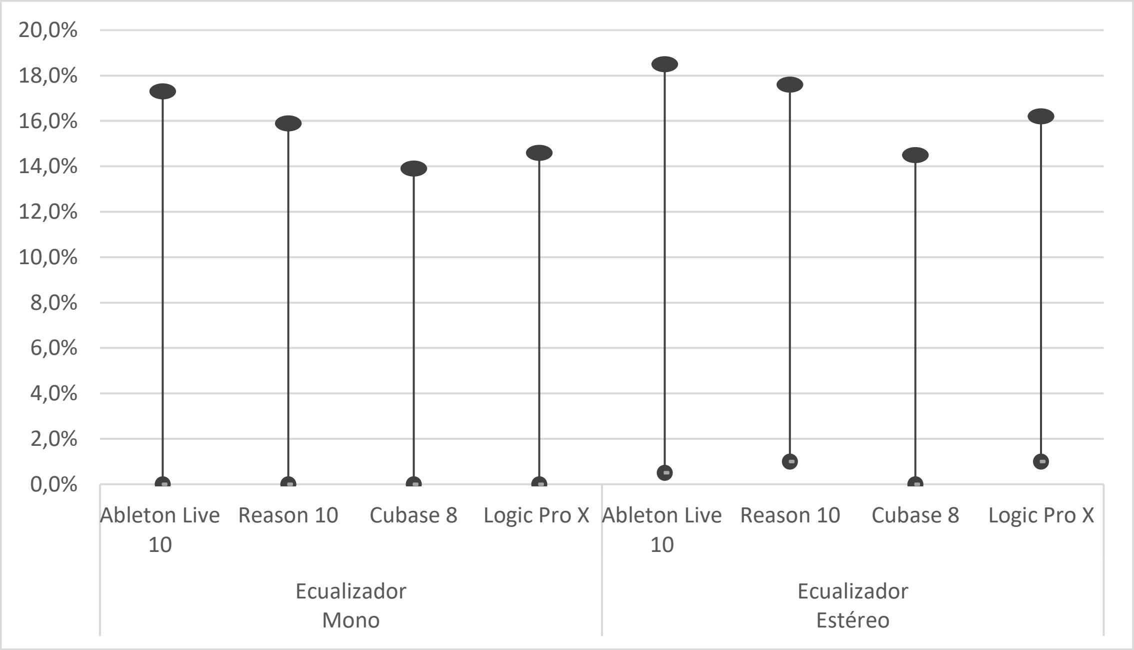


Figura 42. Rango de procesamiento interno de los módulos de ecualización en los programas anfitriones utilizados.

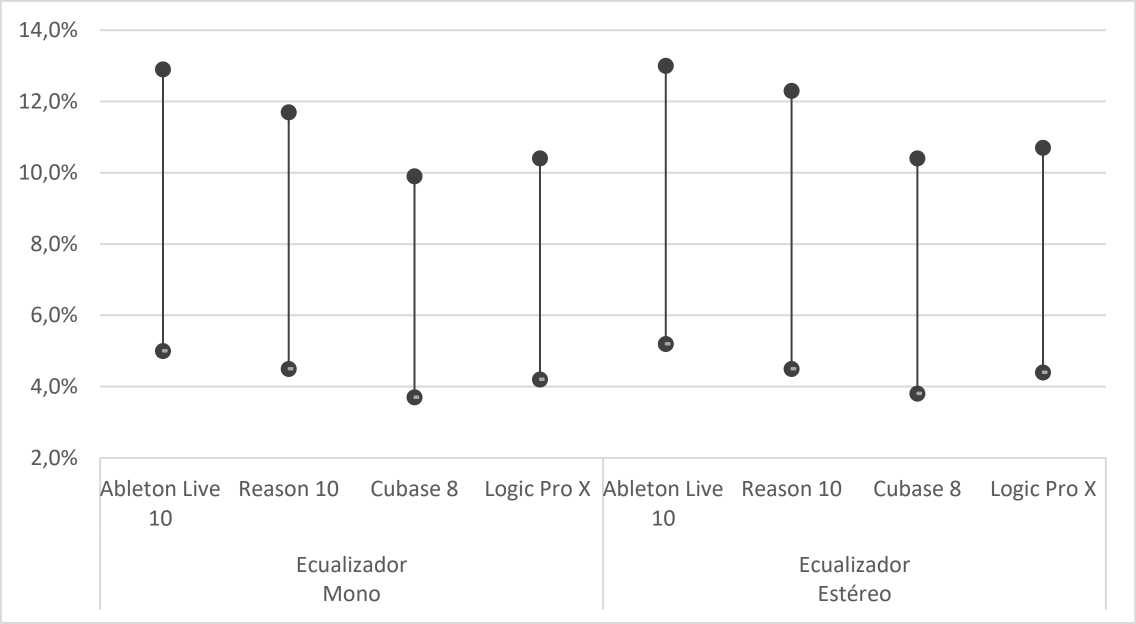


Figura 43. Rango de procesamiento general en el CPU al ejecutar los módulos de ecualización en los programas anfitriones.

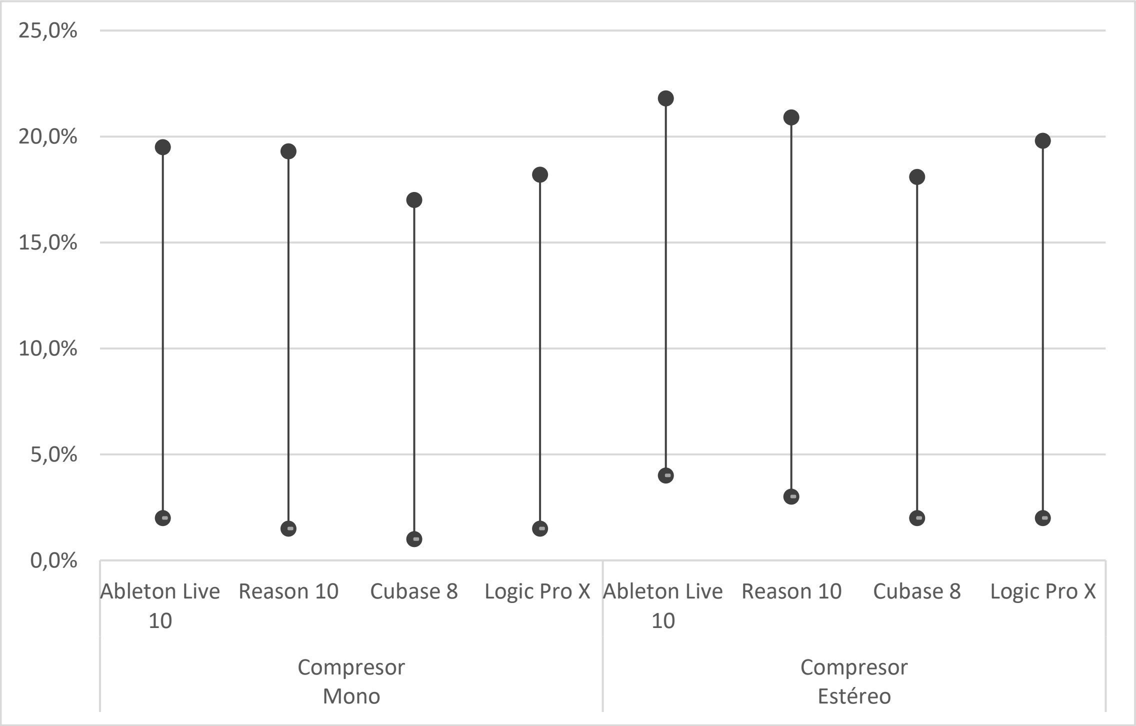


Figura 44. Rango de procesamiento interno de los módulos de compresión en los programas anfitriones utilizados.

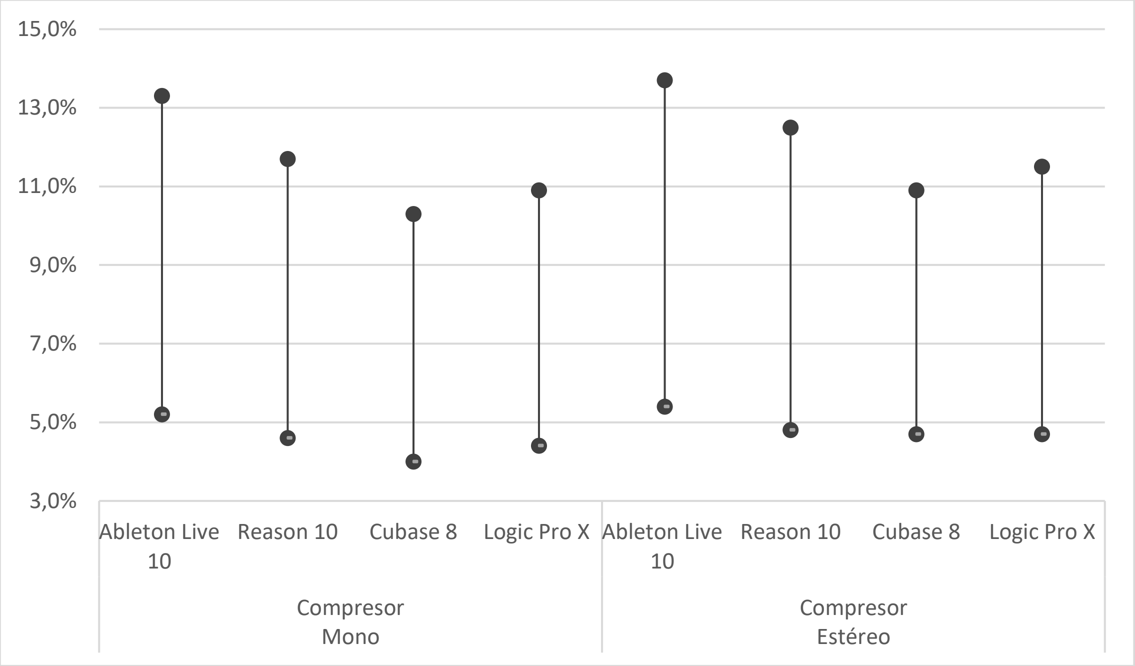


Figura 45. Rango de procesamiento general en el CPU al ejecutar los módulos de compresión en los programas anfitriones.

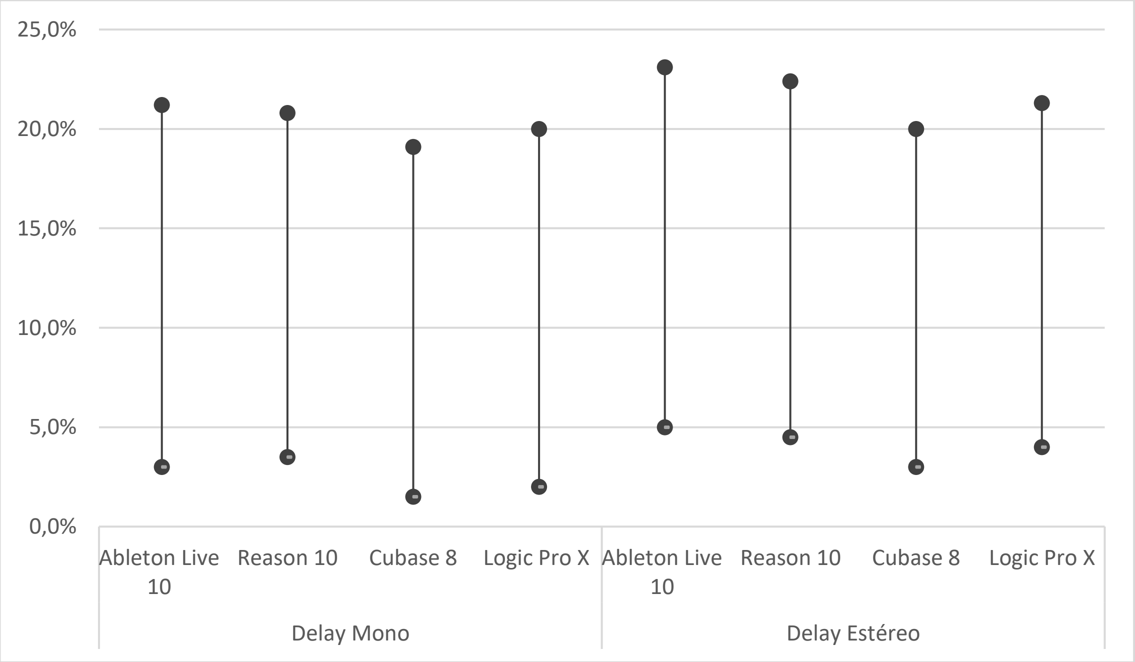


Figura 46. Rango de procesamiento interno de los módulos de *Delay* en los programas anfitriones utilizados.

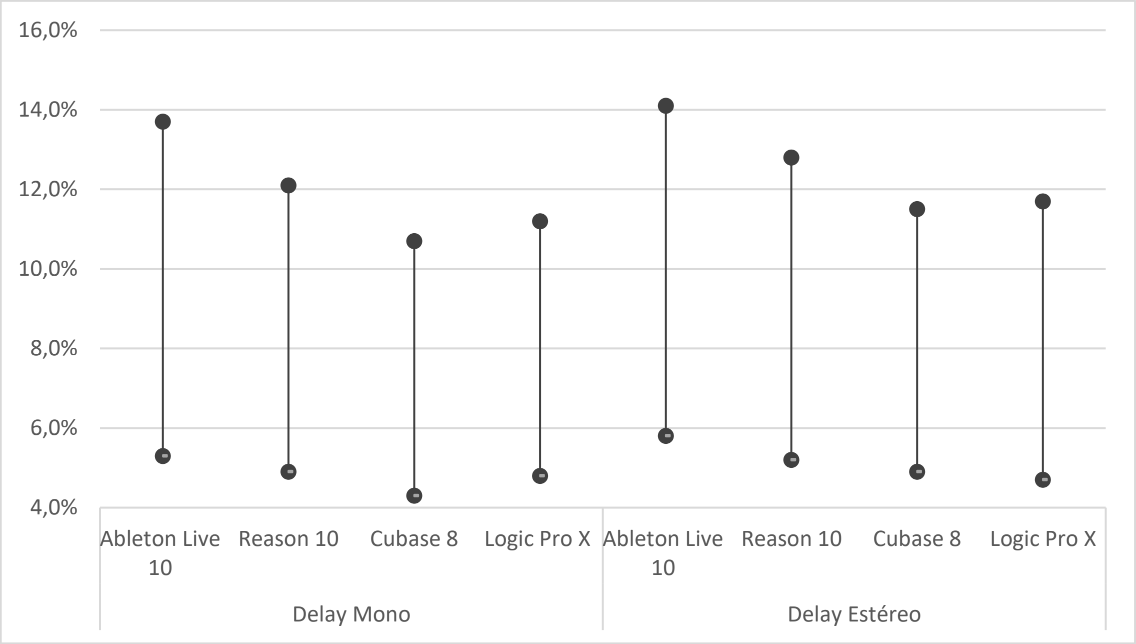


Figura 47. Rango de procesamiento general en el CPU al ejecutar los módulos de *Delay* en los programas anfitriones.

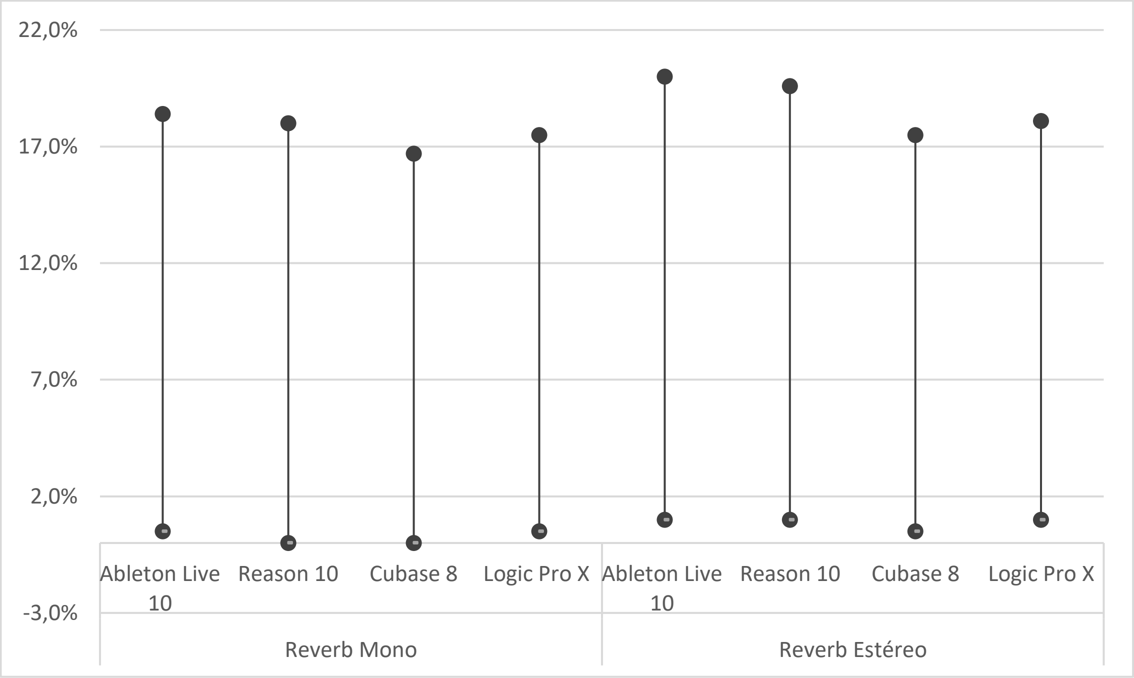


Figura 48. Rango de procesamiento interno de los módulos de reverberación en los programas anfitriones utilizados.

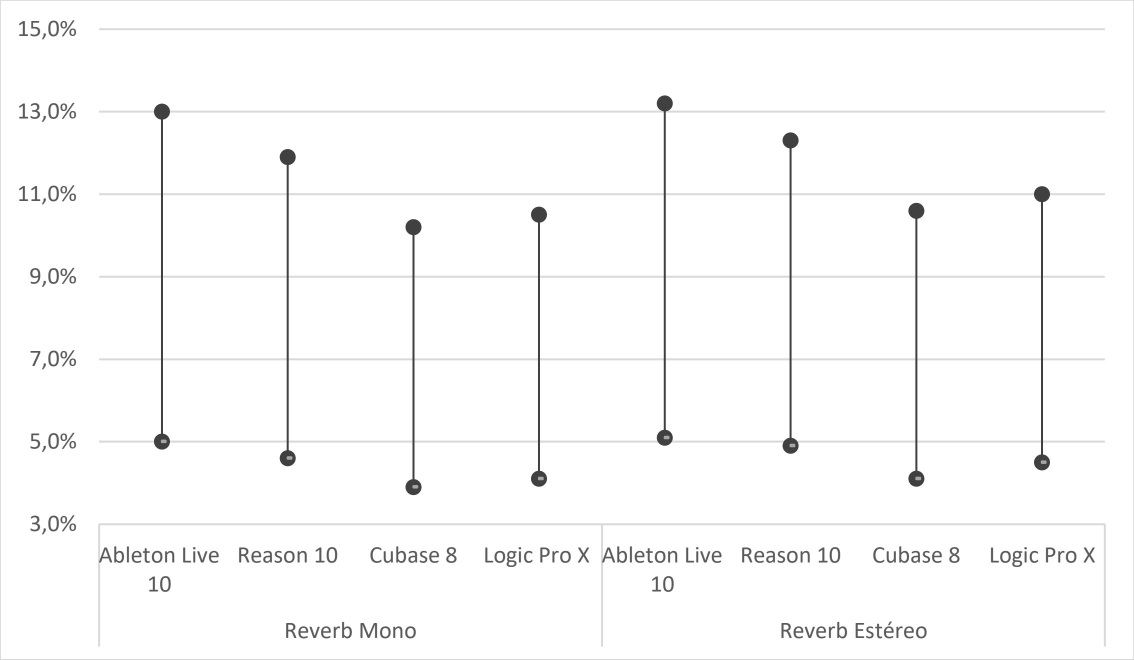


Figura 49. Rango de procesamiento general en el CPU al ejecutar los módulos de reverberación en los programas anfitriones.

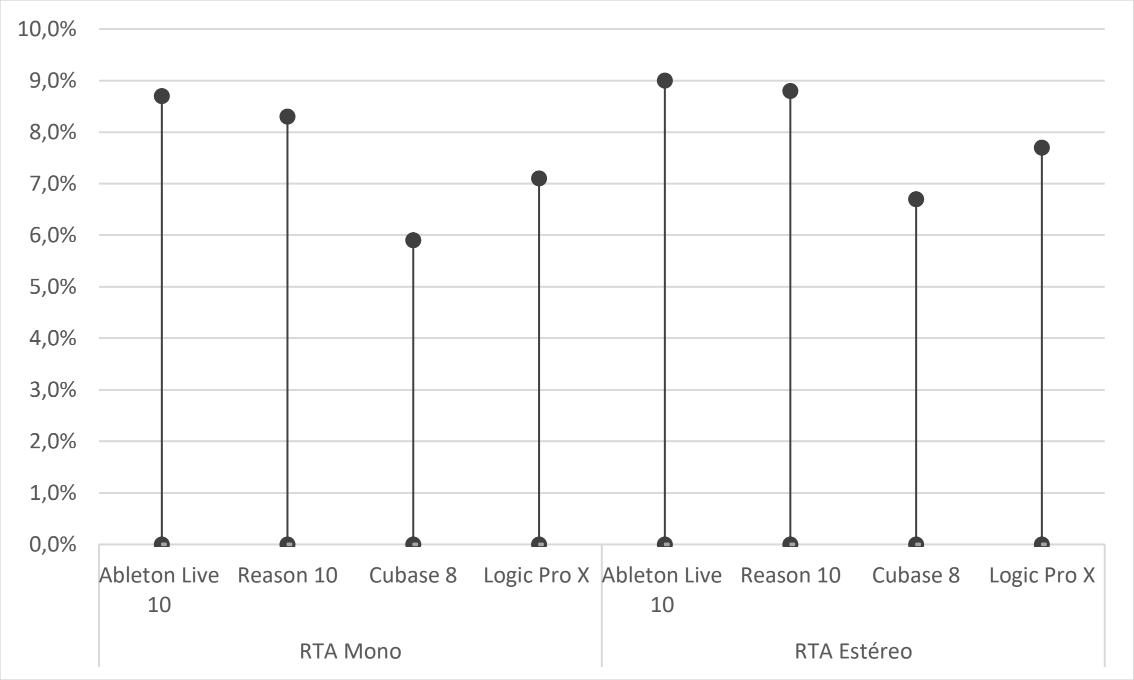


Figura 50. Rango de procesamiento interno de los módulos analizadores de espectro en los programas anfitriones utilizados.

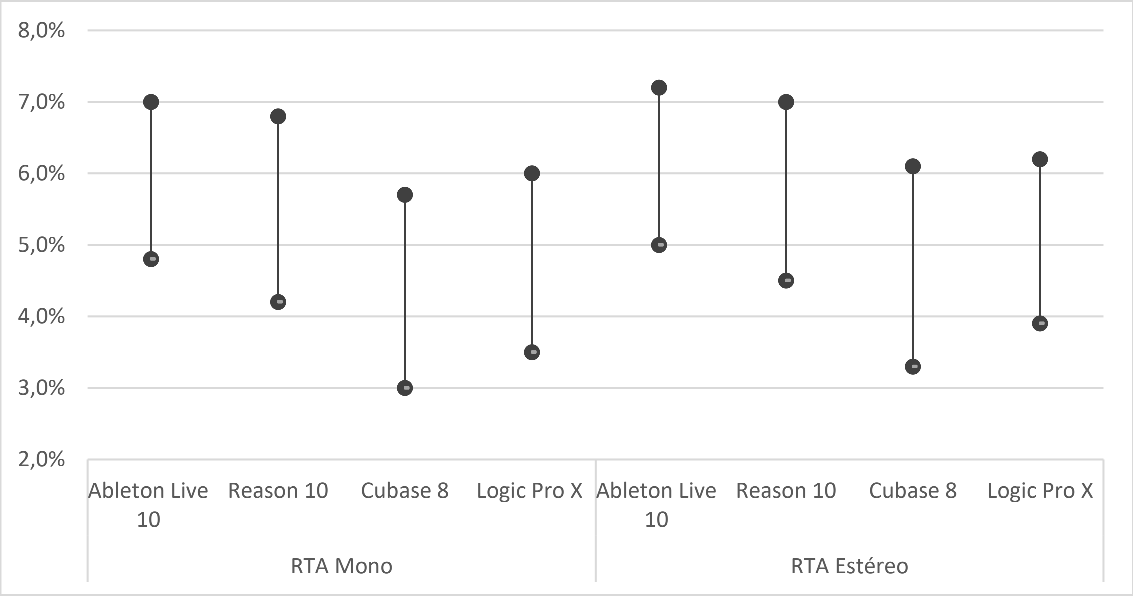


Figura 51. Rango de procesamiento general en el CPU al ejecutar los módulos analizadores de espectro en los programas anfitriones.

En las figuras 42 - 51, los valores mínimos corresponden al procesamiento de cada módulo con tamaños de buffer grandes y frecuencias de muestreo bajas, mientras que los valores máximos corresponden al consumo a tamaños de buffer pequeños y frecuencias de muestreo altas. El módulo que menos recursos computacionales consume es el analizador de espectro debido a que en este, la única operación que se realiza es una transformada rápida de Fourier para poder visualizar el espectro, pero no se procesa el audio de ninguna manera. Por otro lado, los módulos que más recursos consumen son los de compresión y *Delay*; esto se debe a que como estos módulos utilizan parámetros temporales, estos se actualizan al procesar cada muestra y no cada bloque de muestras como los demás parámetros no temporales (filtros, ganancia, etc.), lo cual eleva su consumo de procesamiento.

Es importante mencionar que en todas las pruebas de procesamiento realizadas no se observaron diferencias en los datos obtenidos entre el uso de las dos interfaces de audio con las que se trabajó. Esto se debe a que los *plugins* desarrollados, se diseñaron para ser procesados de manera nativa, es decir, por el procesador del ordenador que los esté ejecutando y no por un dispositivo externo.

Un dato interesante que se encontró durante la realización de las pruebas generales e individuales (figuras 32 - 51) fue que, al tener la interfaz de usuario abierta, el consumo de procesamiento aumenta hasta en un 20%, sin embargo, lo curioso de este efecto encontrado fue que el valor de aumento toma lugar en el procesamiento del CPU, más no en el del software anfitrión, sin importar cual sea este, mostrando de esta manera, que estos programas optimizan el consumo de recursos dando prioridad al procesamiento de la señal de audio y dejando las acciones secundarias como la ejecución de la interfaz, en este caso al GPU. Los valores mostrados en toda esta sección (4.1) se obtuvieron con dichas interfaces de usuario abiertas.

4.3 Medición de características de funcionamiento

Una vez verificada la validez de cada módulo y evaluado su procesamiento individual y en conjunto, se realizaron pruebas en las cuales se midieron las características propias de cada uno de los *plugins*. En estas pruebas se evaluó la precisión de los parámetros de los módulos para estimar si los valores que estos entregan son correctos. Para estas pruebas se utilizó otros módulos fabricados por terceros que correspondieron a cada efecto desarrollado en particular; sin embargo, como cada *plugin* tiene secciones en común como la etapa de inversión de polaridad y las etapas de ganancia de entrada y salida, cuyas características fueron evaluadas de la misma manera para todos los módulos.

4.3.1 Evaluación de la etapa de inversión de polaridad

En esta prueba se evaluó el correcto funcionamiento de la etapa de inversión de polaridad de cada *plugin* desarrollado. Para esta medición se utilizó el módulo *Inphase* de la compañía *Waves Ltd*, el cual puede identificar y tratar desfase en señales sonoras y mostrar su correlación (indicador del balance estéreo de una señal sonora). Primeramente, se verificó que la etapa de inversión de fase con un tono puro a 1 kHz y se renderizó la señal original y la señal procesada solamente con esta etapa encendida, sin realizarse ningún otro procesamiento de los módulos.

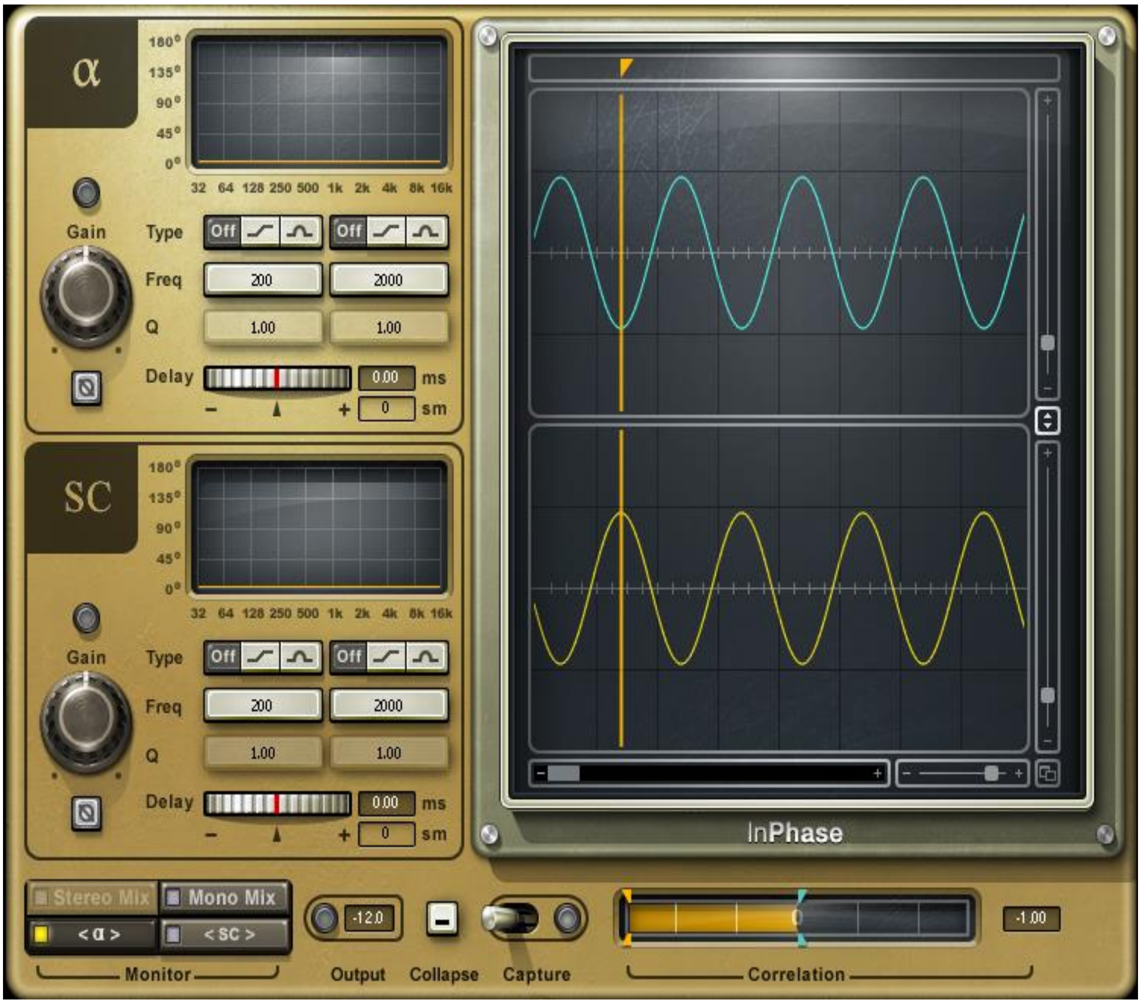


Figura 52: Comparación de señal con polaridad invertida con el módulo a señal original.

Como se observa en la figura 52, la señal procesada (amarillo) está completamente invertida en comparación con la original (azul) y por lo tanto su correlación es de -1. Con esto se comprueba que el funcionamiento de esta etapa es el correcto y actúa de la manera en la que está diseñado para hacerlo.

4.3.2 Evaluación de las etapas de ganancia de entrada y salida

Esta también fue evaluada de la misma manera para todos los módulos, debido a que se implementó de manera idéntica para todos estos. Esta prueba consistió simplemente en analizar la variación del nivel de varias señales sonoras distintas, al pasar por los módulos. Para esto se modificó las secciones de la implementación de cada efecto para que este no se aplique. Es decir, en el ecualizador, se dejó totalmente abiertos los filtros y cada una de las bandas con un valor de ganancia de 0.0 dB, para que no se aplique el efecto a la señal. En los módulos de compresión, *Delay* y Reverberación, se ajustaron los parámetros de mezcla o cantidad de señal procesada a 0,00% para que no se aplique el efecto en cada módulo. De esta manera se evaluó solamente las etapas de ganancia de entrada y salida, igual que en la prueba anterior. Se utilizaron tres tipos de señales: ruido rosa, ruido blanco y un tono puro de frecuencia de 1 kHz. Al realizar esta prueba, lo primero que se determinó es que en ningún módulo se modifica el nivel de las señales procesadas solamente con ejecutarlo, además se verificó que estas etapas evaluadas funcionaban de igual manera para todos los *plugins*. Para realizar la evaluación mencionada, se utilizó un módulo llamado *WLM Meter Plus*, el cual es un medidor de nivel muy preciso desarrollado por la compañía *Waves Ltd*. El módulo de medición fue configurado según la norma ITU 1770, establecida por la unión internacional de telecomunicaciones, y la unidad de medición fue el LUFS (unidad de volumen de escala completa). Se creó una instancia del módulo de medición antes y uno después de cada *plugin* analizado para ver la variación de la salida sobre la entrada. Al realizar algunas combinaciones de nivel de entrada y ganancia aplicada en los módulos, se pudo comparar la relación entre la dicha ganancia y las unidades de volumen de escala completa esta representa. (tabla 5).

Tabla 5.

Valores de la medición de las etapas de entrada y salida.

Tipo de señal (Tono puro 1 kHz, Ruido Rosa, Ruido blanco)	Nivel de entrada (dBFS)	LUFS de entrada	Ganancia aplicada desde el módulo (dB)	LUFS de salida
	-12	-21	-6, -3, 0, 3, 6, 9, 12	-27, -24, -21, -18, -15, -12, -9
	-9	-14	-6, -3, 0, 3, 6, 9, 12	-20, -17, -14, -11, -8, -5, -2
	-7,5	-11	-6, -3, 0, 3, 6	-13.5, -10.5, -7.5, -4.5, -1.5

En general, no se observaron diferencias entre los tres tipos de señales utilizadas, ni tampoco diferencias si se aplica la ganancia de entrada del módulo o la de salida. A partir de esta prueba, se determinó que el efecto que tienen las etapas evaluadas es de 1 LUFS por cada dB de ganancia aplicada (Tabla 5). Al analizar de manera espectral estas dos etapas, no se encontró diferencias notables en todo el rango de frecuencia humanamente audibles al aplicar valores de ganancia de entrada o salida.

4.3.3 Mediciones de las características del Ecualizador

Una vez evaluadas las etapas que los módulos tienen en común, el siguiente paso fue evaluar la parte característica del efecto correspondiente a cada módulo. Como se trata de 4 efectos distintos y analizador de espectro, las pruebas para cada uno de ellos fueron realizada de diferente manera. Para medir los rasgos de este módulo fue necesario evaluar el orden de los filtros implementados, ya que, estos en su diseño no se conocían, debido a que al desarrollarlos se utilizó un factor de calidad de 1 y no un orden en específico.

Para esta medición se utilizó un módulo que originalmente es un ecualizador con analizador de espectro en tiempo real a manera de función de transferencia. Este módulo se llama “F6 RTA” y fue desarrollado por la compañía *Waves Ltd*. Todas las bandas disponibles en el mismo para que solamente funcione como analizador de espectro y permita evaluar el comportamiento de los distintos parámetros de los módulos desarrollados.

Para seleccionar el tipo de señales a utilizar, se probó con señales de ruido rosa y ruido blanco cuyos espectros se muestran en las figuras 53 y 54, sin embargo, en el caso del ruido blanco la respuesta no fue plana y para evitar posibles errores de interpretación al momento de realizar pruebas posteriores, se decidió solamente utilizar ruido rosa, ya que con este tipo de señal se obtuvo una respuesta mucho más plana a lo largo del espectro. Cabe mencionar que en esta prueba se utilizaron otros tipos de señales las cuales no mostraron un espectro uniforme para analizar evaluar las características de los parámetros del ecualizador. Sin embargo, para las pruebas de algunos de los demás módulos no fue necesario utilizar señales de ruido rosa o ya que la evaluación de su funcionamiento fue distinta para cada *plugin.*


Ruido Rosa

A: Full Reset

RESPONSE RMS

BANDING 1/6

AVERAGING Medium

MA On

MAX Off

SOURCE Mono

RTA Post On

Figura 53. Señal de prueba (ruido rosa).


Ruido Blanco

A: Full Reset*

RESPONSE RMS
BANDING 1/6
AVERAGING Medium
MA On
MAX Off
SOURCE Mono
RTA Post
On
545 Hz (C#5 -30 Cents)
RTA: -26.3 dB

Figura 54. Señal de prueba (ruido blanco).

Como se puede observar en las figuras 53 y 54, el analizador de espectro utilizado muestra una pendiente al utilizar ruido blanco y un espectro plano al utilizar ruido rosa, lo cual de acuerdo con la teoría de este tipo de señales no es correcto. Por este tipo de errores siempre es importante conocer la teoría detrás de los recursos que se utilizan para no cometer errores en la realización e interpretación de pruebas y resultados en los que se utilice software de terceros.

Figura 55. Orden de los filtros de paso del ecualizador. Frecuencia de corte del filtro de paso alto: 1 kHz. Frecuencia de corte del filtro de paso bajo: 4 kHz.

El orden de los filtros (*figura 55*), tanto el de paso alto como el de paso bajo corresponde a una caída 6 dB/Oct, es decir, estos dos filtros son de primer orden; por lo tanto, se añadió el orden a la etiqueta de los filtros en el módulo para que se proporcione esta información en la interfaz de usuario. Después de verificar el orden de los filtros de paso, se evaluaron los filtros tipo *peak* aplicando diferentes valores de ganancia a cada banda de frecuencia de estos.

Tabla 6.

Evaluación de filtros peak de módulo de ecualización.

Ecualizador							
Banda de frecuencia (Hz, q:3)	**Aplicada desde el módulo**			**Medida**			**Diferencia**
	125	**700**	**4000**	**125**	**700**	**4000**	
Ganancia	-18	-18	-18	-17,82	-17,982	-17,73	0,156
	-12	-12	-12	-11,88	-11,988	-11,82	0,104
	-6	-6	-6	-5,94	-5,994	-5,91	0,052
	0	0	0	0	0	0	0
	6	6	6	5,94	5,994	5,91	0,052
	12	12	12	11,88	11,988	11,82	0,104
	18	18	18	17,82	17,982	17,73	0,156

Con los datos obtenidos que se muestran en la tabla 6, se realizó un promedio de las diferencias entre los valores de ganancia aplicados y los medidos, con lo cual se pudo determinar que el error promedio de estos filtros es de 0,09 dB. También se evaluó el funcionamiento de los tres filtros *peak* simultáneamente, para observar las posibles interacciones entre estas y el rango de funcionamiento de cada una de ellas. Para ello se colocaron valores de ganancia máxima y mínima en cada banda de estos filtros.

Figura 56. Implementación de los tres filtros *peak* simultáneamente con ganancia máxima positiva.

VIEW
RESPONSE: RMS
SLOPE
BANDING: 1/6
AVERAGING: Medium
MA: On
MAX: Off
SOURCE: Mono
RTA: Post
On
545 Hz (C#5 -30 Cents)
RTA: -5.0 dB

Figura 57. Implementación de los tres filtros *peak* con ganancia máxima negativa.

Como se puede observar en las figuras 56 y 57, el solapamiento entre las bandas de frecuencia es prácticamente nulo o muy bajo lo cual permite que cada uno de esos filtros se utilice tanto de manera individual o en conjunto. Esto también muestra que cada uno de estos filtros tiene un rango aproximación de acción debido a su factor de calidad Q (Tabla 7).

Tabla 7.

Rangos aproximados de las bandas de los filtros peak del ecualizador.

factor de calidad: 3			
Banda de frecuencia del filtro (Hz)	**125**	**700**	**4000**
Rango aproximado	63 Hz - 250 Hz	260 Hz - 1500 Hz	1700 Hz - 10000 Hz

En base a todas las pruebas realizadas en este módulo, se comprobó que la precisión entre los datos reales y los desarrollados son muy similares por lo cual este módulo tiene una exactitud aceptable.

4.3.4 Medición de las características del compresor

Para la evaluación de este *plugin*, se realizaron dos pruebas. La primera consistió en la comparación de distintas señales musicales procesadas por el módulo en cuestión y por otros módulos de compresión de terceros. Todos estos compresores se configuraron con los mismos valores de los parámetros para que los resultados sean comparables. Los compresores utilizados se en esta prueba fueron: *H Comp*, *Audio Track*, *Scheps Omni Channel*, Compresor estándar de *Logic Pro-X* y el Compresor desarrollado.

Se utilizaron 2 señales de audio distintas, y en cada una se utilizaron distintos parámetros de compresión. Es importante mencionar que para los compresores que tenían disponible el parámetro de suavizado o *Knee*, en este se colocó un valor de este en 0 para que no influya en el procesamiento de las señales. Como se puede observar en las figuras 58 y 59, ninguna de las señales comprimidas con los distintos compresores es idéntica a las demás a pesar de que los módulos fueron configurados con los mismos valores en sus parámetros, sin embargo, se puede notar que en todos los casos la dinámica de la señal fue modificada. Es importante notar que las señales procesadas son similares (*figuras* 58 y 59), pero ninguna es idéntica, lo cual puede dar una idea de quecada uno de estos módulos ha sido codificado con algoritmos distintos produciendo de esta manera resultados diferentes.

Esta primera prueba se realizó no con el fin de medir la exactitud de los parámetros del compresor desarrollado sino de comparar su funcionamiento con el de otros módulos con el mismo efecto. El propósito también fue poder notar visiblemente las diferencias que tienen las señales de audio procesadas procesados por distintos compresores configurados con sus parámetros en los mismos valores (*figura 58, 59*).

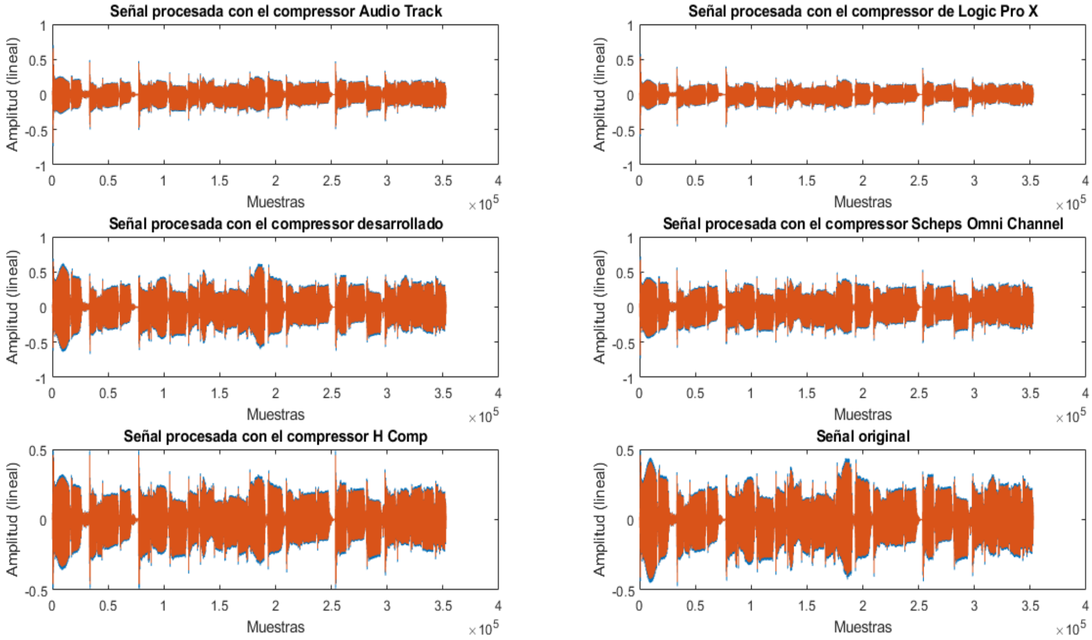

Señal procesada con el compressor Audio Track
Señal procesada con el compressor de Logic Pro X
Señal procesada con el compressor desarrollado
Señal procesada con el compressor Scheps Omni Channel
Señal procesada con el compressor H Comp
Señal original

Figura 58. Primera señal procesada con el efecto de compresión. Parámetros: ataque: 15 ms, relajación: 200 ms, *Treshold*: -20 dB, Ratio: 7,0:1, OutGain: + 3 dB.

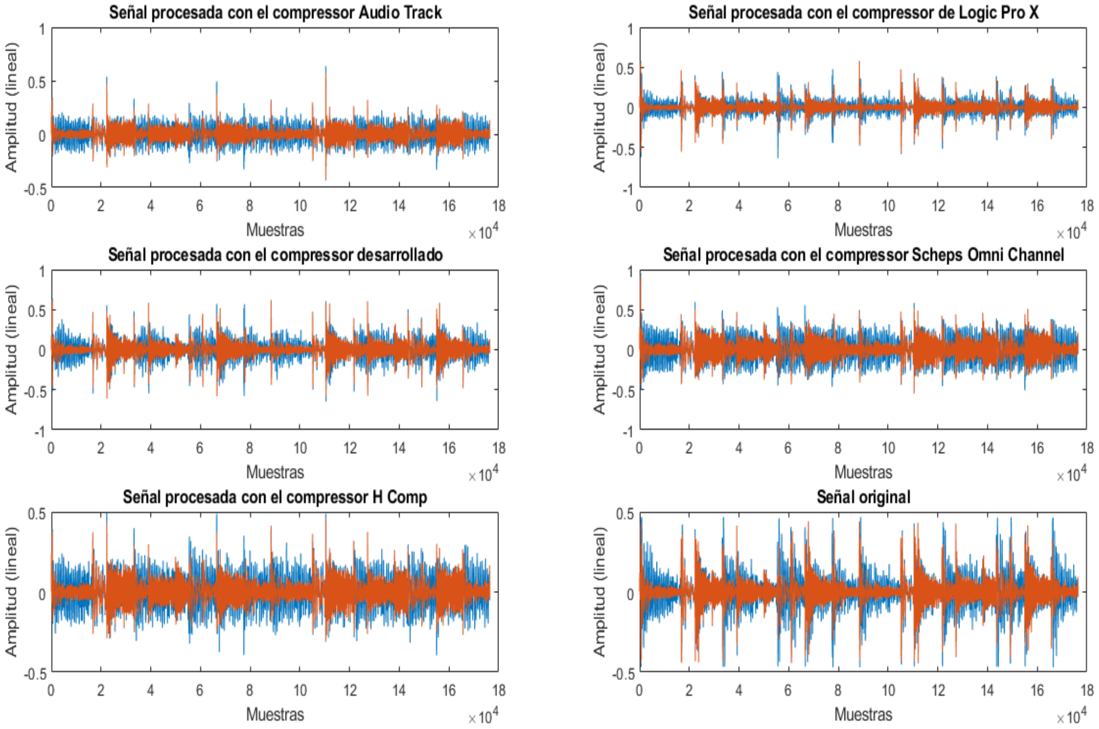


Figura 59. Segunda señal de audio de comparación de compresores. Parámetros: Ataque: 1 ms, relajación: 30 ms, *Treshold*: -18 dB, Ratio: 10,0:1, *Out Gain*: + 3dB.

Las figuras 58 y 59 representan las señales de audio estereofónicas comprimidas con distintos módulos, incluyendo el desarrollado. Como se trata de señales estereofónicas, las que tienen color naranja representan el canal izquierdo de las señales mientras que las que tienen color azul representan el canal derecho. Para evaluar la precisión de los parámetros de se procesó un tono puro de 1 kHz y se evaluó si los valores de su amplitud procesada correspondían a los parámetros de procesamiento. La señal utilizada se tenía una amplitud de -3 dBFS y los parámetros del compresor para esta prueba fueron los siguientes: *Treshold:* -18 dB, Ratio: 10,5:1, ganancia de salida: 0 dB.

Tabla 8.

Resultados prueba de precisión de compresión.

Parámetros de compresión			
Ataque	**Relajación**	**Treshold**	**Ratio**
26 ms	1 ms	- 18 dB	10,5:1
Escala	**Nivel sin comprimir**	**Nivel Comprimido**	**Diferencia de niveles**
Logarítmica (dBFS)	-3,0	-16,7	13,7

Al analizar los valores mostrados en la tabla 8, los valores de niveles comprimidos que se presentan corresponden al momento inmediatamente después de que la señal alcanzó su máximo compresión, es decir, cuando transcurrió el tiempo de ataque. Como la relación de compresión utilizada fue de 10,5 a 1, se espera que por cada 10,5 dB de señal de entrada que sean superiores al umbral seleccionado que pase por el compresor debe salir solamente 1 dB.

Para verificar que los valores de estos parámetros son precisos se realizó una regla de tres simple considerando el nivel esperado sobre el umbral de acuerdo con los parámetros seleccionados de lo cual se obtuvo un valor de 1,3047. Ahora, al realizar una simple diferencia entre el nivel obtenido sobre el umbral establecido se obtiene el valor de 1,3 con lo cual se verifica que todos estos parámetros funcionan con un error de 0,0047 dB, lo cual es bastante exacto.

De igual manera fue necesario determinar la precisión de los parámetros temporales del módulo para verificar que funcionen correctamente, para lo cual se exportó la señal original y la señal comprimida y se realizó un análisis de su amplitud y del tiempo en que esta se comprimió y dejó de comprimirse. Con el fin de tener una mayor facilidad de realizar este análisis, las señales correspondientes fueron analizadas en el software Pro Tools, y fue grabada con una frecuencia de muestreo de 44100 Hz. El tiempo de ataque utilizado en esta prueba fue de 26 ms.

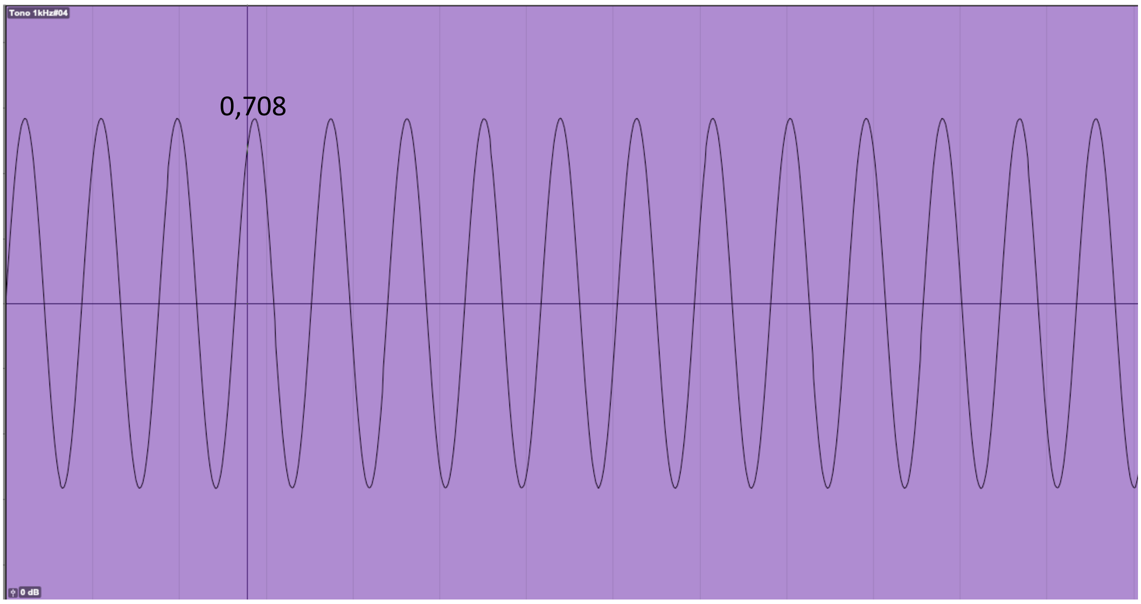


Figura 60. Señal original sin compresión. Tono puro a 1 kHz.

Con las herramientas del software mencionado se logró determinar el nivel de la señal original en escala lineal es de 0,708 (figura 59), lo cual corresponde a los -3 dBFS de señal de entrada mostrados en la tabla 8. Cabe mencionar que la señal utilizada es bastante exigente para el *plugin* debido a que es un tono puro sin embargo, este la procesa sin ningún problema como se muestra en la tabla 8 y en las figuras 61, 62.


78685

Start 78685
End 79833
Length 1148

Cursor 78627 -3047516

Figura 61. Verificación de tiempo de ataque.

En la figura 61 se puede ver que la amplitud final de la señal comprimida en escala lineal es de 0,146, la cual corresponde a los 16,7 dBFS mostrados en la tabla 8. Además, se tiene seleccionado todo el tiempo que se demora el módulo en implementar la compresión completa, el cual está enmarcado en la esquina derecha en escala de muestras de audio. Según la figura 61, este *plugin* tarda 1148 muestras en alcanzar la compresión máxima, lo cual de acuerdo con la frecuencia de muestreo con la que se trabajó (44100 Hz), corresponde a 26,032 milisegundos. Tomando en cuenta que el tiempo de ataque establecido fue de 26 milisegundos, se ha determinado que el error de este parámetro es de 0,032 milisegundos, lo cual representa que la exactitud de este parámetro es muy buena.

Para realizar la prueba de precisión del tiempo de relajación se utilizó el mismo nivel de señal de entrada -3 dBFS, pero se modificaron los parámetros de umbral y relación de compresión, los cuales par este caso fueron: Treshold: -9 dB, Ratio: 6,5 y tiempo de relajación de 1 milisegundo. Para evaluar este parámetro se esperó a que la señal esté completamente comprimida y se automatizó el parámetro de Treshold para que cambie a 0,0 dB y dejase de comprimir, se renderizó la señal con estas características y se analizó la sección en la cual el módulo deja de comprimir para determinar el tiempo que se demoró en hacerlo como se muestra en figura 62.

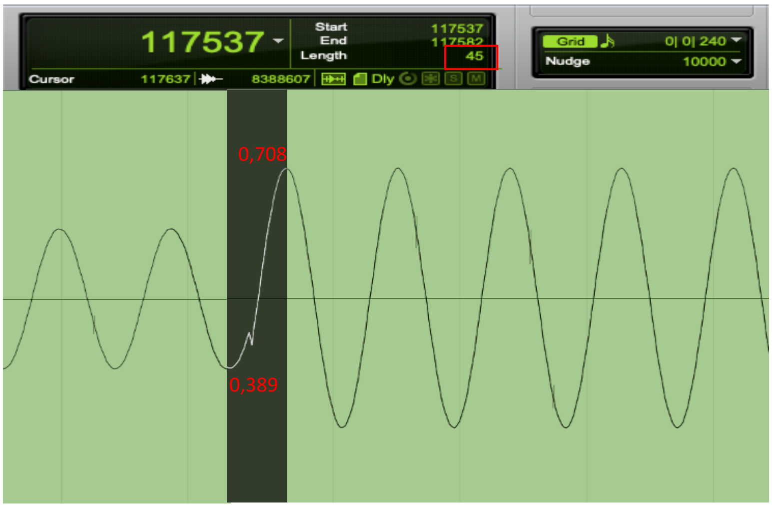

Start 117537
End 117582
Length 45

Figura 62. Evaluación de precisión del tiempo de relajación del módulo de compresión.

Como se puede observar en la figura 62, se puede observar que el tiempo que se demora el módulo en dejar de comprimir es de 45 muestras, que a una frecuencia de muestreo de 44100 corresponde a 1,02 milisegundos. Entonces el error de este parámetro en el módulo desarrollado es de 0,02 milisegundos, similar al error del tiempo de ataque (0,032 ms), pero ligeramente menor. Con estas pruebas se comprobó la precisión y funcionamiento correcto del *plugin* de compresión desarrollado. Es importante mencionar que todas las pruebas realizadas en este módulo se realizaron con el parámetro de mezcla al 100 %. En general el funcionamiento de los tiempos de ataque y relajación de un compresor suelen indicar el lapso que transcurre entre el momento en que se detecta el nivel de la señal a comprimir o a dejar de hacerlo y el momento en el que se alcanza el 90% del nivel final de compresión o el 90% del nivel descomprimido respectivamente. Sin embargo, De acuerdo con los valores obtenidos en las pruebas realizadas en el módulo de compresión desarrollado, se determinó que el funcionamiento de este es de tal manera que sus parámetros corresponden al 100% de los lapsos temporales mencionados.

4.3.5 Medición de las características del módulo de *Delay*

Lo primero que se probó en este módulo fue el que resultado tenía el parámetro de retroalimentación sobre las señales procesadas por dicho *plugin*, así que se realizaron pruebas similares a las de evaluación de las etapas de ganancia de entrada.

Se utilizó como señal de prueba un tono puro de frecuencia 1 kHz. Para realizar esta prueba se dejó estáticos todos los parámetros del módulo a excepción del de retroalimentación, modificando y observando los cambios en la amplitud de salida de la señal.

Tabla 9.

Resultados de las pruebas de retroalimentación (Feedback) del plugin de Delay

Nivel de entrada	**Retroalimentación**	**Nivel de salida**
-14 LUFS	0%	-14 LUFS
	25%	-13 LUFS
	50%	-12 LUFS
	75%	-10 LUFS
	99%	-8 LUFS

Con los datos de la tabla 9 se logró determinar que el parámetro de retroalimentación del módulo correspondiente se relaciona con la amplitud de la señal según la ecuación 5 anteriormente presentada.

Es importante mencionar que la ecuación 5 solo tiene que ver con la dependencia del nivel de salida en relación con el parámetro evaluado, pero existen otras variables que pueden afectar dichos niveles, las cuales no fueron consideradas en esta prueba.

Después de establecer el efecto del parámetro de *Feedback*, al igual que con el compresor, para el *plugin* de *Delay*, también se realizó una comparación sobre su funcionamiento junto con otros módulos desarrollados por terceros, en la cual se procesó una señal impulsiva. En cada módulo se configuró el tiempo de retardo en 1 segundo, retroalimentación al 50%, mezcla 50%, sin ganancia de entrada ni salida. Los módulos de comparación utilizados fueron: *H* Delay, *Stereo Delay* de Logic Pro-X, *GTR Stomp Delay* y el módulo desarrolado.

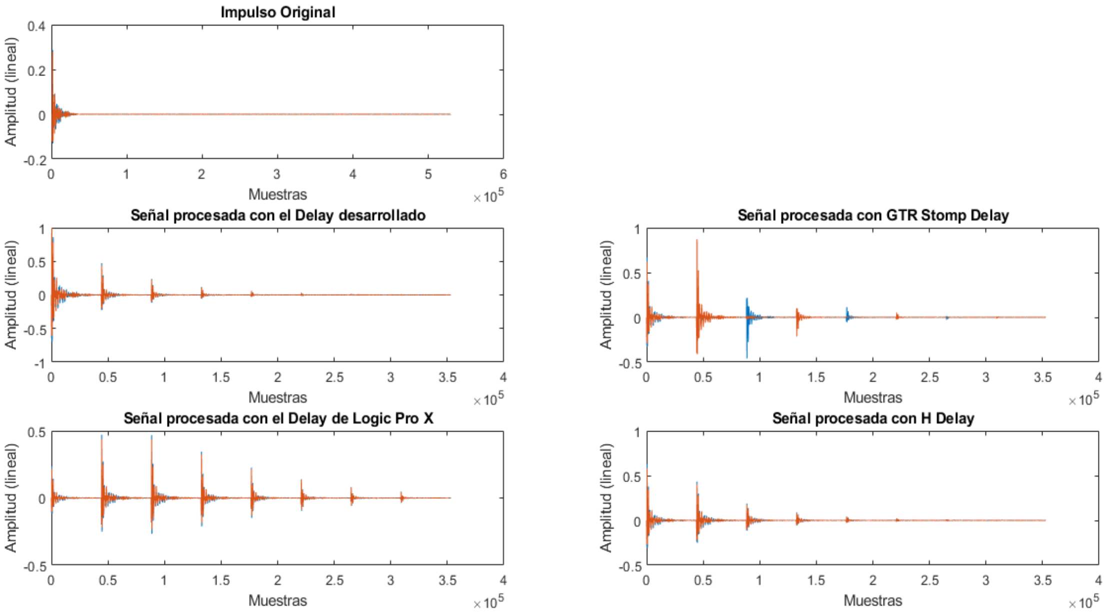


Figura 63. Pruebas de comparación de módulos de *Delay.* Naranja: canal izquierdo de la señal. Azul: canal derecho de la señal.

Como se puede observar en la figura 63, casi todos los módulos comparados entregan señales similares, solamente el módulo de *GTR Stomp Delay* tiene la particularidad de que actúa en una especie de *Delay* tipo ping pong, debido a la a amplitud de las repeticiones de la señal, varían de una canal a otro. Esto simplemente se debe a ese módulo de *Delay* en particular (*GTR Stomp Delay*) y no al ajuste de los parámetros utilizados.

Al analizar la figura 63, se puede observar que, en promedio, en todos los módulos la señal crea ecos que producen aproximadamente cinco y siete repeticiones, antes de que su amplitud se extinga, entonces considerando que, se utilizó un tiempo de retardo de 1 segundo se puede afirmar que las señales tuvieron ecos de hasta 7 siete segundos con la configuración de parámetros antes mencionada.

Después de realizar la prueba de comparación con otros módulos de *Delay*¸ la segunda prueba realizada fue una verificación de la precisión del tiempo de retardo del *plugin*. Esto se realizó tanto para el modo de tiempo libre como para el modo de tiempo sincronizado con el anfitrión. La señal utilizada fue el mismo impulso de la prueba anterior, así como los ajustes de retroalimentación y mezcla y en el caso del tiempo sincronizado se utilizó un valor de bpm de 60.

Figura 64. Resultados de pruebas de tiempo de retardo modo libre (1, 2 y 3 segundos de retardo del eco).

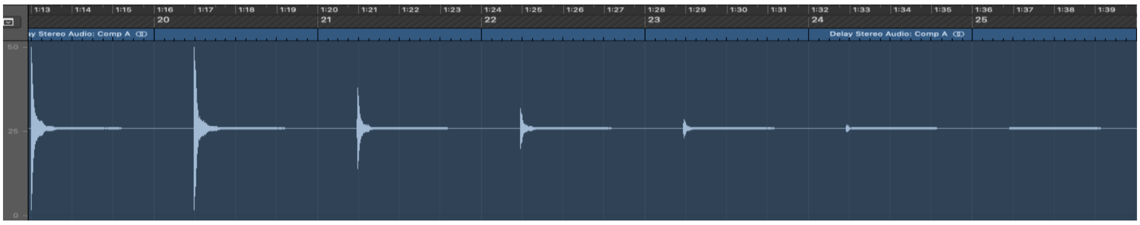


Figura 65. Resultados de pruebas de tiempo de retardo modo libre (4 segundos de retardo del eco).

Como se puede observar en las figuras 64 y 65, los tiempos probados en modo libre dieron resultados correctos, y se determinó además que la retroalimentación se aplica de acuerdo con el parámetro de retardo, es decir que las señales con tiempos de *Delay* más altos van a tardar más en extinguirse que las señales con valores más cortos de este parámetro.

Figura 66. Tiempos de retardo sincronizados 60 bpm: 1 compas – blanca.

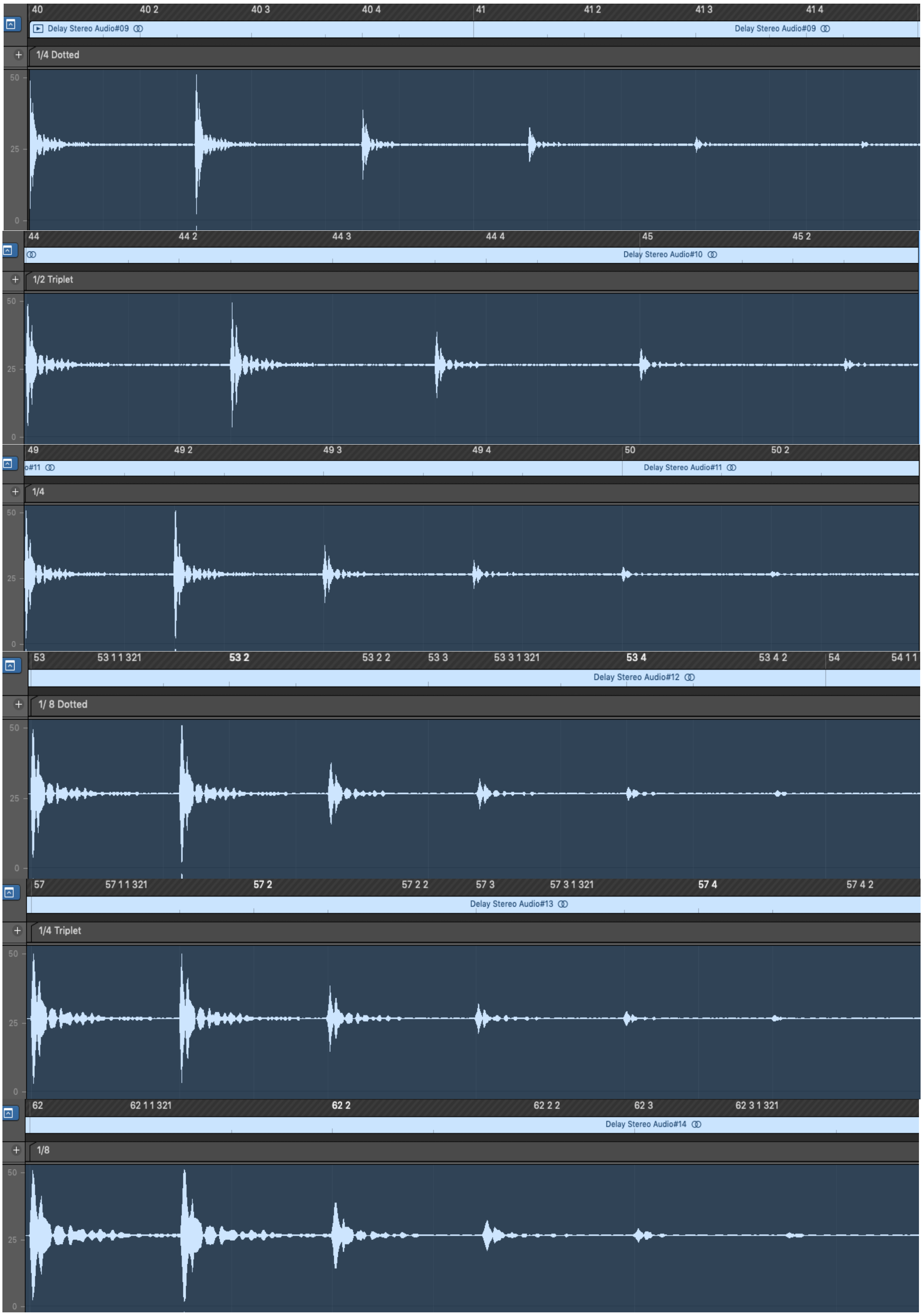


Figura 67. Tiempos de retardo modo sincronizado: Negra con punto – Corchea.

Figura 68. Tiempos de retardo modo sincronizado: Semicorchea con punto – Tresillo de semicorchea.

Según lo mostrado en las figuras 66, 67 y 68, se pudo determinar que los tiempos de *Delay* sincronizados con el software anfitrión son correctos ya que el retardo creado corresponde a las divisiones de los compases del tiempo establecido para cada a las figuras musicales establecidas como tiempos de retardo.

Con todas estas pruebas realizadas sobre este módulo, se comprobó que este tenía un funcionamiento adecuado y que respondía de manera correcta conforme con su diseño y desarrollo. Es importante mencionar que este fue el módulo que más problemas presentó, tanto durante la etapa de desarrollo como en la etapa de pruebas, y dentro de esa última, en primera instancia, se encontraron problemas al modificar los tiempos de retardo, creando discontinuidades y eventos sonoros indeseados, por lo que fue necesario aumentar el suavizado de cambio de los parámetros de este *plugin* incluso más que en el de los demás, por lo que las señales procesadas finales, las cuales se han mostrado, dan evidencia de que estos problemas se lograron superar.

Como última consideración a tener en cuenta dentro de lo que se determinó en las pruebas realizadas en este módulo es que si se realizan cambios en el tiempo de *Delay* mientras se está reproduciendo la señal, ya sea en modo libre o sincronizado, se altera el tono de dicha señal generando modulaciones de acuerdo a la velocidad con la que se modifique el parámetro temporal. Se pudo verificar que este fenómeno se da en la mayoría de los efectos de *Delay* de *plugins* de terceros. No obstante, esto solo se da mientras se modifica el parámetro indicado, volviendo a su funcionamiento normal cuando este queda nuevamente estático.

Este fenómeno se da por la interpolación añadida al módulo para evitar discontinuidades y ruidos indeseados y puede llegar a sonar de manera similar a las modulaciones que producían las antiguas máquinas de *Delay* de cinta al manipular la longitud de esta. Por estas razones se considera que este no es un fenómeno que se pueda catalogar como un defecto del *plugin* sino todo lo contrario, ya que se puede utilizar para crear efectos interesantes según la creatividad de quien lo utilice.

4.3.6 Medición de las características del módulo de reverberación

En primer lugar, se evaluaron los filtros de paso de este módulo, los cuales solamente se aplican a la señal reverberada para tener un mayor control sobre el especto de esta. Como estos filtros fueron diseñados de la misma manera que los del módulo de ecualización se esperaba resultados idénticos a las pruebas de este (figura 55), sin embargo, se pudo observar que debido a la reverberación aplicada la caída de los filtros no era exactamente igual. Es decir, en lugar de tener una caída de 6 dB/Oct se tuvo caída de 7,2 - 7,5 dB/Oct (figura 69). Esto no quiere decir que los filtros sean defectuosos, sino que el efecto implementado modifica el espectro de la señal procesado causando esta diferencia de entre 1,2 y 1,5 dB en el orden de los filtros. Al igual que en el ecualizador, aquí también se utilizó ruido rosa.

Figura 69. Prueba de orden de filtros en el módulo de reverberación. Parámetros: *Size*: 0.5, *Width*: 0.5, *Damping*: 0.5, *Dry:* 0 %. *Wet:* 100 %. Frecuencia de corte de paso alto: 1 kHz. Frecuencia de corte de paso bajo: 4 kHz. Señal utilizada: ruido rosa.

También se evaluó el efecto que tenían los tres parámetros principales del módulo (*Size, Width* y *damping*), para lo cual, se fueron modificando cada uno de estos parámetros acústicos. Al hacerlo, cada uno de estos valores afecto el espectro de la señal procesada el cual se vio alterado de diferentes maneras. Los valores que quedaron fijos fueron los de la señal original y de la señal reverberada que tuvieron un valor de 0% y 100 % respectivamente. Además, los filtros quedaron completamente abiertos para esta prueba.

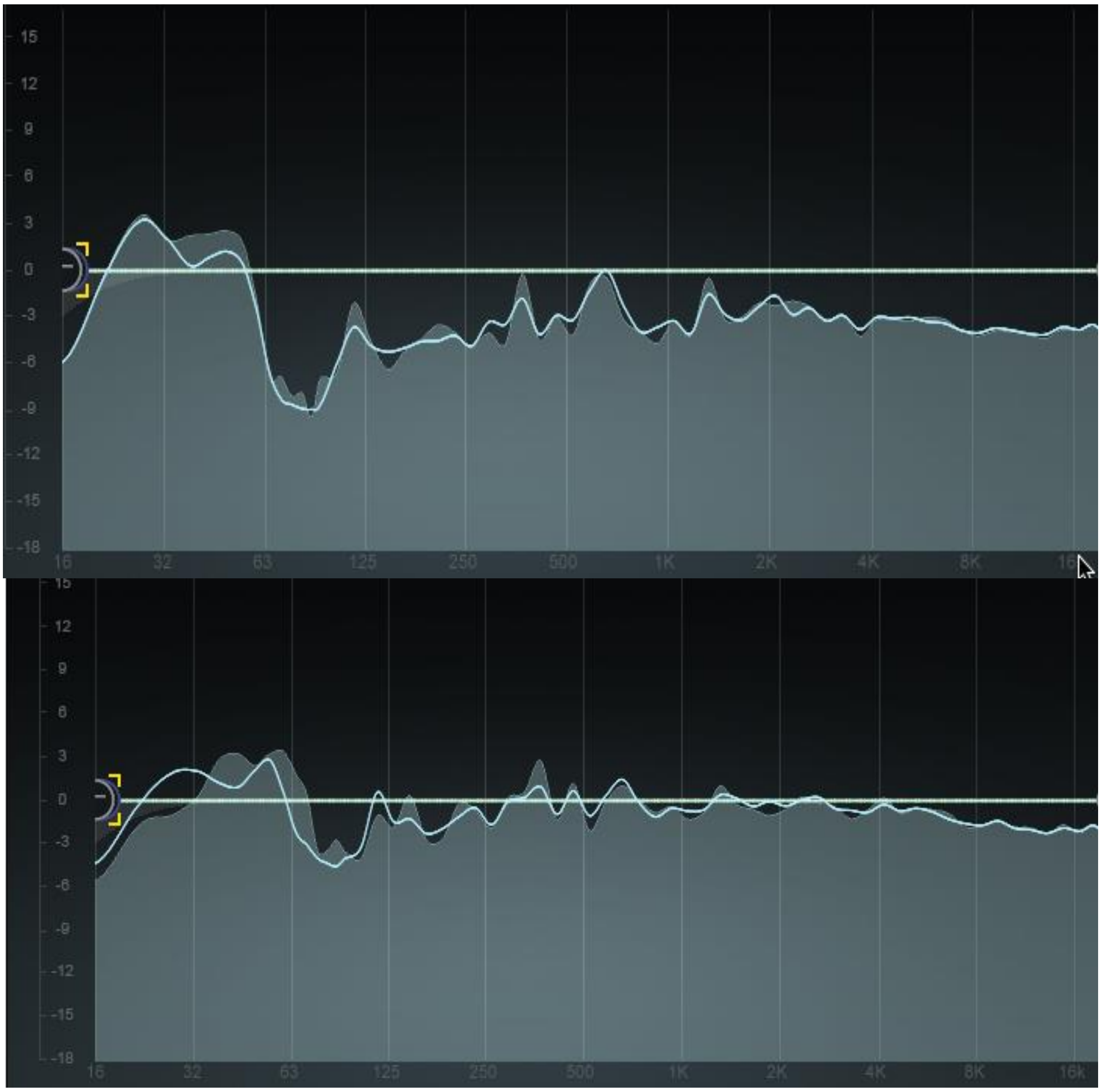


Figura 70. Arriba: Espectro de la señal reverberada con parámetros acústicos al mínimo. Size: 0, Width: 0, Damping: 1. Abajo: Espectro de la señal reverberada con parámetros acústicos a la mitad. Size: 0.5, Width: 0.5, Damping: 0.5.

VIEW
RESPONSE RMS
SLOPE
BANDING 1/6
AVERAGING Medium
MA On
MAX Off
SOURCE L+R
RTA Post On

Figura 71. Espectro de la señal procesada con parámetros acústicos al máximo. Size: 1, Width: 1, Damping: 0.

Al realizar una comparación entre las figuras 70 y 71, se puede observar que el parámetro de amortiguamiento actúa a manera de una especie de filtro similar a un tipo *High Shelf* para altas frecuencias aproximadamente desde 1,5 kHz sobre la señal procesada. También está relacionado con lo que vendría a representar la absorción acústica en altas frecuencias de la “sala hipotética” de reverberación de la señal, siendo directamente proporcional a esta. Se puede observar que los niveles de las frecuencias bajas y medias se mantienen constantes al modificar el amortiguamiento, determinando que este no tiene ningún efecto sobre esas frecuencias, sino su nivel estará seguramente relacionado con los otros parámetros acústicos del módulo. Con los parámetros acústicos al mínimo (figura 70), se produce una caída de más de 9 dB entre la banda frecuencia del 63 a 125 Hz de la señal reverberada. En cambio, al maximizar estos parámetros (figura 71), todo el espectro sube entre 6 y 9 dB aproximadamente. El parámetro de profundidad (*Width*) es el menos notorio en este efecto mientras que el de tamaño (*Size*) es el que más influye en la reverberación.

En general, como estos parámetros pertenecen a la clase de reverberación que viene dada en el marco de referencia y sus valores están normalizados, no se puede cuantificar de manera muy precisa cuales parámetros acústicos reales son los que les corresponden. Sin embargo, con esta prueba realizada se pudo tener una idea más clara del efecto sobre una señal que tiene dichos parámetros dentro del *plugin* desarrollado.

La última prueba realizada en este módulo consistió en determinar el valor máximo, medio y mínimo del tiempo de reverberación que pueden entregar el módulo construido, considerando que en su diseño no se establecieron parámetros temporales. Para tener una idea del tiempo de reverberación (TR) que el *plugin* puede alcanzar se tomó en cuenta la definición del T60. En este caso la fuente sonora fue la misma señal impulsiva que se utilizó en las pruebas de medición de características del módulo de *Delay* y se utilizaron las tres combinaciones de los parámetros acústicos del módulo mostradas en las figuras 70 y 71. De igual manera, en esta prueba el nivel de la señal no procesada se fijó en 0% y el de la procesada en 100%. Se renderizó un audio con cada combinación de parámetros realizada y se analizó la amplitud de cada audio en el programa *Matlab.*

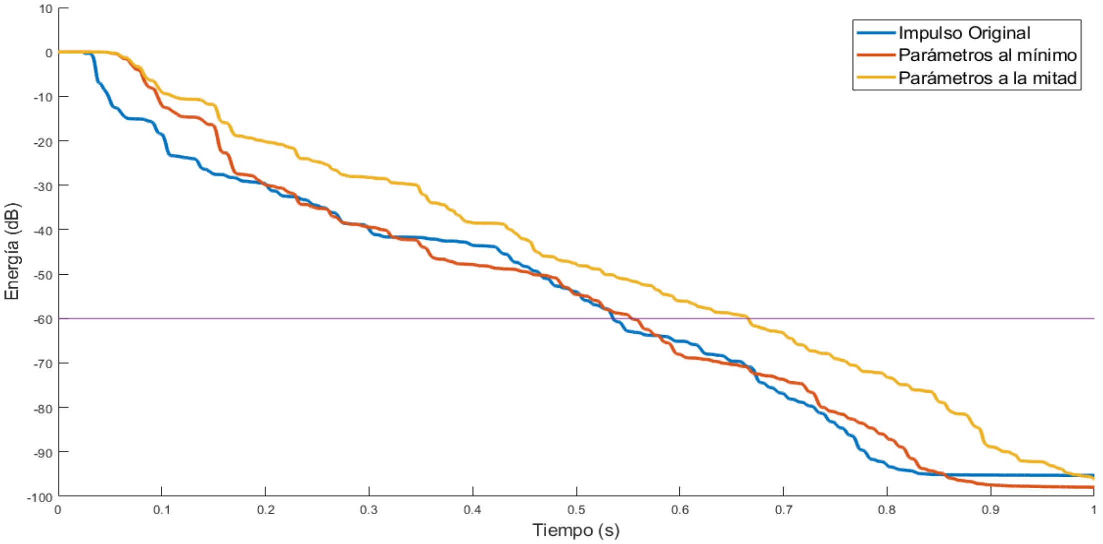


Figura 72. Energía (dB) vs Tiempo(s) del impulso original, señal reverberada con parámetros acústicos al mínimo y con parámetros acústicos a la mitad.

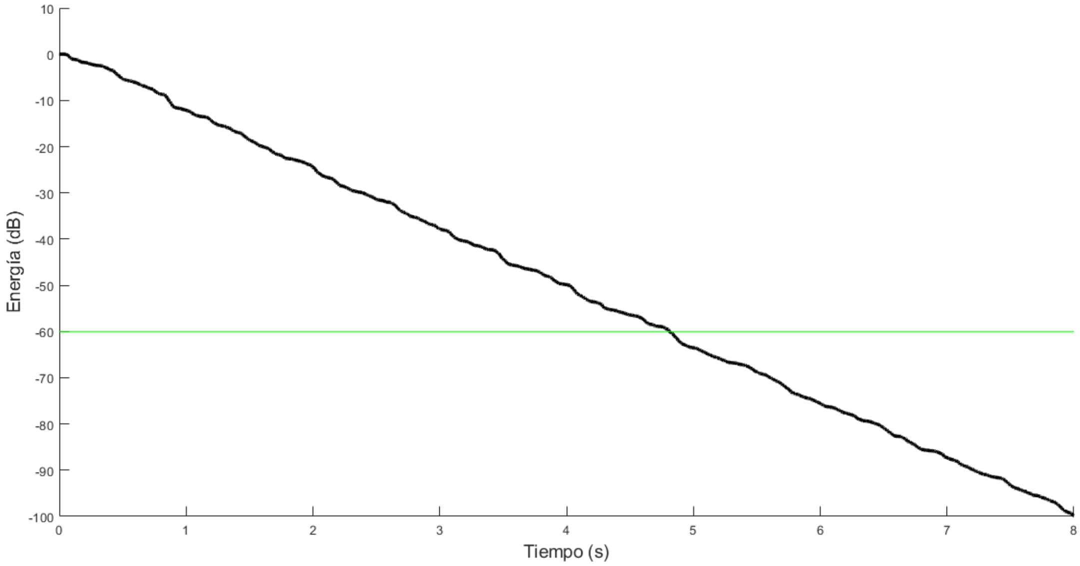


Figura 73. Energía (dB) vs Tiempo (s) de señal reverberada con parámetros acústicos al máximo.

En el caso de la figura 72, se puede observar que en el impulso original se tiene un TR≈ 0,53 s, mientras que al procesar la señal con parámetros acústicos al mínimo (*Size:* 0, *Width:* 0, *Damping:* 1), se obtuvo un TR≈ 0,57 s. Esto significa que el *plugin* en cuestión como mínimo añade aproximadamente 0,04 segundos de reverberación a la señal procesada. También se puede observar (cf. *Figura 72*), que en el caso el que los parámetros acústicos se configuraron a la mitad (*Size:* 0.5, *Width:* 0.5, *Damping:* 0.5), se obtiene un TR≈ 0,67 s, es decir solamente 0,1 segundos superior al caso en el que los parámetros acústicos se configuraron al mínimo. Por último, en la figura 73 se tiene que con los parámetros acústicos al máximo (*Size: 1, Width: 1, Damping: 0*), se llega a obtener un TR≈ 4,8 s, el cual viene a ser el máximo posible para alcanzar con este *plugin*. Entonces, con la información proporcionada por las figuras 72 y 73, se puede afirmar que el módulo de reverberación desarrollado puede simular desde espacios pequeños (TR < 0,6 s) hasta salas muy grandes (TR ≈ 4,8 s), lo cual significa que no tiene limitaciones significativas para cubrir una amplia gama de espacios acústicos. También se puede notar que la proporción en la que los parámetros acústicos del *plugin* de reverberación influyen en el TR no es lineal, ya que, al tener dichos parámetros al mínimo y a la mitad (figura 72), el cambio del TR es muy pequeño, mientras que al tener los parámetros al máximo (figura 73), el cambio es mucho más grande.

Es importante mencionar también, que se puede observar que si bien hay una relación directa entre los parámetros de tamaño y profundidad (*Size* y *Width*) con el tiempo de reverberación y una relación indirecta con el parámetro de amortiguamiento (*Damping*), no se puede determinar exactamente como varía el TR de manera exacta en función de estos tres parámetros. Esto se podría determinar con sistemas de ecuaciones y datos estadísticos muy tediosos, sin embargo, no es necesario ya que puede controlar la reverberación solamente con los parámetros con los que ha sido diseñado el módulo.

4.3.7 Medición del analizador de espectro

Como este módulo no es correspondiente a uno de procesamiento en sí de señales que se modifican y su salida varía con respecto a su entrada, no fueron necesarias muchas pruebas, solamente la verificación de que la escala utilizada para el nivel y para la frecuencia sea la correcta, pero también se probaron tres tipos de señales para observar su representación: ruido rosa, ruido blanco y tonos puros. Al aplicar la señal de ruido rosa, la cual tiene energía que decae proporcionalmente a la frecuencia con pendiente negativa, se puede observar que el módulo desarrollado si muestra de manera un poco escueta dicha pendiente, siendo la representación de este tipo de señal aceptable para tener una idea del funcionamiento del *plugin* en cuestión. En cambio, al aplicar ruido blanco el espectro el plano. Como es mostrado en la figura 74, se cumplen las dos condiciones mencionadas sobre los tipos de ruido utilizados en esta prueba (rosa y blanco). Esto indica que el *plugin* funciona correctamente en cuanto a la representación de este tipo de espectro a diferencia de *plugins* desarrollados por terceros, que muestran el espectro del ruido rosa como el de ruido blanco y viceversa, como es el caso del analizador de espectro utilizado para pruebas del ecualizador (figuras 53, 54).

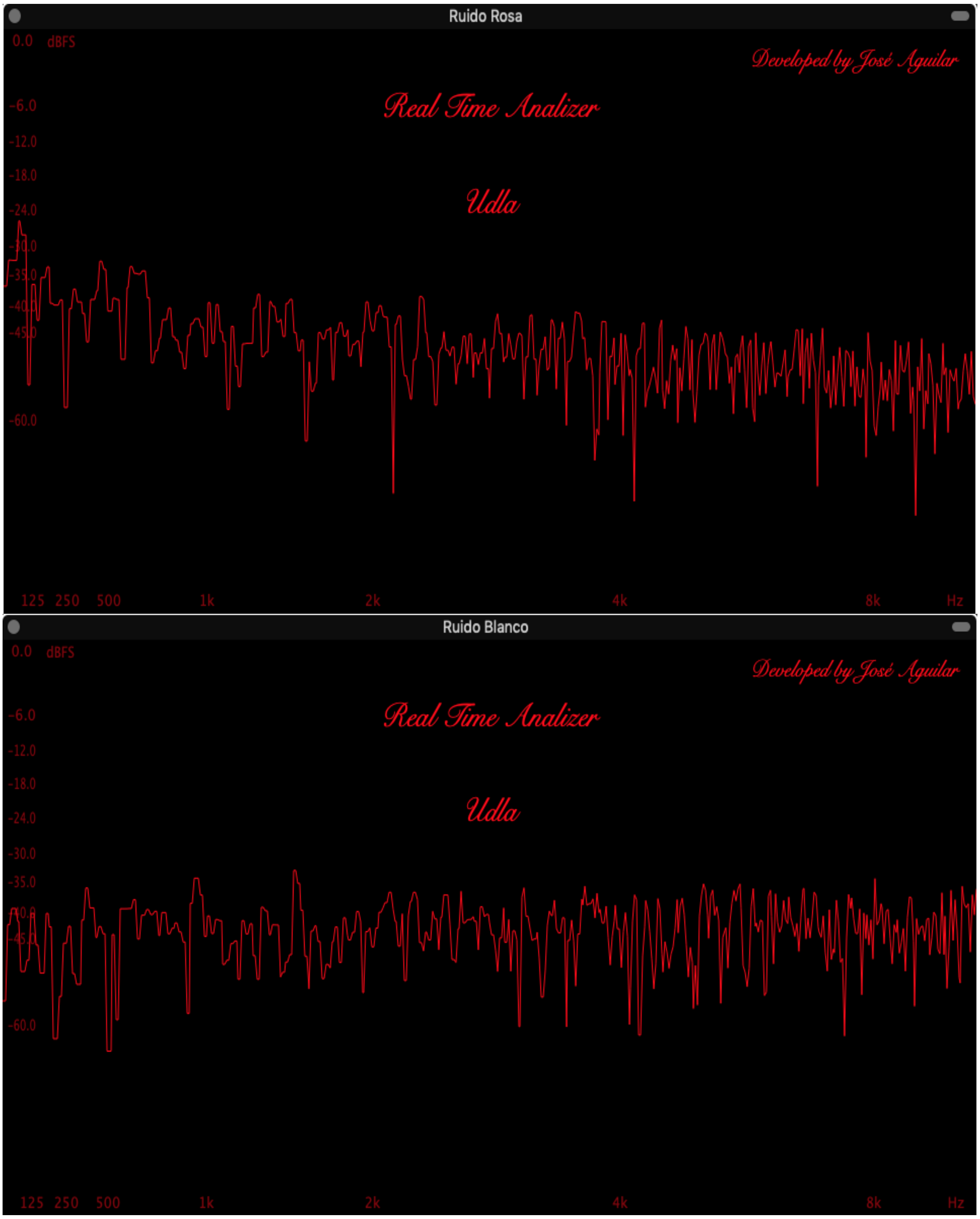


Figura 74. Señales probadas en el analizador de espectro. Arriba: Ruido rosa. Abajo: Ruido blanco.

Para determinar que escala de nivel colocada sea correcta, se utilizó como señal un tono puro de 1 kHz, al cual se le fue modificando el nivel de entrada a los valores marcados en dicha escala (Tabla 10).

Tabla 10.

Niveles de señal en la escala del analizador de espectro a 1 kHz.

Nivel de entrada (dBFS)	Nivel en la escala colocada (dBFS)
-6	-6
-12	-12
-18	-18
-24	-24
-30	-30
-35	-35
-40	-40
-45	-45
-60	-60

Los niveles obtenidos en la tabla 10 fueron correspondientes a los de la escala colocada, lo cual es coherente tomando en cuenta que en la fase de diseño se colocó la escala en base a valores probados de niveles de tonos puros. De igual manera para la escala de frecuencia se utilizó un tono puro de -6 dBFS de nivel y se varió la frecuencia de este (Tabla 11).

Tabla 11.

Escala de frecuencias evaluadas en el analizador de espectro a -6 dBFS.

Frecuencia de entrada (Hz)	Frecuencia en la escala colocada (Hz)
125	125
250	250
500	500
1000	1000
2000	2000
4000	4000
8000	8000

Esta prueba dio resultados de frecuencia correspondientes a los valores reales probados de la misma manera que en la prueba de precisión de nivel (Tabla 10).

4.4 Funcionamiento de las interfaces de usuario

Durante todas las pruebas realizadas se comprobó que las interfaces de usuario de todos los módulos funcionan correctamente y que sus parámetros son automatizables, además se pueden guardar *presets* o estados de configuración de los módulos para abrirlos en otros momentos de ser necesario. La única consideración que se debe realizar en este tema es que para que en los módulos de señales estereofónicas funcionen correctamente los parámetros de enlace de los dos canales de las señales, se debe abrir la interfaz de usuario cada vez que se modifique uno de los parámetros de este tipo.

No es necesario dejar la interfaz abierta todo el tiempo, sino simplemente abrirla cuando el parámetro de enlazamiento de canal se ha modificado o se quiere modificar y después se puede volver a cerrar la interfaz y modificar los demás parámetros para que este enlazamiento sea efectivo. Esto se debe a que el algoritmo de esta sección se codificó en la implementación de la interfaz, y dicha codificación no se ejecuta si esta no está abierta.

En todos los programas anfitriones utilizados, se verificó que la latencia de cada uno de los módulos fue de 0 muestras de audio como se estableció desde la etapa de desarrollo. Esto no significa que la latencia real será de 0 muestras sino solamente la latencia relativa, es decir, el valor total de la real dependerá del tamaño del buffer de procesamiento con el que se trabaje.

En la sección de anexos, se encuentran las figuras complementarias de absolutamente todas las pruebas realizadas que se han presentado en esta parte del documento, como los detalles de las pruebas de procesamiento y figuras de todas las pruebas que se han detallado en esta sección de resultados.

5 CONCLUSIONES Y RECOMENDACIONES

5.1 Conclusiones

En el desarrollo de este proyecto se construyeron 10 módulos de audio, 5 para señales monofónicas y 5 para señales estereofónicas, capaces de procesar señales de audio digital. Según las pruebas realizadas, se puede afirmar que dichos módulos cumplen con criterios y estándares de *plugins* profesionales disponibles en el mercado de la industria de software de audio digital. Todos los módulos creados procesan señales digitales de audio en tiempo real con latencia de relativa de 0 muestras al menos en los softwares anfitriones utilizados en las pruebas (*figura* 40, 41), lo cual significa que la latencia real dependerá del tamaño del buffer de procesamiento, de la de frecuencia de muestreo y de los tipos de conversores analógicos digitales y viceversa con los que se configure el software anfitrión que ejecute los *plugins*. Generalmente, los tamaños de este buffer suelen estar entre los valores de 32 a 2048 muestras, lo cual, dependiendo de la selección de la frecuencia de muestreo equivale a tiempos de entre 0,16 y 46,44 milisegundos, un rango de tiempo que permite utilizar los módulos para aplicaciones en vivo, así como para postproducción. La latencia generada por los conversores dependerá de su tipo y configuración.

El error máximo de actualización de los parámetros no temporales de los módulos desarrollados también dependerá de este tamaño de buffer, lo cual no quiere decir que este sea el error de precisión de los *plugins* sino solamente el error de actualización estos parámetros, siendo este un punto muy importante de enfatizar. En cambio, los parámetros temporales se actualizan con una precisión de muestra a muestra permitiendo obtener una exactitud muy acertada según los valores mostrados en los resultados. Todo esto fue realizado así, ya que, fue la única manera en la que se pudieron optimizar los módulos para que su consumo de procesamiento sea comparable con *plugins* desarrollados por terceros.

El hecho de tener los cuatro efectos de implementación y el analizador de espectro en módulos separados es mucho más práctico que unirlos en un solo *plugin*¸ debido a que, de esta manera, se permite al usuario decidir en qué orden desea procesar las señales sonoras con los efectos, así como permitir utilizar cada efecto más de una vez en una misma cadena de procesamiento. Además de esto, se admite la inversión de polaridad a la entrada de cada *plugin* y manipulación de ganancia tanto en la entrada como la salida de cada efecto.

Con todo lo presentado en este documento, se puede afirmar que cada uno de los módulos desarrollados funcionan de manera estable y bastante precisa en todos los programas utilizados como anfitriones durante la etapa de pruebas (Tabla 1); además, son funcionales para los sistemas operativos *Windows* 10 Pro-2019 y *MacOS Mojave* 10.14.6. Es muy probable que también funcionen correctamente en las DAWs que permitan los formatos de *plugins* VST, VST3 o Audio Unit. Además, cada uno de los módulos es comparable con *plugins* desarrollados por terceros, sin embargo, los resultados en señales procesadas nunca serán los mismos (figuras 59, 63), ya que, los algoritmos del *plugin* de cada empresa son distintos incluso si se trata de un mismo tipo de efecto.

A pesar de que en el diseño y desarrollo de los *plugins* no se concibió que estos sean de alta precisión, el error hallado en el funcionamiento de los *plugins* en cuestión fue muy pequeño, siendo de menos de 0,05 milisegundos para los parámetros temporales y de menos de 0,01 dB para los parámetros de nivel. Esto indica que los valores mostrados en dichos parámetros son confiables y con una exactitud muy alta.

Gracias a la optimización implementada en los módulos, el procesamiento consumido fue relativamente bajo, permitiendo que estos puedan ser utilizados tanto en condiciones de baja demanda de recursos computacionales como en casos de alta demanda, sin provocar fallos o sobrecargas del sistema. Lo cual indica una estabilidad muy alta en la ejecución de estos *plugins*.

Con respecto a la interfaz de usuario, debido al suavizado añadido cuando se varían de sus parámetros, permite manipularlos desde estas o con automatizaciones, sin que se produzcan discontinuidades indeseadas ni eventos sonoros indeterminados en las señales procesadas. El analizador de espectro desarrollado, entrega información confiable y real sobre el nivel frecuencial de las señales que pasan a través de él (*figura* 74*, tabla* 10, 11), brindando de esta manera, una idea muy clara al usuario, sobre los datos disponibles de dichas señales.

5.2 Recomendaciones

Las principales aplicaciones para los módulos desarrollados tienen que ver con su uso en ingeniería de mezcla para procesar distintas señales de audio en sesiones de trabajo en diferentes programas anfitriones, sin embargo, una aplicación no tan notoria a primera vista, puede ser su uso en diseño sonoro. Debido a que, en sus instancias en estéreo, se pueden modificar sus parámetros de cada canal de manera individual, se pueden construir sonidos poco convencionales cuyo alcance dependerá de la creatividad de quien los utilice.

Si se dispone de recursos computaciones limitados y se desea optimizar el procesamiento de ejecución de los módulos aún más de lo que ya vienen dados desde su desarrollo, se recomienda modificar sus parámetros desde la sección de automatización del software anfitrión que se esté utilizando para que, al no abrir las interfaces de usuario, o abrirlas lo menos posibles, se optimice mucho más el consumo de procesamiento requerido. En estos casos también es recomendable no ejecutar otros programas que no sean absolutamente necesarios mientras se trabaja en el procesamiento del audio.

REFERENCIAS

Adimulam, M. K., Movva, K. K., Veeramachaneni, S., Muthukrishnan, N. M., & Srinivas, M. B. (2010). *A low power*, *variable resolution two-step flash ADC*. *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI - GLSVLSI '10*. Recuperado el 24 de octubre de 2019 de doi:10.1145/1785481.1785492

Ashraf, K., Elizalde, B., Iandola, F., Moskewicz, M., Bernd, J., Friedland, G., & Keutzer, K. (2015). *Audio-Based Multimedia Event Detection with DNNs and Sparse Sampling*. *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval* - *ICMR* '15. Recuperado el 24 de octubre de 2019 de doi:10.1145/2671188.2749396

Baelde, M., Biernacki, C., & Greff, R. (2019). *Real-Time monophonic and polyphonic audio classification from power spectra. Pattern Recognition*, 92, 82-92. Recuperado el 2 de noviembre de 2019 de doi:10.1016/j.patcog.2019.03.017

Bessell, D. (2013). *Dynamic Convolution Modeling, a Hybrid Synthesis Strategy. Computer Music Journal*, 37(1), 44-51. Recuperado el 3 de noviembre de 2019 de doi:10.1162/comj_a_00159

Butcher, M. (2014, November 18). *Music Hardware Maker ROLI Acquires JUCE, A Key Music Industry Framework,* Recuperado el 10 de noviembre de 2019 de https://techcrunch.com/2014/11/18/music-hardware-maker-roli-acquires-juce-a-key-music-industry-framework/

D'angelo, S., & Valimaki, V. (2014). *Generalized Moog Ladder Filter: Part I– Linear Analysis and Parameterization. IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(12), 1825-1832. Recuperado el 10 de noviembre de 2019 de doi:10.1109/taslp.2014.2352495

Goussios, C. A., Tsinikas, N., & Kitsiou, N. (2015). *Echoes of reverb. Proceedings of the Audio Mostly 2015 on Interaction with Sound - AM* '15. Recuperado el 15 de noviembre de 2019 de doi:10.1145/2814895.2814915

Hamada, M. (2012). *A learning system for audio compression. Proceedings of the 14th International Conference on Information Integration and Web-based Applications & Services - IIWAS* '12. Recuperado el 15 de noviembre de 2019 de doi:10.1145/2428736.2428795

Hodgson, J. (2010). *A field guide to equalisation and dynamics processing on rock and electronica records. Popular Music*, 29(2), 283-297. Recuperado el 17 de noviembre de 2019 de doi:10.1017/s0261143010000085

Jiang, Y., & Xu, Y. (2014). *Fast computation of the multidimensional discrete Fourier transform and discrete backward Fourier transform on sparse grids. Mathematics of Computation*, 83(289), 2347-2384. Recuperado el 17 de noviembre de 2019 de doi:10.1090/s0025-5718-2014-02785-3

Lazzarini, V., & Timoney, J. (2013). *Synthesis of Resonance by Nonlinear Distortion Methods. Computer Music Journal*, 37(1), 35-43. Recuperado el 20 de noviembre de 2019 de doi:10.1162/comj_a_00160

Lim, F., Zhang, W., Habets, E. A., & Naylor, P. A. (2014). *Robust Multichannel Dereverberation using Relaxed Multichannel Least Squares. IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(9), 1379-1390. Recuperado el 21 de noviembre de 2019 de doi:10.1109/taslp.2014.2329632

Oppenheim, A. V., Willsky, A. S., & Young, I. T. (1990). *Signal and systems. Prentice Hall of India.* New Jersey, United States of América.

Pirkle, W. (2017). *Designing Audio Effect plugins in C++ : With digital audio signal processing theory. CRC Press.* New York, United States of América.

Pirkle, W. (2017). *Designing software synthesizer plugins in C++: For rackafx, vst3, and audio units. CRC Press.* New York, United States of América.

Parga, C. J., & Calleja, M. A. (2015). *UML:* Aplicaciones en *Java y* C*. Ra-Ma.* Editorial *Ra-Ma*. Madrid, España.

Ramo, J., Valimaki, V., & Bank, B. (2014). *High-Precision Parallel Graphic Equalizer. IEEE/ACM Transactions on Audio, Speech, and Language*

Processing, 22(12), 1894-1904. Recuperado el 23 de noviembre de 2019 de doi:10.1109/taslp.2014.2354241

Sabin, A. T., & Pardo, B. (2009). *A method for rapid personalization of audio equalization parameters. Proceedings of the Seventeen ACM International Conference on Multimedia* - MM '09. Recuperado el 24 de noviembre de 2019 de doi:10.1145/1631272.1631410

Samarasinghe, P., Abhayapala, T., Poletti, M., & Betlehem, T. (2015). *An Efficient Parameterization of the Room Transfer Function. IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(12), 2217-2227. Recuperado el 24 de noviembre de 2019 de doi:10.1109/taslp.2015.2475173

Schildt, H. (2009). *C++ soluciones de programación. McGraw-Hill*.

Silva, P. M., Mattos, C. L., & Junior, A. H. (2019). *Audio Plugin Recommendation Systems for Music Production. 2019 8th Brazilian Conference on Intelligent Systems (BRACIS)*. Recuperado el 27 de noviembre de 2019 de doi:10.1109/bracis.2019.00152

Srivatsan, K., & Venkatesan, N. (2020). *Farrow structure based FIR filter design using hybrid optimization. AEU - International Journal of Electronics and Communications*, 114, 153020. Recuperado el 23 de enero de 2020 de doi:10.1016/j.aeue.2019.153020

Stankovic, L., & Brajovic, M. (2018). *Analysis of the Reconstruction of Sparse Signals in the DCT Domain Applied to Audio Signals. IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(7), 1220-1235. Recuperado el 29 de noviembre de 2019 de doi:10.1109/taslp.2018.2819819

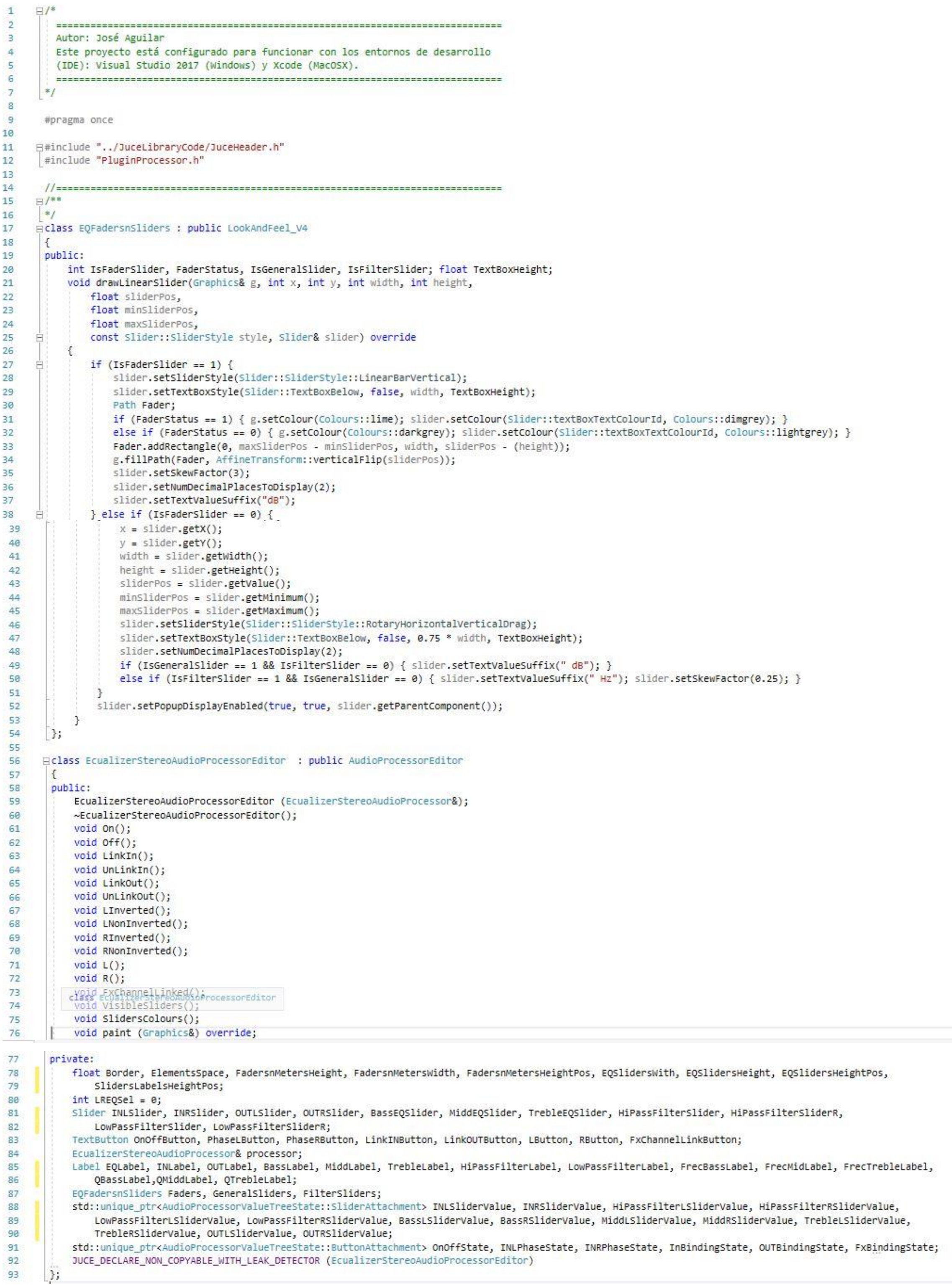
Tanev, G., & Božinovski, A. (2014). *Virtual Studio Technology inside Music Production. ICT Innovations 2013 Advances in Intelligent Systems and Computing*, 231-241. Recuperado el 2 de diciembre de 2019 de doi:10.1007/978-3-319-01466-1_22

Widmark, S. (2018). *Causal IIR Audio Precompensator Filters Subject to Quadratic Constraints. IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(10), 1897-1912. Recuperado el 2 de diciembre de 2019 de doi:10.1109/taslp.2018.2839355

Zivanovic, M. (2015). *Harmonic Bandwidth Companding for Separation of Overlapping Harmonics in Pitched Signals. IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 1-1. Recuperado el 3 de diciembre de 2019 de doi:10.1109/taslp.2015.2412464

ANEXOS

Anexo 1. Algoritmo del módulo de ecualización



```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"

//==============================================================================
/**
*/
class EQFadersnSliders : public LookAndFeel_V4
{
public:
    int IsFaderSlider, FaderStatus, IsGeneralSlider, IsFilterSlider; float TextBoxHeight;
    void drawLinearSlider(Graphics& g, int x, int y, int width, int height,
        float sliderPos,
        float minSliderPos,
        float maxSliderPos,
        const Slider::SliderStyle style, Slider& slider) override
    {
        if (IsFaderSlider == 1) {
            slider.setSliderStyle(Slider::SliderStyle::LinearBarVertical);
            slider.setTextBoxStyle(Slider::TextBoxBelow, false, width, TextBoxHeight);
            Path Fader;
            if (FaderStatus == 1) { g.setColour(Colours::lime); slider.setColour(Slider::textBoxTextColourId, Colours::dimgrey); }
            else if (FaderStatus == 0) { g.setColour(Colours::darkgrey); slider.setColour(Slider::textBoxTextColourId, Colours::lightgrey); }
            Fader.addRectangle(0, maxSliderPos - minSliderPos, width, sliderPos - (height));
            g.fillPath(Fader, AffineTransform::verticalFlip(sliderPos));
            slider.setSkewFactor(3);
            slider.setNumDecimalPlacesToDisplay(2);
            slider.setTextValueSuffix("dB");
        } else if (IsFaderSlider == 0) {
            x = slider.getX();
            y = slider.getY();
            width = slider.getWidth();
            height = slider.getHeight();
            sliderPos = slider.getValue();
            minSliderPos = slider.getMinimum();
            maxSliderPos = slider.getMaximum();
            slider.setSliderStyle(Slider::SliderStyle::RotaryHorizontalVerticalDrag);
            slider.setTextBoxStyle(Slider::TextBoxBelow, false, 0.75 * width, TextBoxHeight);
            slider.setNumDecimalPlacesToDisplay(2);
            if (IsGeneralSlider == 1 && IsFilterSlider == 0) { slider.setTextValueSuffix(" dB"); }
            else if (IsFilterSlider == 1 && IsGeneralSlider == 0) { slider.setTextValueSuffix(" Hz"); slider.setSkewFactor(0.25); }
        }
        slider.setPopupDisplayEnabled(true, true, slider.getParentComponent());
    }
};

class EcualizerStereoAudioProcessorEditor  : public AudioProcessorEditor
{
public:
    EcualizerStereoAudioProcessorEditor (EcualizerStereoAudioProcessor&);
    ~EcualizerStereoAudioProcessorEditor();
    void On();
    void Off();
    void LinkIn();
    void UnLinkIn();
    void LinkOut();
    void UnLinkOut();
    void LInverted();
    void LNonInverted();
    void RInverted();
    void RNonInverted();
    void L();
    void R();
    void FxChannelLinked();
    void VisibleSliders();
    void SlidersColours();
    void paint (Graphics&) override;

private:
    float Border, ElementsSpace, FadersnMetersHeight, FadersnMetersWidth, FadersnMetersHeightPos, EQSlidersWith, EQSlidersHeight, EQSlidersHeightPos,
        SlidersLabelsHeightPos;
    int LREQSel = 0;
    Slider INLSlider, INRSlider, OUTLSlider, OUTRSlider, BassEQSlider, MiddEQSlider, TrebleEQSlider, HiPassFilterSlider, HiPassFilterSliderR,
        LowPassFilterSlider, LowPassFilterSliderR;
    TextButton OnOffButton, PhaseLButton, PhaseRButton, LinkINButton, LinkOUTButton, LButton, RButton, FxChannelLinkButton;
    EcualizerStereoAudioProcessor& processor;
    Label EQLabel, INLabel, OUTLabel, BassLabel, MiddLabel, TrebleLabel, HiPassFilterLabel, LowPassFilterLabel, FrecBassLabel, FrecMidLabel, FrecTrebleLabel,
        QBassLabel,QMiddLabel, QTrebleLabel;
    EQFadersnSliders Faders, GeneralSliders, FilterSliders;
    std::unique_ptr<AudioProcessorValueTreeState::SliderAttachment> INLSliderValue, INRSliderValue, HiPassFilterLSliderValue, HiPassFilterRSliderValue,
        LowPassFilterLSliderValue, LowPassFilterRSliderValue, BassLSliderValue, BassRSliderValue, MiddLSliderValue, MiddRSliderValue, TrebleLSliderValue,
        TrebleRSliderValue, OUTLSliderValue, OUTRSliderValue;
    std::unique_ptr<AudioProcessorValueTreeState::ButtonAttachment> OnOffState, INLPhaseState, INRPhaseState, InBindingState, OUTBindingState, FxBindingState;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (EcualizerStereoAudioProcessorEditor)
};
```

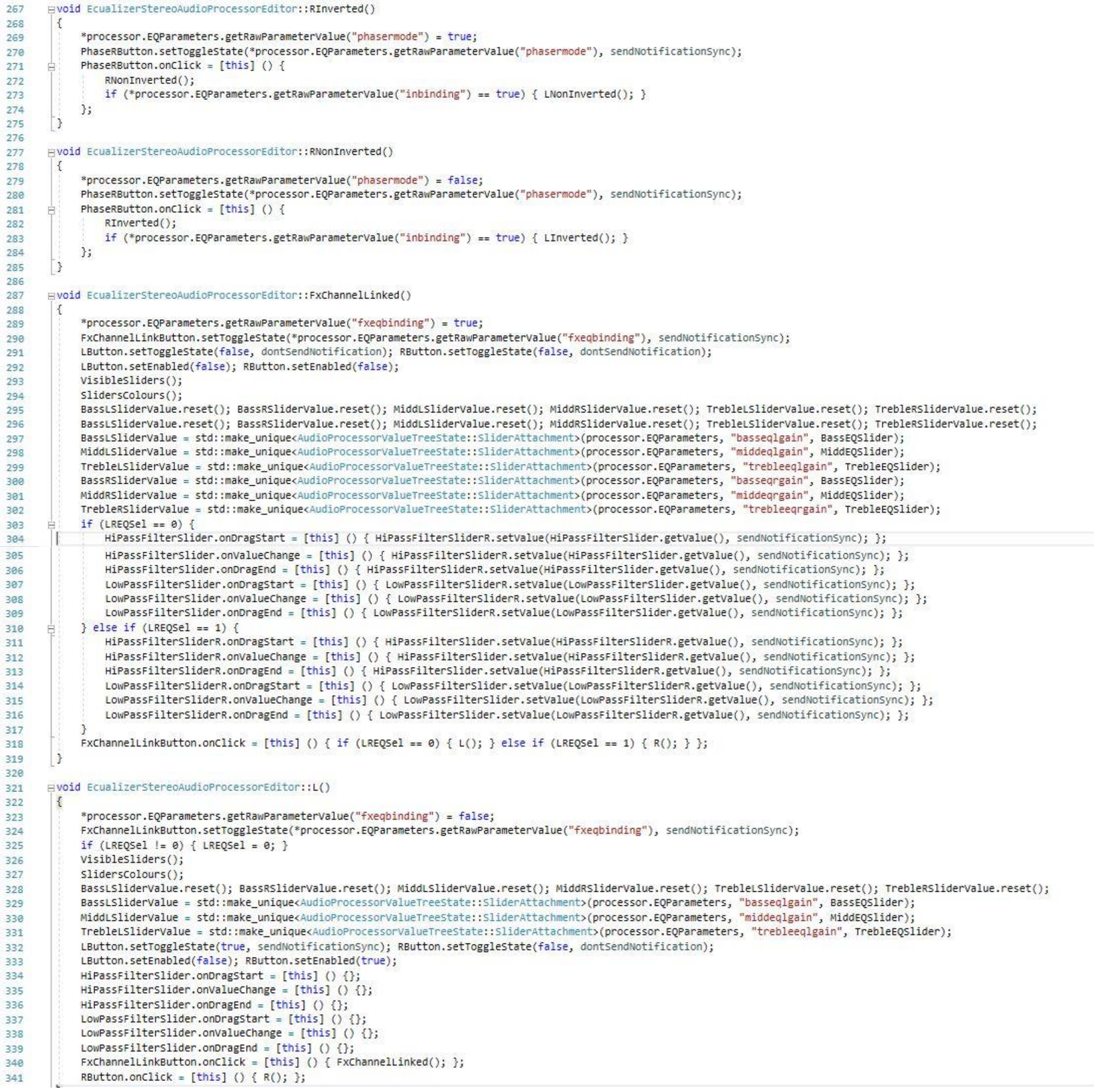
```
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
EcualizerStereoAudioProcessorEditor::EcualizerStereoAudioProcessorEditor (EcualizerStereoAudioProcessor& p)
: AudioProcessorEditor (&p), processor (p)
{
    if (getParentWidth() <= 1400 && getParentHeight() <= 800) { setSize(getParentWidth() / 1.25, 0.45 * getParentHeight()); }
    else { setSize(getParentWidth() / 1.7, 0.35 * getParentHeight()); }
    OnOffState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.EQParameters, "eqstate", OnOffButton);
    INLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters,"inlgain",INLSlider);
    INRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters,"inrgain",INRSlider);
    OUTLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters,"outlgain",OUTLSlider);
    OUTRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters,"outrgain",OUTRSlider);
    OnOffState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.EQParameters, "eqstate", OnOffButton);
    INLPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.EQParameters, "phaselmode", PhaseLButton);
    INRPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.EQParameters, "phasermode", PhaseRButton);
    InBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.EQParameters, "inbinding", LinkINButton);
    OUTBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.EQParameters, "outbinding", LinkOUTButton);
    FxBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.EQParameters, "fxeqbinding", FxChannelLinkButton);
    HiPassFilterLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "hipasseqfilterl", HiPassFilterSlider);
    HiPassFilterRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "hipasseqfilterr", HiPassFilterSliderR);
    LowPassFilterLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "lowpasseqfilterl", LowPassFilterSlider);
    LowPassFilterRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "lowpasseqfilterr", LowPassFilterSliderR);
    INLSlider.setRange(-100.0f, 12.0f, 0.01); INRSlider.setRange(-100.0f, 12.0f, 0.01); OUTLSlider.setRange(-100.0f, 12.0f, 0.01); OUTRSlider.setRange(-100.0f, 12.0f, 0.01);
    HiPassFilterSlider.setRange(20.f, 20000.f, 0.01); LowPassFilterSlider.setRange(20.f, 20000.f, 0.01);  BassEQSlider.setRange(-18.0f, 18.0f, 0.01);
    MiddEQSlider.setRange(-18.0f, 18.0f, 0.01);
    HiPassFilterSliderR.setRange(20.f, 20000.f, 0.01); LowPassFilterSliderR.setRange(20.f, 20000.f, 0.01); TrebleEQSlider.setRange(-18.0f, 18.0f, 0.01);
    DBG("EQ STEREO");
    Border = jmin(getWidth(), getHeight()) / 30;
    ElementsSpace = jmin(getWidth(), getHeight()) / 20;
    FadersnMetersWidth = getWidth() / 25;
    FadersnMetersHeight = 0.65 * getHeight();
    FadersnMetersHeightPos = 0.175 * getHeight();
    SlidersLabelsHeightPos = getHeight() / 2 - ElementsSpace - 0.25 * Border;
    EQSlidersWith = (getWidth() - 2 * (5.5 * ElementsSpace + 3 * FadersnMetersWidth + Border)) / 3;
    EQSlidersHeightPos = getHeight() / 2;
    EQSlidersHeight = FadersnMetersHeight / 2;
    Faders.IsFaderSlider = 1; Faders.IsGeneralSlider = 0; Faders.TextBoxHeight = ElementsSpace; Faders.IsFilterSlider = 0;
    GeneralSliders.IsFaderSlider =0; GeneralSliders.IsGeneralSlider = 1; GeneralSliders.TextBoxHeight = ElementsSpace; GeneralSliders.IsFilterSlider = 0;
    FilterSliders.IsFaderSlider =0; FilterSliders.IsGeneralSlider = 0; FilterSliders.TextBoxHeight = ElementsSpace; FilterSliders.IsFilterSlider = 1;
    EQLabel.setText("Equalizer", sendNotificationSync); INLabel.setText("In", sendNotificationSync); OUTLabel.setText("Out", sendNotificationSync);
    HiPassFilterLabel.setText("Hi Pass 6 dB/Oct", sendNotificationSync);
    LowPassFilterLabel.setText("Low Pass 6 dB/Oct", sendNotificationSync); BassLabel.setText("Bass", sendNotificationSync); MiddLabel.setText("Middle", sendNotificationSync);
    TrebleLabel.setText("Treble", sendNotificationSync);
    LinkINButton.setButtonText("LINK"); FxChannelLinkButton.setButtonText(LinkINButton.getButtonText()); LinkOUTButton.setButtonText(LinkINButton.getButtonText());
    LButton.setButtonText("L"); RButton.setButtonText("R"); PhaseLButton.setButtonText(CharPointer_UTF8("Ø")); OnOffButton.setButtonText(CharPointer_UTF8("•"));
    PhaseRButton.setButtonText(PhaseLButton.getButtonText());
    FrecBassLabel.setText("125 Hz", sendNotificationSync); FrecMidLabel.setText("700 Hz", sendNotificationSync);
    FrecTrebleLabel.setText("4 kHz", sendNotificationSync); QBassLabel.setText("q:3", sendNotificationSync);
    QMiddLabel.setText("q:3", sendNotificationSync); QTrebleLabel.setText("q:3", sendNotificationSync);
    INLSlider.setLookAndFeel(&Faders); INRSlider.setLookAndFeel(&Faders); OUTLSlider.setLookAndFeel(&Faders); OUTRSlider.setLookAndFeel(&Faders);
    HiPassFilterSlider.setLookAndFeel(&FilterSliders); LowPassFilterSlider.setLookAndFeel(&FilterSliders);
    HiPassFilterSliderR.setLookAndFeel(&FilterSliders); LowPassFilterSliderR.setLookAndFeel(&FilterSliders);
    BassEQSlider.setLookAndFeel(&GeneralSliders); MiddEQSlider.setLookAndFeel(&GeneralSliders); TrebleEQSlider.setLookAndFeel(&GeneralSliders);
    EQLabel.setBounds(getWidth() / 2 - 2 * FadersnMetersWidth, ((FadersnMetersHeightPos + Border) / 2) - 0.75 * ElementsSpace, 4 * FadersnMetersWidth, 4 * ElementsSpace);
    INLSlider.setBounds(Border + 1.5 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    INRSlider.setBounds(INLSlider.getX() + INLSlider.getWidth() + 2.f * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    INLabel.setBounds(INRSlider.getX() - 2 * ElementsSpace, (getHeight() * 0.5) - ElementsSpace, 2 * ElementsSpace, 2 * ElementsSpace);
    LinkINButton.setBounds(INRSlider.getX() - 2 * ElementsSpace - 0.75 * FadersnMetersWidth, getHeight() - FadersnMetersHeightPos + 0.875 * ElementsSpace,
        1.5 * FadersnMetersWidth + 2 * ElementsSpace, 1.75 * ElementsSpace);
    PhaseLButton.setBounds(INLSlider.getX(), EQLabel.getY(), INLSlider.getWidth(), 1.5 * ElementsSpace);
    PhaseRButton.setBounds(INRSlider.getX(), EQLabel.getY(), INRSlider.getWidth(), 1.5 * ElementsSpace);
    HiPassFilterLabel.setBounds(Border + 3 * FadersnMetersWidth + 4.5 * ElementsSpace, EQLabel.getY() - 0.5 * Border, 0.5 * (2 * EQSlidersWith + 0.75 * ElementsSpace),
        1.75 * ElementsSpace);
    HiPassFilterSlider.setBounds(HiPassFilterLabel.getX(), EQLabel.getY() + ElementsSpace, HiPassFilterLabel.getWidth(), EQSlidersHeight);
    HiPassFilterSliderR.setBounds(HiPassFilterSlider.getBounds());
    LowPassFilterLabel.setBounds(getWidth() - 3 * FadersnMetersWidth - Border - 4.5 * ElementsSpace - EQSlidersWith, HiPassFilterLabel.getY(),
        HiPassFilterLabel.getWidth(), 1.75 * ElementsSpace);
    LowPassFilterSlider.setBounds(LowPassFilterLabel.getX(), HiPassFilterSlider.getY(), EQSlidersWith, EQSlidersHeight);
    LowPassFilterSliderR.setBounds(LowPassFilterSlider.getBounds());
    BassLabel.setBounds(HiPassFilterLabel.getX(), SlidersLabelsHeightPos, EQSlidersWith, HiPassFilterLabel.getHeight());
    BassEQSlider.setBounds(BassLabel.getX(), EQSlidersHeightPos, EQSlidersWith, EQSlidersHeight);
    MiddLabel.setBounds(BassLabel.getX() + BassLabel.getWidth() +  1.f * ElementsSpace, BassLabel.getY(), EQSlidersWith, BassLabel.getHeight());
    MiddEQSlider.setBounds(MiddLabel.getX(), EQSlidersHeightPos, EQSlidersWith, EQSlidersHeight);
    FxChannelLinkButton.setBounds((getWidth() * 0.5) - (LinkINButton.getWidth() * 0.5), MiddLabel.getY() - LinkINButton.getHeight() - 0.5 * ElementsSpace,
        LinkINButton.getWidth(), LinkINButton.getHeight());
    LButton.setBounds(FxChannelLinkButton.getX() - 2.5 * ElementsSpace, FxChannelLinkButton.getY(), 1.5 * ElementsSpace, FxChannelLinkButton.getHeight());
    RButton.setBounds(FxChannelLinkButton.getX() + FxChannelLinkButton.getWidth() + ElementsSpace, FxChannelLinkButton.getY(), LButton.getWidth(),
        FxChannelLinkButton.getHeight());
    TrebleLabel.setBounds(MiddLabel.getX() + MiddLabel.getWidth() + ElementsSpace, SlidersLabelsHeightPos, EQSlidersWith, MiddLabel.getHeight());
    TrebleEQSlider.setBounds(TrebleLabel.getX(), EQSlidersHeightPos, EQSlidersWith, EQSlidersHeight);
    OUTLSlider.setBounds(getWidth() - Border - 3 * FadersnMetersWidth  - 3.5 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    OUTRSlider.setBounds(OUTLSlider.getX() + OUTLSlider.getWidth() + 2 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    LinkOUTButton.setBounds(OUTRSlider.getX() - 2 * ElementsSpace - 0.75 * FadersnMetersWidth, LinkINButton.getY(), LinkINButton.getWidth(), LinkINButton.getHeight());
    OUTLabel.setBounds(OUTRSlider.getX() - 2 * ElementsSpace, INLabel.getY(), INLabel.getWidth(), INLabel.getHeight());
    OnOffButton.setBounds(OUTRSlider.getX(), EQLabel.getY(), OUTRSlider.getWidth(), PhaseLButton.getHeight());
    FrecBassLabel.setBounds(BassLabel.getX() + 0.325 * EQSlidersWith, LinkINButton.getY(), 0.3 * EQSlidersWith, LinkINButton.getHeight());
    FrecMidLabel.setBounds(MiddLabel.getX() + 0.325 * EQSlidersWith, LinkINButton.getY(), FrecBassLabel.getWidth(), FrecBassLabel.getHeight());
    FrecTrebleLabel.setBounds(TrebleLabel.getX() + 0.325 * EQSlidersWith, LinkINButton.getY(), FrecMidLabel.getWidth(), FrecMidLabel.getHeight());
    QBassLabel.setBounds(BassLabel.getX() + 0.675 * EQSlidersWith, FrecBassLabel.getY(), 0.2 * EQSlidersWith, FrecBassLabel.getHeight());
    QMiddLabel.setBounds(MiddLabel.getX() + 0.675 * EQSlidersWith, FrecMidLabel.getY(), QBassLabel.getWidth(), FrecMidLabel.getHeight());
    QTrebleLabel.setBounds(TrebleLabel.getX() + 0.675 * EQSlidersWith, FrecTrebleLabel.getY(), QMiddLabel.getWidth(), FrecTrebleLabel.getHeight());
    OnOffButton.setColour(TextButton::textColourOffId, Colours::white); OnOffButton.setColour(TextButton::buttonColourId, Colours::black);
    PhaseLButton.setColour(TextButton::textColourOffId, Colours::white); PhaseLButton.setColour(TextButton::buttonColourId, Colours::black);
    PhaseRButton.setColour(TextButton::textColourOffId, Colours::white); PhaseRButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkINButton.setColour(TextButton::textColourOffId, Colours::white); LinkINButton.setColour(TextButton::buttonColourId, Colours::black);
    LButton.setColour(TextButton::textColourOffId, Colours::white); LButton.setColour(TextButton::buttonColourId, Colours::black);
    RButton.setColour(TextButton::textColourOffId, Colours::white); RButton.setColour(TextButton::buttonColourId, Colours::black);
    FxChannelLinkButton.setColour(TextButton::textColourOffId, Colours::white); FxChannelLinkButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkOUTButton.setColour(TextButton::textColourOffId, Colours::white); LinkOUTButton.setColour(TextButton::buttonColourId, Colours::black);
    OnOffButton.setColour(TextButton::textColourOffId, Colours::white); OnOffButton.setColour(TextButton::buttonColourId, Colours::black);
    OnOffButton.setColour(TextButton::textColourOffId, Colours::darkgrey); OnOffButton.setColour(TextButton::buttonColourId, Colours::black);
    FrecBassLabel.setJustificationType(Justification::Flags::centred); FrecMidLabel.setJustificationType(Justification::Flags::centred);
    FrecTrebleLabel.setJustificationType(Justification::Flags::centred); QBassLabel.setJustificationType(Justification::Flags::centred);
    QMiddLabel.setJustificationType(Justification::Flags::centred); QTrebleLabel.setJustificationType(Justification::Flags::centred);
```
`

```
    EQLabel.setJustificationType(Justification::Flags::centred); INLabel.setJustificationType(Justification::Flags::centred);
    OUTLabel.setJustificationType(Justification::Flags::centred);
    HiPassFilterLabel.setJustificationType(Justification::Flags::centred); LowPassFilterLabel.setJustificationType(Justification::Flags::centred);
    BassLabel.setJustificationType(Justification::Flags::centred); MiddLabel.setJustificationType(Justification::Flags::centred);
    TrebleLabel.setJustificationType(Justification::Flags::centred);
    LButton.setColour(TextButton::textColourOnId, Colours::black); RButton.setColour(TextButton::textColourOnId, Colours::black);
    PhaseLButton.setColour(TextButton::textColourOnId, Colours::black); PhaseRButton.setColour(TextButton::textColourOnId, Colours::black);
    LinkINButton.setColour(TextButton::textColourOnId, Colours::black); FxChannelLinkButton.setColour(TextButton::textColourOnId, Colours::black);
    OnOffButton.setColour(TextButton::textColourOnId, Colours::black); LinkOUTButton.setColour(TextButton::textColourOnId, Colours::black);
    addAndMakeVisible(EQLabel); addAndMakeVisible(&INLSlider); addAndMakeVisible(&INRSlider); addAndMakeVisible(INLabel); addAndMakeVisible(&LinkOUTButton);
    addAndMakeVisible(&PhaseLButton); addAndMakeVisible(&PhaseRButton); addAndMakeVisible(HiPassFilterLabel); addAndMakeVisible(&HiPassFilterSlider);
    addAndMakeVisible(LowPassFilterLabel); addAndMakeVisible(&LowPassFilterSlider); addAndMakeVisible(BassLabel); addAndMakeVisible(&BassEQSlider);
    addAndMakeVisible(MiddLabel); addAndMakeVisible(&MiddEQSlider); addAndMakeVisible(TrebleLabel); addAndMakeVisible(&TrebleEQSlider);
    addAndMakeVisible(OUTLabel);
    addAndMakeVisible(&OnOffButton);
    addAndMakeVisible(&LinkINButton); addAndMakeVisible(&FxChannelLinkButton);
    addAndMakeVisible(&LButton); addAndMakeVisible(&RButton); addAndMakeVisible(&OUTLSlider);
    addAndMakeVisible(&OUTRSlider);     addAndMakeVisible(&HiPassFilterSliderR); addAndMakeVisible(&LowPassFilterSliderR);
    addAndMakeVisible(&FrecBassLabel); addAndMakeVisible(&FrecMidLabel); addAndMakeVisible(&FrecTrebleLabel);
    addAndMakeVisible(&QBassLabel); addAndMakeVisible(&QMiddLabel); addAndMakeVisible(&QTrebleLabel);
    if (*processor.EQParameters.getRawParameterValue("eqstate") == true) { On(); }
    else { Off(); }
    if (*processor.EQParameters.getRawParameterValue("phaselmode") == true) { LInverted(); }
    else { LNonInverted(); }
    if (*processor.EQParameters.getRawParameterValue("phasermode") == true) { RInverted(); }
    else { RNonInverted(); }
    if (*processor.EQParameters.getRawParameterValue("inbinding") == true) { LinkIn(); }
    else { UnLinkIn(); }
    if (*processor.EQParameters.getRawParameterValue("outbinding") == true) { LinkOut(); }
    else { UnLinkOut(); }
    if (*processor.EQParameters.getRawParameterValue("fxeqbinding") == true) { FxChannelLinked(); }
    else { if (LREQSel == 0) { L(); } else if (LREQSel == 1) { R(); } }
}

EcualizerStereoAudioProcessorEditor::~EcualizerStereoAudioProcessorEditor()
{
    INLSlider.setLookAndFeel(nullptr); INRSlider.setLookAndFeel(nullptr); OUTLSlider.setLookAndFeel(nullptr); OUTRSlider.setLookAndFeel(nullptr);
    HiPassFilterSlider.setLookAndFeel(nullptr); HiPassFilterSliderR.setLookAndFeel(nullptr); LowPassFilterSlider.setLookAndFeel(nullptr);
```

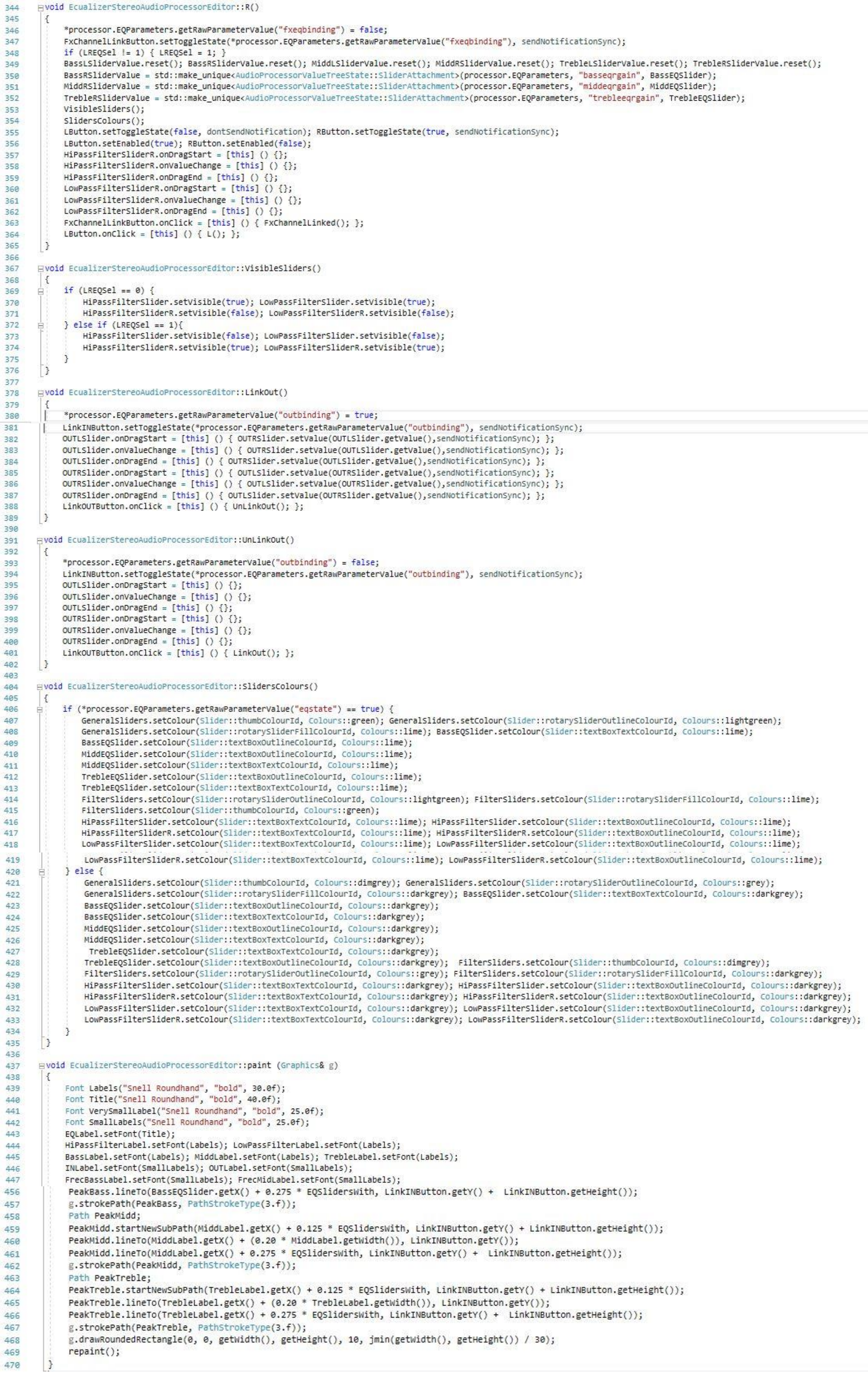
```
    LowPassFilterSliderR.setLookAndFeel(nullptr);
    BassEQSlider.setLookAndFeel(nullptr); MiddEQSlider.setLookAndFeel(nullptr);
    TrebleEQSlider.setLookAndFeel(nullptr);


}

void EcualizerStereoAudioProcessorEditor::On()
{
    *processor.EQParameters.getRawParameterValue("eqstate") = true;
    OnOffButton.setToggleState(*processor.EQParameters.getRawParameterValue("eqstate"), sendNotificationSync);
    Faders.FaderStatus = 1;
    SlidersColours();
    INLabel.setColour(Label::textColourId, Colours::lime); OUTLabel.setColour(Label::textColourId, Colours::lime); EQLabel.setColour(Label::textColourId, Colours::lime);
    HiPassFilterLabel.setColour(Label::textColourId, Colours::lime); LowPassFilterLabel.setColour(Label::textColourId, Colours::lime);
    BassLabel.setColour(Label::textColourId, Colours::lime); MiddLabel.setColour(Label::textColourId, Colours::lime); TrebleLabel.setColour(Label::textColourId, Colours::lime);
    FrecBassLabel.setColour(Label::textColourId, Colours::lime); FrecMidLabel.setColour(Label::textColourId, Colours::lime);
    FrecTrebleLabel.setColour(Label::textColourId, Colours::lime);
    QBassLabel.setColour(Label::textColourId, Colours::lime); QMiddLabel.setColour(Label::textColourId, Colours::lime); QTrebleLabel.setColour(Label::textColourId, Colours::lime);
    LButton.setColour(TextButton::buttonOnColourId, Colours::lime); LButton.setColour(TextButton::textColourOffId, Colours::white);
    LButton.setColour(TextButton::buttonColourId, Colours::black); RButton.setColour(TextButton::buttonOnColourId, Colours::lime);
    RButton.setColour(TextButton::textColourOffId, Colours::white); RButton.setColour(TextButton::buttonColourId, Colours::black);
    FxChannelLinkButton.setColour(TextButton::buttonOnColourId, Colours::lime); FxChannelLinkButton.setColour(TextButton::textColourOffId, Colours::white);
    FxChannelLinkButton.setColour(TextButton::buttonColourId, Colours::black); LinkINButton.setColour(TextButton::buttonOnColourId, Colours::lime);
    LinkINButton.setColour(TextButton::textColourOffId, Colours::white); LinkINButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkOUTButton.setColour(TextButton::buttonOnColourId, Colours::lime); LinkOUTButton.setColour(TextButton::textColourOffId, Colours::white);
    LinkOUTButton.setColour(TextButton::buttonColourId, Colours::black); PhaseLButton.setColour(TextButton::buttonOnColourId, Colours::lime);
    PhaseLButton.setColour(TextButton::textColourOffId, Colours::white); PhaseLButton.setColour(TextButton::buttonColourId, Colours::black);
    PhaseRButton.setColour(TextButton::buttonOnColourId, Colours::lime); PhaseRButton.setColour(TextButton::textColourOffId, Colours::white);
    PhaseRButton.setColour(TextButton::buttonColourId, Colours::black); OnOffButton.setColour(TextButton::buttonOnColourId, Colours::lime);
    OnOffButton.onClick = [this] () { Off(); };
}

void EcualizerStereoAudioProcessorEditor::Off()
{
    *processor.EQParameters.getRawParameterValue("eqstate") = false;
```

```
         CompressorL.setRelease(*OnebandCompressorParameters.getRawParameterValue("releasetime_l"));
        }
        RTRD = ReleaseTimeSmoothR - *OnebandCompressorParameters.getRawParameterValue("releasetime_r");
        if (ReleaseTimeSmoothR != *OnebandCompressorParameters.getRawParameterValue("releasetime_r")) {
            ReleaseTimeSmoothR = ReleaseTimeSmoothR - (((RTRD + 0.001) * (RTRD)) / (RTRD));
            CompressorR.setRelease(ReleaseTimeSmoothR);
        } else {
         CompressorR.setRelease(*OnebandCompressorParameters.getRawParameterValue("releasetime_r"));
        }
        TLD = TresholdSmoothL - *OnebandCompressorParameters.getRawParameterValue("tresholdl");
        if (TresholdSmoothL != *OnebandCompressorParameters.getRawParameterValue("tresholdl")) {
            TresholdSmoothL = TresholdSmoothL - (((TLD + 0.001) * (TLD)) / (TLD));
            CompressorL.setTreshold(TresholdSmoothL);
        } else {
            CompressorL.setTreshold(*OnebandCompressorParameters.getRawParameterValue("tresholdl"));
        }
        TRD = TresholdSmoothR - *OnebandCompressorParameters.getRawParameterValue("tresholdr");
        if (TresholdSmoothR != *OnebandCompressorParameters.getRawParameterValue("tresholdr")) {
            TresholdSmoothR = TresholdSmoothR - (((TRD + 0.001) * (TRD)) / (TRD));
            CompressorR.setTreshold(TresholdSmoothR);
        } else {
            CompressorR.setTreshold(*OnebandCompressorParameters.getRawParameterValue("tresholdr"));
        }
}

void OneBandCompressorAudioProcessor::processBlock (AudioBuffer<float>& buffer, MidiBuffer& midiMessages)
{
    if (SelectedSampleRate != getSampleRate()) {
        SelectedSampleRate = getSampleRate();
    }
    ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();
    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());
    if (*OnebandCompressorParameters.getRawParameterValue("compressorstate") == true) {
        PolarityL = *OnebandCompressorParameters.getRawParameterValue("phaselmode") ? -1.0: 1.0;
        PolarityR = *OnebandCompressorParameters.getRawParameterValue("phasermode") ? -1.0: 1.0;
```

```cpp
    PhaseRButton.onClick = [this] () {
        RNonInverted();
        if (*processor.DelayParameters.getRawParameterValue("inbinding") == true) {
            LNonInverted();
        }
    };
}

void CompatibilityAudioProcessorEditor::RNonInverted()
{
    *processor.DelayParameters.getRawParameterValue("phasermode") = false;
    PhaseRButton.setToggleState(*processor.DelayParameters.getRawParameterValue("phasermode"), sendNotificationSync);
    PhaseRButton.onClick = [this] () {
        RInverted();
        if (*processor.DelayParameters.getRawParameterValue("inbinding") == true) {
            LInverted();
        }
    };
}

void CompatibilityAudioProcessorEditor::SyncTempo()
{
    SyncState();
    DBG("Sync Tempo");
    if (*processor.DelayParameters.getRawParameterValue("fxdelaybinding") == true) {
        DelayNoteLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "notetime_l", NoteTimeSlider);
        DelayNoteRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "notetime_r", NoteTimeSlider);

    } else {
        if (LRDelaySel == 0) {
            DelayNoteLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "notetime_l", NoteTimeSlider);
        } else if (LRDelaySel == 1) {
            DelayNoteRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "notetime_r", NoteTimeSlider);
        }
    }
    NoteTimeSlider.setVisible(true);
    TimeSlider.setVisible(false);
    SyncTempoButton.onClick = [this] () {
```



```cpp
void EcualizerStereoAudioProcessorEditor::RInverted()
{
    *processor.EQParameters.getRawParameterValue("phasermode") = true;
    PhaseRButton.setToggleState(*processor.EQParameters.getRawParameterValue("phasermode"), sendNotificationSync);
    PhaseRButton.onClick = [this] () {
        RNonInverted();
        if (*processor.EQParameters.getRawParameterValue("inbinding") == true) { LNonInverted(); }
    };
}

void EcualizerStereoAudioProcessorEditor::RNonInverted()
{
    *processor.EQParameters.getRawParameterValue("phasermode") = false;
    PhaseRButton.setToggleState(*processor.EQParameters.getRawParameterValue("phasermode"), sendNotificationSync);
    PhaseRButton.onClick = [this] () {
        RInverted();
        if (*processor.EQParameters.getRawParameterValue("inbinding") == true) { LInverted(); }
    };
}

void EcualizerStereoAudioProcessorEditor::FxChannelLinked()
{
    *processor.EQParameters.getRawParameterValue("fxeqbinding") = true;
    FxChannelLinkButton.setToggleState(*processor.EQParameters.getRawParameterValue("fxeqbinding"), sendNotificationSync);
    LButton.setToggleState(false, dontSendNotification); RButton.setToggleState(false, dontSendNotification);
    LButton.setEnabled(false); RButton.setEnabled(false);
    VisibleSliders();
    SlidersColours();
    BassLSliderValue.reset(); BassRSliderValue.reset(); MiddLSliderValue.reset(); MiddRSliderValue.reset(); TrebleLSliderValue.reset(); TrebleRSliderValue.reset();
    BassLSliderValue.reset(); BassRSliderValue.reset(); MiddLSliderValue.reset(); MiddRSliderValue.reset(); TrebleLSliderValue.reset(); TrebleRSliderValue.reset();
    BassLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "basseqlgain", BassEQSlider);
    MiddLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "middeqlgain", MiddEQSlider);
    TrebleLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "trebleeqlgain", TrebleEQSlider);
    BassRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "basseqrgain", BassEQSlider);
    MiddRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "middeqrgain", MiddEQSlider);
    TrebleRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "trebleeqrgain", TrebleEQSlider);
    if (LREQSel == 0) {
        HiPassFilterSlider.onDragStart = [this] () { HiPassFilterSliderR.setValue(HiPassFilterSlider.getValue(), sendNotificationSync); };
        HiPassFilterSlider.onValueChange = [this] () { HiPassFilterSliderR.setValue(HiPassFilterSlider.getValue(), sendNotificationSync); };
        HiPassFilterSlider.onDragEnd = [this] () { HiPassFilterSliderR.setValue(HiPassFilterSlider.getValue(), sendNotificationSync); };
        LowPassFilterSlider.onDragStart = [this] () { LowPassFilterSliderR.setValue(LowPassFilterSlider.getValue(), sendNotificationSync); };
        LowPassFilterSlider.onValueChange = [this] () { LowPassFilterSliderR.setValue(LowPassFilterSlider.getValue(), sendNotificationSync); };
        LowPassFilterSlider.onDragEnd = [this] () { LowPassFilterSliderR.setValue(LowPassFilterSlider.getValue(), sendNotificationSync); };
    } else if (LREQSel == 1) {
        HiPassFilterSliderR.onDragStart = [this] () { HiPassFilterSlider.setValue(HiPassFilterSliderR.getValue(), sendNotificationSync); };
        HiPassFilterSliderR.onValueChange = [this] () { HiPassFilterSlider.setValue(HiPassFilterSliderR.getValue(), sendNotificationSync); };
        HiPassFilterSliderR.onDragEnd = [this] () { HiPassFilterSlider.setValue(HiPassFilterSliderR.getValue(), sendNotificationSync); };
        LowPassFilterSliderR.onDragStart = [this] () { LowPassFilterSlider.setValue(LowPassFilterSliderR.getValue(), sendNotificationSync); };
        LowPassFilterSliderR.onValueChange = [this] () { LowPassFilterSlider.setValue(LowPassFilterSliderR.getValue(), sendNotificationSync); };
        LowPassFilterSliderR.onDragEnd = [this] () { LowPassFilterSlider.setValue(LowPassFilterSliderR.getValue(), sendNotificationSync); };
    }
    FxChannelLinkButton.onClick = [this] () { if (LREQSel == 0) { L(); } else if (LREQSel == 1) { R(); } };
}

void EcualizerStereoAudioProcessorEditor::L()
{
    *processor.EQParameters.getRawParameterValue("fxeqbinding") = false;
    FxChannelLinkButton.setToggleState(*processor.EQParameters.getRawParameterValue("fxeqbinding"), sendNotificationSync);
    if (LREQSel != 0) { LREQSel = 0; }
    VisibleSliders();
    SlidersColours();
    BassLSliderValue.reset(); BassRSliderValue.reset(); MiddLSliderValue.reset(); MiddRSliderValue.reset(); TrebleLSliderValue.reset(); TrebleRSliderValue.reset();
    BassLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "basseqlgain", BassEQSlider);
    MiddLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "middeqlgain", MiddEQSlider);
    TrebleLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "trebleeqlgain", TrebleEQSlider);
    LButton.setToggleState(true, sendNotificationSync); RButton.setToggleState(false, dontSendNotification);
    LButton.setEnabled(false); RButton.setEnabled(true);
    HiPassFilterSlider.onDragStart = [this] () {};
    HiPassFilterSlider.onValueChange = [this] () {};
    HiPassFilterSlider.onDragEnd = [this] () {};
    LowPassFilterSlider.onDragStart = [this] () {};
    LowPassFilterSlider.onValueChange = [this] () {};
    LowPassFilterSlider.onDragEnd = [this] () {};
    FxChannelLinkButton.onClick = [this] () { FxChannelLinked(); };
    RButton.onClick = [this] () { R(); };
```

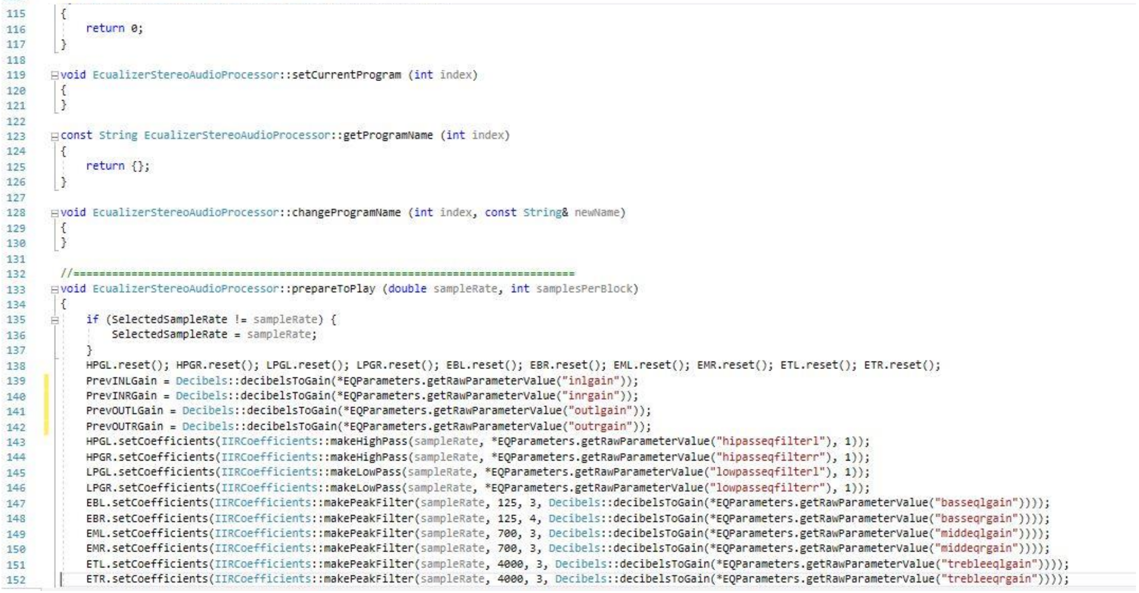


```cpp
void EcualizerStereoAudioProcessorEditor::R()
{
    *processor.EQParameters.getRawParameterValue("fxeqbinding") = false;
    FxChannelLinkButton.setToggleState(*processor.EQParameters.getRawParameterValue("fxeqbinding"), sendNotificationSync);
    if (LREQSel != 1) { LREQSel = 1; }
    BassLSliderValue.reset(); BassRSliderValue.reset(); MiddLSliderValue.reset(); MiddRSliderValue.reset(); TrebleLSliderValue.reset(); TrebleRSliderValue.reset();
    BassRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "basseqrgain", BassEQSlider);
    MiddRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "middeqrgain", MiddEQSlider);
    TrebleRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.EQParameters, "trebleeqrgain", TrebleEQSlider);
    VisibleSliders();
    SlidersColours();
    LButton.setToggleState(false, dontSendNotification); RButton.setToggleState(true, sendNotificationSync);
    LButton.setEnabled(true); RButton.setEnabled(false);
    HiPassFilterSliderR.onDragStart = [this] () {};
    HiPassFilterSliderR.onValueChange = [this] () {};
    HiPassFilterSliderR.onDragEnd = [this] () {};
    LowPassFilterSliderR.onDragStart = [this] () {};
    LowPassFilterSliderR.onValueChange = [this] () {};
    LowPassFilterSliderR.onDragEnd = [this] () {};
    FxChannelLinkButton.onClick = [this] () { FxChannelLinked(); };
    LButton.onClick = [this] () { L(); };
}

void EcualizerStereoAudioProcessorEditor::VisibleSliders()
{
    if (LREQSel == 0) {
        HiPassFilterSlider.setVisible(true); LowPassFilterSlider.setVisible(true);
        HiPassFilterSliderR.setVisible(false); LowPassFilterSliderR.setVisible(false);
    } else if (LREQSel == 1){
        HiPassFilterSlider.setVisible(false); LowPassFilterSlider.setVisible(false);
        HiPassFilterSliderR.setVisible(true); LowPassFilterSliderR.setVisible(true);
    }
}

void EcualizerStereoAudioProcessorEditor::LinkOut()
{
    *processor.EQParameters.getRawParameterValue("outbinding") = true;
    LinkINButton.setToggleState(*processor.EQParameters.getRawParameterValue("outbinding"), sendNotificationSync);
    OUTLSlider.onDragStart = [this] () { OUTRSlider.setValue(OUTLSlider.getValue(),sendNotificationSync); };
    OUTLSlider.onValueChange = [this] () { OUTRSlider.setValue(OUTLSlider.getValue(),sendNotificationSync); };
    OUTLSlider.onDragEnd = [this] () { OUTRSlider.setValue(OUTLSlider.getValue(),sendNotificationSync); };
    OUTRSlider.onDragStart = [this] () { OUTLSlider.setValue(OUTRSlider.getValue(),sendNotificationSync); };
    OUTRSlider.onValueChange = [this] () { OUTLSlider.setValue(OUTRSlider.getValue(),sendNotificationSync); };
    OUTRSlider.onDragEnd = [this] () { OUTLSlider.setValue(OUTRSlider.getValue(),sendNotificationSync); };
    LinkOUTButton.onClick = [this] () { UnLinkOut(); };
}

void EcualizerStereoAudioProcessorEditor::UnLinkOut()
{
    *processor.EQParameters.getRawParameterValue("outbinding") = false;
    LinkINButton.setToggleState(*processor.EQParameters.getRawParameterValue("outbinding"), sendNotificationSync);
    OUTLSlider.onDragStart = [this] () {};
    OUTLSlider.onValueChange = [this] () {};
    OUTLSlider.onDragEnd = [this] () {};
    OUTRSlider.onDragStart = [this] () {};
    OUTRSlider.onValueChange = [this] () {};
    OUTRSlider.onDragEnd = [this] () {};
    LinkOUTButton.onClick = [this] () { LinkOut(); };
}

void EcualizerStereoAudioProcessorEditor::SlidersColours()
{
    if (*processor.EQParameters.getRawParameterValue("eqstate") == true) {
        GeneralSliders.setColour(Slider::thumbColourId, Colours::green); GeneralSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgreen);
        GeneralSliders.setColour(Slider::rotarySliderFillColourId, Colours::lime); BassEQSlider.setColour(Slider::textBoxTextColourId, Colours::lime);
        BassEQSlider.setColour(Slider::textBoxOutlineColourId, Colours::lime);
        MiddEQSlider.setColour(Slider::textBoxOutlineColourId, Colours::lime);
        MiddEQSlider.setColour(Slider::textBoxTextColourId, Colours::lime);
        TrebleEQSlider.setColour(Slider::textBoxOutlineColourId, Colours::lime);
        TrebleEQSlider.setColour(Slider::textBoxTextColourId, Colours::lime);
        FilterSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgreen); FilterSliders.setColour(Slider::rotarySliderFillColourId, Colours::lime);
        FilterSliders.setColour(Slider::thumbColourId, Colours::green);
        HiPassFilterSlider.setColour(Slider::textBoxTextColourId, Colours::lime); HiPassFilterSlider.setColour(Slider::textBoxOutlineColourId, Colours::lime);
        HiPassFilterSliderR.setColour(Slider::textBoxTextColourId, Colours::lime); HiPassFilterSliderR.setColour(Slider::textBoxOutlineColourId, Colours::lime);
        LowPassFilterSlider.setColour(Slider::textBoxTextColourId, Colours::lime); LowPassFilterSlider.setColour(Slider::textBoxOutlineColourId, Colours::lime);
        LowPassFilterSliderR.setColour(Slider::textBoxTextColourId, Colours::lime); LowPassFilterSliderR.setColour(Slider::textBoxOutlineColourId, Colours::lime);
    } else {
        GeneralSliders.setColour(Slider::thumbColourId, Colours::dimgrey); GeneralSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::grey);
        GeneralSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey); BassEQSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey);
        BassEQSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        BassEQSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey);
        MiddEQSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        MiddEQSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey);
         TrebleEQSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey);
        TrebleEQSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);  FilterSliders.setColour(Slider::thumbColourId, Colours::dimgrey);
        FilterSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::grey); FilterSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey);
        HiPassFilterSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); HiPassFilterSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        HiPassFilterSliderR.setColour(Slider::textBoxTextColourId, Colours::darkgrey); HiPassFilterSliderR.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        LowPassFilterSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); LowPassFilterSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        LowPassFilterSliderR.setColour(Slider::textBoxTextColourId, Colours::darkgrey); LowPassFilterSliderR.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
    }
}

void EcualizerStereoAudioProcessorEditor::paint (Graphics& g)
{
    Font Labels("Snell Roundhand", "bold", 30.0f);
    Font Title("Snell Roundhand", "bold", 40.0f);
    Font VerySmallLabel("Snell Roundhand", "bold", 25.0f);
    Font SmallLabels("Snell Roundhand", "bold", 25.0f);
    EQLabel.setFont(Title);
    HiPassFilterLabel.setFont(Labels); LowPassFilterLabel.setFont(Labels);
    BassLabel.setFont(Labels); MiddLabel.setFont(Labels); TrebleLabel.setFont(Labels);
    INLabel.setFont(SmallLabels); OUTLabel.setFont(SmallLabels);
    FrecBassLabel.setFont(SmallLabels); FrecMidLabel.setFont(SmallLabels);
    PeakBass.lineTo(BassEQSlider.getX() + 0.275 * EQSlidersWith, LinkINButton.getY() +  LinkINButton.getHeight());
    g.strokePath(PeakBass, PathStrokeType(3.f));
    Path PeakMidd;
    PeakMidd.startNewSubPath(MiddLabel.getX() + 0.125 * EQSlidersWith, LinkINButton.getY() + LinkINButton.getHeight());
    PeakMidd.lineTo(MiddLabel.getX() + (0.20 * MiddLabel.getWidth()), LinkINButton.getY());
    PeakMidd.lineTo(MiddLabel.getX() + 0.275 * EQSlidersWith, LinkINButton.getY() +  LinkINButton.getHeight());
    g.strokePath(PeakMidd, PathStrokeType(3.f));
    Path PeakTreble;
    PeakTreble.startNewSubPath(TrebleLabel.getX() + 0.125 * EQSlidersWith, LinkINButton.getY() + LinkINButton.getHeight());
    PeakTreble.lineTo(TrebleLabel.getX() + (0.20 * TrebleLabel.getWidth()), LinkINButton.getY());
    PeakTreble.lineTo(TrebleLabel.getX() + 0.275 * EQSlidersWith, LinkINButton.getY() +  LinkINButton.getHeight());
    g.strokePath(PeakTreble, PathStrokeType(3.f));
    g.drawRoundedRectangle(0, 0, getWidth(), getHeight(), 10, jmin(getWidth(), getHeight()) / 30);
    repaint();
}
```

```
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#pragma once

#include "../JuceLibraryCode/JuceHeader.h"

//==============================================================================
/**
*/
class EcualizerStereoAudioProcessor  : public AudioProcessor
{
public:
    //==============================================================================
    EcualizerStereoAudioProcessor();
    ~EcualizerStereoAudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif

    void processBlock (AudioBuffer<float>&, MidiBuffer&) override;

    //==============================================================================
    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==============================================================================
    const String getName() const override;
    const String getName() const override;

    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;

    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const String getProgramName (int index) override;
    void changeProgramName (int index, const String& newName) override;

    //==============================================================================
    void getStateInformation (MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;
    void updateParameters();
    AudioProcessorValueTreeState EQParameters;
    AudioProcessorValueTreeState::ParameterLayout createParameterLayout();

private:
    float PolarityL, PolarityR, CurINLGain, CurINRGain, PrevINLGain, PrevINRGain, PrevOUTLGain, PrevOUTRGain, CurOutLGain, CurOutRGain, HPLSmooth,
        LPLSmooth, BassLSmooth, MiddLSmooth, TrebleLSmooth, SelectedSampleRate, HPRSmooth, LPRSmooth, BassRSmooth, MiddRSmooth, TrebleRSmooth;
    IIRFilter HPGL, HPGR, LPGL, LPGR, EBL, EBR, EML, EMR, ETL, ETR;
    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (EcualizerStereoAudioProcessor)
};
```

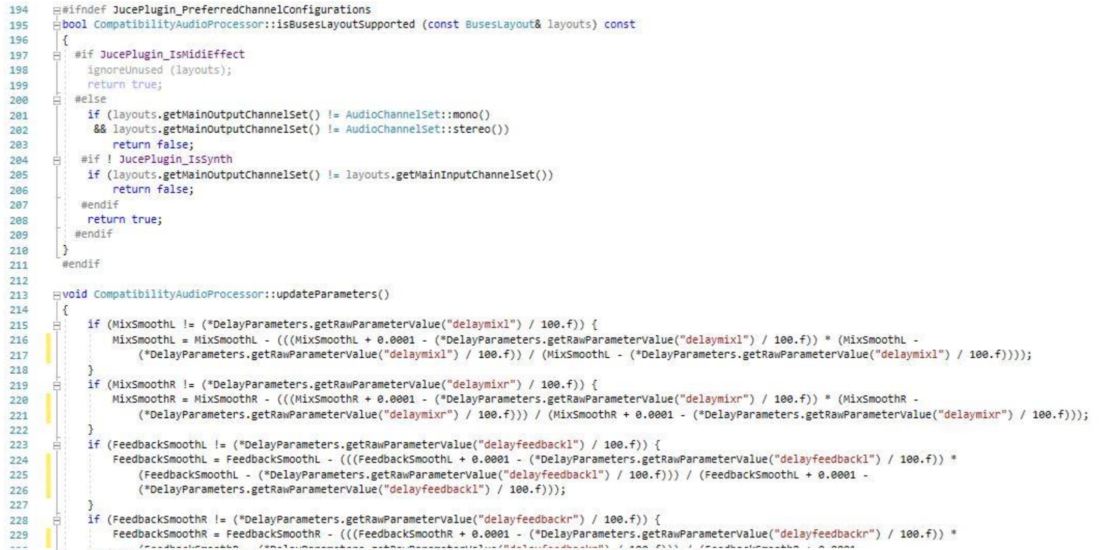
```
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
EcualizerStereoAudioProcessor::EcualizerStereoAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties().withInput("INPUT", AudioChannelSet::stereo(), true)
                                        .withOutput("OUTPUT", AudioChannelSet::stereo(), true)
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", AudioChannelSet::stereo(), true)
                     #endif
                       ), EQParameters(*this, nullptr, "Parameters", createParameterLayout())
#endif
{
}

EcualizerStereoAudioProcessor::~EcualizerStereoAudioProcessor()
{
}

AudioProcessorValueTreeState::ParameterLayout EcualizerStereoAudioProcessor::createParameterLayout()
{
    std::vector<std::unique_ptr<RangedAudioParameter>> EQParameters;
    /*Parámetros Generales*/
    auto EQState = std::make_unique<AudioParameterBool>("eqstate", "Plugin State", true);
    auto INBinding = std::make_unique<AudioParameterBool>("inbinding", "Link In", true);
    auto OUTBinding = std::make_unique<AudioParameterBool>("outbinding", "Link Out", true);
```

```cpp
    auto PhaseLMode = std::make_unique<AudioParameterBool>("phaselmode", "Polarity L", false);
    auto PhaseRMode = std::make_unique<AudioParameterBool>("phasermode", "Polarity R", false);
    auto FxEQBinding = std::make_unique<AudioParameterBool>("fxeqbinding", "Fx Channel Link", true);
    /*Parámetros de entrada y salida*/
    auto INLGainParamater = std::make_unique<AudioParameterFloat>("inlgain", "IN L Gain", -100.0f, 12.0f, 0.0f);
    auto INRGainParameter = std::make_unique<AudioParameterFloat>("inrgain", "IN R Gain", -100.0f, 12.0f, 0.0f);
    auto OUTLGainParameter = std::make_unique<AudioParameterFloat>("outlgain", "OUT L Gain", -100.0f, 12.0f, 0.0f);
    auto OUTRGainParameter = std::make_unique<AudioParameterFloat>("outrgain", "OUT R Gain", -100.0f, 12.0f, 0.0f);
    /*Plugin Parameteres*/
    auto HipassEQFilterLParameter = std::make_unique<AudioParameterFloat>("hipasseqfilterl", "HP Filter L", 20.f, 20000.f, 20.f);
    auto HipassEQFilterRParameter = std::make_unique<AudioParameterFloat>("hipasseqfilterr", "HP Filter R", 20.f, 20000.f, 20.f);
    auto LowPassEQFilterLParameter = std::make_unique<AudioParameterFloat>("lowpasseqfilterl", "LP Filter L", 20.f, 20000.f, 20000.f);
    auto LowPassEQFilterRParameter = std::make_unique<AudioParameterFloat>("lowpasseqfilterr", "LP Filter R", 20.f, 20000.f, 20000.f);
    auto BassEQLParameter = std::make_unique<AudioParameterFloat >("basseqlgain", "Bass EQ L", -18.0f, 18.0f, 0.0f);
    auto MiddEQLParameter = std::make_unique<AudioParameterFloat >("middeqlgain", "Midd EQ L", -18.0f, 18.0f, 0.0f);
    auto TrebleEQLParameter = std::make_unique<AudioParameterFloat >("trebleeqlgain", "Treble EQ L", -18.0f, 18.0f, 0.0f);
    auto BassEQRParameter = std::make_unique<AudioParameterFloat >("basseqrgain", "Bass EQ R", -18.0f, 18.0f, 0.0f);
    auto MiddEQRParameter = std::make_unique<AudioParameterFloat >("middeqrgain", "Midd EQ R", -18.0f, 18.0f, 0.0f);
    auto TrebleEQRParameter = std::make_unique<AudioParameterFloat >("trebleeqrgain", "Treble EQ R", -18.0f, 18.0f, 0.0f);
    /*Push Back*/
    EQParameters.push_back(std::move(EQState)); EQParameters.push_back(std::move(INBinding)); EQParameters.push_back(std::move(PhaseLMode));
    EQParameters.push_back(std::move(PhaseRMode));
    EQParameters.push_back(std::move(INLGainParamater)); EQParameters.push_back(std::move(INRGainParameter)); EQParameters.push_back(std::move(FxEQBinding));
    EQParameters.push_back(std::move(HipassEQFilterLParameter));
    EQParameters.push_back(std::move(HipassEQFilterRParameter)); EQParameters.push_back(std::move(LowPassEQFilterLParameter));
    EQParameters.push_back(std::move(LowPassEQFilterRParameter));
    EQParameters.push_back(std::move(BassEQLParameter)); EQParameters.push_back(std::move(BassEQRParameter)); EQParameters.push_back(std::move(MiddEQLParameter));
    EQParameters.push_back(std::move(MiddEQRParameter)); EQParameters.push_back(std::move(TrebleEQLParameter)); EQParameters.push_back(std::move(TrebleEQRParameter));
    EQParameters.push_back(std::move(OUTBinding)); EQParameters.push_back(std::move(OUTLGainParameter)); EQParameters.push_back(std::move(OUTRGainParameter));
    return { EQParameters.begin(), EQParameters.end() };
}

//==============================================================================
const String EcualizerStereoAudioProcessor::getName() const
{
    return JucePlugin_Name;
}

bool EcualizerStereoAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}

bool EcualizerStereoAudioProcessor::producesMidi() const
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
    return false;
   #endif
}

bool EcualizerStereoAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
   #else
    return false;
   #endif
}

double EcualizerStereoAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int EcualizerStereoAudioProcessor::getNumPrograms()
{
    return 1;
}

```



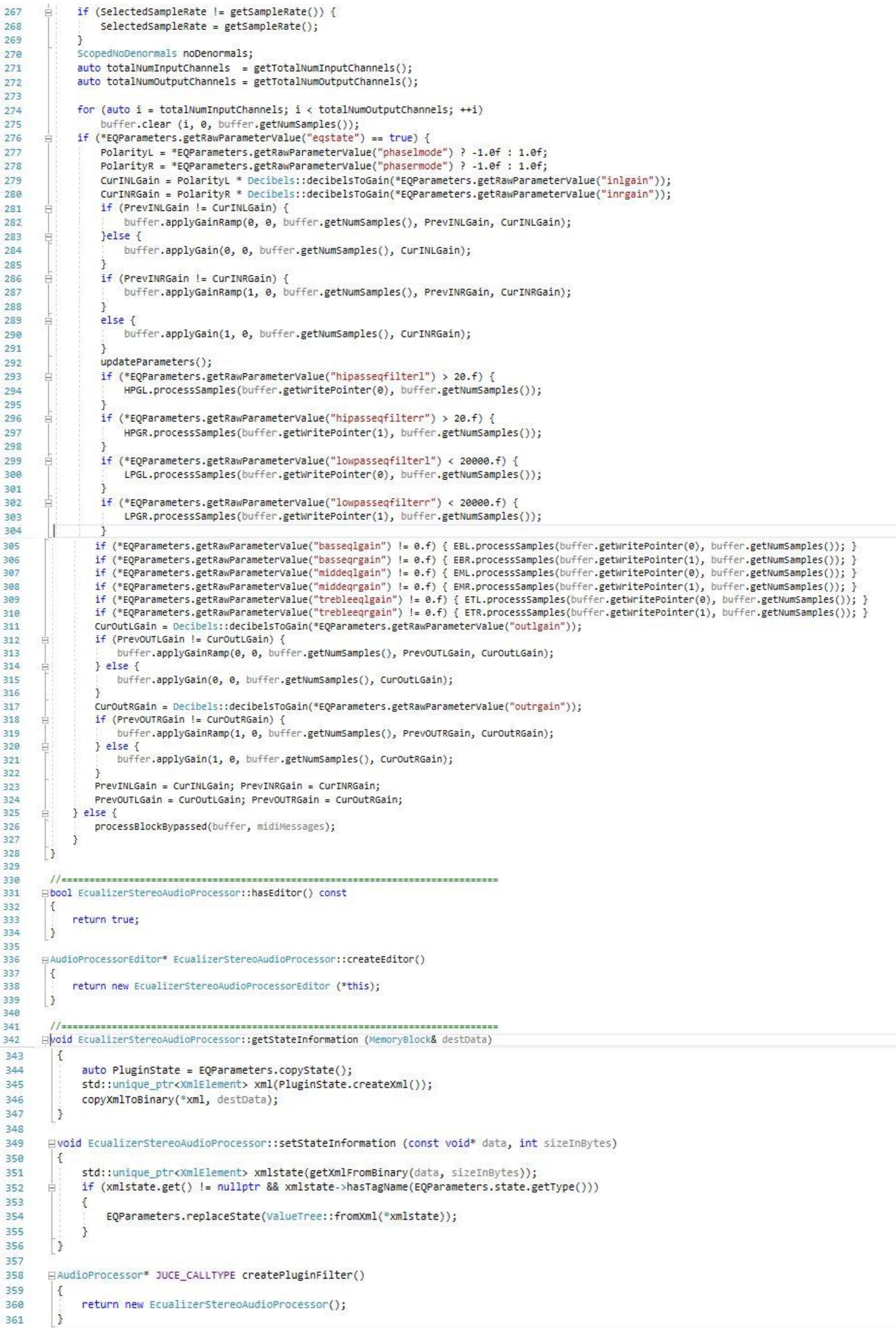
```cpp
{
    return 0;
}

void EcualizerStereoAudioProcessor::setCurrentProgram (int index)
{
}

const String EcualizerStereoAudioProcessor::getProgramName (int index)
{
    return {};
}

void EcualizerStereoAudioProcessor::changeProgramName (int index, const String& newName)
{
}

//==============================================================================
void EcualizerStereoAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    if (SelectedSampleRate != sampleRate) {
        SelectedSampleRate = sampleRate;
    }
    HPGL.reset(); HPGR.reset(); LPGL.reset(); LPGR.reset(); EBL.reset(); EBR.reset(); EML.reset(); EMR.reset(); ETL.reset(); ETR.reset();
    PrevINLGain = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("inlgain"));
    PrevINRGain = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("inrgain"));
    PrevOUTLGain = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("outlgain"));
    PrevOUTRGain = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("outrgain"));
    HPGL.setCoefficients(IIRCoefficients::makeHighPass(sampleRate, *EQParameters.getRawParameterValue("hipasseqfilterl"), 1));
    HPGR.setCoefficients(IIRCoefficients::makeHighPass(sampleRate, *EQParameters.getRawParameterValue("hipasseqfilterr"), 1));
    LPGL.setCoefficients(IIRCoefficients::makeLowPass(sampleRate, *EQParameters.getRawParameterValue("lowpasseqfilterl"), 1));
    LPGR.setCoefficients(IIRCoefficients::makeLowPass(sampleRate, *EQParameters.getRawParameterValue("lowpasseqfilterr"), 1));
    EBL.setCoefficients(IIRCoefficients::makePeakFilter(sampleRate, 125, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("basseqlgain"))));
    EBR.setCoefficients(IIRCoefficients::makePeakFilter(sampleRate, 125, 4, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("basseqrgain"))));
    EML.setCoefficients(IIRCoefficients::makePeakFilter(sampleRate, 700, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqlgain"))));
    EMR.setCoefficients(IIRCoefficients::makePeakFilter(sampleRate, 700, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqrgain"))));
    ETL.setCoefficients(IIRCoefficients::makePeakFilter(sampleRate, 4000, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqlgain"))));
    ETR.setCoefficients(IIRCoefficients::makePeakFilter(sampleRate, 4000, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqrgain"))));
```

```cpp
    HPLSmooth = *EQParameters.getRawParameterValue("hipasseqfilterl");
    HPRSmooth = *EQParameters.getRawParameterValue("hipasseqfilterr");
    LPLSmooth = *EQParameters.getRawParameterValue("lowpasseqfilterl");
    LPRSmooth = *EQParameters.getRawParameterValue("lowpasseqfilterr");
    BassLSmooth = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("basseqlgain"));
    BassRSmooth = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("basseqrgain"));
    MiddLSmooth = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqlgain"));
    MiddRSmooth = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqrgain"));
    TrebleLSmooth = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqlgain"));
    TrebleLSmooth = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqrgain"));

}

void EcualizerStereoAudioProcessor::releaseResources()
{
    HPGL.reset(); HPGR.reset(); LPGL.reset(); LPGR.reset(); EBL.reset(); EBR.reset(); EML.reset(); EMR.reset(); ETL.reset(); ETR.reset();
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool EcualizerStereoAudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    ignoreUnused (layouts);
    return true;
  #else
    if (layouts.getMainOutputChannelSet() != AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != AudioChannelSet::stereo())
        return false;
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() != layouts.getMainInputChannelSet())
        return false;
   #endif
    return true;
  #endif
}
#endif

void EcualizerStereoAudioProcessor::updateParameters()
```



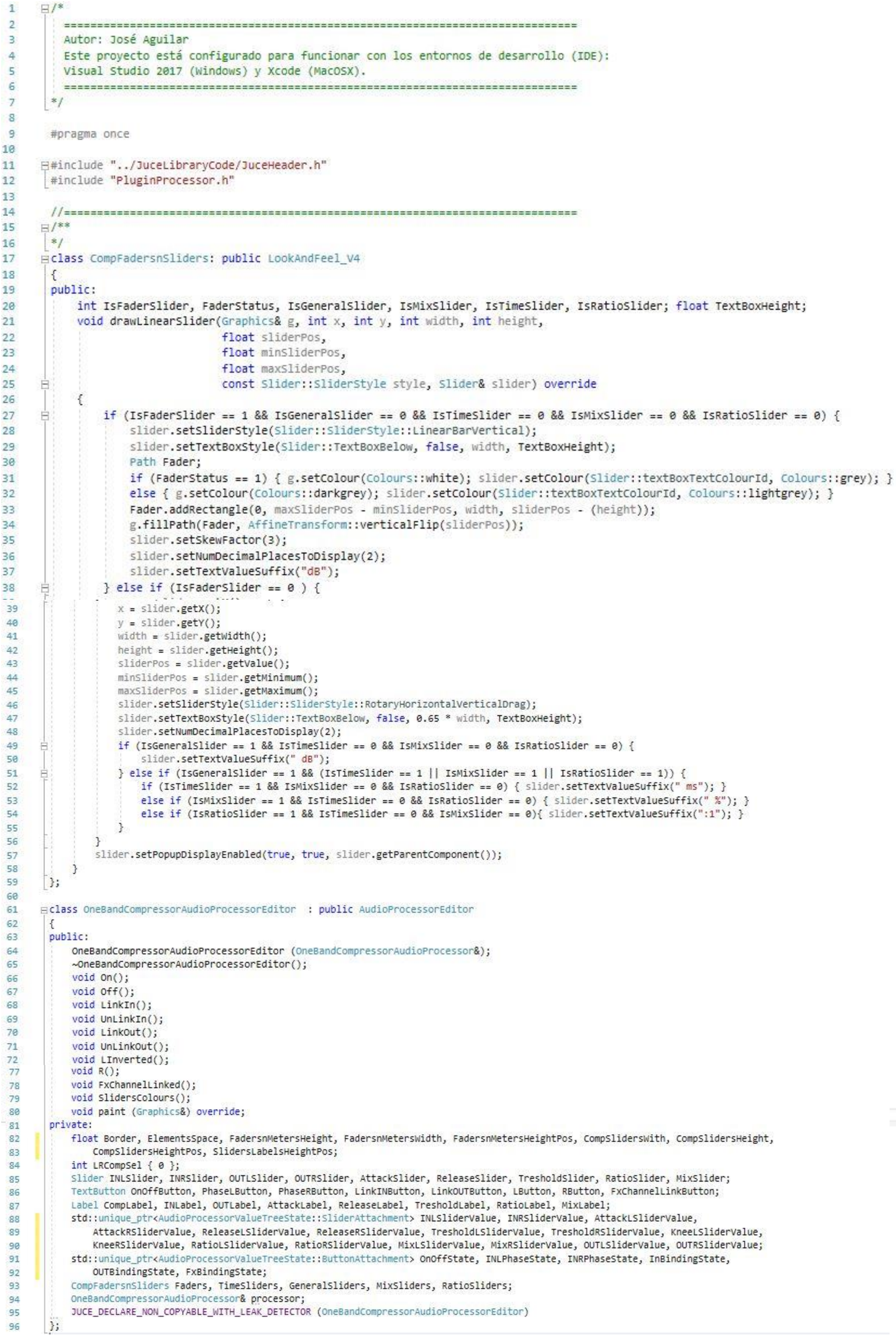
```cpp
#ifndef JucePlugin_PreferredChannelConfigurations
bool CompatibilityAudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    ignoreUnused (layouts);
    return true;
  #else
    if (layouts.getMainOutputChannelSet() != AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != AudioChannelSet::stereo())
        return false;
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() != layouts.getMainInputChannelSet())
        return false;
   #endif
    return true;
  #endif
}
#endif

void CompatibilityAudioProcessor::updateParameters()
{
    if (MixSmoothL != (*DelayParameters.getRawParameterValue("delaymixl") / 100.f)) {
        MixSmoothL = MixSmoothL - (((MixSmoothL + 0.0001 - (*DelayParameters.getRawParameterValue("delaymixl") / 100.f)) * (MixSmoothL -
            (*DelayParameters.getRawParameterValue("delaymixl") / 100.f)) / (MixSmoothL - (*DelayParameters.getRawParameterValue("delaymixl") / 100.f))));
    }
    if (MixSmoothR != (*DelayParameters.getRawParameterValue("delaymixr") / 100.f)) {
        MixSmoothR = MixSmoothR - (((MixSmoothR + 0.0001 - (*DelayParameters.getRawParameterValue("delaymixr") / 100.f)) * (MixSmoothR -
            (*DelayParameters.getRawParameterValue("delaymixr") / 100.f))) / (MixSmoothR + 0.0001 - (*DelayParameters.getRawParameterValue("delaymixr") / 100.f)));
    }
    if (FeedbackSmoothL != (*DelayParameters.getRawParameterValue("delayfeedbackl") / 100.f)) {
        FeedbackSmoothL = FeedbackSmoothL - (((FeedbackSmoothL + 0.0001 - (*DelayParameters.getRawParameterValue("delayfeedbackl") / 100.f)) *
            (FeedbackSmoothL - (*DelayParameters.getRawParameterValue("delayfeedbackl") / 100.f))) / (FeedbackSmoothL + 0.0001 -
            (*DelayParameters.getRawParameterValue("delayfeedbackl") / 100.f)));
    }
    if (FeedbackSmoothR != (*DelayParameters.getRawParameterValue("delayfeedbackr") / 100.f)) {
        FeedbackSmoothR = FeedbackSmoothR - (((FeedbackSmoothR + 0.0001 - (*DelayParameters.getRawParameterValue("delayfeedbackr") / 100.f)) *
```

```cpp
    if (BassRSmooth != Decibels::decibelsToGain(*EQParameters.getRawParameterValue("basseqrgain"))) {
        BassRSmooth = BassRSmooth - (((BRDifR + 0.001) * BRDifR) / BRDifR);
        EBR.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 125, 3, BassRSmooth));
    } else {
        EBR.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 125, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("basseqrgain"))));
    }
    MLDifL = MiddLSmooth - Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqlgain"));
    if (MiddLSmooth != Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqlgain"))) {
        MiddLSmooth = MiddLSmooth - (((MLDifL + 0.001) * MLDifL) / MLDifL);
        EML.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 700, 3, MiddLSmooth));
    } else {
        EML.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 700, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqlgain"))));
    }
    MRDifR = MiddRSmooth - Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqrgain"));
    if (MiddRSmooth != Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqrgain"))) {
        MiddRSmooth = MiddRSmooth - (((MRDifR + 0.001) * MRDifR) / MRDifR);
        EMR.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 700, 3, MiddRSmooth));
    } else {
        EMR.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 700, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("middeqrgain"))));
    }
    TLDifL = TrebleLSmooth - Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqlgain"));
    if (TrebleLSmooth != Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqlgain"))) {
        TrebleLSmooth = TrebleLSmooth - (((TLDifL + 0.001) * TLDifL) / TLDifL);
        ETL.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 4000, 3, TrebleLSmooth));
    } else {
        ETL.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 4000, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqlgain"))));
    }
    TRDifR = TrebleRSmooth - Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqrgain"));
    if (TrebleRSmooth != Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqrgain"))) {
        TrebleRSmooth = TrebleRSmooth - (((TRDifR + 0.001) * TRDifR) / TRDifR);
        ETR.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 4000, 3, TrebleRSmooth));
    } else {
        ETR.setCoefficients(IIRCoefficients::makePeakFilter(SelectedSampleRate, 4000, 3, Decibels::decibelsToGain(*EQParameters.getRawParameterValue("trebleeqrgain"))));
    }
}

void EcualizerStereoAudioProcessor::processBlock (AudioBuffer<float>& buffer, MidiBuffer& midiMessages)
{
```



```cpp
    if (SelectedSampleRate != getSampleRate()) {
        SelectedSampleRate = getSampleRate();
    }
    ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());
    if (*EQParameters.getRawParameterValue("eqstate") == true) {
        PolarityL = *EQParameters.getRawParameterValue("phaselmode") ? -1.0f : 1.0f;
        PolarityR = *EQParameters.getRawParameterValue("phasermode") ? -1.0f : 1.0f;
        CurINLGain = PolarityL * Decibels::decibelsToGain(*EQParameters.getRawParameterValue("inlgain"));
        CurINRGain = PolarityR * Decibels::decibelsToGain(*EQParameters.getRawParameterValue("inrgain"));
        if (PrevINLGain != CurINLGain) {
            buffer.applyGainRamp(0, 0, buffer.getNumSamples(), PrevINLGain, CurINLGain);
        }else {
            buffer.applyGain(0, 0, buffer.getNumSamples(), CurINLGain);
        }
        if (PrevINRGain != CurINRGain) {
            buffer.applyGainRamp(1, 0, buffer.getNumSamples(), PrevINRGain, CurINRGain);
        }
        else {
            buffer.applyGain(1, 0, buffer.getNumSamples(), CurINRGain);
        }
        updateParameters();
        if (*EQParameters.getRawParameterValue("hipasseqfilterl") > 20.f) {
            HPGL.processSamples(buffer.getWritePointer(0), buffer.getNumSamples());
        }
        if (*EQParameters.getRawParameterValue("hipasseqfilterr") > 20.f) {
            HPGR.processSamples(buffer.getWritePointer(1), buffer.getNumSamples());
        }
        if (*EQParameters.getRawParameterValue("lowpasseqfilterl") < 20000.f) {
            LPGL.processSamples(buffer.getWritePointer(0), buffer.getNumSamples());
        }
        if (*EQParameters.getRawParameterValue("lowpasseqfilterr") < 20000.f) {
            LPGR.processSamples(buffer.getWritePointer(1), buffer.getNumSamples());
        }
        if (*EQParameters.getRawParameterValue("basseqlgain") != 0.f) { EBL.processSamples(buffer.getWritePointer(0), buffer.getNumSamples()); }
        if (*EQParameters.getRawParameterValue("basseqrgain") != 0.f) { EBR.processSamples(buffer.getWritePointer(1), buffer.getNumSamples()); }
        if (*EQParameters.getRawParameterValue("middeqlgain") != 0.f) { EML.processSamples(buffer.getWritePointer(0), buffer.getNumSamples()); }
        if (*EQParameters.getRawParameterValue("middeqrgain") != 0.f) { EMR.processSamples(buffer.getWritePointer(1), buffer.getNumSamples()); }
        if (*EQParameters.getRawParameterValue("trebleeqlgain") != 0.f) { ETL.processSamples(buffer.getWritePointer(0), buffer.getNumSamples()); }
        if (*EQParameters.getRawParameterValue("trebleeqrgain") != 0.f) { ETR.processSamples(buffer.getWritePointer(1), buffer.getNumSamples()); }
        CurOutLGain = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("outlgain"));
        if (PrevOUTLGain != CurOutLGain) {
            buffer.applyGainRamp(0, 0, buffer.getNumSamples(), PrevOUTLGain, CurOutLGain);
        } else {
            buffer.applyGain(0, 0, buffer.getNumSamples(), CurOutLGain);
        }
        CurOutRGain = Decibels::decibelsToGain(*EQParameters.getRawParameterValue("outrgain"));
        if (PrevOUTRGain != CurOutRGain) {
            buffer.applyGainRamp(1, 0, buffer.getNumSamples(), PrevOUTRGain, CurOutRGain);
        } else {
            buffer.applyGain(1, 0, buffer.getNumSamples(), CurOutRGain);
        }
        PrevINLGain = CurINLGain; PrevINRGain = CurINRGain;
        PrevOUTLGain = CurOutLGain; PrevOUTRGain = CurOutRGain;
    } else {
        processBlockBypassed(buffer, midiMessages);
    }
}

//==============================================================================
bool EcualizerStereoAudioProcessor::hasEditor() const
{
    return true;
}

AudioProcessorEditor* EcualizerStereoAudioProcessor::createEditor()
{
    return new EcualizerStereoAudioProcessorEditor (*this);
}

//==============================================================================
void EcualizerStereoAudioProcessor::getStateInformation (MemoryBlock& destData)
{
    auto PluginState = EQParameters.copyState();
    std::unique_ptr<XmlElement> xml(PluginState.createXml());
    copyXmlToBinary(*xml, destData);
}

void EcualizerStereoAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    std::unique_ptr<XmlElement> xmlstate(getXmlFromBinary(data, sizeInBytes));
    if (xmlstate.get() != nullptr && xmlstate->hasTagName(EQParameters.state.getType()))
    {
        EQParameters.replaceState(ValueTree::fromXml(*xmlstate));
    }
}

AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new EcualizerStereoAudioProcessor();
}
```

Anexo 2. Algoritmo del módulo de compresión



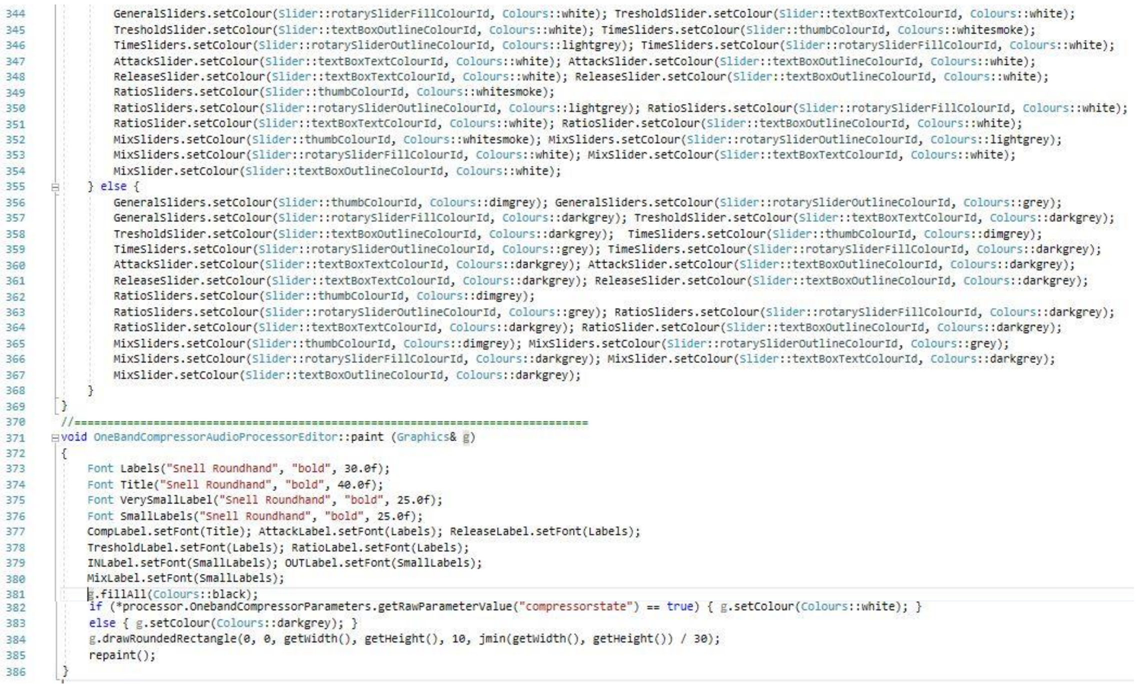
```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo (IDE):
  Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"

//==============================================================================
/**
*/
class CompFadersnSliders: public LookAndFeel_V4
{
public:
    int IsFaderSlider, FaderStatus, IsGeneralSlider, IsMixSlider, IsTimeSlider, IsRatioSlider; float TextBoxHeight;
    void drawLinearSlider(Graphics& g, int x, int y, int width, int height,
                          float sliderPos,
                          float minSliderPos,
                          float maxSliderPos,
                          const Slider::SliderStyle style, Slider& slider) override
    {
        if (IsFaderSlider == 1 && IsGeneralSlider == 0 && IsTimeSlider == 0 && IsMixSlider == 0 && IsRatioSlider == 0) {
            slider.setSliderStyle(Slider::SliderStyle::LinearBarVertical);
            slider.setTextBoxStyle(Slider::TextBoxBelow, false, width, TextBoxHeight);
            Path Fader;
            if (FaderStatus == 1) { g.setColour(Colours::white); slider.setColour(Slider::textBoxTextColourId, Colours::grey); }
            else { g.setColour(Colours::darkgrey); slider.setColour(Slider::textBoxTextColourId, Colours::lightgrey); }
            Fader.addRectangle(0, maxSliderPos - minSliderPos, width, sliderPos - (height));
            g.fillPath(Fader, AffineTransform::verticalFlip(sliderPos));
            slider.setSkewFactor(3);
            slider.setNumDecimalPlacesToDisplay(2);
            slider.setTextValueSuffix("dB");
        } else if (IsFaderSlider == 0 ) {
            x = slider.getX();
            y = slider.getY();
            width = slider.getWidth();
            height = slider.getHeight();
            sliderPos = slider.getValue();
            minSliderPos = slider.getMinimum();
            maxSliderPos = slider.getMaximum();
            slider.setSliderStyle(Slider::SliderStyle::RotaryHorizontalVerticalDrag);
            slider.setTextBoxStyle(Slider::TextBoxBelow, false, 0.65 * width, TextBoxHeight);
            slider.setNumDecimalPlacesToDisplay(2);
            if (IsGeneralSlider == 1 && IsTimeSlider == 0 && IsMixSlider == 0 && IsRatioSlider == 0) {
                slider.setTextValueSuffix(" dB");
            } else if (IsGeneralSlider == 1 && (IsTimeSlider == 1 || IsMixSlider == 1 || IsRatioSlider == 1)) {
                if (IsTimeSlider == 1 && IsMixSlider == 0 && IsRatioSlider == 0) { slider.setTextValueSuffix(" ms"); }
                else if (IsMixSlider == 1 && IsTimeSlider == 0 && IsRatioSlider == 0) { slider.setTextValueSuffix(" %"); }
                else if (IsRatioSlider == 1 && IsTimeSlider == 0 && IsMixSlider == 0){ slider.setTextValueSuffix(":1"); }
            }
        }
        slider.setPopupDisplayEnabled(true, true, slider.getParentComponent());
    }
};

class OneBandCompressorAudioProcessorEditor  : public AudioProcessorEditor
{
public:
    OneBandCompressorAudioProcessorEditor (OneBandCompressorAudioProcessor&);
    ~OneBandCompressorAudioProcessorEditor();
    void On();
    void Off();
    void LinkIn();
    void UnLinkIn();
    void LinkOut();
    void UnLinkOut();
    void LInverted();
    void R();
    void FxChannelLinked();
    void SlidersColours();
    void paint (Graphics&) override;
private:
    float Border, ElementsSpace, FadersnMetersHeight, FadersnMetersWidth, FadersnMetersHeightPos, CompSlidersWith, CompSlidersHeight,
        CompSlidersHeightPos, SlidersLabelsHeightPos;
    int LRCompSel { 0 };
    Slider INLSlider, INRSlider, OUTLSlider, OUTRSlider, AttackSlider, ReleaseSlider, TresholdSlider, RatioSlider, MixSlider;
    TextButton OnOffButton, PhaseLButton, PhaseRButton, LinkINButton, LinkOUTButton, LButton, RButton, FxChannelLinkButton;
    Label CompLabel, INLabel, OUTLabel, AttackLabel, ReleaseLabel, TresholdLabel, RatioLabel, MixLabel;
    std::unique_ptr<AudioProcessorValueTreeState::SliderAttachment> INLSliderValue, INRSliderValue, AttackLSliderValue,
        AttackRSliderValue, ReleaseLSliderValue, ReleaseRSliderValue, TresholdLSliderValue, TresholdRSliderValue, KneeLSliderValue,
        KneeRSliderValue, RatioLSliderValue, RatioRSliderValue, MixLSliderValue, MixRSliderValue, OUTLSliderValue, OUTRSliderValue;
    std::unique_ptr<AudioProcessorValueTreeState::ButtonAttachment> OnOffState, INLPhaseState, INRPhaseState, InBindingState,
        OUTBindingState, FxBindingState;
    CompFadersnSliders Faders, TimeSliders, GeneralSliders, MixSliders, RatioSliders;
    OneBandCompressorAudioProcessor& processor;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (OneBandCompressorAudioProcessorEditor)
};
```

```
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo (IDE):
  Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
OneBandCompressorAudioProcessorEditor::OneBandCompressorAudioProcessorEditor(OneBandCompressorAudioProcessor& p)
    : AudioProcessorEditor(&p), processor(p)
{
    if (getParentWidth() <= 1400 && getParentHeight() <= 800) { setSize(getParentWidth() / 1.25, 0.45 * getParentHeight()); }
    else { setSize(getParentWidth() / 1.7, 0.35 * getParentHeight()); }
    OnOffState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.OnebandCompressorParameters, "compressorstate", OnOffButton);
    INLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "inlgain", INLSlider);
    INRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "inrgain", INRSlider);
    OUTLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "outlgain", OUTLSlider);
    OUTRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "outrgain", OUTRSlider);
    OnOffState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.OnebandCompressorParameters, "eqstate", OnOffButton);
    INLPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.OnebandCompressorParameters, "phaselmode", PhaseLButton);
    INRPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.OnebandCompressorParameters, "phasermode", PhaseRButton);
    InBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.OnebandCompressorParameters, "inbinding", LinkINButton);
    OUTBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.OnebandCompressorParameters, "outbinding", LinkOUTButton);
    FxBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.OnebandCompressorParameters, "fxcompbinding", FxChannelLinkButton);
    INLSlider.setRange(-100.f, 12.f, 0.01); INRSlider.setRange(-100.f, 12.f, 0.01); OUTLSlider.setRange(-100.f, 12.f, 0.01); OUTRSlider.setRange(-100.f, 12.f, 0.01);
    AttackSlider.setRange(0.1f, 1000.f, 0.01); ReleaseSlider.setRange(1.f, 1500.f, 0.01);
    TresholdSlider.setRange(-60.f, 0.f, 0.01);
    RatioSlider.setRange(1.f, 30.f, 0.01); MixSlider.setRange(0.f, 100.f, 0.01);
    DBG("Compressor Stereo");
    Border = jmin(getWidth(), getHeight()) / 30;
    ElementsSpace = jmin(getWidth(), getHeight()) / 20;
    FadersnMetersWidth = getWidth() / 25;
    FadersnMetersHeight = 0.65 * getHeight();
    FadersnMetersHeightPos = 0.175 * getHeight();
    SlidersLabelsHeightPos = getHeight() / 2 - ElementsSpace - 0.25 * Border;
    CompSlidersWith = (getWidth() - 2 * (5.5 * ElementsSpace + 3 * FadersnMetersWidth + Border)) / 3;
    CompSlidersHeightPos = getHeight() / 2;
    CompSlidersHeight = FadersnMetersHeight / 2;
    Faders.IsFaderSlider = 1; Faders.IsGeneralSlider = 0; Faders.TextBoxHeight = ElementsSpace; Faders.IsRatioSlider = 0; Faders.IsTimeSlider = 0; Faders.IsMixSlider = 0;
    GeneralSliders.IsFaderSlider = 0; GeneralSliders.IsGeneralSlider = 1; GeneralSliders.TextBoxHeight = ElementsSpace; GeneralSliders.IsRatioSlider = 0;
    GeneralSliders.IsTimeSlider = 0; GeneralSliders.IsMixSlider = 0;
    TimeSliders.IsFaderSlider = 0; TimeSliders.IsGeneralSlider = 1; TimeSliders.TextBoxHeight = ElementsSpace; TimeSliders.IsTimeSlider = 1; TimeSliders.IsRatioSlider = 0;
    TimeSliders.IsMixSlider = 0;
    RatioSliders.IsFaderSlider = 0; RatioSliders.IsGeneralSlider = 1; RatioSliders.TextBoxHeight = ElementsSpace; RatioSliders.IsTimeSlider = 0; RatioSliders.IsRatioSlider = 1;
    RatioSliders.IsMixSlider = 0;
    MixSliders.IsFaderSlider = 0; MixSliders.IsGeneralSlider = 1; MixSliders.TextBoxHeight = ElementsSpace; MixSliders.IsTimeSlider = 0; MixSliders.IsRatioSlider = 0;
    MixSliders.IsMixSlider = 1;
    CompLabel.setText("Compressor", sendNotificationSync); INLabel.setText("In", sendNotificationSync); OUTLabel.setText("Out", sendNotificationSync);
    AttackLabel.setText("Attack", sendNotificationSync); ReleaseLabel.setText("Release", sendNotificationSync); TresholdLabel.setText("Treshold", sendNotificationSync);
    RatioLabel.setText("Ratio", sendNotificationSync); MixLabel.setText("Mix", sendNotificationSync);
    LinkINButton.setButtonText("LINK"); FxChannelLinkButton.setButtonText(LinkINButton.getButtonText()); LinkOUTButton.setButtonText(LinkINButton.getButtonText());
    LButton.setButtonText("L"); RButton.setButtonText("R"); PhaseLButton.setButtonText(CharPointer_UTF8("Ø")); OnOffButton.setButtonText(CharPointer_UTF8("•"));
    PhaseRButton.setButtonText(PhaseLButton.getButtonText());
    INLSlider.setLookAndFeel(&Faders); INRSlider.setLookAndFeel(&Faders); OUTLSlider.setLookAndFeel(&Faders); OUTRSlider.setLookAndFeel(&Faders);
    AttackSlider.setLookAndFeel(&TimeSliders);
    ReleaseSlider.setLookAndFeel(&TimeSliders);
    TresholdSlider.setLookAndFeel(&GeneralSliders);
    RatioSlider.setLookAndFeel(&RatioSliders);
    MixSlider.setLookAndFeel(&MixSliders);
    CompLabel.setBounds(getWidth() / 2 - 2 * FadersnMetersWidth, ((FadersnMetersHeightPos + Border) / 2) - 0.75 * ElementsSpace, 4 * FadersnMetersWidth, 4 * ElementsSpace);
    INLSlider.setBounds(Border + 1.25 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    INRSlider.setBounds(INLSlider.getX() + INLSlider.getWidth() + 2.f * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    INLabel.setBounds(INRSlider.getX() - 2 * ElementsSpace, (getHeight() * 0.5) - ElementsSpace, 2 * ElementsSpace, 2 * ElementsSpace);
    LinkINButton.setBounds(INRSlider.getX() - 2 * ElementsSpace - 0.75 * FadersnMetersWidth, getHeight() - FadersnMetersHeightPos + 0.875 * ElementsSpace,
        1.5 * FadersnMetersWidth + 2 * ElementsSpace, 1.75 * ElementsSpace);
    PhaseLButton.setBounds(INLSlider.getX(), CompLabel.getY(), INLSlider.getWidth(), 1.5 * ElementsSpace);
    PhaseRButton.setBounds(INRSlider.getX(), CompLabel.getY(), INRSlider.getWidth(), 1.5 * ElementsSpace);
    AttackLabel.setBounds(Border + 3 * FadersnMetersWidth + 4.5 * ElementsSpace, CompLabel.getY() - 0.5 * Border, CompSlidersWith, 1.75 * ElementsSpace);
    AttackSlider.setBounds(AttackLabel.getX(), CompLabel.getY() + ElementsSpace, AttackLabel.getWidth(), CompSlidersHeight);
    TresholdLabel.setBounds(AttackLabel.getX(), SlidersLabelsHeightPos, CompSlidersWith, AttackLabel.getHeight());
    TresholdSlider.setBounds(TresholdLabel.getX(), CompSlidersHeightPos, CompSlidersWith, CompSlidersHeight);
    RatioLabel.setBounds(TresholdLabel.getX() + TresholdLabel.getWidth() + ElementsSpace, TresholdLabel.getY(), CompSlidersWith, TresholdLabel.getHeight());
    RatioSlider.setBounds(RatioLabel.getX(), CompSlidersHeightPos, CompSlidersWith, CompSlidersHeight);
    MixLabel.setBounds(RatioLabel.getX() + RatioLabel.getWidth() + ElementsSpace, SlidersLabelsHeightPos, CompSlidersWith, RatioLabel.getHeight());
    MixSlider.setBounds(MixLabel.getX(), CompSlidersHeightPos, CompSlidersWith, CompSlidersHeight);
    ReleaseLabel.setBounds(MixLabel.getX(), AttackLabel.getY(), CompSlidersWith, AttackLabel.getHeight());
    ReleaseSlider.setBounds(ReleaseLabel.getX(), AttackSlider.getY(), ReleaseLabel.getWidth(), CompSlidersHeight);
    OUTLSlider.setBounds(getWidth() - Border - 3.25 * ElementsSpace - 3 * FadersnMetersWidth, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    OUTRSlider.setBounds(OUTLSlider.getX() + OUTLSlider.getWidth() + 2 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    OUTLabel.setBounds(OUTRSlider.getX() - 2 * ElementsSpace, INLabel.getY(), 2 * ElementsSpace, INLabel.getHeight());
    OnOffButton.setBounds(OUTRSlider.getX(), PhaseLButton.getY(), OUTRSlider.getWidth(), PhaseLButton.getHeight());
    LinkOUTButton.setBounds(OUTLabel.getX() - 0.75 * FadersnMetersWidth, LinkINButton.getY(), LinkINButton.getWidth(), LinkINButton.getHeight());
    FxChannelLinkButton.setBounds((0.5 * getWidth()) - (LinkINButton.getWidth() * 0.5 ), LinkINButton.getY(), LinkINButton.getWidth(), LinkINButton.getHeight());
    LButton.setBounds(FxChannelLinkButton.getX() - 0.5 * ElementsSpace - 0.25 * CompSlidersWith, FxChannelLinkButton.getY(), 0.25 * CompSlidersWith, FxChannelLinkButton.getHeight());
    RButton.setBounds(FxChannelLinkButton.getX() + FxChannelLinkButton.getWidth() + 0.5 * ElementsSpace, FxChannelLinkButton.getY(),
        0.25 * CompSlidersWith, FxChannelLinkButton.getHeight());
    CompLabel.setJustificationType(Justification::Flags::centred); INLabel.setJustificationType(Justification::Flags::centred);
    OUTLabel.setJustificationType(Justification::Flags::centred);
    AttackLabel.setJustificationType(Justification::Flags::centred); ReleaseLabel.setJustificationType(Justification::Flags::centred);
    TresholdLabel.setJustificationType(Justification::Flags::centred); RatioLabel.setJustificationType(Justification::Flags::centred);
    MixLabel.setJustificationType(Justification::Flags::centred);
    PhaseLButton.setColour(TextButton::textColourOnId, Colours::black); PhaseRButton.setColour(TextButton::textColourOnId, Colours::black);
    LinkINButton.setColour(TextButton::textColourOnId, Colours::black); LButton.setColour(TextButton::textColourOnId, Colours::black);
    RButton.setColour(TextButton::textColourOnId, Colours::black); FxChannelLinkButton.setColour(TextButton::textColourOnId, Colours::black);
    LinkOUTButton.setColour(TextButton::textColourOnId, Colours::black);
    addAndMakeVisible(&CompLabel); addAndMakeVisible(&INLabel); addAndMakeVisible(&OUTLabel); addAndMakeVisible(&AttackLabel); addAndMakeVisible(&ReleaseLabel);
    addAndMakeVisible(&TresholdLabel); addAndMakeVisible(&RatioLabel); addAndMakeVisible(&MixLabel);
    addAndMakeVisible(&INLSlider); addAndMakeVisible(&INRSlider); addAndMakeVisible(&PhaseLButton); addAndMakeVisible(&PhaseRButton); addAndMakeVisible(&LinkINButton);
    addAndMakeVisible(&AttackSlider); addAndMakeVisible(&ReleaseSlider); addAndMakeVisible(&TresholdSlider); addAndMakeVisible(&RatioSlider);
    addAndMakeVisible(&MixSlider);
    addAndMakeVisible(&OUTLSlider); addAndMakeVisible(&OUTRSlider); addAndMakeVisible(&FxChannelLinkButton); addAndMakeVisible(&LButton);
    addAndMakeVisible(&LinkOUTButton); addAndMakeVisible(&OUTLSlider); addAndMakeVisible(&OUTRSlider); addAndMakeVisible(&LinkOUTButton); addAndMakeVisible(&OnOffButton);
    addAndMakeVisible(&RButton);
    if (*processor.OnebandCompressorParameters.getRawParameterValue("compressorstate") == true) { On(); }
    else { Off(); }
    if (*processor.OnebandCompressorParameters.getRawParameterValue("phaselmode") == false) { LNonInverted(); }
    else { LInverted(); }
    if (*processor.OnebandCompressorParameters.getRawParameterValue("phasermode") == false) { RNonInverted(); }
    else { RInverted(); }
    if (*processor.OnebandCompressorParameters.getRawParameterValue("inbinding") == true) { LinkIn(); }
```

```cpp
    else { UnLinkIn(); }
    if (*processor.OnebandCompressorParameters.getRawParameterValue("fxcompbinding") == true) { FxChannelLinked(); }
    else { if (LRCompSel == 0) { L(); } else if (LRCompSel == 1) { R(); } }
    if (*processor.OnebandCompressorParameters.getRawParameterValue("outbinding") == true) { LinkOut(); }
    else { UnLinkOut(); }
}
OneBandCompressorAudioProcessorEditor::~OneBandCompressorAudioProcessorEditor()
{
    INLSlider.setLookAndFeel(nullptr); INRSlider.setLookAndFeel(nullptr); OUTLSlider.setLookAndFeel(nullptr); OUTRSlider.setLookAndFeel(nullptr);
    AttackSlider.setLookAndFeel(nullptr); ReleaseSlider.setLookAndFeel(nullptr);
    TresholdSlider.setLookAndFeel(nullptr); RatioSlider.setLookAndFeel(nullptr); MixSlider.setLookAndFeel(nullptr);
}

void OneBandCompressorAudioProcessorEditor::On()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("compressorstate") = true;
    OnOffButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("compressorstate"), sendNotificationSync);
    Faders.FaderStatus = 1;
    SlidersColours();
    PhaseLButton.setColour(TextButton::textColourOffId, Colours::white); PhaseLButton.setColour(TextButton::buttonColourId, Colours::black);
    PhaseLButton.setColour(TextButton::buttonOnColourId, Colours::white); PhaseRButton.setColour(TextButton::textColourOffId, Colours::white);
    PhaseRButton.setColour(TextButton::buttonColourId, Colours::black); PhaseRButton.setColour(TextButton::buttonOnColourId, Colours::white);
    LinkINButton.setColour(TextButton::textColourOffId, Colours::white); LinkINButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkINButton.setColour(TextButton::buttonOnColourId, Colours::white); LButton.setColour(TextButton::buttonColourId, Colours::black);
    LButton.setColour(TextButton::textColourOffId, Colours::white); LButton.setColour(TextButton::buttonOnColourId, Colours::white);
    RButton.setColour(TextButton::textColourOffId, Colours::white); RButton.setColour(TextButton::buttonColourId, Colours::black);
    RButton.setColour(TextButton::buttonOnColourId, Colours::white); FxChannelLinkButton.setColour(TextButton::textColourOffId, Colours::white);
    FxChannelLinkButton.setColour(TextButton::buttonColourId, Colours::black); FxChannelLinkButton.setColour(TextButton::buttonOnColourId, Colours::white);
    LinkOUTButton.setColour(TextButton::textColourOffId, Colours::white); LinkOUTButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkOUTButton.setColour(TextButton::buttonOnColourId, Colours::white); OnOffButton.setColour(TextButton::textColourOnId, Colours::black);
    OnOffButton.setColour(TextButton::buttonOnColourId, Colours::white);
    INLabel.setColour(Label::textColourId, Colours::white); OUTLabel.setColour(Label::textColourId, Colours::white); CompLabel.setColour(Label::textColourId, Colours::white);
    AttackLabel.setColour(Label::textColourId, Colours::white); ReleaseLabel.setColour(Label::textColourId, Colours::white);
    TresholdLabel.setColour(Label::textColourId, Colours::white);
    RatioLabel.setColour(Label::textColourId, Colours::white); MixLabel.setColour(Label::textColourId, Colours::white);
    OnOffButton.onClick = [this]() { Off(); };
}
void OneBandCompressorAudioProcessorEditor::Off()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("compressorstate") = false;
    OnOffButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("compressorstate"), sendNotificationSync);
    Faders.FaderStatus = 0;
    SlidersColours();
    PhaseLButton.setColour(TextButton::textColourOffId, Colours::darkgrey); PhaseLButton.setColour(TextButton::buttonColourId, Colours::black);
    PhaseLButton.setColour(TextButton::textColourOnId, Colours::black); PhaseLButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    PhaseRButton.setColour(TextButton::textColourOffId, Colours::darkgrey); PhaseRButton.setColour(TextButton::buttonColourId, Colours::black);
    PhaseRButton.setColour(TextButton::textColourOnId, Colours::black); PhaseRButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    LinkINButton.setColour(TextButton::textColourOffId, Colours::darkgrey); LinkINButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkINButton.setColour(TextButton::textColourOnId, Colours::black); LinkINButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    LButton.setColour(TextButton::textColourOffId, Colours::darkgrey); LButton.setColour(TextButton::buttonColourId, Colours::black);
    LButton.setColour(TextButton::textColourOnId, Colours::black); LButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    RButton.setColour(TextButton::textColourOffId, Colours::darkgrey); RButton.setColour(TextButton::buttonColourId, Colours::black);
    RButton.setColour(TextButton::textColourOnId, Colours::black); RButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    FxChannelLinkButton.setColour(TextButton::textColourOffId, Colours::darkgrey); FxChannelLinkButton.setColour(TextButton::buttonColourId, Colours::black);
    FxChannelLinkButton.setColour(TextButton::textColourOnId, Colours::black); FxChannelLinkButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    LinkOUTButton.setColour(TextButton::textColourOffId, Colours::darkgrey); LinkOUTButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkOUTButton.setColour(TextButton::textColourOnId, Colours::black); LinkOUTButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    OnOffButton.setColour(TextButton::textColourOffId, Colours::darkgrey); OnOffButton.setColour(TextButton::buttonColourId, Colours::black);
    INLabel.setColour(Label::textColourId, Colours::darkgrey); OUTLabel.setColour(Label::textColourId, Colours::darkgrey); CompLabel.setColour(Label::textColourId, Colours::darkgrey);
    AttackLabel.setColour(Label::textColourId, Colours::darkgrey); ReleaseLabel.setColour(Label::textColourId, Colours::darkgrey);
    TresholdLabel.setColour(Label::textColourId, Colours::darkgrey);
    RatioLabel.setColour(Label::textColourId, Colours::darkgrey); MixLabel.setColour(Label::textColourId, Colours::darkgrey);
    OnOffButton.onClick = [this]() { On(); };
}

void OneBandCompressorAudioProcessorEditor::LinkIn()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("inbinding") = true;
    LinkINButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("inbinding"), sendNotificationSync);
    if (*processor.OnebandCompressorParameters.getRawParameterValue("phaselmode") == true && *processor.OnebandCompressorParameters.getRawParameterValue("phasermode") == false) {
        RInverted();
    } else if (*processor.OnebandCompressorParameters.getRawParameterValue("phasermode") == true && *processor.OnebandCompressorParameters.getRawParameterValue("phaselmode") == false)
        LInverted();
```

```cpp
        INLSlider.onValueChange = [this] () { INRSlider.setValue(INLSlider.getValue(), sendNotificationSync); };
        INLSlider.onDragEnd = [this] () { INRSlider.setValue(INRSlider.getValue(), sendNotificationSync); };
        INRSlider.onDragStart = [this] () { INLSlider.setValue(INRSlider.getValue(), sendNotificationSync); };
        INRSlider.onValueChange = [this] () { INLSlider.setValue(INRSlider.getValue(), sendNotificationSync); };
        INRSlider.onDragEnd = [this] () { INLSlider.setValue(INRSlider.getValue(), sendNotificationSync); };
        LinkINButton.onClick = [this]() { UnLinkIn(); };
    }

    void OneBandCompressorAudioProcessorEditor::UnLinkIn()
    {
        *processor.OnebandCompressorParameters.getRawParameterValue("inbinding") = false;
        LinkINButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("inbinding"), sendNotificationSync);
        INLSlider.onDragStart = [this]() {};
        INLSlider.onValueChange = [this]() {};
        INLSlider.onDragEnd = [this]() {};
        INRSlider.onDragStart = [this]() {};
        INRSlider.onValueChange = [this]() {};
        INRSlider.onDragEnd = [this]() {};
        LinkINButton.onClick = [this]() { LinkIn(); };
    }

    void OneBandCompressorAudioProcessorEditor::LInverted()
    {
        *processor.OnebandCompressorParameters.getRawParameterValue("phaselmode") = true;
        PhaseLButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("phaselmode"), sendNotificationSync);
        PhaseLButton.onClick = [this]() {
            LNonInverted();
            if (*processor.OnebandCompressorParameters.getRawParameterValue("inbinding") == true) { RNonInverted(); }
        };
    }

    void OneBandCompressorAudioProcessorEditor::LNonInverted()
    {
        *processor.OnebandCompressorParameters.getRawParameterValue("phaselmode") = false;
        PhaseLButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("phaselmode"), sendNotificationSync);
        PhaseLButton.onClick = [this]() {
            LInverted();
            if (*processor.OnebandCompressorParameters.getRawParameterValue("inbinding") == true) { RInverted(); }
```

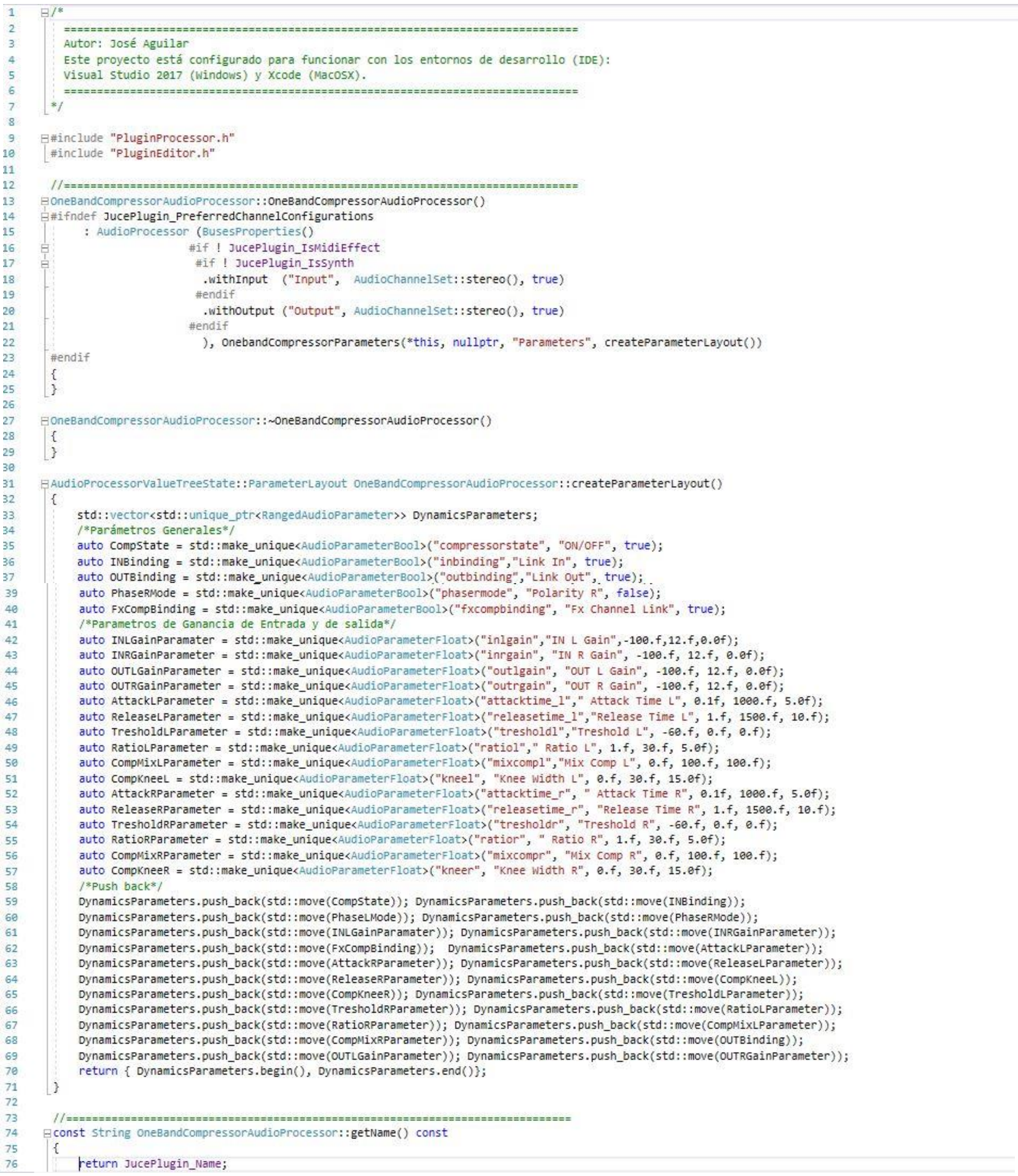
```cpp
}

void OneBandCompressorAudioProcessorEditor::RInverted()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("phasermode") = true;
    PhaseRButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("phasermode"), sendNotificationSync);
    PhaseRButton.onClick = [this]() {
        RNonInverted();
        if (*processor.OnebandCompressorParameters.getRawParameterValue("inbinding") == true) { LNonInverted(); }
    };
}

void OneBandCompressorAudioProcessorEditor::RNonInverted()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("phasermode") = false;
    PhaseRButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("phasermode"), sendNotificationSync);
    PhaseRButton.onClick = [this]() {
        RInverted();
        if (*processor.OnebandCompressorParameters.getRawParameterValue("inbinding") == true) { LInverted(); }
    };
}

void OneBandCompressorAudioProcessorEditor::FxChannelLinked()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("fxcompbinding") = true;
    FxChannelLinkButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("fxcompbinding"), sendNotificationSync);
    LButton.setToggleState(false, dontSendNotification);
    RButton.setToggleState(false, dontSendNotification);
    LButton.setEnabled(false); RButton.setEnabled(false);
    SlidersColours();
    AttackLSliderValue.reset(); ReleaseLSliderValue.reset(); TresholdLSliderValue.reset(); RatioLSliderValue.reset(); MixLSliderValue.reset();
    AttackRSliderValue.reset(); ReleaseRSliderValue.reset(); TresholdRSliderValue.reset(); RatioRSliderValue.reset(); MixRSliderValue.reset();
    AttackLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "attacktime_l", AttackSlider);
    ReleaseLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "releasetime_l", ReleaseSlider);
    TresholdLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "tresholdl", TresholdSlider);
    RatioLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "ratiol", RatioSlider);
    MixLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "mixcompl", MixSlider);
    AttackRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "attacktime_r", AttackSlider);
    ReleaseRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "releasetime_r", ReleaseSlider);
    TresholdRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "tresholdr", TresholdSlider);
    RatioRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "ratior", RatioSlider);
    MixRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "mixcompr", MixSlider);
    FxChannelLinkButton.onClick = [this]() { if (LRCompSel == 0) { L(); } else if (LRCompSel == 1) { R(); } };
}

void OneBandCompressorAudioProcessorEditor::L()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("fxcompbinding") = false;
    FxChannelLinkButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("fxcompbinding"), sendNotificationSync);
    if (LRCompSel != 0) { LRCompSel = 0; }
    SlidersColours();
    LButton.setToggleState(true, sendNotificationSync); RButton.setToggleState(false, dontSendNotification);
    AttackLSliderValue.reset(); ReleaseLSliderValue.reset(); TresholdLSliderValue.reset(); RatioLSliderValue.reset(); MixLSliderValue.reset();
    AttackRSliderValue.reset(); ReleaseRSliderValue.reset(); TresholdRSliderValue.reset(); RatioRSliderValue.reset(); MixRSliderValue.reset();
    LButton.setEnabled(false); RButton.setEnabled(true);
    AttackLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "attacktime_l", AttackSlider);
    ReleaseLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "releasetime_l", ReleaseSlider);
    TresholdLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "tresholdl", TresholdSlider);
    RatioLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "ratiol", RatioSlider);
    MixLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "mixcompl", MixSlider);
    FxChannelLinkButton.onClick = [this]() { FxChannelLinked(); };
    RButton.onClick = [this]() { R(); };
}

void OneBandCompressorAudioProcessorEditor::R()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("fxcompbinding") = false;
    FxChannelLinkButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("fxcompbinding"), sendNotificationSync);
    if (LRCompSel != 1) { LRCompSel = 1; }
    LButton.setToggleState(false, dontSendNotification); RButton.setToggleState(true, sendNotificationSync);
    LButton.setEnabled(true); RButton.setEnabled(false);
    SlidersColours();
    AttackLSliderValue.reset(); ReleaseLSliderValue.reset(); TresholdLSliderValue.reset(); RatioLSliderValue.reset(); MixLSliderValue.reset();
    AttackRSliderValue.reset(); ReleaseRSliderValue.reset(); TresholdRSliderValue.reset(); RatioRSliderValue.reset(); MixRSliderValue.reset();
    AttackRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "attacktime_r", AttackSlider);
    TresholdRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "tresholdr", TresholdSlider);
    RatioRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "ratior", RatioSlider);
    MixRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.OnebandCompressorParameters, "mixcompr", MixSlider);
    FxChannelLinkButton.onClick = [this]() { FxChannelLinked(); };
    LButton.onClick = [this]() { L(); };
}

void OneBandCompressorAudioProcessorEditor::LinkOut()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("outbinding") = true;
    LinkOUTButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("outbinding"), sendNotificationSync);
    OUTLSlider.onDragStart = [this]() { OUTRSlider.setValue(OUTLSlider.getValue(), sendNotificationSync); };
    OUTLSlider.onValueChange = [this]() { OUTRSlider.setValue(OUTLSlider.getValue(), sendNotificationSync); };
    OUTLSlider.onDragEnd = [this]() { OUTRSlider.setValue(OUTLSlider.getValue(), sendNotificationSync); };
    OUTRSlider.onDragStart = [this]() { OUTLSlider.setValue(OUTRSlider.getValue(), sendNotificationSync); };
    OUTRSlider.onValueChange = [this]() { OUTLSlider.setValue(OUTRSlider.getValue(), sendNotificationSync); };
    OUTRSlider.onDragEnd = [this]() { OUTLSlider.setValue(OUTRSlider.getValue(), sendNotificationSync); };
    LinkOUTButton.onClick = [this]() { UnLinkOut(); };
}

void OneBandCompressorAudioProcessorEditor::UnLinkOut()
{
    *processor.OnebandCompressorParameters.getRawParameterValue("outbinding") = false;
    LinkOUTButton.setToggleState(*processor.OnebandCompressorParameters.getRawParameterValue("outbinding"), sendNotificationSync);
    OUTLSlider.onDragStart = [this]() {};
    OUTLSlider.onValueChange = [this]() {};
    OUTLSlider.onDragEnd = [this]() {};
    OUTRSlider.onDragStart = [this]() {};
    OUTRSlider.onValueChange = [this]() {};
    OUTRSlider.onDragEnd = [this]() {};
    LinkOUTButton.onClick = [this]() { LinkOut(); };
}


void OneBandCompressorAudioProcessorEditor::SlidersColours()
{
    if (*processor.OnebandCompressorParameters.getRawParameterValue("compressorstate") == true) {
        GeneralSliders.setColour(Slider::thumbColourId, Colours::whitesmoke); GeneralSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgrey);
```



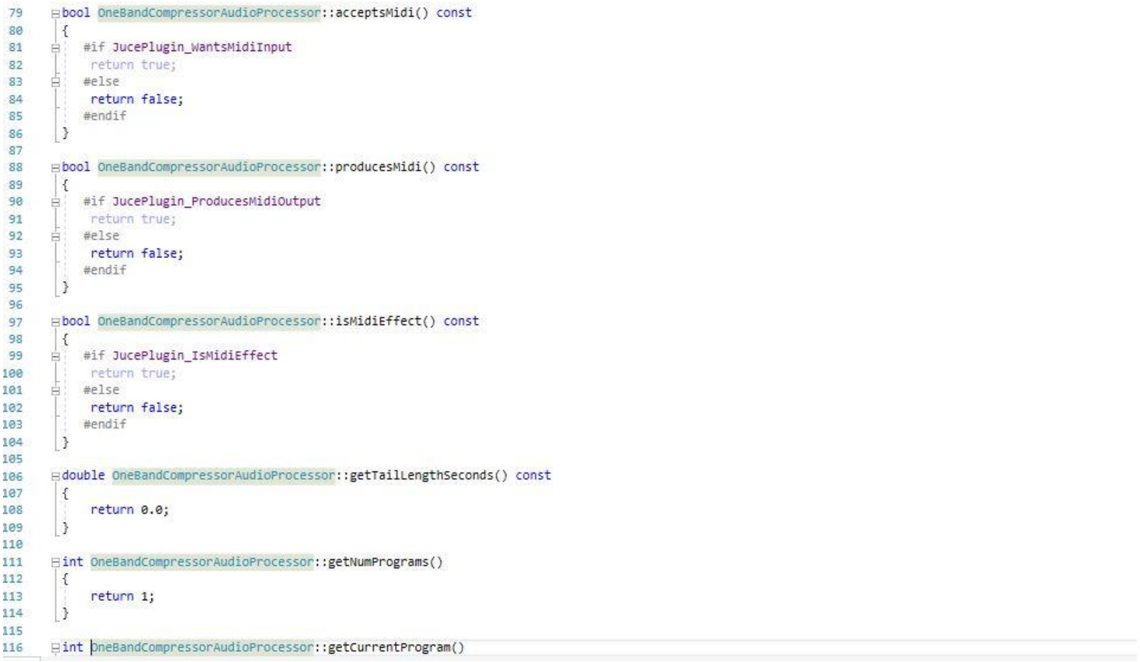
```cpp
        GeneralSliders.setColour(Slider::rotarySliderFillColourId, Colours::white); TresholdSlider.setColour(Slider::textBoxTextColourId, Colours::white);
        TresholdSlider.setColour(Slider::textBoxOutlineColourId, Colours::white); TimeSliders.setColour(Slider::thumbColourId, Colours::whitesmoke);
        TimeSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgrey); TimeSliders.setColour(Slider::rotarySliderFillColourId, Colours::white);
        AttackSlider.setColour(Slider::textBoxTextColourId, Colours::white); AttackSlider.setColour(Slider::textBoxOutlineColourId, Colours::white);
        ReleaseSlider.setColour(Slider::textBoxTextColourId, Colours::white); ReleaseSlider.setColour(Slider::textBoxOutlineColourId, Colours::white);
        RatioSliders.setColour(Slider::thumbColourId, Colours::whitesmoke);
        RatioSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgrey); RatioSliders.setColour(Slider::rotarySliderFillColourId, Colours::white);
        RatioSlider.setColour(Slider::textBoxTextColourId, Colours::white); RatioSlider.setColour(Slider::textBoxOutlineColourId, Colours::white);
        MixSliders.setColour(Slider::thumbColourId, Colours::whitesmoke); MixSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgrey);
        MixSliders.setColour(Slider::rotarySliderFillColourId, Colours::white); MixSlider.setColour(Slider::textBoxTextColourId, Colours::white);
        MixSlider.setColour(Slider::textBoxOutlineColourId, Colours::white);
    } else {
        GeneralSliders.setColour(Slider::thumbColourId, Colours::dimgrey); GeneralSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::grey);
        GeneralSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey); TresholdSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey);
        TresholdSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);  TimeSliders.setColour(Slider::thumbColourId, Colours::dimgrey);
        TimeSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::grey); TimeSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey);
        AttackSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); AttackSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        ReleaseSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); ReleaseSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        RatioSliders.setColour(Slider::thumbColourId, Colours::dimgrey);
        RatioSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::grey); RatioSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey);
        RatioSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); RatioSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        MixSliders.setColour(Slider::thumbColourId, Colours::dimgrey); MixSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::grey);
        MixSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey); MixSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey);
        MixSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
    }
}
//==============================================================================
void OneBandCompressorAudioProcessorEditor::paint (Graphics& g)
{
    Font Labels("Snell Roundhand", "bold", 30.0f);
    Font Title("Snell Roundhand", "bold", 40.0f);
    Font VerySmallLabel("Snell Roundhand", "bold", 25.0f);
    Font SmallLabels("Snell Roundhand", "bold", 25.0f);
    CompLabel.setFont(Title); AttackLabel.setFont(Labels); ReleaseLabel.setFont(Labels);
    TresholdLabel.setFont(Labels); RatioLabel.setFont(Labels);
    INLabel.setFont(SmallLabels); OUTLabel.setFont(SmallLabels);
    MixLabel.setFont(SmallLabels);
    g.fillAll(Colours::black);
    if (*processor.OnebandCompressorParameters.getRawParameterValue("compressorstate") == true) { g.setColour(Colours::white); }
    else { g.setColour(Colours::darkgrey); }
    g.drawRoundedRectangle(0, 0, getWidth(), getHeight(), 10, jmin(getWidth(), getHeight()) / 30);
    repaint();
}
```

```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#pragma once

#include "../JuceLibraryCode/JuceHeader.h"

//==============================================================================

class SmoothShaper
{
public:
    void PrepareForPlay(float samplerate) {
        m_SampleRate = samplerate;
        Update();
    };
    void processAudioSample(float& sample) {
        if (sample > m_Smooth) {
            m_Smooth += m_Attack * (sample - m_Smooth);
        } else if (sample < m_Smooth) {
            m_Smooth += m_Release * (sample - m_Smooth);
        }
        sample = m_Smooth;
    };
    void setAttack(float Attack) { m_AttackTime = Attack; Update(); };
    void setRelease(float Release) { m_ReleaseTime = Release; Update(); };
private:
    float m_Smooth, m_SampleRate, m_AttackTime, m_ReleaseTime, m_Attack, m_Release;
    void Update() {
        m_Attack = Calculate(m_AttackTime);
        m_Release = Calculate(m_ReleaseTime);
    };
    float Calculate(float Time){
        if (Time < 0.f || m_SampleRate <= 0.f) {
            return 1.f;
        }
        return 1.f - exp(-1.f / (Time * 0.001 * m_SampleRate));
    };
};

class Compressor
{
public:
    void PrepareForThePlay(float samplerate) {
        m_SmoothShaper.PrepareForPlay(samplerate);
    };
    void processIndividualSample(float& sample) {
        float DetectionSignal = sample;
        DetectionSignal = fabs(DetectionSignal);
        m_SmoothShaper.processAudioSample(DetectionSignal);
        DetectionSignal = AmptodB(DetectionSignal);
        float Gain;
        if (DetectionSignal > m_Treshold)  {
            float Scale = 1.f - (1.f / m_Ratio);
            Gain = Scale * (m_Treshold - DetectionSignal);
    };
    float dBtoAmp (float dB){
        return pow(10.f, dB / 20.f);
    };
};

class OneBandCompressorAudioProcessor  : public AudioProcessor
{
public:
    //==============================================================================
    OneBandCompressorAudioProcessor();
    ~OneBandCompressorAudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif

    void processBlock (AudioBuffer<float>&, MidiBuffer&) override;

    //==============================================================================
    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==============================================================================
    const String getName() const override;

    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;

    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
```

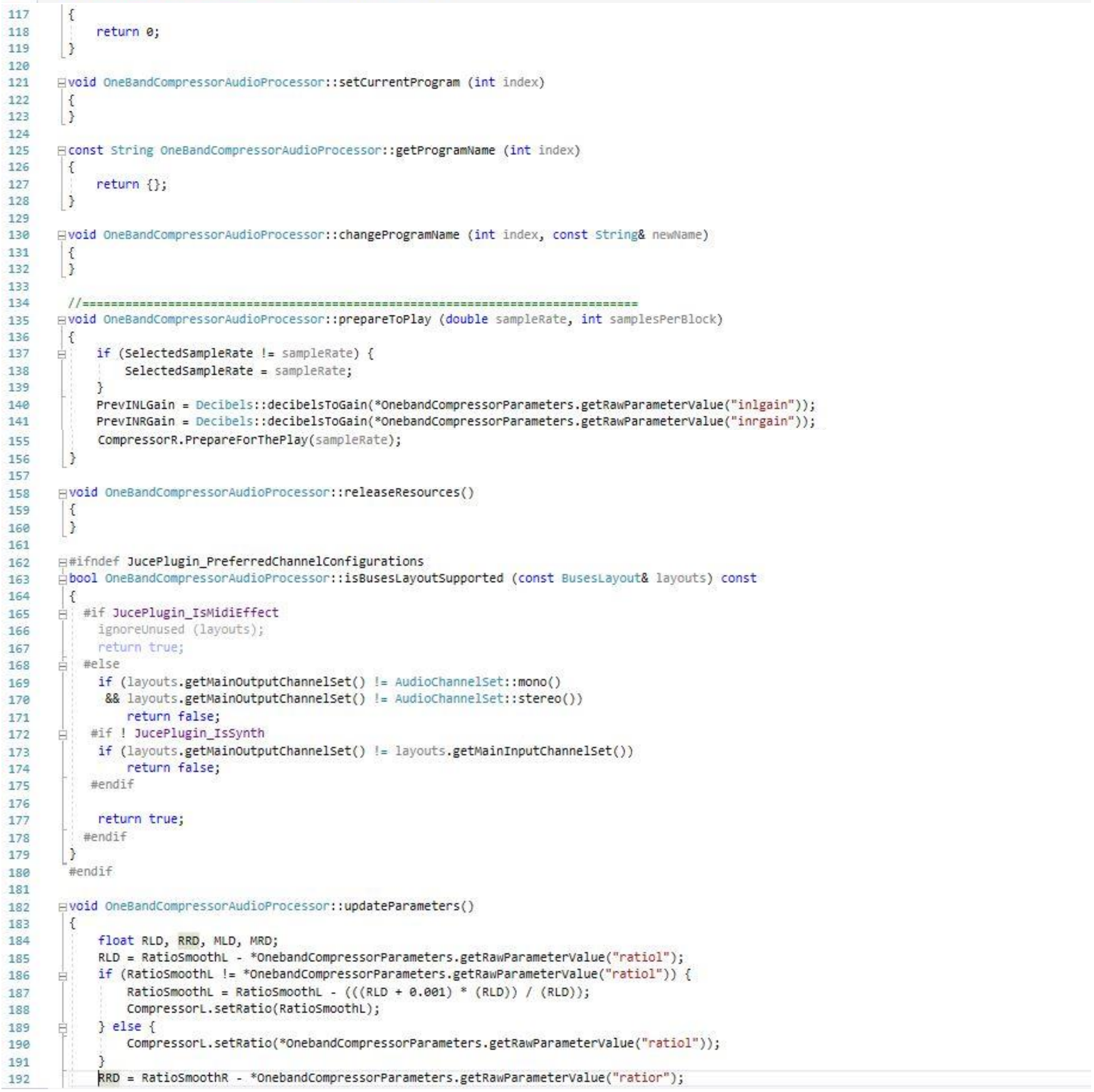
```
    void setCurrentProgram (int index) override;
    const String getProgramName (int index) override;
    void changeProgramName (int index, const String& newName) override;

    //==============================================================================
    void getStateInformation (MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;
    void updateParameters();
    void updateTimeParameters();
    AudioProcessorValueTreeState OnebandCompressorParameters;
    AudioProcessorValueTreeState::ParameterLayout createParameterLayout();

private:
    Compressor CompressorL, CompressorR;
    float AttackTimeSmoothL, AttackTimeSmoothR, ReleaseTimeSmoothL, ReleaseTimeSmoothR, TresholdSmoothL, TresholdSmoothR, RatioSmoothL, RatioSmoothR,
        PrevINLGain, PrevINRGain, PrevOUTLGain, PrevOUTRGain, PolarityL, PolarityR, CurINLGain, CurINRGain, CurOutLGain, CurOutRGain, MixSmoothL,
        MixSmoothR, SelectedSampleRate;

    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (OneBandCompressorAudioProcessor)
};
```



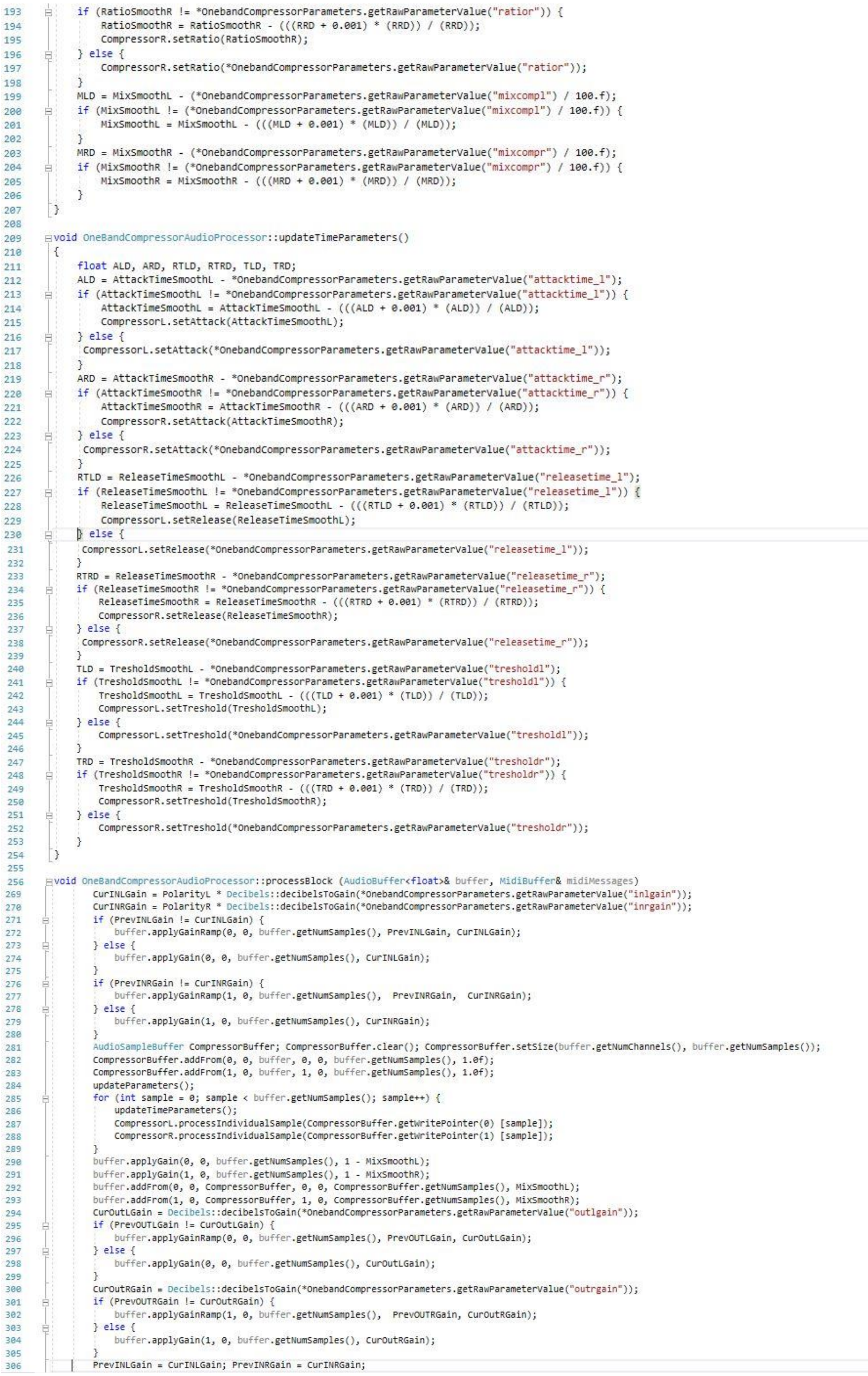
```
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo (IDE):
  Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
OneBandCompressorAudioProcessor::OneBandCompressorAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", AudioChannelSet::stereo(), true)
                     #endif
                       ), OnebandCompressorParameters(*this, nullptr, "Parameters", createParameterLayout())
#endif
{
}

OneBandCompressorAudioProcessor::~OneBandCompressorAudioProcessor()
{
}

AudioProcessorValueTreeState::ParameterLayout OneBandCompressorAudioProcessor::createParameterLayout()
{
    std::vector<std::unique_ptr<RangedAudioParameter>> DynamicsParameters;
    /*Parámetros Generales*/
    auto CompState = std::make_unique<AudioParameterBool>("compressorstate", "ON/OFF", true);
    auto INBinding = std::make_unique<AudioParameterBool>("inbinding","Link In", true);
    auto OUTBinding = std::make_unique<AudioParameterBool>("outbinding","Link Out", true);
     auto PhaseRMode = std::make_unique<AudioParameterBool>("phasermode", "Polarity R", false);
     auto FxCompBinding = std::make_unique<AudioParameterBool>("fxcompbinding", "Fx Channel Link", true);
     /*Parametros de Ganancia de Entrada y de salida*/
     auto INLGainParamater = std::make_unique<AudioParameterFloat>("inlgain","IN L Gain",-100.f,12.f,0.0f);
     auto INRGainParameter = std::make_unique<AudioParameterFloat>("inrgain", "IN R Gain", -100.f, 12.f, 0.0f);
     auto OUTLGainParameter = std::make_unique<AudioParameterFloat>("outlgain", "OUT L Gain", -100.f, 12.f, 0.0f);
     auto OUTRGainParameter = std::make_unique<AudioParameterFloat>("outrgain", "OUT R Gain", -100.f, 12.f, 0.0f);
     auto AttackLParameter = std::make_unique<AudioParameterFloat>("attacktime_l"," Attack Time L", 0.1f, 1000.f, 5.0f);
     auto ReleaseLParameter = std::make_unique<AudioParameterFloat>("releasetime_l","Release Time L", 1.f, 1500.f, 10.f);
     auto TresholdLParameter = std::make_unique<AudioParameterFloat>("tresholdl","Treshold L", -60.f, 0.f, 0.f);
     auto RatioLParameter = std::make_unique<AudioParameterFloat>("ratiol"," Ratio L", 1.f, 30.f, 5.0f);
     auto CompMixLParameter = std::make_unique<AudioParameterFloat>("mixcompl","Mix Comp L", 0.f, 100.f, 100.f);
     auto CompKneeL = std::make_unique<AudioParameterFloat>("kneel", "Knee Width L", 0.f, 30.f, 15.0f);
     auto AttackRParameter = std::make_unique<AudioParameterFloat>("attacktime_r", " Attack Time R", 0.1f, 1000.f, 5.0f);
     auto ReleaseRParameter = std::make_unique<AudioParameterFloat>("releasetime_r", "Release Time R", 1.f, 1500.f, 10.f);
     auto TresholdRParameter = std::make_unique<AudioParameterFloat>("tresholdr", "Treshold R", -60.f, 0.f, 0.f);
     auto RatioRParameter = std::make_unique<AudioParameterFloat>("ratior", " Ratio R", 1.f, 30.f, 5.0f);
     auto CompMixRParameter = std::make_unique<AudioParameterFloat>("mixcompr", "Mix Comp R", 0.f, 100.f, 100.f);
     auto CompKneeR = std::make_unique<AudioParameterFloat>("kneer", "Knee Width R", 0.f, 30.f, 15.0f);
     /*Push back*/
     DynamicsParameters.push_back(std::move(CompState)); DynamicsParameters.push_back(std::move(INBinding));
     DynamicsParameters.push_back(std::move(PhaseLMode)); DynamicsParameters.push_back(std::move(PhaseRMode));
     DynamicsParameters.push_back(std::move(INLGainParamater)); DynamicsParameters.push_back(std::move(INRGainParameter));
     DynamicsParameters.push_back(std::move(FxCompBinding));  DynamicsParameters.push_back(std::move(AttackLParameter));
     DynamicsParameters.push_back(std::move(AttackRParameter)); DynamicsParameters.push_back(std::move(ReleaseLParameter));
     DynamicsParameters.push_back(std::move(ReleaseRParameter)); DynamicsParameters.push_back(std::move(CompKneeL));
     DynamicsParameters.push_back(std::move(CompKneeR)); DynamicsParameters.push_back(std::move(TresholdLParameter));
     DynamicsParameters.push_back(std::move(TresholdRParameter)); DynamicsParameters.push_back(std::move(RatioLParameter));
     DynamicsParameters.push_back(std::move(RatioRParameter)); DynamicsParameters.push_back(std::move(CompMixLParameter));
     DynamicsParameters.push_back(std::move(CompMixRParameter)); DynamicsParameters.push_back(std::move(OUTBinding));
     DynamicsParameters.push_back(std::move(OUTLGainParameter)); DynamicsParameters.push_back(std::move(OUTRGainParameter));
     return { DynamicsParameters.begin(), DynamicsParameters.end()};
}

 //==============================================================================
const String OneBandCompressorAudioProcessor::getName() const
{
    return JucePlugin_Name;
```



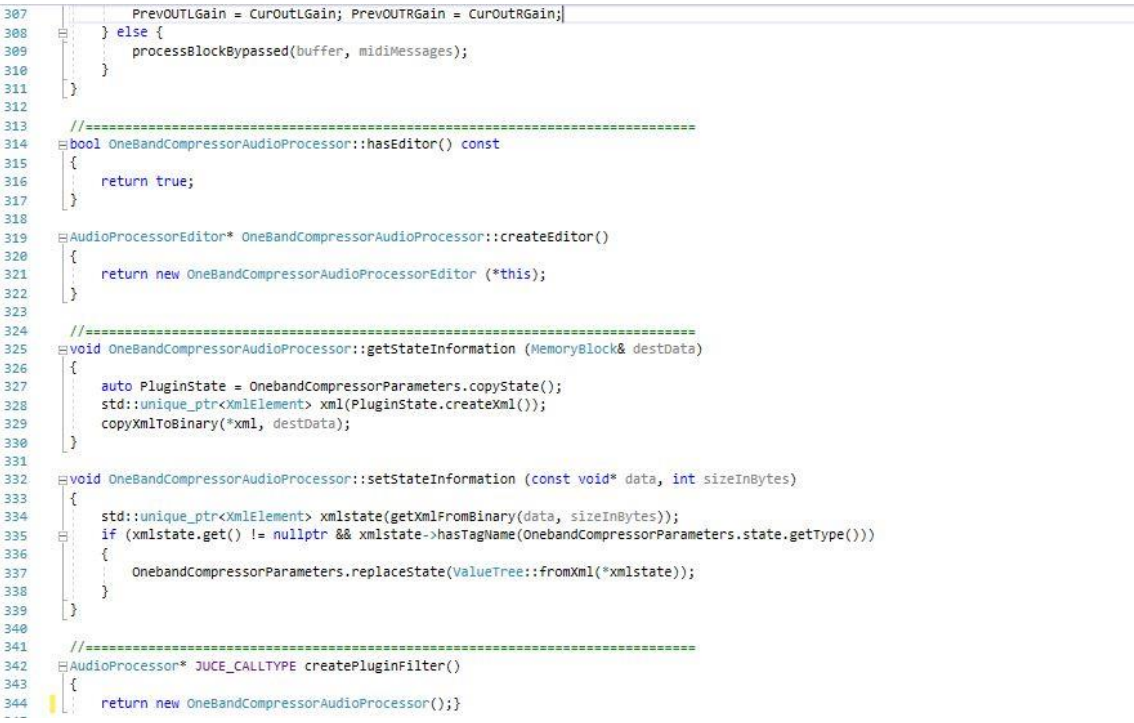
```cpp
bool OneBandCompressorAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}

bool OneBandCompressorAudioProcessor::producesMidi() const
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
    return false;
   #endif
}

bool OneBandCompressorAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
   #else
    return false;
   #endif
}

double OneBandCompressorAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int OneBandCompressorAudioProcessor::getNumPrograms()
{
    return 1;
}

int OneBandCompressorAudioProcessor::getCurrentProgram()
```



```cpp
{
    return 0;
}

void OneBandCompressorAudioProcessor::setCurrentProgram (int index)
{
}

const String OneBandCompressorAudioProcessor::getProgramName (int index)
{
    return {};
}

void OneBandCompressorAudioProcessor::changeProgramName (int index, const String& newName)
{
}

//==============================================================================
void OneBandCompressorAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    if (SelectedSampleRate != sampleRate) {
        SelectedSampleRate = sampleRate;
    }
    PrevINLGain = Decibels::decibelsToGain(*OnebandCompressorParameters.getRawParameterValue("inlgain"));
    PrevINRGain = Decibels::decibelsToGain(*OnebandCompressorParameters.getRawParameterValue("inrgain"));
    CompressorR.PrepareForThePlay(sampleRate);
}

void OneBandCompressorAudioProcessor::releaseResources()
{
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool OneBandCompressorAudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    ignoreUnused (layouts);
    return true;
  #else
    if (layouts.getMainOutputChannelSet() != AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != AudioChannelSet::stereo())
        return false;
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() != layouts.getMainInputChannelSet())
        return false;
   #endif

    return true;
  #endif
}
#endif

void OneBandCompressorAudioProcessor::updateParameters()
{
    float RLD, RRD, MLD, MRD;
    RLD = RatioSmoothL - *OnebandCompressorParameters.getRawParameterValue("ratiol");
    if (RatioSmoothL != *OnebandCompressorParameters.getRawParameterValue("ratiol")) {
        RatioSmoothL = RatioSmoothL - (((RLD + 0.001) * (RLD)) / (RLD));
        CompressorL.setRatio(RatioSmoothL);
    } else {
        CompressorL.setRatio(*OnebandCompressorParameters.getRawParameterValue("ratiol"));
    }
    RRD = RatioSmoothR - *OnebandCompressorParameters.getRawParameterValue("ratior");
```

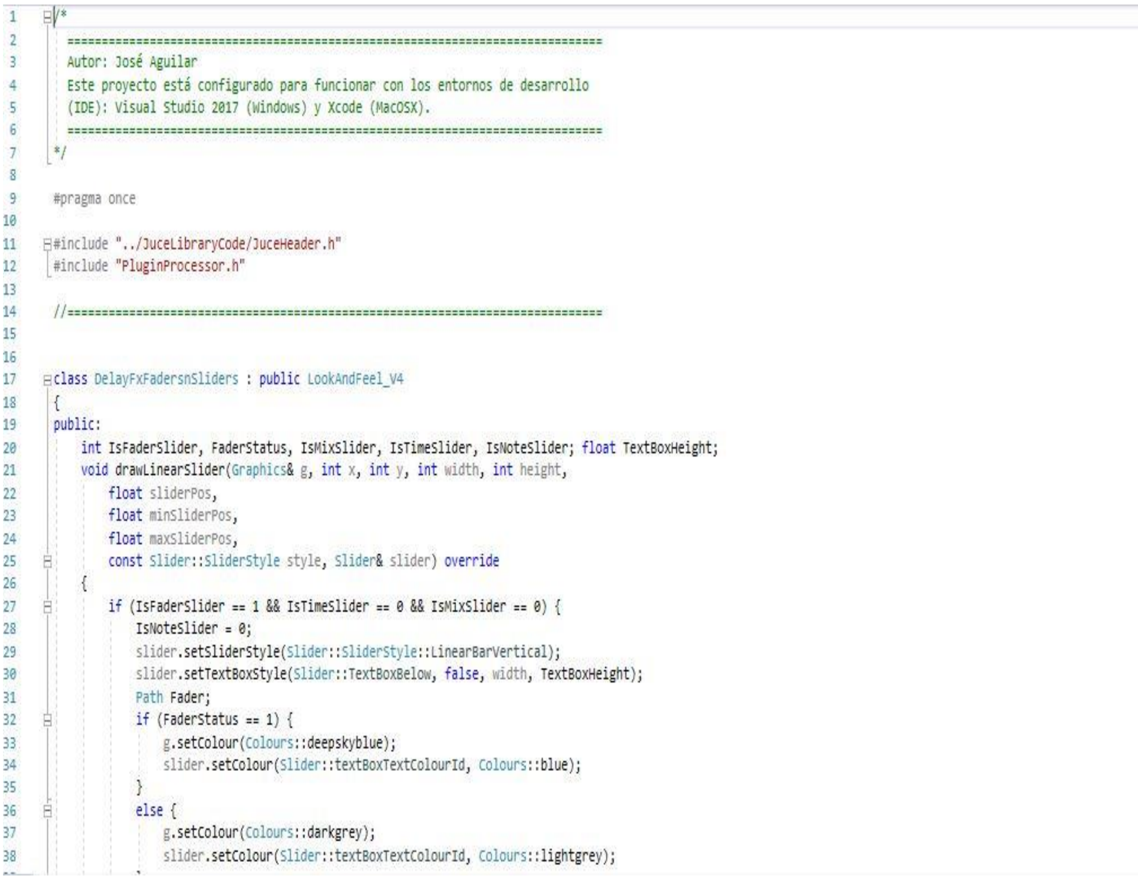


```
    if (RatioSmoothR != *OnebandCompressorParameters.getRawParameterValue("ratior")) {
        RatioSmoothR = RatioSmoothR - (((RRD + 0.001) * (RRD)) / (RRD));
        CompressorR.setRatio(RatioSmoothR);
    } else {
        CompressorR.setRatio(*OnebandCompressorParameters.getRawParameterValue("ratior"));
    }
    MLD = MixSmoothL - (*OnebandCompressorParameters.getRawParameterValue("mixcompl") / 100.f);
    if (MixSmoothL != (*OnebandCompressorParameters.getRawParameterValue("mixcompl") / 100.f)) {
        MixSmoothL = MixSmoothL - (((MLD + 0.001) * (MLD)) / (MLD));
    }
    MRD = MixSmoothR - (*OnebandCompressorParameters.getRawParameterValue("mixcompr") / 100.f);
    if (MixSmoothR != (*OnebandCompressorParameters.getRawParameterValue("mixcompr") / 100.f)) {
        MixSmoothR = MixSmoothR - (((MRD + 0.001) * (MRD)) / (MRD));
    }
}

void OneBandCompressorAudioProcessor::updateTimeParameters()
{
    float ALD, ARD, RTLD, RTRD, TLD, TRD;
    ALD = AttackTimeSmoothL - *OnebandCompressorParameters.getRawParameterValue("attacktime_l");
    if (AttackTimeSmoothL != *OnebandCompressorParameters.getRawParameterValue("attacktime_l")) {
        AttackTimeSmoothL = AttackTimeSmoothL - (((ALD + 0.001) * (ALD)) / (ALD));
        CompressorL.setAttack(AttackTimeSmoothL);
    } else {
     CompressorL.setAttack(*OnebandCompressorParameters.getRawParameterValue("attacktime_l"));
    }
    ARD = AttackTimeSmoothR - *OnebandCompressorParameters.getRawParameterValue("attacktime_r");
    if (AttackTimeSmoothR != *OnebandCompressorParameters.getRawParameterValue("attacktime_r")) {
        AttackTimeSmoothR = AttackTimeSmoothR - (((ARD + 0.001) * (ARD)) / (ARD));
        CompressorR.setAttack(AttackTimeSmoothR);
    } else {
     CompressorR.setAttack(*OnebandCompressorParameters.getRawParameterValue("attacktime_r"));
    }
    RTLD = ReleaseTimeSmoothL - *OnebandCompressorParameters.getRawParameterValue("releasetime_l");
    if (ReleaseTimeSmoothL != *OnebandCompressorParameters.getRawParameterValue("releasetime_l")) {
        ReleaseTimeSmoothL = ReleaseTimeSmoothL - (((RTLD + 0.001) * (RTLD)) / (RTLD));
        CompressorL.setRelease(ReleaseTimeSmoothL);
    } else {
     CompressorL.setRelease(*OnebandCompressorParameters.getRawParameterValue("releasetime_l"));
    }
    RTRD = ReleaseTimeSmoothR - *OnebandCompressorParameters.getRawParameterValue("releasetime_r");
    if (ReleaseTimeSmoothR != *OnebandCompressorParameters.getRawParameterValue("releasetime_r")) {
        ReleaseTimeSmoothR = ReleaseTimeSmoothR - (((RTRD + 0.001) * (RTRD)) / (RTRD));
        CompressorR.setRelease(ReleaseTimeSmoothR);
    } else {
     CompressorR.setRelease(*OnebandCompressorParameters.getRawParameterValue("releasetime_r"));
    }
    TLD = TresholdSmoothL - *OnebandCompressorParameters.getRawParameterValue("tresholdl");
    if (TresholdSmoothL != *OnebandCompressorParameters.getRawParameterValue("tresholdl")) {
        TresholdSmoothL = TresholdSmoothL - (((TLD + 0.001) * (TLD)) / (TLD));
        CompressorL.setTreshold(TresholdSmoothL);
    } else {
        CompressorL.setTreshold(*OnebandCompressorParameters.getRawParameterValue("tresholdl"));
    }
    TRD = TresholdSmoothR - *OnebandCompressorParameters.getRawParameterValue("tresholdr");
    if (TresholdSmoothR != *OnebandCompressorParameters.getRawParameterValue("tresholdr")) {
        TresholdSmoothR = TresholdSmoothR - (((TRD + 0.001) * (TRD)) / (TRD));
        CompressorR.setTreshold(TresholdSmoothR);
    } else {
        CompressorR.setTreshold(*OnebandCompressorParameters.getRawParameterValue("tresholdr"));
    }
}

void OneBandCompressorAudioProcessor::processBlock (AudioBuffer<float>& buffer, MidiBuffer& midiMessages)
        CurINLGain = PolarityL * Decibels::decibelsToGain(*OnebandCompressorParameters.getRawParameterValue("inlgain"));
        CurINRGain = PolarityR * Decibels::decibelsToGain(*OnebandCompressorParameters.getRawParameterValue("inrgain"));
        if (PrevINLGain != CurINLGain) {
            buffer.applyGainRamp(0, 0, buffer.getNumSamples(), PrevINLGain, CurINLGain);
        } else {
            buffer.applyGain(0, 0, buffer.getNumSamples(), CurINLGain);
        }
        if (PrevINRGain != CurINRGain) {
            buffer.applyGainRamp(1, 0, buffer.getNumSamples(),  PrevINRGain,  CurINRGain);
        } else {
            buffer.applyGain(1, 0, buffer.getNumSamples(), CurINRGain);
        }
        AudioSampleBuffer CompressorBuffer; CompressorBuffer.clear(); CompressorBuffer.setSize(buffer.getNumChannels(), buffer.getNumSamples());
        CompressorBuffer.addFrom(0, 0, buffer, 0, 0, buffer.getNumSamples(), 1.0f);
        CompressorBuffer.addFrom(1, 0, buffer, 1, 0, buffer.getNumSamples(), 1.0f);
        updateParameters();
        for (int sample = 0; sample < buffer.getNumSamples(); sample++) {
            updateTimeParameters();
            CompressorL.processIndividualSample(CompressorBuffer.getWritePointer(0) [sample]);
            CompressorR.processIndividualSample(CompressorBuffer.getWritePointer(1) [sample]);
        }
        buffer.applyGain(0, 0, buffer.getNumSamples(), 1 - MixSmoothL);
        buffer.applyGain(1, 0, buffer.getNumSamples(), 1 - MixSmoothR);
        buffer.addFrom(0, 0, CompressorBuffer, 0, 0, CompressorBuffer.getNumSamples(), MixSmoothL);
        buffer.addFrom(1, 0, CompressorBuffer, 1, 0, CompressorBuffer.getNumSamples(), MixSmoothR);
        CurOutLGain = Decibels::decibelsToGain(*OnebandCompressorParameters.getRawParameterValue("outlgain"));
        if (PrevOUTLGain != CurOutLGain) {
            buffer.applyGainRamp(0, 0, buffer.getNumSamples(), PrevOUTLGain, CurOutLGain);
        } else {
            buffer.applyGain(0, 0, buffer.getNumSamples(), CurOutLGain);
        }
        CurOutRGain = Decibels::decibelsToGain(*OnebandCompressorParameters.getRawParameterValue("outrgain"));
        if (PrevOUTRGain != CurOutRGain) {
            buffer.applyGainRamp(1, 0, buffer.getNumSamples(),  PrevOUTRGain, CurOutRGain);
        } else {
            buffer.applyGain(1, 0, buffer.getNumSamples(), CurOutRGain);
        }
        PrevINLGain = CurINLGain; PrevINRGain = CurINRGain;
```



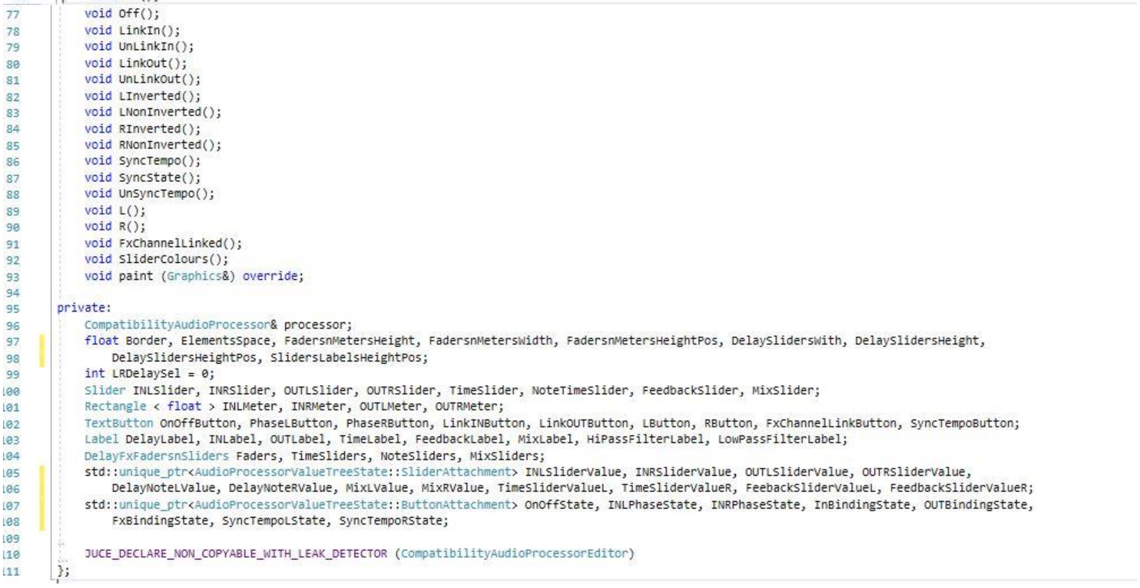
```
            PrevOUTLGain = CurOutLGain; PrevOUTRGain = CurOutRGain;
        } else {
            processBlockBypassed(buffer, midiMessages);
        }
}

//==============================================================================
bool OneBandCompressorAudioProcessor::hasEditor() const
{
    return true;
}

AudioProcessorEditor* OneBandCompressorAudioProcessor::createEditor()
{
    return new OneBandCompressorAudioProcessorEditor (*this);
}

//==============================================================================
void OneBandCompressorAudioProcessor::getStateInformation (MemoryBlock& destData)
{
    auto PluginState = OnebandCompressorParameters.copyState();
    std::unique_ptr<XmlElement> xml(PluginState.createXml());
    copyXmlToBinary(*xml, destData);
}

void OneBandCompressorAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    std::unique_ptr<XmlElement> xmlstate(getXmlFromBinary(data, sizeInBytes));
    if (xmlstate.get() != nullptr && xmlstate->hasTagName(OnebandCompressorParameters.state.getType()))
    {
        OnebandCompressorParameters.replaceState(ValueTree::fromXml(*xmlstate));
    }
}

//==============================================================================
AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new OneBandCompressorAudioProcessor();}
```

Anexo 3. Algoritmo del módulo de *Delay*



```
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"

//==============================================================================


class DelayFxFadersnSliders : public LookAndFeel_V4
{
public:
    int IsFaderSlider, FaderStatus, IsMixSlider, IsTimeSlider, IsNoteSlider; float TextBoxHeight;
    void drawLinearSlider(Graphics& g, int x, int y, int width, int height,
        float sliderPos,
        float minSliderPos,
        float maxSliderPos,
        const Slider::SliderStyle style, Slider& slider) override
    {
        if (IsFaderSlider == 1 && IsTimeSlider == 0 && IsMixSlider == 0) {
            IsNoteSlider = 0;
            slider.setSliderStyle(Slider::SliderStyle::LinearBarVertical);
            slider.setTextBoxStyle(Slider::TextBoxBelow, false, width, TextBoxHeight);
            Path Fader;
            if (FaderStatus == 1) {
                g.setColour(Colours::deepskyblue);
                slider.setColour(Slider::textBoxTextColourId, Colours::blue);
            }
            else {
                g.setColour(Colours::darkgrey);
                slider.setColour(Slider::textBoxTextColourId, Colours::lightgrey);
```

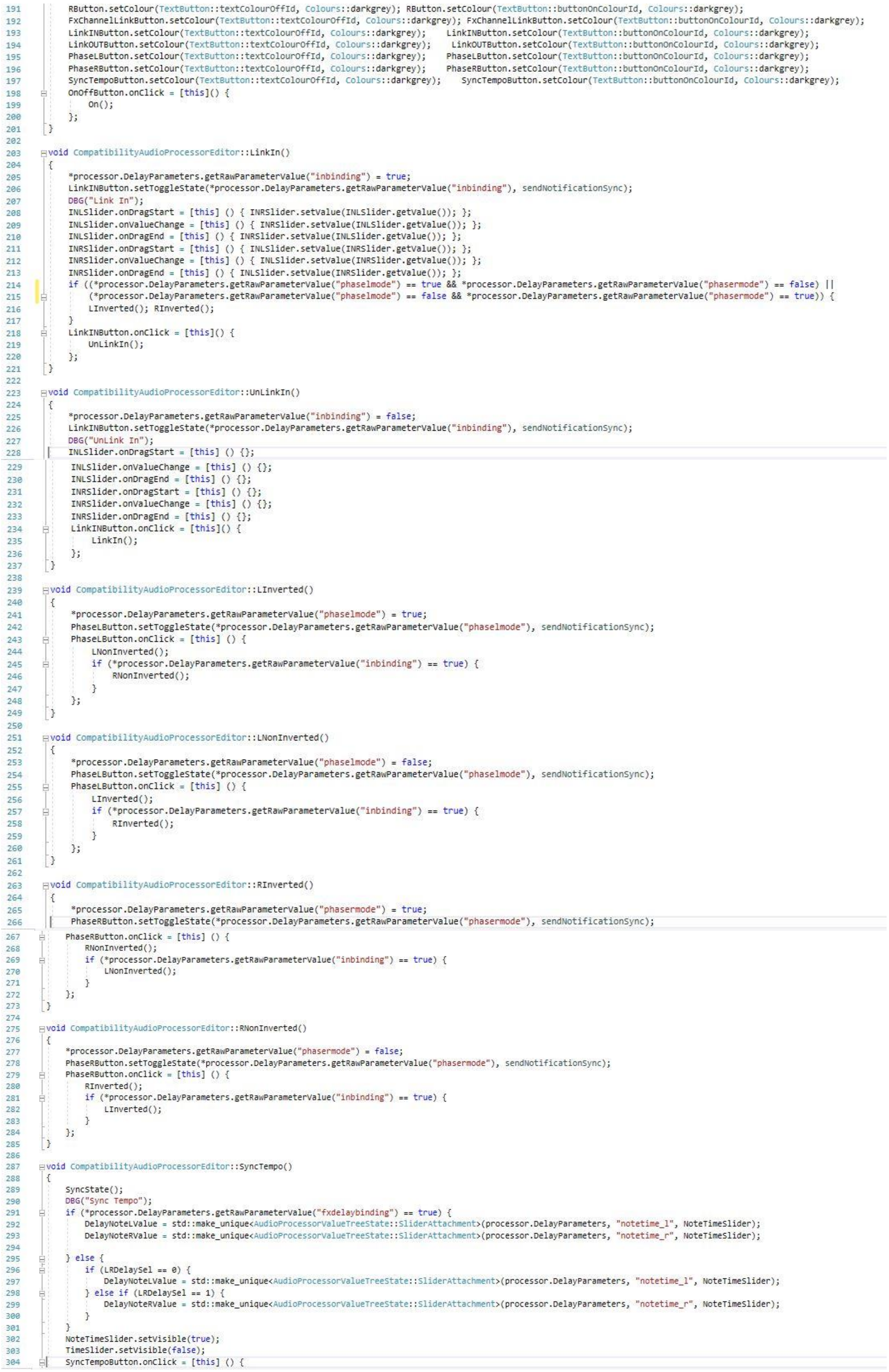
```cpp
            }
            Fader.addRectangle(0, maxSliderPos - minSliderPos, width, sliderPos - (height));
            g.fillPath(Fader, AffineTransform::verticalFlip(sliderPos));
            slider.setSkewFactor(3);
            slider.setNumDecimalPlacesToDisplay(2);
            slider.setTextValueSuffix("dB");
        }
        else if (IsFaderSlider == 0) {
            x = slider.getX();
            y = slider.getY();
            width = slider.getWidth();
            height = slider.getHeight();
            sliderPos = slider.getValue();
            minSliderPos = slider.getMinimum();
            maxSliderPos = slider.getMaximum();
            slider.setSliderStyle(Slider::SliderStyle::RotaryHorizontalVerticalDrag);
            slider.setTextBoxStyle(Slider::TextBoxBelow, false, 0.65 * width, TextBoxHeight);
            slider.setNumDecimalPlacesToDisplay(2);
            if (IsTimeSlider == 1 && IsMixSlider == 0) {
                if (IsNoteSlider == 0) {
                    slider.setTextValueSuffix("  ms");
                }
            } else if (IsTimeSlider == 0 && IsMixSlider == 1) {
                slider.setTextValueSuffix(" %");
            }
        }
        slider.setPopupDisplayEnabled(true, true, slider.getParentComponent());
    }
};

class CompatibilityAudioProcessorEditor  : public AudioProcessorEditor
{
public:
    CompatibilityAudioProcessorEditor (CompatibilityAudioProcessor&);
    ~CompatibilityAudioProcessorEditor();

    //==============================================================================
    void On();
```



```cpp
    void Off();
    void LinkIn();
    void UnLinkIn();
    void LinkOut();
    void UnLinkOut();
    void LInverted();
    void LNonInverted();
    void RInverted();
    void RNonInverted();
    void SyncTempo();
    void SyncState();
    void UnSyncTempo();
    void L();
    void R();
    void FxChannelLinked();
    void SliderColours();
    void paint (Graphics&) override;

private:
    CompatibilityAudioProcessor& processor;
    float Border, ElementsSpace, FadersnMetersHeight, FadersnMetersWidth, FadersnMetersHeightPos, DelaySlidersWith, DelaySlidersHeight,
        DelaySlidersHeightPos, SlidersLabelsHeightPos;
    int LRDelaySel = 0;
    Slider INLSlider, INRSlider, OUTLSlider, OUTRSlider, TimeSlider, NoteTimeSlider, FeedbackSlider, MixSlider;
    Rectangle < float > INLMeter, INRMeter, OUTLMeter, OUTRMeter;
    TextButton OnOffButton, PhaseLButton, PhaseRButton, LinkINButton, LinkOUTButton, LButton, RButton, FxChannelLinkButton, SyncTempoButton;
    Label DelayLabel, INLabel, OUTLabel, TimeLabel, FeedbackLabel, MixLabel, HiPassFilterLabel, LowPassFilterLabel;
    DelayFxFadersnSliders Faders, TimeSliders, NoteSliders, MixSliders;
    std::unique_ptr<AudioProcessorValueTreeState::SliderAttachment> INLSliderValue, INRSliderValue, OUTLSliderValue, OUTRSliderValue,
        DelayNoteLValue, DelayNoteRValue, MixLValue, MixRValue, TimeSliderValueL, TimeSliderValueR, FeebackSliderValueL, FeedbackSliderValueR;
    std::unique_ptr<AudioProcessorValueTreeState::ButtonAttachment> OnOffState, INLPhaseState, INRPhaseState, InBindingState, OUTBindingState,
        FxBindingState, SyncTempoLState, SyncTempoRState;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (CompatibilityAudioProcessorEditor)
};
```

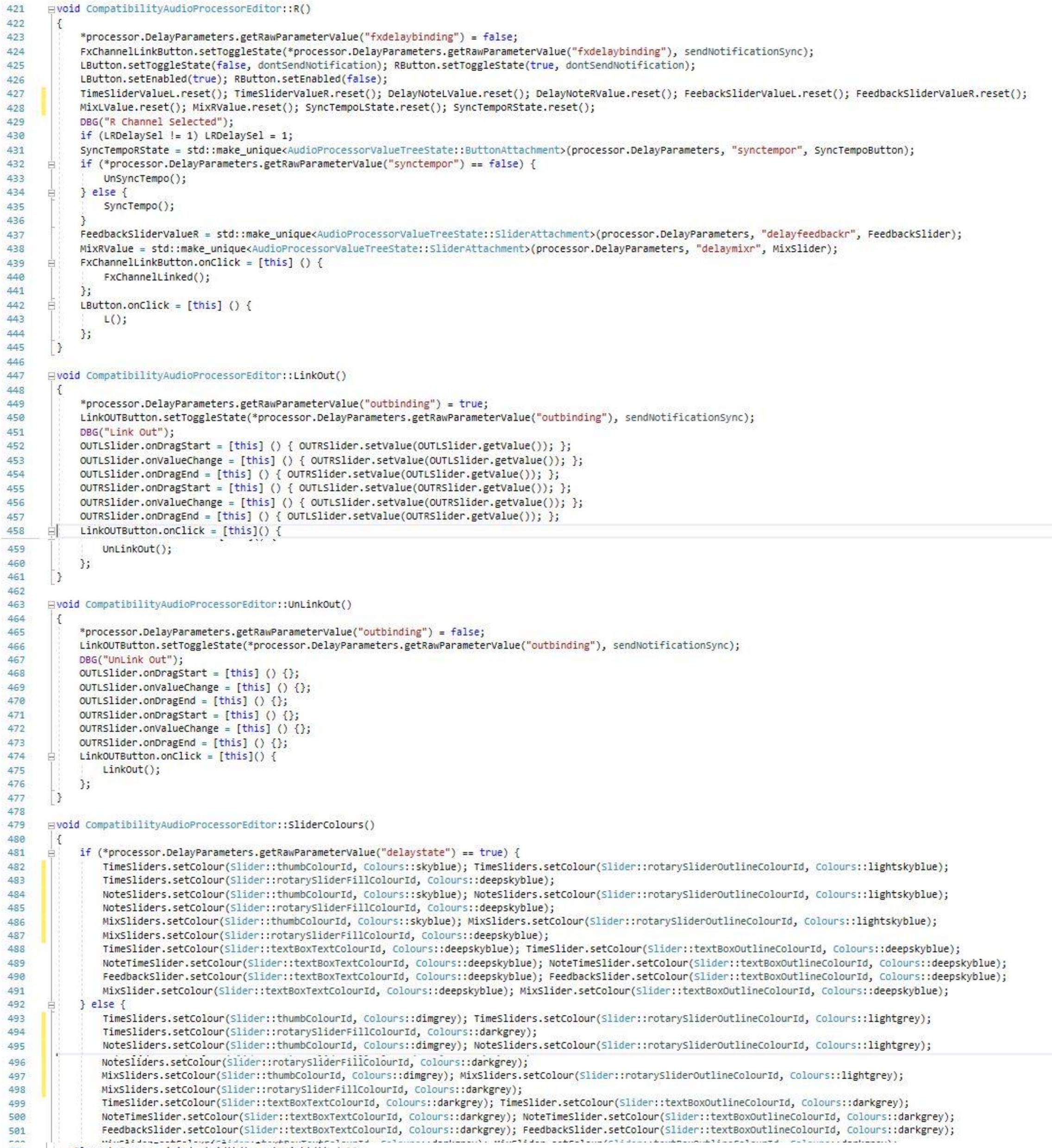
```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
CompatibilityAudioProcessorEditor::CompatibilityAudioProcessorEditor(CompatibilityAudioProcessor& p)
    : AudioProcessorEditor(&p), processor(p)
{
    if (getParentWidth() <= 1400 && getParentHeight() <= 800) {
        setSize(getParentWidth() / 1.25, 0.45 * getParentHeight());
    }
    else {
        setSize(getParentWidth() / 1.7, 0.35 * getParentHeight());
    }
    OnOffState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "delaystate", OnOffButton);
    INLPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "phaselmode", PhaseLButton);
    INRPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "phasermode", PhaseRButton);
    InBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "inbinding", LinkINButton);
    OUTBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "outbinding", LinkOUTButton);
    FxBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "fxdelaybinding", FxChannelLinkButton);
    INLPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "phaselmode", PhaseLButton);
    INRPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "phasermode", PhaseRButton);
    INLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "inlgain", INLSlider);
    INRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "inrgain", INRSlider);
    OUTLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "outlgain", OUTLSlider);
    OUTRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "outrgain", OUTRSlider);
    INLSlider.setRange(-100.f, 12.f, 0.01); INRSlider.setRange(-100.f, 12.f, 0.01); OUTLSlider.setRange(-100.f, 12.f, 0.01); OUTRSlider.setRange(-100.f, 12.f, 0.01);
    TimeSlider.setRange(1, 4000, 1); FeedbackSlider.setRange(0.f, 100.f, 0.01); MixSlider.setRange(0.f, 100.f, 0.01);
    MixSlider.setRange(0.0f, 100.0f, 0.01); FeedbackSlider.setRange(0.0f, 100.0f, 0.01);
    DBG("Delay Stereo");
    Border = jmin(getWidth(), getHeight()) / 30;
```

```cpp
        LinkINButton.getWidth(), LinkINButton.getHeight());
    MixLabel.setBounds(FeedbackLabel.getX() + FeedbackLabel.getWidth() + ElementsSpace, FeedbackLabel.getY(), FeedbackLabel.getWidth(), FeedbackLabel.getHeight());
    MixSlider.setBounds(MixLabel.getX(), DelaySlidersHeightPos, DelaySlidersWith, DelaySlidersHeight);
    FxChannelLinkButton.setBounds(getWidth() * 0.5 - LinkINButton.getWidth() * 0.5, LinkINButton.getY(), LinkINButton.getWidth(), LinkINButton.getHeight());
    LButton.setBounds(FxChannelLinkButton.getX() - 2.5 * ElementsSpace, FxChannelLinkButton.getY(), 1.5 * ElementsSpace, FxChannelLinkButton.getHeight());
    RButton.setBounds(FxChannelLinkButton.getX() + FxChannelLinkButton.getWidth() + ElementsSpace, FxChannelLinkButton.getY(), LButton.getWidth(), FxChannelLinkButton.getHeight());
    OUTLSlider.setBounds(getWidth() - Border - 3 * FadersnMetersWidth  - 3.5 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    OUTRSlider.setBounds(OUTLSlider.getX() + OUTLSlider.getWidth() + 2 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    LinkOUTButton.setBounds(OUTRSlider.getX() - 2 * ElementsSpace - 0.75 * FadersnMetersWidth, LinkINButton.getY(), LinkINButton.getWidth(), LinkINButton.getHeight());
    OUTLabel.setBounds(OUTRSlider.getX() - 2 * ElementsSpace, INLabel.getY(), INLabel.getWidth(), INLabel.getHeight());
    OnOffButton.setBounds(OUTRSlider.getX(), DelayLabel.getY(), OUTRSlider.getWidth(), PhaseLButton.getHeight());
    PhaseLButton.setColour(TextButton::buttonColourId, Colours::black); PhaseRButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkINButton.setColour(TextButton::buttonColourId, Colours::black); LinkOUTButton.setColour(TextButton::buttonColourId, Colours::black);
    OnOffButton.setColour(TextButton::textColourOffId, Colours::darkgrey); OnOffButton.setColour(TextButton::buttonColourId, Colours::black);
    FxChannelLinkButton.setColour(TextButton::buttonColourId, Colours::black); LButton.setColour(TextButton::buttonColourId, Colours::black);
    RButton.setColour(TextButton::buttonColourId, Colours::black); SyncTempoButton.setColour(TextButton::buttonColourId, Colours::black);
    PhaseLButton.setColour(TextButton::textColourOnId, Colours::black); PhaseRButton.setColour(TextButton::textColourOnId, Colours::black);
    LinkINButton.setColour(TextButton::textColourOnId, Colours::black); LinkOUTButton.setColour(TextButton::textColourOnId, Colours::black);
    OnOffButton.setColour(TextButton::textColourOnId, Colours::black); OnOffButton.setColour(TextButton::textColourOnId, Colours::black);
    FxChannelLinkButton.setColour(TextButton::textColourOnId, Colours::black); LButton.setColour(TextButton::textColourOnId, Colours::black);
    RButton.setColour(TextButton::textColourOnId, Colours::black); SyncTempoButton.setColour(TextButton::textColourOnId, Colours::black);
    OnOffButton.setColour(TextButton::buttonOnColourId, Colours::deepskyblue);
    addAndMakeVisible(&DelayLabel); addAndMakeVisible(&INLabel); addAndMakeVisible(&OUTLabel); addAndMakeVisible(&TimeLabel); addAndMakeVisible(&FeedbackLabel);
    addAndMakeVisible(&MixLabel);
    addAndMakeVisible(MixLabel); addAndMakeVisible(&OnOffButton); addAndMakeVisible(&FxChannelLinkButton); addAndMakeVisible(&LButton); addAndMakeVisible(&RButton);
    addAndMakeVisible(&LinkINButton);
    addAndMakeVisible(&LinkOUTButton); addAndMakeVisible(&INLSlider); addAndMakeVisible(&INRSlider); addAndMakeVisible(&OUTLSlider); addAndMakeVisible(&OUTRSlider);
    addAndMakeVisible(&FeedbackSlider);
    addAndMakeVisible(&MixSlider); addAndMakeVisible(&FxChannelLinkButton); addAndMakeVisible(&LButton); addAndMakeVisible(&RButton); addAndMakeVisible(&PhaseLButton);
    addAndMakeVisible(&PhaseRButton);
    addAndMakeVisible(&SyncTempoButton); addAndMakeVisible(&TimeSlider);
    addAndMakeVisible(&NoteTimeSlider);
    DelayLabel.setJustificationType(Justification::Flags::centred); INLabel.setJustificationType(Justification::Flags::centred);
    OUTLabel.setJustificationType(Justification::Flags::centred); TimeLabel.setJustificationType(Justification::Flags::centred);
    FeedbackLabel.setJustificationType(Justification::Flags::centred); MixLabel.setJustificationType(Justification::Flags::centred);
    if (*processor.DelayParameters.getRawParameterValue("delaystate") == true) {
        On();
    } else {
        Off();
    }
    if (*processor.DelayParameters.getRawParameterValue("phaselmode") == false) {
        LNonInverted();
    } else {
        LInverted();
    }
    if (*processor.DelayParameters.getRawParameterValue("phasermode") == false) {
        RNonInverted();
    } else {
        RInverted();
    }
    if (*processor.DelayParameters.getRawParameterValue("inbinding") == true) {
        LinkIn();
    } else {
        UnLinkIn();
    }
    if (*processor.DelayParameters.getRawParameterValue("outbinding") == true) {
        LinkOut();
    } else {
        UnLinkOut();
    }
    if (*processor.DelayParameters.getRawParameterValue("fxdelaybinding") == true) {
        FxChannelLinked();
    } else {
        if (LRDelaySel == 0) {
            L();
        } else if (LRDelaySel == 1) {
            R();
        }
    }
}

CompatibilityAudioProcessorEditor::~CompatibilityAudioProcessorEditor()
{
    INLSlider.setLookAndFeel(nullptr); INRSlider.setLookAndFeel(nullptr); OUTLSlider.setLookAndFeel(nullptr); OUTRSlider.setLookAndFeel(nullptr);
    TimeSlider.setLookAndFeel(nullptr); NoteTimeSlider.setLookAndFeel(nullptr); FeedbackSlider.setLookAndFeel(nullptr); MixSlider.setLookAndFeel(nullptr);
}
void CompatibilityAudioProcessorEditor::On()
{
    *processor.DelayParameters.getRawParameterValue("delaystate") = true;
    OnOffButton.setToggleState(*processor.DelayParameters.getRawParameterValue("delaystate"), sendNotificationSync);
    DBG("On");
    Faders.FaderStatus = 1;
    SliderColours();
    INLabel.setColour(Label::textColourId, Colours::deepskyblue); OUTLabel.setColour(Label::textColourId, Colours::deepskyblue);
    DelayLabel.setColour(Label::textColourId, Colours::deepskyblue);
    TimeLabel.setColour(Label::textColourId, Colours::deepskyblue); FeedbackLabel.setColour(Label::textColourId, Colours::deepskyblue);
    MixLabel.setColour(Label::textColourId, Colours::deepskyblue);
    LButton.setColour(TextButton::textColourOffId, Colours::deepskyblue);    LButton.setColour(TextButton::buttonOnColourId, Colours::deepskyblue);
    RButton.setColour(TextButton::textColourOffId, Colours::deepskyblue);    RButton.setColour(TextButton::buttonOnColourId, Colours::deepskyblue);
    FxChannelLinkButton.setColour(TextButton::textColourOffId, Colours::deepskyblue); FxChannelLinkButton.setColour(TextButton::buttonOnColourId, Colours::deepskyblue);
    LinkINButton.setColour(TextButton::textColourOffId, Colours::deepskyblue);    LinkINButton.setColour(TextButton::buttonOnColourId, Colours::deepskyblue);
    LinkOUTButton.setColour(TextButton::textColourOffId, Colours::deepskyblue);    LinkOUTButton.setColour(TextButton::buttonOnColourId, Colours::deepskyblue);
    PhaseLButton.setColour(TextButton::textColourOffId, Colours::deepskyblue);    PhaseLButton.setColour(TextButton::buttonOnColourId, Colours::deepskyblue);
    PhaseRButton.setColour(TextButton::textColourOffId, Colours::deepskyblue);    PhaseRButton.setColour(TextButton::buttonOnColourId, Colours::deepskyblue);
    SyncTempoButton.setColour(TextButton::textColourOffId, Colours::deepskyblue);    SyncTempoButton.setColour(TextButton::buttonOnColourId, Colours::deepskyblue);
    OnOffButton.onClick = [this]() {
        Off();
    };
}

void CompatibilityAudioProcessorEditor::Off()
{
    *processor.DelayParameters.getRawParameterValue("delaystate") = false;
    OnOffButton.setToggleState(*processor.DelayParameters.getRawParameterValue("delaystate"), sendNotificationSync);
    DBG("Off");
    Faders.FaderStatus = 0;
    SliderColours();
    INLabel.setColour(Label::textColourId, Colours::darkgrey); OUTLabel.setColour(Label::textColourId, Colours::darkgrey);
    DelayLabel.setColour(Label::textColourId, Colours::darkgrey);
    TimeLabel.setColour(Label::textColourId, Colours::darkgrey); FeedbackLabel.setColour(Label::textColourId, Colours::darkgrey);
    MixLabel.setColour(Label::textColourId, Colours::darkgrey);
    LinkOUTButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    LButton.setColour(TextButton::textColourOffId, Colours::darkgrey); LButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
```



```cpp
    RButton.setColour(TextButton::textColourOffId, Colours::darkgrey); RButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    FxChannelLinkButton.setColour(TextButton::textColourOffId, Colours::darkgrey); FxChannelLinkButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    LinkINButton.setColour(TextButton::textColourOffId, Colours::darkgrey);    LinkINButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    LinkOUTButton.setColour(TextButton::textColourOffId, Colours::darkgrey);    LinkOUTButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    PhaseLButton.setColour(TextButton::textColourOffId, Colours::darkgrey);    PhaseLButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    PhaseRButton.setColour(TextButton::textColourOffId, Colours::darkgrey);    PhaseRButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    SyncTempoButton.setColour(TextButton::textColourOffId, Colours::darkgrey);    SyncTempoButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey);
    OnOffButton.onClick = [this]() {
        On();
    };
}

void CompatibilityAudioProcessorEditor::LinkIn()
{
    *processor.DelayParameters.getRawParameterValue("inbinding") = true;
    LinkINButton.setToggleState(*processor.DelayParameters.getRawParameterValue("inbinding"), sendNotificationSync);
    DBG("Link In");
    INLSlider.onDragStart = [this] () { INRSlider.setValue(INLSlider.getValue()); };
    INLSlider.onValueChange = [this] () { INRSlider.setValue(INLSlider.getValue()); };
    INLSlider.onDragEnd = [this] () { INRSlider.setValue(INLSlider.getValue()); };
    INRSlider.onDragStart = [this] () { INLSlider.setValue(INRSlider.getValue()); };
    INRSlider.onValueChange = [this] () { INLSlider.setValue(INRSlider.getValue()); };
    INRSlider.onDragEnd = [this] () { INLSlider.setValue(INRSlider.getValue()); };
    if ((*processor.DelayParameters.getRawParameterValue("phaselmode") == true && *processor.DelayParameters.getRawParameterValue("phasermode") == false) ||
        (*processor.DelayParameters.getRawParameterValue("phaselmode") == false && *processor.DelayParameters.getRawParameterValue("phasermode") == true)) {
        LInverted(); RInverted();
    }
    LinkINButton.onClick = [this]() {
        UnLinkIn();
    };
}

void CompatibilityAudioProcessorEditor::UnLinkIn()
{
    *processor.DelayParameters.getRawParameterValue("inbinding") = false;
    LinkINButton.setToggleState(*processor.DelayParameters.getRawParameterValue("inbinding"), sendNotificationSync);
    DBG("UnLink In");
    INLSlider.onDragStart = [this] () {};
    INLSlider.onValueChange = [this] () {};
    INLSlider.onDragEnd = [this] () {};
    INRSlider.onDragStart = [this] () {};
    INRSlider.onValueChange = [this] () {};
    INRSlider.onDragEnd = [this] () {};
    LinkINButton.onClick = [this]() {
        LinkIn();
    };
}

void CompatibilityAudioProcessorEditor::LInverted()
{
    *processor.DelayParameters.getRawParameterValue("phaselmode") = true;
    PhaseLButton.setToggleState(*processor.DelayParameters.getRawParameterValue("phaselmode"), sendNotificationSync);
    PhaseLButton.onClick = [this] () {
        LNonInverted();
        if (*processor.DelayParameters.getRawParameterValue("inbinding") == true) {
            RNonInverted();
        }
    };
}

void CompatibilityAudioProcessorEditor::LNonInverted()
{
    *processor.DelayParameters.getRawParameterValue("phaselmode") = false;
    PhaseLButton.setToggleState(*processor.DelayParameters.getRawParameterValue("phaselmode"), sendNotificationSync);
    PhaseLButton.onClick = [this] () {
        LInverted();
        if (*processor.DelayParameters.getRawParameterValue("inbinding") == true) {
            RInverted();
        }
    };
}

void CompatibilityAudioProcessorEditor::RInverted()
{
    *processor.DelayParameters.getRawParameterValue("phasermode") = true;
    PhaseRButton.setToggleState(*processor.DelayParameters.getRawParameterValue("phasermode"), sendNotificationSync);
    PhaseRButton.onClick = [this] () {
        RNonInverted();
        if (*processor.DelayParameters.getRawParameterValue("inbinding") == true) {
            LNonInverted();
        }
    };
}

void CompatibilityAudioProcessorEditor::RNonInverted()
{
    *processor.DelayParameters.getRawParameterValue("phasermode") = false;
    PhaseRButton.setToggleState(*processor.DelayParameters.getRawParameterValue("phasermode"), sendNotificationSync);
    PhaseRButton.onClick = [this] () {
        RInverted();
        if (*processor.DelayParameters.getRawParameterValue("inbinding") == true) {
            LInverted();
        }
    };
}

void CompatibilityAudioProcessorEditor::SyncTempo()
{
    SyncState();
    DBG("Sync Tempo");
    if (*processor.DelayParameters.getRawParameterValue("fxdelaybinding") == true) {
        DelayNoteLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "notetime_l", NoteTimeSlider);
        DelayNoteRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "notetime_r", NoteTimeSlider);

    } else {
        if (LRDelaySel == 0) {
            DelayNoteLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "notetime_l", NoteTimeSlider);
        } else if (LRDelaySel == 1) {
            DelayNoteRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "notetime_r", NoteTimeSlider);
        }
    }
    NoteTimeSlider.setVisible(true);
    TimeSlider.setVisible(false);
    SyncTempoButton.onClick = [this] () {
```

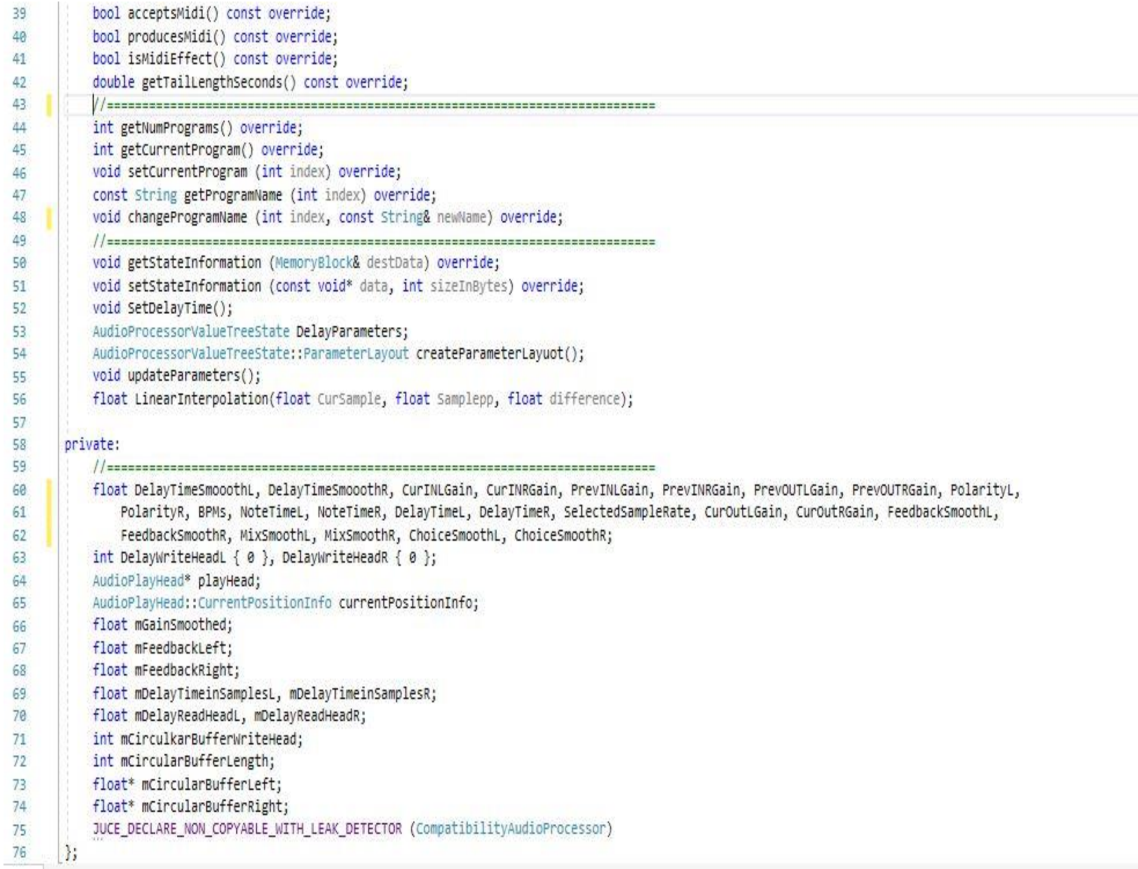
```cpp
        if (*processor.DelayParameters.getRawParameterValue("fxdelaybinding") == true) {
            *processor.DelayParameters.getRawParameterValue("synctempol") = false;
            *processor.DelayParameters.getRawParameterValue("synctempor") = false;
        } else {
            if (LRDelaySel == 0) {
                *processor.DelayParameters.getRawParameterValue("synctempol") = false;
            } else if (LRDelaySel == 1) {
                *processor.DelayParameters.getRawParameterValue("synctempor") = false;
            }
        }
        UnSyncTempo();
    };
}

void CompatibilityAudioProcessorEditor::SyncState()
{
    TimeSliderValueL.reset(); TimeSliderValueR.reset(); DelayNoteLValue.reset(); DelayNoteRValue.reset();
    if (*processor.DelayParameters.getRawParameterValue("fxdelaybinding") == true || LRDelaySel == 0) {
        SyncTempoButton.setToggleState(*processor.DelayParameters.getRawParameterValue("synctempol"), sendNotificationSync);
    } else {
        SyncTempoButton.setToggleState(*processor.DelayParameters.getRawParameterValue("synctempor"), sendNotificationSync);
    }
}

void CompatibilityAudioProcessorEditor::UnSyncTempo()
{
    SyncState();
    DBG("UnSync Tempo");
    if (*processor.DelayParameters.getRawParameterValue("fxdelaybinding") == true) {
        TimeSliderValueL = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delaytime_l", TimeSlider);
        TimeSliderValueR = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delaytime_r", TimeSlider);

    } else {
        if (LRDelaySel == 0) {
            TimeSliderValueL = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delaytime_l", TimeSlider);
        } else if (LRDelaySel == 1) {
            TimeSliderValueR = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delaytime_r", TimeSlider);
    TimeSlider.setVisible(true);
    NoteTimeSlider.setVisible(false);
    SyncTempoButton.onClick = [this] () {
        if (*processor.DelayParameters.getRawParameterValue("fxdelaybinding") == true) {
            *processor.DelayParameters.getRawParameterValue("synctempol") = true;
            *processor.DelayParameters.getRawParameterValue("synctempor") = true;
        } else {
            if (LRDelaySel == 0) {
                *processor.DelayParameters.getRawParameterValue("synctempol") = true;
            } else if (LRDelaySel == 1) {
                *processor.DelayParameters.getRawParameterValue("synctempor") = true;
            }
        }
        SyncTempo();
    };
}

void CompatibilityAudioProcessorEditor::FxChannelLinked()
{
    *processor.DelayParameters.getRawParameterValue("fxdelaybinding") = true;
    FxChannelLinkButton.setToggleState(*processor.DelayParameters.getRawParameterValue("fxdelaybinding"), sendNotificationSync);
    LButton.setToggleState(false, dontSendNotification); RButton.setToggleState(false, dontSendNotification);
    LButton.setEnabled(false); RButton.setEnabled(false);
    FeebackSliderValueL.reset(); FeedbackSliderValueR.reset();  MixLValue.reset(); MixRValue.reset(); SyncTempoLState.reset(); SyncTempoRState.reset();
    TimeSliderValueL.reset(); TimeSliderValueR.reset(); DelayNoteLValue.reset(); DelayNoteRValue.reset();
    DBG("Fx Channel Linked");
    *processor.DelayParameters.getRawParameterValue("synctempor") = *processor.DelayParameters.getRawParameterValue("synctempol");
    SyncTempoLState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "synctempol", SyncTempoButton);
    SyncTempoRState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "synctempor", SyncTempoButton);
    if (*processor.DelayParameters.getRawParameterValue("synctempol") == false && *processor.DelayParameters.getRawParameterValue("synctempor") == false) {
        UnSyncTempo();
    } else {
        SyncTempo();
    }
    FeebackSliderValueL = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delayfeedbackl", FeedbackSlider);
    FeedbackSliderValueR = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delayfeedbackr", FeedbackSlider);
    MixLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delaymixl", MixSlider);
    MixRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delaymixr", MixSlider);
    FxChannelLinkButton.onClick = [this] (){
        if (LRDelaySel == 0) {
            LButton.setToggleState(true, dontSendNotification);
            RButton.setToggleState(false, dontSendNotification);
            L();
        } else if (LRDelaySel == 1){
            RButton.setToggleState(true, dontSendNotification);
            LButton.setToggleState(false, dontSendNotification);
            R();
        }
    };
}

void CompatibilityAudioProcessorEditor::L()
{
    *processor.DelayParameters.getRawParameterValue("fxdelaybinding") = false;
    FxChannelLinkButton.setToggleState(*processor.DelayParameters.getRawParameterValue("fxdelaybinding"), sendNotificationSync);
    LButton.setToggleState(true, dontSendNotification); RButton.setToggleState(false, dontSendNotification);
    LButton.setEnabled(false); RButton.setEnabled(true);
    TimeSliderValueL.reset(); TimeSliderValueR.reset(); DelayNoteLValue.reset(); DelayNoteRValue.reset(); FeebackSliderValueL.reset(); FeedbackSliderValueR.reset();
    MixLValue.reset(); MixRValue.reset(); SyncTempoLState.reset(); SyncTempoRState.reset();
    DBG("L Channel Selected");
    if (LRDelaySel != 0) { LRDelaySel = 0; }
    SyncTempoLState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "synctempol", SyncTempoButton);
    if (*processor.DelayParameters.getRawParameterValue("synctempol") == false) {
        UnSyncTempo();
    } else {
        SyncTempo();
    }
    FeebackSliderValueL = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delayfeedbackl", FeedbackSlider);
    MixLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delaymixl", MixSlider);
    FxChannelLinkButton.onClick = [this] () {
        FxChannelLinked();
    };
    RButton.onClick = [this] () {
        R();
    };
```



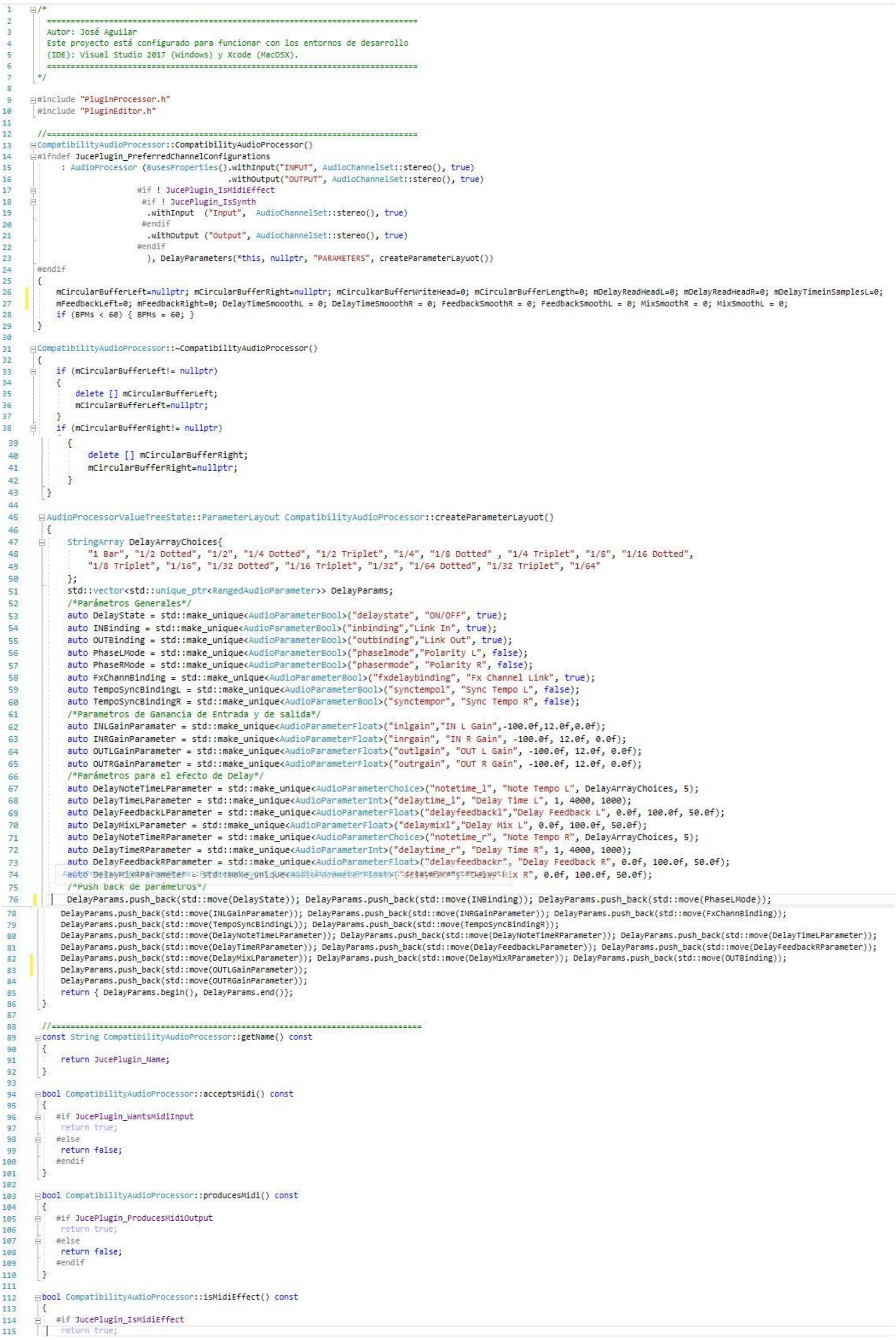
```cpp
void CompatibilityAudioProcessorEditor::R()
{
    *processor.DelayParameters.getRawParameterValue("fxdelaybinding") = false;
    FxChannelLinkButton.setToggleState(*processor.DelayParameters.getRawParameterValue("fxdelaybinding"), sendNotificationSync);
    LButton.setToggleState(false, dontSendNotification); RButton.setToggleState(true, dontSendNotification);
    LButton.setEnabled(true); RButton.setEnabled(false);
    TimeSliderValueL.reset(); TimeSliderValueR.reset(); DelayNoteLValue.reset(); DelayNoteRValue.reset(); FeebackSliderValueL.reset(); FeedbackSliderValueR.reset();
    MixLValue.reset(); MixRValue.reset(); SyncTempoLState.reset(); SyncTempoRState.reset();
    DBG("R Channel Selected");
    if (LRDelaySel != 1) LRDelaySel = 1;
    SyncTempoRState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.DelayParameters, "synctempor", SyncTempoButton);
    if (*processor.DelayParameters.getRawParameterValue("synctempor") == false) {
        UnSyncTempo();
    } else {
        SyncTempo();
    }
    FeedbackSliderValueR = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delayfeedbackr", FeedbackSlider);
    MixRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.DelayParameters, "delaymixr", MixSlider);
    FxChannelLinkButton.onClick = [this] () {
        FxChannelLinked();
    };
    LButton.onClick = [this] () {
        L();
    };
}

void CompatibilityAudioProcessorEditor::LinkOut()
{
    *processor.DelayParameters.getRawParameterValue("outbinding") = true;
    LinkOUTButton.setToggleState(*processor.DelayParameters.getRawParameterValue("outbinding"), sendNotificationSync);
    DBG("Link Out");
    OUTLSlider.onDragStart = [this] () { OUTRSlider.setValue(OUTLSlider.getValue()); };
    OUTLSlider.onValueChange = [this] () { OUTRSlider.setValue(OUTLSlider.getValue()); };
    OUTLSlider.onDragEnd = [this] () { OUTRSlider.setValue(OUTLSlider.getValue()); };
    OUTRSlider.onDragStart = [this] () { OUTLSlider.setValue(OUTRSlider.getValue()); };
    OUTRSlider.onValueChange = [this] () { OUTLSlider.setValue(OUTRSlider.getValue()); };
    OUTRSlider.onDragEnd = [this] () { OUTLSlider.setValue(OUTRSlider.getValue()); };
    LinkOUTButton.onClick = [this]() {
        UnLinkOut();
    };
}

void CompatibilityAudioProcessorEditor::UnLinkOut()
{
    *processor.DelayParameters.getRawParameterValue("outbinding") = false;
    LinkOUTButton.setToggleState(*processor.DelayParameters.getRawParameterValue("outbinding"), sendNotificationSync);
    DBG("UnLink Out");
    OUTLSlider.onDragStart = [this] () {};
    OUTLSlider.onValueChange = [this] () {};
    OUTLSlider.onDragEnd = [this] () {};
    OUTRSlider.onDragStart = [this] () {};
    OUTRSlider.onValueChange = [this] () {};
    OUTRSlider.onDragEnd = [this] () {};
    LinkOUTButton.onClick = [this]() {
        LinkOut();
    };
}

void CompatibilityAudioProcessorEditor::SliderColours()
{
    if (*processor.DelayParameters.getRawParameterValue("delaystate") == true) {
        TimeSliders.setColour(Slider::thumbColourId, Colours::skyblue); TimeSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightskyblue);
        TimeSliders.setColour(Slider::rotarySliderFillColourId, Colours::deepskyblue);
        NoteSliders.setColour(Slider::thumbColourId, Colours::skyblue); NoteSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightskyblue);
        NoteSliders.setColour(Slider::rotarySliderFillColourId, Colours::deepskyblue);
        MixSliders.setColour(Slider::thumbColourId, Colours::skyblue); MixSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightskyblue);
        MixSliders.setColour(Slider::rotarySliderFillColourId, Colours::deepskyblue);
        TimeSlider.setColour(Slider::textBoxTextColourId, Colours::deepskyblue); TimeSlider.setColour(Slider::textBoxOutlineColourId, Colours::deepskyblue);
        NoteTimeSlider.setColour(Slider::textBoxTextColourId, Colours::deepskyblue); NoteTimeSlider.setColour(Slider::textBoxOutlineColourId, Colours::deepskyblue);
        FeedbackSlider.setColour(Slider::textBoxTextColourId, Colours::deepskyblue); FeedbackSlider.setColour(Slider::textBoxOutlineColourId, Colours::deepskyblue);
        MixSlider.setColour(Slider::textBoxTextColourId, Colours::deepskyblue); MixSlider.setColour(Slider::textBoxOutlineColourId, Colours::deepskyblue);
    } else {
        TimeSliders.setColour(Slider::thumbColourId, Colours::dimgrey); TimeSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgrey);
        TimeSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey);
        NoteSliders.setColour(Slider::thumbColourId, Colours::dimgrey); NoteSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgrey);
        NoteSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey);
        MixSliders.setColour(Slider::thumbColourId, Colours::dimgrey); MixSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgrey);
        MixSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey);
        TimeSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); TimeSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        NoteTimeSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); NoteTimeSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        FeedbackSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); FeedbackSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
```

```cpp
    ElementsSpace = jmin(getWidth(), getHeight()) / 20;
    FadersnMetersWidth = getWidth() / 25;
    FadersnMetersHeight = 0.65 * getHeight();
    FadersnMetersHeightPos = 0.175 * getHeight();
    DelaySlidersWith = ((getWidth() - 2 * (5.5 * ElementsSpace + 3 * FadersnMetersWidth + Border)) / 3);
    DelaySlidersHeight = FadersnMetersHeight - 5.75 * ElementsSpace;
    SlidersLabelsHeightPos = getHeight() * 0.5 - DelaySlidersHeight * 0.5 - 1.85 * ElementsSpace;
    DelaySlidersHeightPos = FadersnMetersHeightPos + FadersnMetersHeight * 0.5 - DelaySlidersHeight * 0.5;
    Faders.IsFaderSlider = 1; Faders.IsTimeSlider = 0; Faders.IsMixSlider = 0; Faders.TextBoxHeight = ElementsSpace;
    TimeSliders.IsFaderSlider = 0; TimeSliders.IsTimeSlider = 1; TimeSliders.IsMixSlider = 0; TimeSliders.TextBoxHeight = ElementsSpace; TimeSliders.IsNoteSlider = 0;
    NoteSliders.IsFaderSlider = 0; NoteSliders.IsTimeSlider = 1; NoteSliders.IsMixSlider = 0; NoteSliders.TextBoxHeight = ElementsSpace; NoteSliders.IsNoteSlider = 1;
    MixSliders.IsFaderSlider = 0; MixSliders.IsTimeSlider = 0; MixSliders.IsMixSlider = 1; MixSliders.TextBoxHeight = ElementsSpace;
    DelayLabel.setText("Delay", sendNotificationSync); INLabel.setText("In", sendNotificationSync); OUTLabel.setText("Out", sendNotificationSync);
    HiPassFilterLabel.setText("Hi Pass", sendNotificationSync);
    LowPassFilterLabel.setText("Low Pass", sendNotificationSync); MixLabel.setText("Mix", sendNotificationSync);
    TimeLabel.setText("Time", sendNotificationSync); FeedbackLabel.setText("Feedback", sendNotificationSync);
    LinkINButton.setButtonText("LINK"); FxChannelLinkButton.setButtonText(LinkINButton.getButtonText());
    LinkOUTButton.setButtonText(LinkINButton.getButtonText()); LButton.setButtonText("L"); RButton.setButtonText("R");
    PhaseLButton.setButtonText(CharPointer_UTF8("Ø"));
    OnOffButton.setButtonText(CharPointer_UTF8("•"));
    PhaseRButton.setButtonText(PhaseLButton.getButtonText()); SyncTempoButton.setButtonText("SYNC");
    INLSlider.setLookAndFeel(&Faders); INRSlider.setLookAndFeel(&Faders); OUTLSlider.setLookAndFeel(&Faders); OUTRSlider.setLookAndFeel(&Faders);
    TimeSlider.setLookAndFeel(&TimeSliders);
    NoteTimeSlider.setLookAndFeel(&NoteSliders); FeedbackSlider.setLookAndFeel(&MixSliders); MixSlider.setLookAndFeel(&MixSliders);
    DelayLabel.setBounds(getWidth() / 2 - 3 * FadersnMetersWidth, ((FadersnMetersHeightPos + Border) / 2) - 0.75 * ElementsSpace, 6 * FadersnMetersWidth, 4 * ElementsSpace);
    INLSlider.setBounds(Border + 1.5 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    INRSlider.setBounds(INLSlider.getX() + INLSlider.getWidth() + 2.f * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    INLabel.setBounds(INRSlider.getX() - 2 * ElementsSpace, (getHeight() * 0.5) - ElementsSpace, 2 * ElementsSpace, 2 * ElementsSpace);
    LinkINButton.setBounds(INRSlider.getX() - 2 * ElementsSpace - 0.75 * FadersnMetersWidth, getHeight() - FadersnMetersHeightPos + 0.875 * ElementsSpace,
        1.5 * FadersnMetersWidth + 2 * ElementsSpace, 1.75 * ElementsSpace);
    PhaseLButton.setBounds(INLSlider.getX(), DelayLabel.getY(), INLSlider.getWidth(), 1.5 * ElementsSpace);
    PhaseRButton.setBounds(INRSlider.getX(), DelayLabel.getY(), INRSlider.getWidth(), 1.5 * ElementsSpace);
    TimeLabel.setBounds(Border + 3 * FadersnMetersWidth + 4.5* ElementsSpace, SlidersLabelsHeightPos, DelaySlidersWith, 1.75 * ElementsSpace);
    TimeSlider.setBounds(TimeLabel.getX(), DelaySlidersHeightPos, DelaySlidersWith, DelaySlidersHeight);
    NoteTimeSlider.setBounds(TimeSlider.getBounds());
    FeedbackLabel.setBounds(TimeLabel.getX() + TimeLabel.getWidth() + ElementsSpace, TimeLabel.getY(), TimeLabel.getWidth(), 1.75 * ElementsSpace);
    FeedbackSlider.setBounds(FeedbackLabel.getX(), DelaySlidersHeightPos, DelaySlidersWith, DelaySlidersHeight);
    SyncTempoButton.setBounds(TimeSlider.getX() + TimeSlider.getWidth() * 0.5 - LinkINButton.getWidth() * 0.5, getHeight() - FadersnMetersHeightPos - LinkINButton.getHeight(),
```

```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#define MAX_DELAY_TIME 4
//==============================================================================

class CompatibilityAudioProcessor  : public AudioProcessor
{
public:
    //==============================================================================
    CompatibilityAudioProcessor();
    ~CompatibilityAudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif

    void processBlock (AudioBuffer<float>&, MidiBuffer&) override;

    //==============================================================================
    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==============================================================================
    const String getName() const override;

```



```cpp
    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;
    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const String getProgramName (int index) override;
    void changeProgramName (int index, const String& newName) override;
    //==============================================================================
    void getStateInformation (MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;
    void SetDelayTime();
    AudioProcessorValueTreeState DelayParameters;
    AudioProcessorValueTreeState::ParameterLayout createParameterLayuot();
    void updateParameters();
    float LinearInterpolation(float CurSample, float Samplepp, float difference);

private:
    //==============================================================================
    float DelayTimeSmooothL, DelayTimeSmooothR, CurINLGain, CurINRGain, PrevINLGain, PrevINRGain, PrevOUTLGain, PrevOUTRGain, PolarityL,
        PolarityR, BPMs, NoteTimeL, NoteTimeR, DelayTimeL, DelayTimeR, SelectedSampleRate, CurOutLGain, CurOutRGain, FeedbackSmoothL,
        FeedbackSmoothR, MixSmoothL, MixSmoothR, ChoiceSmoothL, ChoiceSmoothR;
    int DelayWriteHeadL { 0 }, DelayWriteHeadR { 0 };
    AudioPlayHead* playHead;
    AudioPlayHead::CurrentPositionInfo currentPositionInfo;
    float mGainSmoothed;
    float mFeedbackLeft;
    float mFeedbackRight;
    float mDelayTimeinSamplesL, mDelayTimeinSamplesR;
    float mDelayReadHeadL, mDelayReadHeadR;
    int mCirculkarBufferWriteHead;
    int mCircularBufferLength;
    float* mCircularBufferLeft;
    float* mCircularBufferRight;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (CompatibilityAudioProcessor)
};
```

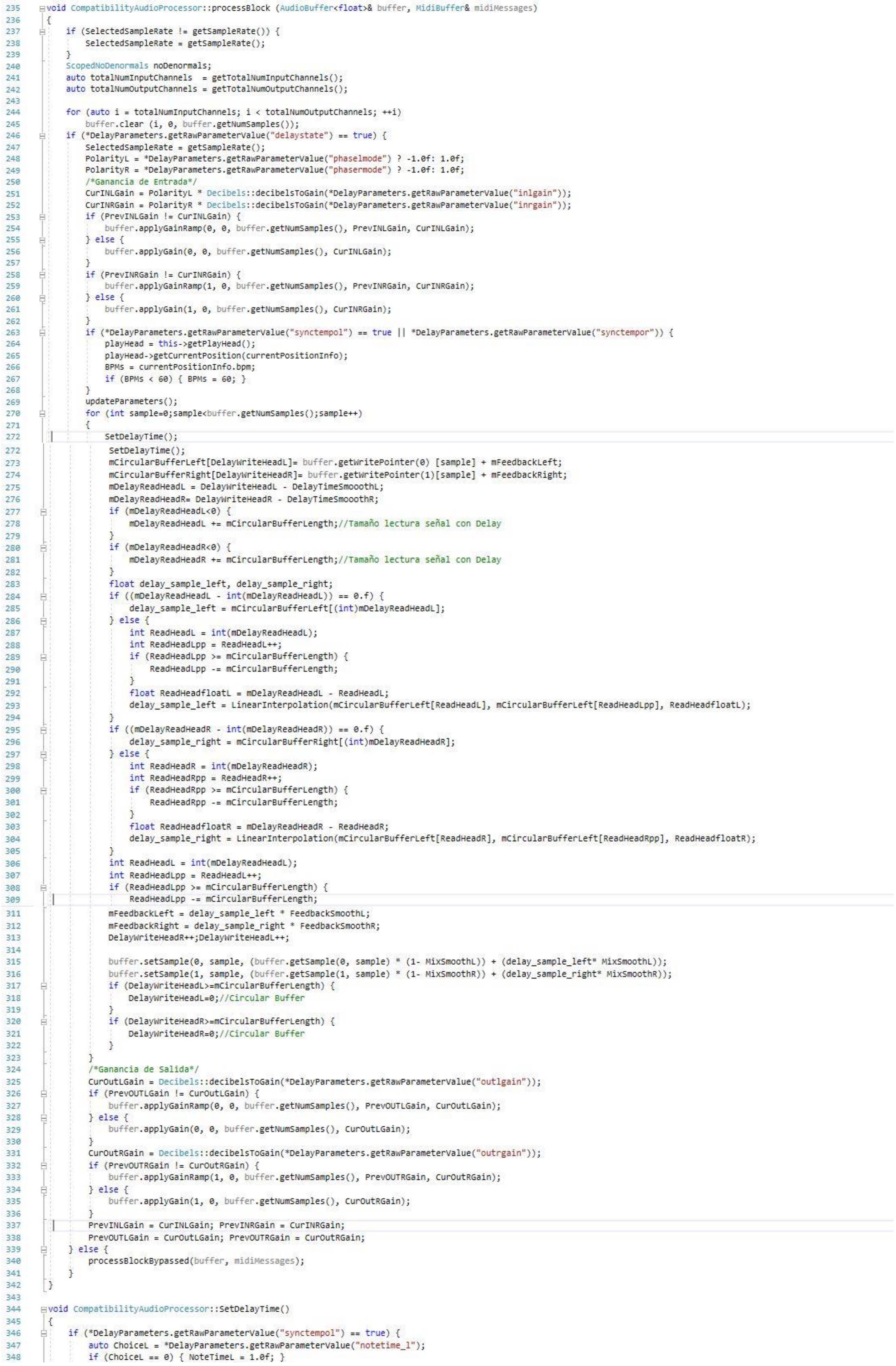


```cpp
/*
  ============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
CompatibilityAudioProcessor::CompatibilityAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties().withInput("INPUT", AudioChannelSet::stereo(), true)
                                        .withOutput("OUTPUT", AudioChannelSet::stereo(), true)
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", AudioChannelSet::stereo(), true)
                     #endif
                       ), DelayParameters(*this, nullptr, "PARAMETERS", createParameterLayuot())
#endif
{
    mCircularBufferLeft=nullptr; mCircularBufferRight=nullptr; mCirculkarBufferWriteHead=0; mCircularBufferLength=0; mDelayReadHeadL=0; mDelayReadHeadR=0; mDelayTimeinSamplesL=0;
    mFeedbackLeft=0; mFeedbackRight=0; DelayTimeSmooothL = 0; DelayTimeSmooothR = 0; FeedbackSmoothR = 0; FeedbackSmoothL = 0; MixSmoothR = 0; MixSmoothL = 0;
    if (BPMs < 60) { BPMs = 60; }
}

CompatibilityAudioProcessor::~CompatibilityAudioProcessor()
{
    if (mCircularBufferLeft!= nullptr)
    {
        delete [] mCircularBufferLeft;
        mCircularBufferLeft=nullptr;
    }
    if (mCircularBufferRight!= nullptr)
    {
        delete [] mCircularBufferRight;
        mCircularBufferRight=nullptr;
    }
}

AudioProcessorValueTreeState::ParameterLayout CompatibilityAudioProcessor::createParameterLayuot()
{
    StringArray DelayArrayChoices{
        "1 Bar", "1/2 Dotted", "1/2", "1/4 Dotted", "1/2 Triplet", "1/4", "1/8 Dotted" , "1/4 Triplet", "1/8", "1/16 Dotted",
        "1/8 Triplet", "1/16", "1/32 Dotted", "1/16 Triplet", "1/32", "1/64 Dotted", "1/32 Triplet", "1/64"
    };
    std::vector<std::unique_ptr<RangedAudioParameter>> DelayParams;
    /*Parámetros Generales*/
    auto DelayState = std::make_unique<AudioParameterBool>("delaystate", "ON/OFF", true);
    auto INBinding = std::make_unique<AudioParameterBool>("inbinding","Link In", true);
    auto OUTBinding = std::make_unique<AudioParameterBool>("outbinding","Link Out", true);
    auto PhaseLMode = std::make_unique<AudioParameterBool>("phaselmode","Polarity L", false);
    auto PhaseRMode = std::make_unique<AudioParameterBool>("phasermode", "Polarity R", false);
    auto FxChannBinding = std::make_unique<AudioParameterBool>("fxdelaybinding", "Fx Channel Link", true);
    auto TempoSyncBindingL = std::make_unique<AudioParameterBool>("synctempol", "Sync Tempo L", false);
    auto TempoSyncBindingR = std::make_unique<AudioParameterBool>("synctempor", "Sync Tempo R", false);
    /*Parametros de Ganancia de Entrada y de salida*/
    auto INLGainParamater = std::make_unique<AudioParameterFloat>("inlgain","IN L Gain",-100.0f,12.0f,0.0f);
    auto INRGainParameter = std::make_unique<AudioParameterFloat>("inrgain", "IN R Gain", -100.0f, 12.0f, 0.0f);
    auto OUTLGainParameter = std::make_unique<AudioParameterFloat>("outlgain", "OUT L Gain", -100.0f, 12.0f, 0.0f);
    auto OUTRGainParameter = std::make_unique<AudioParameterFloat>("outrgain", "OUT R Gain", -100.0f, 12.0f, 0.0f);
    /*Parámetros para el efecto de Delay*/
    auto DelayNoteTimeLParameter = std::make_unique<AudioParameterChoice>("notetime_l", "Note Tempo L", DelayArrayChoices, 5);
    auto DelayTimeLParameter = std::make_unique<AudioParameterInt>("delaytime_l", "Delay Time L", 1, 4000, 1000);
    auto DelayFeedbackLParameter = std::make_unique<AudioParameterFloat>("delayfeedbackl","Delay Feedback L", 0.0f, 100.0f, 50.0f);
    auto DelayMixLParameter = std::make_unique<AudioParameterFloat>("delaymixl","Delay Mix L", 0.0f, 100.0f, 50.0f);
    auto DelayNoteTimeRParameter = std::make_unique<AudioParameterChoice>("notetime_r", "Note Tempo R", DelayArrayChoices, 5);
    auto DelayTimeRParameter = std::make_unique<AudioParameterInt>("delaytime_r", "Delay Time R", 1, 4000, 1000);
    auto DelayFeedbackRParameter = std::make_unique<AudioParameterFloat>("delayfeedbackr", "Delay Feedback R", 0.0f, 100.0f, 50.0f);
    auto DelayMixRParameter = std::make_unique<AudioParameterFloat>("delaymixr", "Delay Mix R", 0.0f, 100.0f, 50.0f);
    /*Push back de parámetros*/
    DelayParams.push_back(std::move(DelayState)); DelayParams.push_back(std::move(INBinding)); DelayParams.push_back(std::move(PhaseLMode));
    DelayParams.push_back(std::move(INLGainParamater)); DelayParams.push_back(std::move(INRGainParameter)); DelayParams.push_back(std::move(FxChannBinding));
    DelayParams.push_back(std::move(TempoSyncBindingL)); DelayParams.push_back(std::move(TempoSyncBindingR));
    DelayParams.push_back(std::move(DelayNoteTimeLParameter)); DelayParams.push_back(std::move(DelayNoteTimeRParameter)); DelayParams.push_back(std::move(DelayTimeLParameter));
    DelayParams.push_back(std::move(DelayTimeRParameter)); DelayParams.push_back(std::move(DelayFeedbackLParameter)); DelayParams.push_back(std::move(DelayFeedbackRParameter));
    DelayParams.push_back(std::move(DelayMixLParameter)); DelayParams.push_back(std::move(DelayMixRParameter)); DelayParams.push_back(std::move(OUTBinding));
    DelayParams.push_back(std::move(OUTLGainParameter));
    DelayParams.push_back(std::move(OUTRGainParameter));
    return { DelayParams.begin(), DelayParams.end()};
}

//==============================================================================
const String CompatibilityAudioProcessor::getName() const
{
    return JucePlugin_Name;
}

bool CompatibilityAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}

bool CompatibilityAudioProcessor::producesMidi() const
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
    return false;
   #endif
}

bool CompatibilityAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
```

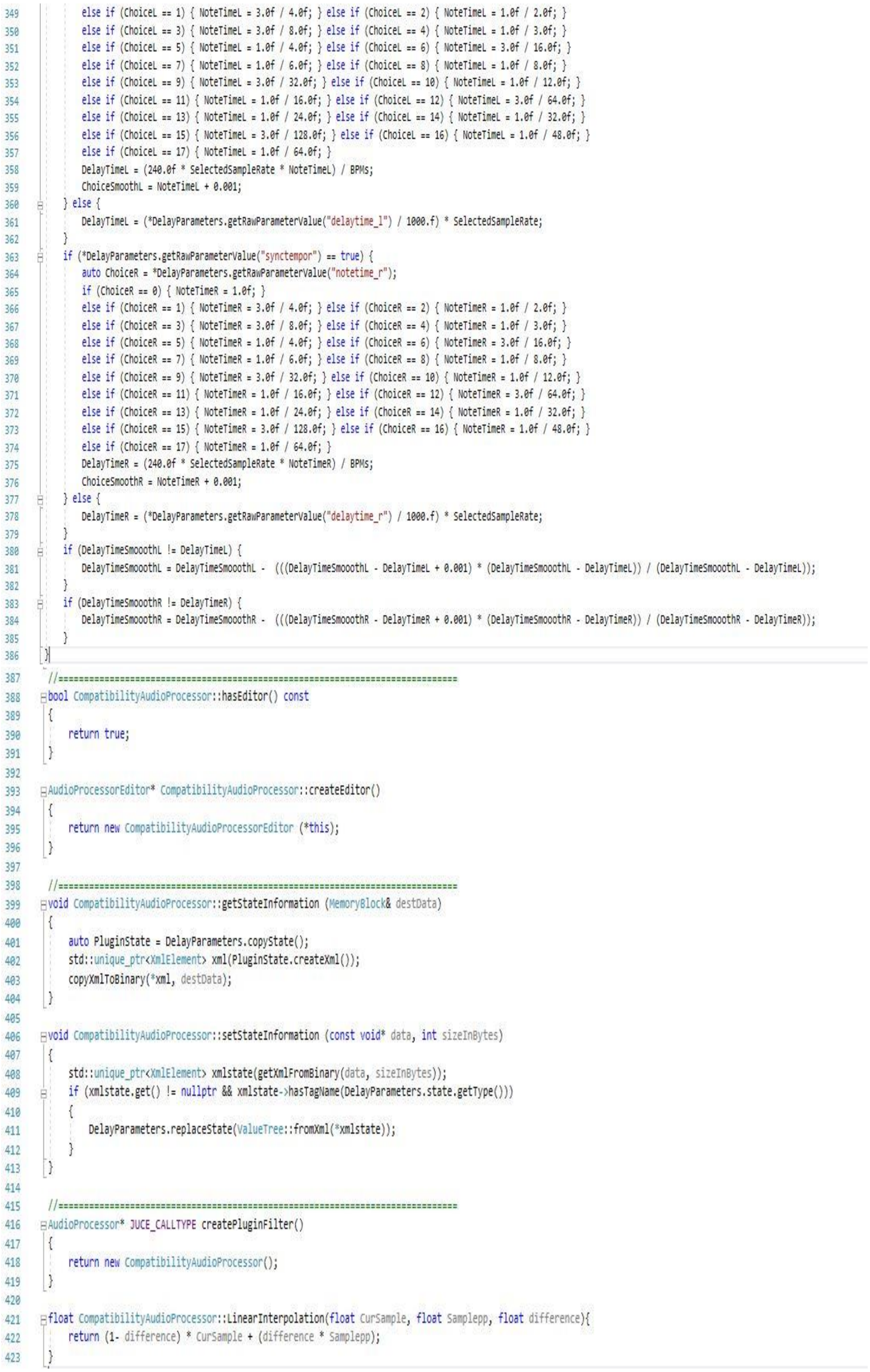
```
    #else
     return false;
    #endif
}

double CompatibilityAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int CompatibilityAudioProcessor::getNumPrograms()
{
    return 1;
}

int CompatibilityAudioProcessor::getCurrentProgram()
{
    return 0;
}

void CompatibilityAudioProcessor::setCurrentProgram (int index)
{
}

const String CompatibilityAudioProcessor::getProgramName (int index)
{
    return {};
}

void CompatibilityAudioProcessor::changeProgramName (int index, const String& newName)
{
}

//==============================================================================
void CompatibilityAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    if (SelectedSampleRate != sampleRate) {
        SelectedSampleRate = sampleRate;
```

```
    mCircularBufferLength= sampleRate* MAX_DELAY_TIME;
    if (mCircularBufferLeft== nullptr)
    {
        mCircularBufferLeft=new float [mCircularBufferLength];
    }
    zeromem(mCircularBufferLeft, mCircularBufferLength*sizeof(float));
    if (mCircularBufferRight== nullptr)
    {
        mCircularBufferRight=new float [mCircularBufferLength];
    }
    zeromem(mCircularBufferRight, mCircularBufferLength*sizeof(float));
    mCirculkarBufferWriteHead=0;
    PrevINLGain = Decibels::decibelsToGain(*DelayParameters.getRawParameterValue("inlgain"));
    PrevINLGain = Decibels::decibelsToGain(*DelayParameters.getRawParameterValue("inrgain"));
    PrevOUTLGain = Decibels::decibelsToGain(*DelayParameters.getRawParameterValue("outlgain"));
    PrevOUTRGain = Decibels::decibelsToGain(*DelayParameters.getRawParameterValue("outrgain"));
    if (BPMs < 60) { BPMs = 60; }
    SetDelayTime();
    FeedbackSmoothL = *DelayParameters.getRawParameterValue("delayfeedbackl") / 100.f;
    FeedbackSmoothR = *DelayParameters.getRawParameterValue("delayfeedbackr") / 100.f;
    MixSmoothL = *DelayParameters.getRawParameterValue("delaymixl") / 100.f;
    MixSmoothR = *DelayParameters.getRawParameterValue("delaymixr") / 100.f;
    DelayTimeSmooothL = DelayTimeL; DelayTimeSmooothR = DelayTimeR;
}

void CompatibilityAudioProcessor::releaseResources()
{
    if (mCircularBufferLeft!= nullptr)
    {
        delete [] mCircularBufferLeft;
        mCircularBufferLeft=nullptr;
    }
    if (mCircularBufferRight!= nullptr)
    {
        delete [] mCircularBufferRight;
        mCircularBufferRight=nullptr;
```

```
#ifndef JucePlugin_PreferredChannelConfigurations
bool CompatibilityAudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    ignoreUnused (layouts);
    return true;
  #else
    if (layouts.getMainOutputChannelSet() != AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != AudioChannelSet::stereo())
        return false;
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() != layouts.getMainInputChannelSet())
        return false;
   #endif
    return true;
  #endif
}
#endif

void CompatibilityAudioProcessor::updateParameters()
{
    if (MixSmoothL != (*DelayParameters.getRawParameterValue("delaymixl") / 100.f)) {
        MixSmoothL = MixSmoothL - (((MixSmoothL + 0.0001 - (*DelayParameters.getRawParameterValue("delaymixl") / 100.f)) * (MixSmoothL -
            (*DelayParameters.getRawParameterValue("delaymixl") / 100.f)) / (MixSmoothL - (*DelayParameters.getRawParameterValue("delaymixl") / 100.f))));
    }
    if (MixSmoothR != (*DelayParameters.getRawParameterValue("delaymixr") / 100.f)) {
        MixSmoothR = MixSmoothR - (((MixSmoothR + 0.0001 - (*DelayParameters.getRawParameterValue("delaymixr") / 100.f)) * (MixSmoothR -
            (*DelayParameters.getRawParameterValue("delaymixr") / 100.f))) / (MixSmoothR + 0.0001 - (*DelayParameters.getRawParameterValue("delaymixr") / 100.f)));
    }
    if (FeedbackSmoothL != (*DelayParameters.getRawParameterValue("delayfeedbackl") / 100.f)) {
        FeedbackSmoothL = FeedbackSmoothL - (((FeedbackSmoothL + 0.0001 - (*DelayParameters.getRawParameterValue("delayfeedbackl") / 100.f)) *
            (FeedbackSmoothL - (*DelayParameters.getRawParameterValue("delayfeedbackl") / 100.f))) / (FeedbackSmoothL + 0.0001 -
            (*DelayParameters.getRawParameterValue("delayfeedbackl") / 100.f)));
    }
    if (FeedbackSmoothR != (*DelayParameters.getRawParameterValue("delayfeedbackr") / 100.f)) {
        FeedbackSmoothR = FeedbackSmoothR - (((FeedbackSmoothR + 0.0001 - (*DelayParameters.getRawParameterValue("delayfeedbackr") / 100.f)) *
            (FeedbackSmoothR - (*DelayParameters.getRawParameterValue("delayfeedbackr") / 100.f))) / (FeedbackSmoothR + 0.0001 -
            (*DelayParameters.getRawParameterValue("delayfeedbackr") / 100.f)));
```



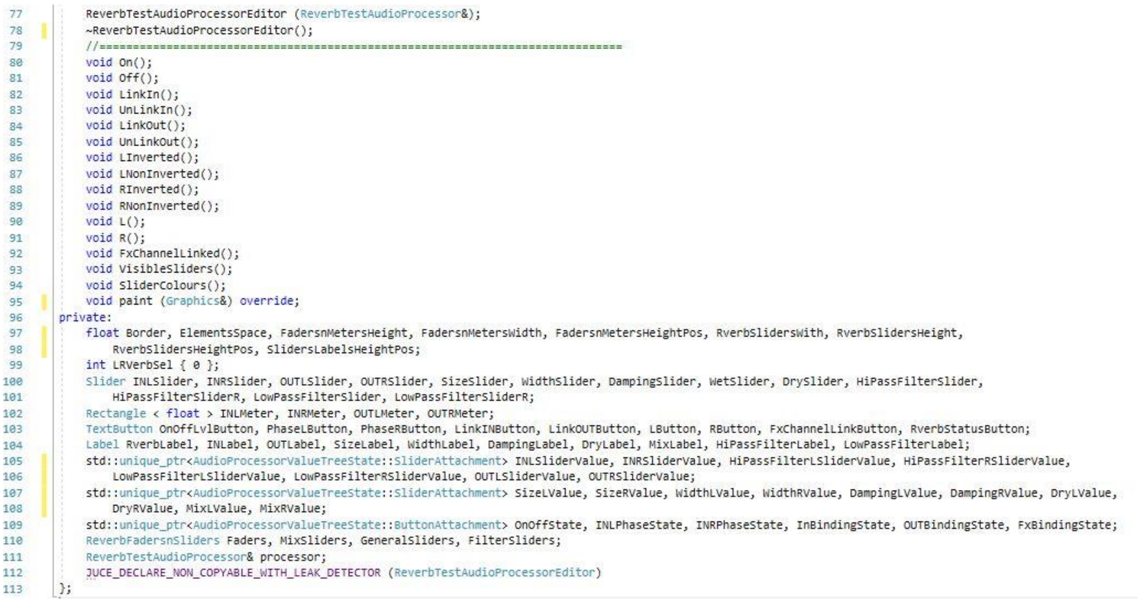
```cpp
void CompatibilityAudioProcessor::processBlock (AudioBuffer<float>& buffer, MidiBuffer& midiMessages)
{
    if (SelectedSampleRate != getSampleRate()) {
        SelectedSampleRate = getSampleRate();
    }
    ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());
    if (*DelayParameters.getRawParameterValue("delaystate") == true) {
        SelectedSampleRate = getSampleRate();
        PolarityL = *DelayParameters.getRawParameterValue("phaselmode") ? -1.0f: 1.0f;
        PolarityR = *DelayParameters.getRawParameterValue("phasermode") ? -1.0f: 1.0f;
        /*Ganancia de Entrada*/
        CurINLGain = PolarityL * Decibels::decibelsToGain(*DelayParameters.getRawParameterValue("inlgain"));
        CurINRGain = PolarityR * Decibels::decibelsToGain(*DelayParameters.getRawParameterValue("inrgain"));
        if (PrevINLGain != CurINLGain) {
            buffer.applyGainRamp(0, 0, buffer.getNumSamples(), PrevINLGain, CurINLGain);
        } else {
            buffer.applyGain(0, 0, buffer.getNumSamples(), CurINLGain);
        }
        if (PrevINRGain != CurINRGain) {
            buffer.applyGainRamp(1, 0, buffer.getNumSamples(), PrevINRGain, CurINRGain);
        } else {
            buffer.applyGain(1, 0, buffer.getNumSamples(), CurINRGain);
        }
        if (*DelayParameters.getRawParameterValue("synctempol") == true || *DelayParameters.getRawParameterValue("synctempor")) {
            playHead = this->getPlayHead();
            playHead->getCurrentPosition(currentPositionInfo);
            BPMs = currentPositionInfo.bpm;
            if (BPMs < 60) { BPMs = 60; }
        }
        updateParameters();
        for (int sample=0;sample<buffer.getNumSamples();sample++)
        {
            SetDelayTime();
            SetDelayTime();
            mCircularBufferLeft[DelayWriteHeadL]= buffer.getWritePointer(0) [sample] + mFeedbackLeft;
            mCircularBufferRight[DelayWriteHeadR]= buffer.getWritePointer(1)[sample] + mFeedbackRight;
            mDelayReadHeadL = DelayWriteHeadL - DelayTimeSmooothL;
            mDelayReadHeadR= DelayWriteHeadR - DelayTimeSmooothR;
            if (mDelayReadHeadL<0) {
                mDelayReadHeadL += mCircularBufferLength;//Tamaño lectura señal con Delay
            }
            if (mDelayReadHeadR<0) {
                mDelayReadHeadR += mCircularBufferLength;//Tamaño lectura señal con Delay
            }
            float delay_sample_left, delay_sample_right;
            if ((mDelayReadHeadL - int(mDelayReadHeadL)) == 0.f) {
                delay_sample_left = mCircularBufferLeft[(int)mDelayReadHeadL];
            } else {
                int ReadHeadL = int(mDelayReadHeadL);
                int ReadHeadLpp = ReadHeadL++;
                if (ReadHeadLpp >= mCircularBufferLength) {
                    ReadHeadLpp -= mCircularBufferLength;
                }
                float ReadHeadfloatL = mDelayReadHeadL - ReadHeadL;
                delay_sample_left = LinearInterpolation(mCircularBufferLeft[ReadHeadL], mCircularBufferLeft[ReadHeadLpp], ReadHeadfloatL);
            }
            if ((mDelayReadHeadR - int(mDelayReadHeadR)) == 0.f) {
                delay_sample_right = mCircularBufferRight[(int)mDelayReadHeadR];
            } else {
                int ReadHeadR = int(mDelayReadHeadR);
                int ReadHeadRpp = ReadHeadR++;
                if (ReadHeadRpp >= mCircularBufferLength) {
                    ReadHeadRpp -= mCircularBufferLength;
                }
                float ReadHeadfloatR = mDelayReadHeadR - ReadHeadR;
                delay_sample_right = LinearInterpolation(mCircularBufferLeft[ReadHeadR], mCircularBufferLeft[ReadHeadRpp], ReadHeadfloatR);
            }
            int ReadHeadL = int(mDelayReadHeadL);
            int ReadHeadLpp = ReadHeadL++;
            if (ReadHeadLpp >= mCircularBufferLength) {
                ReadHeadLpp -= mCircularBufferLength;
            mFeedbackLeft = delay_sample_left * FeedbackSmoothL;
            mFeedbackRight = delay_sample_right * FeedbackSmoothR;
            DelayWriteHeadR++;DelayWriteHeadL++;

            buffer.setSample(0, sample, (buffer.getSample(0, sample) * (1- MixSmoothL)) + (delay_sample_left* MixSmoothL));
            buffer.setSample(1, sample, (buffer.getSample(1, sample) * (1- MixSmoothR)) + (delay_sample_right* MixSmoothR));
            if (DelayWriteHeadL>=mCircularBufferLength) {
                DelayWriteHeadL=0;//Circular Buffer
            }
            if (DelayWriteHeadR>=mCircularBufferLength) {
                DelayWriteHeadR=0;//Circular Buffer
            }
        }
        /*Ganancia de Salida*/
        CurOutLGain = Decibels::decibelsToGain(*DelayParameters.getRawParameterValue("outlgain"));
        if (PrevOUTLGain != CurOutLGain) {
            buffer.applyGainRamp(0, 0, buffer.getNumSamples(), PrevOUTLGain, CurOutLGain);
        } else {
            buffer.applyGain(0, 0, buffer.getNumSamples(), CurOutLGain);
        }
        CurOutRGain = Decibels::decibelsToGain(*DelayParameters.getRawParameterValue("outrgain"));
        if (PrevOUTRGain != CurOutRGain) {
            buffer.applyGainRamp(1, 0, buffer.getNumSamples(), PrevOUTRGain, CurOutRGain);
        } else {
            buffer.applyGain(1, 0, buffer.getNumSamples(), CurOutRGain);
        }
        PrevINLGain = CurINLGain; PrevINRGain = CurINRGain;
        PrevOUTLGain = CurOutLGain; PrevOUTRGain = CurOutRGain;
    } else {
        processBlockBypassed(buffer, midiMessages);
    }
}

void CompatibilityAudioProcessor::SetDelayTime()
{
    if (*DelayParameters.getRawParameterValue("synctempol") == true) {
        auto ChoiceL = *DelayParameters.getRawParameterValue("notetime_l");
        if (ChoiceL == 0) { NoteTimeL = 1.0f; }
```



```cpp
        else if (ChoiceL == 1) { NoteTimeL = 3.0f / 4.0f; } else if (ChoiceL == 2) { NoteTimeL = 1.0f / 2.0f; }
        else if (ChoiceL == 3) { NoteTimeL = 3.0f / 8.0f; } else if (ChoiceL == 4) { NoteTimeL = 1.0f / 3.0f; }
        else if (ChoiceL == 5) { NoteTimeL = 1.0f / 4.0f; } else if (ChoiceL == 6) { NoteTimeL = 3.0f / 16.0f; }
        else if (ChoiceL == 7) { NoteTimeL = 1.0f / 6.0f; } else if (ChoiceL == 8) { NoteTimeL = 1.0f / 8.0f; }
        else if (ChoiceL == 9) { NoteTimeL = 3.0f / 32.0f; } else if (ChoiceL == 10) { NoteTimeL = 1.0f / 12.0f; }
        else if (ChoiceL == 11) { NoteTimeL = 1.0f / 16.0f; } else if (ChoiceL == 12) { NoteTimeL = 3.0f / 64.0f; }
        else if (ChoiceL == 13) { NoteTimeL = 1.0f / 24.0f; } else if (ChoiceL == 14) { NoteTimeL = 1.0f / 32.0f; }
        else if (ChoiceL == 15) { NoteTimeL = 3.0f / 128.0f; } else if (ChoiceL == 16) { NoteTimeL = 1.0f / 48.0f; }
        else if (ChoiceL == 17) { NoteTimeL = 1.0f / 64.0f; }
        DelayTimeL = (240.0f * SelectedSampleRate * NoteTimeL) / BPMs;
        ChoiceSmoothL = NoteTimeL + 0.001;
    } else {
        DelayTimeL = (*DelayParameters.getRawParameterValue("delaytime_l") / 1000.f) * SelectedSampleRate;
    }
    if (*DelayParameters.getRawParameterValue("synctempor") == true) {
        auto ChoiceR = *DelayParameters.getRawParameterValue("notetime_r");
        if (ChoiceR == 0) { NoteTimeR = 1.0f; }
        else if (ChoiceR == 1) { NoteTimeR = 3.0f / 4.0f; } else if (ChoiceR == 2) { NoteTimeR = 1.0f / 2.0f; }
        else if (ChoiceR == 3) { NoteTimeR = 3.0f / 8.0f; } else if (ChoiceR == 4) { NoteTimeR = 1.0f / 3.0f; }
        else if (ChoiceR == 5) { NoteTimeR = 1.0f / 4.0f; } else if (ChoiceR == 6) { NoteTimeR = 3.0f / 16.0f; }
        else if (ChoiceR == 7) { NoteTimeR = 1.0f / 6.0f; } else if (ChoiceR == 8) { NoteTimeR = 1.0f / 8.0f; }
        else if (ChoiceR == 9) { NoteTimeR = 3.0f / 32.0f; } else if (ChoiceR == 10) { NoteTimeR = 1.0f / 12.0f; }
        else if (ChoiceR == 11) { NoteTimeR = 1.0f / 16.0f; } else if (ChoiceR == 12) { NoteTimeR = 3.0f / 64.0f; }
        else if (ChoiceR == 13) { NoteTimeR = 1.0f / 24.0f; } else if (ChoiceR == 14) { NoteTimeR = 1.0f / 32.0f; }
        else if (ChoiceR == 15) { NoteTimeR = 3.0f / 128.0f; } else if (ChoiceR == 16) { NoteTimeR = 1.0f / 48.0f; }
        else if (ChoiceR == 17) { NoteTimeR = 1.0f / 64.0f; }
        DelayTimeR = (240.0f * SelectedSampleRate * NoteTimeR) / BPMs;
        ChoiceSmoothR = NoteTimeR + 0.001;
    } else {
        DelayTimeR = (*DelayParameters.getRawParameterValue("delaytime_r") / 1000.f) * SelectedSampleRate;
    }
    if (DelayTimeSmooothL != DelayTimeL) {
        DelayTimeSmooothL = DelayTimeSmooothL -  (((DelayTimeSmooothL - DelayTimeL + 0.001) * (DelayTimeSmooothL - DelayTimeL)) / (DelayTimeSmooothL - DelayTimeL));
    }
    if (DelayTimeSmooothR != DelayTimeR) {
        DelayTimeSmooothR = DelayTimeSmooothR -  (((DelayTimeSmooothR - DelayTimeR + 0.001) * (DelayTimeSmooothR - DelayTimeR)) / (DelayTimeSmooothR - DelayTimeR));
    }
}
//==============================================================================
bool CompatibilityAudioProcessor::hasEditor() const
{
    return true;
}

AudioProcessorEditor* CompatibilityAudioProcessor::createEditor()
{
    return new CompatibilityAudioProcessorEditor (*this);
}

//==============================================================================
void CompatibilityAudioProcessor::getStateInformation (MemoryBlock& destData)
{
    auto PluginState = DelayParameters.copyState();
    std::unique_ptr<XmlElement> xml(PluginState.createXml());
    copyXmlToBinary(*xml, destData);
}

void CompatibilityAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    std::unique_ptr<XmlElement> xmlstate(getXmlFromBinary(data, sizeInBytes));
    if (xmlstate.get() != nullptr && xmlstate->hasTagName(DelayParameters.state.getType()))
    {
        DelayParameters.replaceState(ValueTree::fromXml(*xmlstate));
    }
}

//==============================================================================
AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new CompatibilityAudioProcessor();
}

float CompatibilityAudioProcessor::LinearInterpolation(float CurSample, float Samplepp, float difference){
    return (1- difference) * CurSample + (difference * Samplepp);
}
```

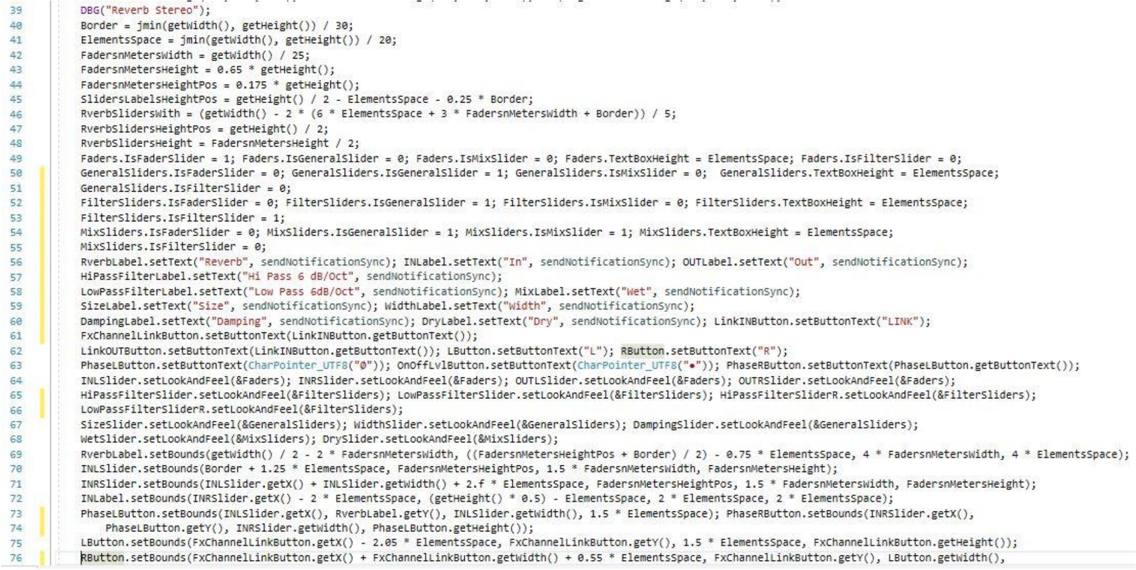
Anexo 4. Algoritmo del módulo de reverberación

```cpp
/*
    ==============================================================================
    Autor: José Aguilar
    Este proyecto está configurado para funcionar con los entornos de desarrollo
    (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
    ==============================================================================
*/


#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"

//==============================================================================
/**
*/

class ReverbFadersnSliders : public LookAndFeel_V4
{
public:
    int IsFaderSlider, FaderStatus, IsGeneralSlider, IsMixSlider, IsFilterSlider, IsHP; float TextBoxHeight;
    void drawLinearSlider(Graphics& g, int x, int y, int width, int height,
        float sliderPos,
        float minSliderPos,
        float maxSliderPos,
        const Slider::SliderStyle style, Slider& slider) override
    {
        if (IsFaderSlider == 1) {
            slider.setSliderStyle(Slider::SliderStyle::LinearBarVertical);
            slider.setTextBoxStyle(Slider::TextBoxBelow, false, width, TextBoxHeight);
            Path Fader;
            if (FaderStatus == 1) {
                g.setColour(Colours::goldenrod);
                slider.setColour(Slider::textBoxTextColourId, Colours::dimgrey);
            }
            else {
                g.setColour(Colours::darkgrey);
                slider.setColour(Slider::textBoxTextColourId, Colours::grey);
            }
            Fader.addRectangle(0, maxSliderPos - minSliderPos, width, sliderPos - (height));
            g.fillPath(Fader, AffineTransform::verticalFlip(sliderPos));
            slider.setSkewFactor(3);
            slider.setNumDecimalPlacesToDisplay(2);
            slider.setTextValueSuffix("dB");
            IsHP = 0;
        } else if (IsFaderSlider == 0) {
            x = slider.getX();
            y = slider.getY();
            width = slider.getWidth();
            height = slider.getHeight();
            sliderPos = slider.getValue();
            minSliderPos = slider.getMinimum();
            maxSliderPos = slider.getMaximum();
            slider.setSliderStyle(Slider::SliderStyle::RotaryHorizontalVerticalDrag);
            slider.setTextBoxStyle(Slider::TextBoxBelow, false, 0.75 * width, TextBoxHeight);
            slider.setNumDecimalPlacesToDisplay(2);
            if (IsGeneralSlider == 1 && (IsMixSlider == 1 || IsFilterSlider == 1)) {
                if (IsMixSlider == 1 || IsFilterSlider == 1) {
                    if (IsMixSlider == 1 && IsFilterSlider == 0) {
                        slider.setTextValueSuffix(" %");
                    }
                    else if (IsFilterSlider == 1 && IsMixSlider == 0) {
                        slider.setTextValueSuffix(" Hz");
                        slider.setSkewFactor(0.25);
                    }
                }
            }
        }
        slider.setPopupDisplayEnabled(true, true, slider.getParentComponent());
    }
};

class ReverbTestAudioProcessorEditor  : public AudioProcessorEditor
{
public:
```



```
    ReverbTestAudioProcessorEditor (ReverbTestAudioProcessor&);
    ~ReverbTestAudioProcessorEditor();
    //==============================================================================
    void On();
    void Off();
    void LinkIn();
    void UnLinkIn();
    void LinkOut();
    void UnLinkOut();
    void LInverted();
    void LNonInverted();
    void RInverted();
    void RNonInverted();
    void L();
    void R();
    void FxChannelLinked();
    void VisibleSliders();
    void SliderColours();
    void paint (Graphics&) override;
private:
    float Border, ElementsSpace, FadersnMetersHeight, FadersnMetersWidth, FadersnMetersHeightPos, RverbSlidersWith, RverbSlidersHeight,
        RverbSlidersHeightPos, SlidersLabelsHeightPos;
    int LRVerbSel { 0 };
    Slider INLSlider, INRSlider, OUTLSlider, OUTRSlider, SizeSlider, WidthSlider, DampingSlider, WetSlider, DrySlider, HiPassFilterSlider,
        HiPassFilterSliderR, LowPassFilterSlider, LowPassFilterSliderR;
    Rectangle < float > INLMeter, INRMeter, OUTLMeter, OUTRMeter;
    TextButton OnOffLvlButton, PhaseLButton, PhaseRButton, LinkINButton, LinkOUTButton, LButton, RButton, FxChannelLinkButton, RverbStatusButton;
    Label RverbLabel, INLabel, OUTLabel, SizeLabel, WidthLabel, DampingLabel, DryLabel, MixLabel, HiPassFilterLabel, LowPassFilterLabel;
    std::unique_ptr<AudioProcessorValueTreeState::SliderAttachment> INLSliderValue, INRSliderValue, HiPassFilterLSliderValue, HiPassFilterRSliderValue,
        LowPassFilterLSliderValue, LowPassFilterRSliderValue, OUTLSliderValue, OUTRSliderValue;
    std::unique_ptr<AudioProcessorValueTreeState::SliderAttachment> SizeLValue, SizeRValue, WidthLValue, WidthRValue, DampingLValue, DampingRValue, DryLValue,
        DryRValue, MixLValue, MixRValue;
    std::unique_ptr<AudioProcessorValueTreeState::ButtonAttachment> OnOffState, INLPhaseState, INRPhaseState, InBindingState, OUTBindingState, FxBindingState;
    ReverbFadersnSliders Faders, MixSliders, GeneralSliders, FilterSliders;
    ReverbTestAudioProcessor& processor;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (ReverbTestAudioProcessorEditor)
};
```

```
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/


#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
ReverbTestAudioProcessorEditor::ReverbTestAudioProcessorEditor (ReverbTestAudioProcessor& p)
    : AudioProcessorEditor (&p), processor (p)
{
    if (getParentWidth() <= 1400 && getParentHeight() <= 800) { setSize(getParentWidth() / 1.25, 0.45 * getParentHeight()); }
    else { setSize(getParentWidth() / 1.7, 0.35 * getParentHeight()); }
    OnOffState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.ReverbParams, "reverbstate", OnOffLvlButton);
    INLPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.ReverbParams, "phaselmode", PhaseLButton);
    INRPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.ReverbParams, "phasermode", PhaseRButton);
    InBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.ReverbParams, "inbinding", LinkINButton);
    OUTBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.ReverbParams, "outbinding", LinkOUTButton);
    FxBindingState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.ReverbParams, "fxverbbinding", FxChannelLinkButton);
    INLPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.ReverbParams, "phaselmode", PhaseLButton);
    INRPhaseState = std::make_unique<AudioProcessorValueTreeState::ButtonAttachment>(processor.ReverbParams, "phasermode", PhaseRButton);
    INLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "inlgain", INLSlider);
    INRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "inrgain", INRSlider);
    OUTLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "outlgain", OUTLSlider);
    OUTRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "outrgain", OUTRSlider);
    HiPassFilterLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "hipassverbfilterl", HiPassFilterSlider);
    LowPassFilterLSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "lowpassverbfilterl", LowPassFilterSlider);
    HiPassFilterRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "hipassverbfilterr", HiPassFilterSliderR);
    LowPassFilterRSliderValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "lowpassverbfilterr", LowPassFilterSliderR);
    INLSlider.setRange(-100.0f, 12.0f, 0.01); INRSlider.setRange(-100.0f, 12.0f, 0.01); OUTLSlider.setRange(-100.f, 12.f, 0.01); OUTRSlider.setRange(-100.f, 12.f, 0.01);
    HiPassFilterSlider.setRange(20.f, 20000.f, 0.01); LowPassFilterSlider.setRange(20, 20000, 0.01); WetSlider.setRange(0.f, 100.f, 0.01);
    DrySlider.setRange(0.f, 100.f, 0.01);
    SizeSlider.setRange(0.f, 1.f,0.01); WidthSlider.setRange(0.f, 1.f, 0.01); DampingSlider.setRange(0.f, 1.f, 0.01);
```

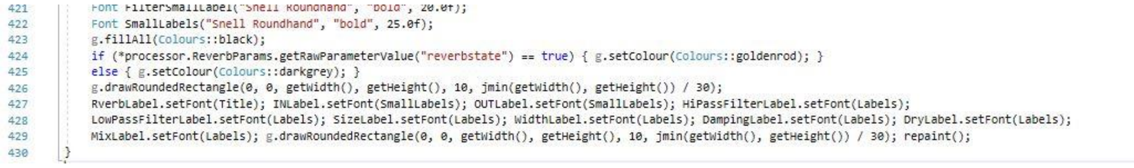


```
    DBG("Reverb Stereo");
    Border = jmin(getWidth(), getHeight()) / 30;
    ElementsSpace = jmin(getWidth(), getHeight()) / 20;
    FadersnMetersWidth = getWidth() / 25;
    FadersnMetersHeight = 0.65 * getHeight();
    FadersnMetersHeightPos = 0.175 * getHeight();
    SlidersLabelsHeightPos = getHeight() / 2 - ElementsSpace - 0.25 * Border;
    RverbSlidersWith = (getWidth() - 2 * (6 * ElementsSpace + 3 * FadersnMetersWidth + Border)) / 5;
    RverbSlidersHeightPos = getHeight() / 2;
    RverbSlidersHeight = FadersnMetersHeight / 2;
    Faders.IsFaderSlider = 1; Faders.IsGeneralSlider = 0; Faders.IsMixSlider = 0; Faders.TextBoxHeight = ElementsSpace; Faders.IsFilterSlider = 0;
    GeneralSliders.IsFaderSlider = 0; GeneralSliders.IsGeneralSlider = 1; GeneralSliders.IsMixSlider = 0;  GeneralSliders.TextBoxHeight = ElementsSpace;
    GeneralSliders.IsFilterSlider = 0;
    FilterSliders.IsFaderSlider = 0; FilterSliders.IsGeneralSlider = 1; FilterSliders.IsMixSlider = 0; FilterSliders.TextBoxHeight = ElementsSpace;
    FilterSliders.IsFilterSlider = 1;
    MixSliders.IsFaderSlider = 0; MixSliders.IsGeneralSlider = 1; MixSliders.IsMixSlider = 1; MixSliders.TextBoxHeight = ElementsSpace;
    MixSliders.IsFilterSlider = 0;
    RverbLabel.setText("Reverb", sendNotificationSync); INLabel.setText("In", sendNotificationSync); OUTLabel.setText("Out", sendNotificationSync);
    HiPassFilterLabel.setText("Hi Pass 6 dB/Oct", sendNotificationSync);
    LowPassFilterLabel.setText("Low Pass 6dB/Oct", sendNotificationSync); MixLabel.setText("Wet", sendNotificationSync);
    SizeLabel.setText("Size", sendNotificationSync); WidthLabel.setText("Width", sendNotificationSync);
    DampingLabel.setText("Damping", sendNotificationSync); DryLabel.setText("Dry", sendNotificationSync); LinkINButton.setButtonText("LINK");
    FxChannelLinkButton.setButtonText(LinkINButton.getButtonText());
    LinkOUTButton.setButtonText(LinkINButton.getButtonText()); LButton.setButtonText("L"); RButton.setButtonText("R");
    PhaseLButton.setButtonText(CharPointer_UTF8("Ø")); OnOffLvlButton.setButtonText(CharPointer_UTF8("•")); PhaseRButton.setButtonText(PhaseLButton.getButtonText());
    INLSlider.setLookAndFeel(&Faders); INRSlider.setLookAndFeel(&Faders); OUTLSlider.setLookAndFeel(&Faders); OUTRSlider.setLookAndFeel(&Faders);
    HiPassFilterSlider.setLookAndFeel(&FilterSliders); LowPassFilterSlider.setLookAndFeel(&FilterSliders); HiPassFilterSliderR.setLookAndFeel(&FilterSliders);
    LowPassFilterSliderR.setLookAndFeel(&FilterSliders);
    SizeSlider.setLookAndFeel(&GeneralSliders); WidthSlider.setLookAndFeel(&GeneralSliders); DampingSlider.setLookAndFeel(&GeneralSliders);
    WetSlider.setLookAndFeel(&MixSliders); DrySlider.setLookAndFeel(&MixSliders);
    RverbLabel.setBounds(getWidth() / 2 - 2 * FadersnMetersWidth, ((FadersnMetersHeightPos + Border) / 2) - 0.75 * ElementsSpace, 4 * FadersnMetersWidth, 4 * ElementsSpace);
    INLSlider.setBounds(Border + 1.25 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    INRSlider.setBounds(INLSlider.getX() + INLSlider.getWidth() + 2.f * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    INLabel.setBounds(INRSlider.getX() - 2 * ElementsSpace, (getHeight() * 0.5) - ElementsSpace, 2 * ElementsSpace, 2 * ElementsSpace);
    PhaseLButton.setBounds(INLSlider.getX(), RverbLabel.getY(), INLSlider.getWidth(), 1.5 * ElementsSpace); PhaseRButton.setBounds(INRSlider.getX(),
        PhaseLButton.getY(), INRSlider.getWidth(), PhaseLButton.getHeight());
    LButton.setBounds(FxChannelLinkButton.getX() - 2.05 * ElementsSpace, FxChannelLinkButton.getY(), 1.5 * ElementsSpace, FxChannelLinkButton.getHeight());
    RButton.setBounds(FxChannelLinkButton.getX() + FxChannelLinkButton.getWidth() + 0.55 * ElementsSpace, FxChannelLinkButton.getY(), LButton.getWidth(),
```

```
        FxChannelLinkButton.getHeight());
    LinkINButton.setBounds(INRSlider.getX() - 2 * ElementsSpace - 0.75 * FadersnMetersWidth, getHeight() - FadersnMetersHeightPos + 0.875 * ElementsSpace,
        1.5 * FadersnMetersWidth + 2 * ElementsSpace, 1.75 * ElementsSpace);
    FxChannelLinkButton.setBounds((0.5 * getWidth()) - (LinkINButton.getWidth() * 0.5 ), LinkINButton.getY(), LinkINButton.getWidth(), LinkINButton.getHeight());
    LButton.setBounds(FxChannelLinkButton.getX() - 0.5 * ElementsSpace - 0.25 * RverbSlidersWith, FxChannelLinkButton.getY(), 0.25 * RverbSlidersWith,
        FxChannelLinkButton.getHeight());
    RButton.setBounds(FxChannelLinkButton.getX() + FxChannelLinkButton.getWidth() + 0.5 * ElementsSpace, FxChannelLinkButton.getY(), 0.25 * RverbSlidersWith,
        FxChannelLinkButton.getHeight());
    HiPassFilterLabel.setBounds(Border + 3 * FadersnMetersWidth + 3.5 * ElementsSpace + 0.5 * RverbSlidersWith, RverbLabel.getY() - 0.5 * Border,
        RverbSlidersWith + 2.75 * ElementsSpace, 1.75 * ElementsSpace);
    HiPassFilterSlider.setBounds(HiPassFilterLabel.getX()  + ElementsSpace, RverbLabel.getY() + ElementsSpace, HiPassFilterLabel.getWidth() - 2 *
        ElementsSpace, RverbSlidersHeight);
    HiPassFilterSliderR.setBounds(HiPassFilterSlider.getBounds());
    LowPassFilterLabel.setBounds(getWidth() - 3 * FadersnMetersWidth - 6.25 * ElementsSpace - Border - 1.5 * RverbSlidersWith, HiPassFilterLabel.getY(),
        HiPassFilterLabel.getWidth(), 1.75 * ElementsSpace);
    LowPassFilterSlider.setBounds(LowPassFilterLabel.getX() + ElementsSpace, LowPassFilterLabel.getY() + ElementsSpace,
        LowPassFilterLabel.getWidth() - 2 * ElementsSpace, RverbSlidersHeight);
    LowPassFilterSliderR.setBounds(LowPassFilterSlider.getBounds());
    SizeLabel.setBounds(Border + 3 * FadersnMetersWidth + 4.5 * ElementsSpace, SlidersLabelsHeightPos, RverbSlidersWith, HiPassFilterLabel.getHeight());
    SizeSlider.setBounds(SizeLabel.getX(), RverbSlidersHeightPos, RverbSlidersWith, RverbSlidersHeight);
    WidthLabel.setBounds(SizeLabel.getX() + SizeLabel.getWidth() + 0.75 * ElementsSpace, SizeLabel.getY(), RverbSlidersWith, SizeLabel.getHeight());
    WidthSlider.setBounds(WidthLabel.getX(), RverbSlidersHeightPos, RverbSlidersWith, RverbSlidersHeight);
    DampingLabel.setBounds(WidthLabel.getX() + WidthLabel.getWidth() + 0.75 * ElementsSpace, WidthLabel.getY(), RverbSlidersWith, WidthLabel.getHeight());
    DampingSlider.setBounds(DampingLabel.getX(), RverbSlidersHeightPos, RverbSlidersWith, RverbSlidersHeight);
    DryLabel.setBounds(DampingLabel.getX() + DampingLabel.getWidth() + 0.75 * ElementsSpace, SlidersLabelsHeightPos, RverbSlidersWith, 1.75 * ElementsSpace);
    DrySlider.setBounds(DryLabel.getX(), RverbSlidersHeightPos, RverbSlidersWith, RverbSlidersHeight);
    MixLabel.setBounds(DryLabel.getX() + DryLabel.getWidth() + 0.75 * ElementsSpace, DryLabel.getY(), RverbSlidersWith, DryLabel.getHeight());
    WetSlider.setBounds(MixLabel.getX(), RverbSlidersHeightPos, RverbSlidersWith, RverbSlidersHeight);
    OUTLSlider.setBounds(getWidth() - Border - 3.25 * ElementsSpace - 3 * FadersnMetersWidth, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    OUTRSlider.setBounds(OUTLSlider.getX() + OUTLSlider.getWidth() + 2 * ElementsSpace, FadersnMetersHeightPos, 1.5 * FadersnMetersWidth, FadersnMetersHeight);
    OUTLabel.setBounds(OUTRSlider.getX() - 2 * ElementsSpace, INLabel.getY(), 2 * ElementsSpace, INLabel.getHeight());
    LinkOUTButton.setBounds(OUTLabel.getX() - 0.75 * FadersnMetersWidth, LinkINButton.getY(), LinkINButton.getWidth(), LinkINButton.getHeight());
    OnOffLvlButton.setBounds(OUTRSlider.getX(), PhaseLButton.getY(), OUTRSlider.getWidth(), PhaseLButton.getHeight());
    INLabel.setJustificationType(Justification::Flags::centred); OUTLabel.setJustificationType(Justification::Flags::centred);
    RverbLabel.setJustificationType(Justification::Flags::centred);
    HiPassFilterLabel.setJustificationType(Justification::Flags::centred); LowPassFilterLabel.setJustificationType(Justification::Flags::centred);
    SizeLabel.setJustificationType(Justification::Flags::centred);
    WidthLabel.setJustificationType(Justification::Flags::centred); DampingLabel.setJustificationType(Justification::Flags::centred);
    DryLabel.setJustificationType(Justification::Flags::centred);
    MixLabel.setJustificationType(Justification::Flags::centred);
    PhaseLButton.setColour(TextButton::textColourOnId, Colours::black); PhaseRButton.setColour(TextButton::textColourOnId, Colours::black);
    LinkINButton.setColour(TextButton::textColourOnId, Colours::black); LButton.setColour(TextButton::textColourOnId, Colours::black);
    RButton.setColour(TextButton::textColourOnId, Colours::black); FxChannelLinkButton.setColour(TextButton::textColourOnId, Colours::black);
    LinkOUTButton.setColour(TextButton::textColourOnId, Colours::black); OnOffLvlButton.setColour(TextButton::textColourOnId, Colours::black);
    PhaseLButton.setColour(TextButton::buttonColourId, Colours::black); PhaseRButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkINButton.setColour(TextButton::buttonColourId, Colours::black); LButton.setColour(TextButton::buttonColourId, Colours::black);
    RButton.setColour(TextButton::buttonColourId, Colours::black); FxChannelLinkButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkOUTButton.setColour(TextButton::buttonColourId, Colours::black); OnOffLvlButton.setColour(TextButton::buttonColourId, Colours::black);
    addAndMakeVisible(&RverbLabel); addAndMakeVisible(&INLabel); addAndMakeVisible(&OUTLabel); addAndMakeVisible(&SizeLabel); addAndMakeVisible(WidthLabel);
    addAndMakeVisible(&DampingLabel); addAndMakeVisible(&DryLabel);
    addAndMakeVisible(&MixLabel); addAndMakeVisible(&OnOffLvlButton); addAndMakeVisible(&FxChannelLinkButton); addAndMakeVisible(&LButton); addAndMakeVisible(&RButton);
    addAndMakeVisible(&LinkINButton); addAndMakeVisible(&HiPassFilterLabel); addAndMakeVisible(&LowPassFilterLabel);
    addAndMakeVisible(&LinkOUTButton); addAndMakeVisible(&INLSlider); addAndMakeVisible(&INRSlider); addAndMakeVisible(&OUTLSlider); addAndMakeVisible(&OUTRSlider);
    addAndMakeVisible(&FxChannelLinkButton); addAndMakeVisible(&LButton); addAndMakeVisible(&RButton); addAndMakeVisible(&PhaseLButton); addAndMakeVisible(&PhaseRButton);
    addAndMakeVisible(&HiPassFilterSlider); addAndMakeVisible(&HiPassFilterSliderR); addAndMakeVisible(&LowPassFilterSliderR);
    addAndMakeVisible(&LowPassFilterSlider); addAndMakeVisible(&SizeSlider); addAndMakeVisible(&WidthSlider); addAndMakeVisible(&DampingSlider);
    addAndMakeVisible(&DrySlider); addAndMakeVisible(&WetSlider);
    if (*processor.ReverbParams.getRawParameterValue("reverbstate") == true) { On(); } else { Off(); }
    if (*processor.ReverbParams.getRawParameterValue("phaselmode") == false) { LNonInverted(); } else { LInverted(); }
    if (*processor.ReverbParams.getRawParameterValue("phasermode") == false) { RNonInverted(); } else { RInverted(); }
    if (*processor.ReverbParams.getRawParameterValue("inbinding") == true) { LinkIn(); } else { UnLinkIn(); }
    if (*processor.ReverbParams.getRawParameterValue("outbinding") == true) { LinkOut(); } else { UnLinkOut(); }
    if (*processor.ReverbParams.getRawParameterValue("fxverbbinding") == true) { FxChannelLinked(); }
    else { if (LRVerbSel == 0) { L(); } else if (LRVerbSel == 1) { R(); } }
}

ReverbTestAudioProcessorEditor::~ReverbTestAudioProcessorEditor()
{
    INLSlider.setLookAndFeel(nullptr); INRSlider.setLookAndFeel(nullptr); OUTLSlider.setLookAndFeel(nullptr); OUTRSlider.setLookAndFeel(nullptr);
    HiPassFilterSlider.setLookAndFeel(nullptr); LowPassFilterSlider.setLookAndFeel(nullptr); HiPassFilterSliderR.setLookAndFeel(nullptr);
    LowPassFilterSliderR.setLookAndFeel(nullptr); SizeSlider.setLookAndFeel(nullptr); WidthSlider.setLookAndFeel(nullptr);
    DampingSlider.setLookAndFeel(nullptr); DrySlider.setLookAndFeel(nullptr); WetSlider.setLookAndFeel(nullptr);
}

//==============================================================================

void ReverbTestAudioProcessorEditor::On()
{
    *processor.ReverbParams.getRawParameterValue("reverbstate") = true;
    OnOffLvlButton.setToggleState(*processor.ReverbParams.getRawParameterValue("reverbstate"), sendNotificationSync);
    DBG("On");
    OnOffLvlButton.setColour(TextButton::buttonOnColourId, Colours::goldenrod);
    Faders.FaderStatus = 1; SliderColours();
    INLabel.setColour(Label::textColourId, Colours::goldenrod); OUTLabel.setColour(Label::textColourId, Colours::goldenrod);
    RverbLabel.setColour(Label::textColourId, Colours::goldenrod); HiPassFilterLabel.setColour(Label::textColourId, Colours::goldenrod);
    LowPassFilterLabel.setColour(Label::textColourId, Colours::goldenrod); SizeLabel.setColour(Label::textColourId, Colours::goldenrod);
    WidthLabel.setColour(Label::textColourId, Colours::goldenrod); DampingLabel.setColour(Label::textColourId, Colours::goldenrod);
    DryLabel.setColour(Label::textColourId, Colours::goldenrod); MixLabel.setColour(Label::textColourId, Colours::goldenrod);
    LButton.setColour(TextButton::buttonOnColourId, Colours::goldenrod); LButton.setColour(TextButton::textColourOffId, Colours::white);
    RButton.setColour(TextButton::buttonOnColourId, Colours::goldenrod); RButton.setColour(TextButton::textColourOffId, Colours::white);
    FxChannelLinkButton.setColour(TextButton::buttonOnColourId, Colours::goldenrod); FxChannelLinkButton.setColour(TextButton::textColourOffId, Colours::white);
    LinkINButton.setColour(TextButton::buttonOnColourId, Colours::goldenrod); LinkINButton.setColour(TextButton::textColourOffId, Colours::white);
    LinkOUTButton.setColour(TextButton::buttonOnColourId, Colours::goldenrod); LinkOUTButton.setColour(TextButton::textColourOffId, Colours::white);
    PhaseLButton.setColour(TextButton::buttonOnColourId, Colours::goldenrod); PhaseLButton.setColour(TextButton::textColourOffId, Colours::white);
    PhaseRButton.setColour(TextButton::buttonOnColourId, Colours::goldenrod); PhaseRButton.setColour(TextButton::textColourOffId, Colours::white);
    OnOffLvlButton.setColour(TextButton::buttonOnColourId, Colours::goldenrod);
    OnOffLvlButton.onClick = [this]() { Off(); };
}

void ReverbTestAudioProcessorEditor::Off()
{
    *processor.ReverbParams.getRawParameterValue("reverbstate") = false;
    OnOffLvlButton.setToggleState(*processor.ReverbParams.getRawParameterValue("reverbstate"), sendNotificationSync);
    DBG("Off");
    Faders.FaderStatus = 0; SliderColours();
    INLabel.setColour(Label::textColourId, Colours::darkgrey); OUTLabel.setColour(Label::textColourId, Colours::darkgrey);
    RverbLabel.setColour(Label::textColourId, Colours::darkgrey); HiPassFilterLabel.setColour(Label::textColourId, Colours::darkgrey);
    LowPassFilterLabel.setColour(Label::textColourId, Colours::darkgrey); SizeLabel.setColour(Label::textColourId, Colours::darkgrey);
    WidthLabel.setColour(Label::textColourId, Colours::darkgrey); DampingLabel.setColour(Label::textColourId, Colours::darkgrey);
    DryLabel.setColour(Label::textColourId, Colours::darkgrey); MixLabel.setColour(Label::textColourId, Colours::darkgrey);
    LinkOUTButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey); LinkINButton.setColour(TextButton::textColourOffId, Colours::darkgrey);
    LButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey); LButton.setColour(TextButton::textColourOffId, Colours::darkgrey);
    RButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey); RButton.setColour(TextButton::textColourOffId, Colours::darkgrey);
    FxChannelLinkButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey); FxChannelLinkButton.setColour(TextButton::textColourOffId, Colours::darkgrey);
```

```cpp
    LinkINButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey); LinkINButton.setColour(TextButton::textColourOffId, Colours::darkgrey);
    PhaseLButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey); PhaseLButton.setColour(TextButton::textColourOffId, Colours::darkgrey);
    PhaseRButton.setColour(TextButton::buttonOnColourId, Colours::darkgrey); PhaseRButton.setColour(TextButton::textColourOffId, Colours::darkgrey);
    OnOffLvlButton.setColour(TextButton::textColourOffId, Colours::darkgrey);
    OnOffLvlButton.onClick = [this]() { On(); };
}

void ReverbTestAudioProcessorEditor::LinkIn()
{
    *processor.ReverbParams.getRawParameterValue("inbinding") = true;
    LinkINButton.setToggleState(*processor.ReverbParams.getRawParameterValue("inbinding"), sendNotificationSync);
    DBG("Link In");
    INLSlider.onDragStart = [this] () { INRSlider.setValue(INLSlider.getValue()); };
    INLSlider.onValueChange = [this] () { INRSlider.setValue(INLSlider.getValue()); };
    INLSlider.onDragEnd = [this] () { INRSlider.setValue(INLSlider.getValue()); };
    INRSlider.onDragStart = [this] () { INLSlider.setValue(INRSlider.getValue()); };
    INRSlider.onValueChange = [this] () { INLSlider.setValue(INRSlider.getValue()); };
    INRSlider.onDragEnd = [this] () { INLSlider.setValue(INRSlider.getValue()); };
    if (*processor.ReverbParams.getRawParameterValue("phaselmode") == true && *processor.ReverbParams.getRawParameterValue("phasermode") == false) {
        RInverted();
    } else if (*processor.ReverbParams.getRawParameterValue("phasermode") == true && *processor.ReverbParams.getRawParameterValue("phaselmode") == false) {
        LInverted();
    }
    LinkINButton.onClick = [this]() { UnLinkIn(); };
}

void ReverbTestAudioProcessorEditor::LInverted()
{
    *processor.ReverbParams.getRawParameterValue("phaselmode") = true;
    PhaseLButton.setToggleState(*processor.ReverbParams.getRawParameterValue("phaselmode"), sendNotificationSync);
    PhaseLButton.onClick = [this] () { LNonInverted(); if (*processor.ReverbParams.getRawParameterValue("inbinding") == true) { RNonInverted(); } };
}

void ReverbTestAudioProcessorEditor::LNonInverted()
{
    *processor.ReverbParams.getRawParameterValue("phaselmode") = false;
    PhaseLButton.setToggleState(*processor.ReverbParams.getRawParameterValue("phaselmode"), sendNotificationSync);
    PhaseLButton.onClick = [this] () { LInverted(); if (*processor.ReverbParams.getRawParameterValue("inbinding") == true) { RInverted(); } };
void ReverbTestAudioProcessorEditor::RInverted()
{
    *processor.ReverbParams.getRawParameterValue("phasermode") = true;
    PhaseRButton.setToggleState(*processor.ReverbParams.getRawParameterValue("phasermode"), sendNotificationSync);
    PhaseRButton.onClick = [this] () { RNonInverted(); if (*processor.ReverbParams.getRawParameterValue("inbinding") == true) { LNonInverted(); } };
}

void ReverbTestAudioProcessorEditor::RNonInverted()
{
    *processor.ReverbParams.getRawParameterValue("phasermode") = false;
    PhaseRButton.setToggleState(*processor.ReverbParams.getRawParameterValue("phasermode"), sendNotificationSync);
    PhaseRButton.onClick = [this] () { RInverted(); if (*processor.ReverbParams.getRawParameterValue("inbinding") == true) { LInverted(); } };
}

void ReverbTestAudioProcessorEditor::UnLinkIn()
{
    *processor.ReverbParams.getRawParameterValue("inbinding") = false;
    LinkINButton.setToggleState(*processor.ReverbParams.getRawParameterValue("inbinding"), sendNotificationSync);
    DBG("UnLink In");
    INLSlider.onDragStart = [this] () {}; INLSlider.onValueChange = [this] () {}; INLSlider.onDragEnd = [this] () {};
    INRSlider.onDragStart = [this] () {}; INRSlider.onValueChange = [this] () {}; INRSlider.onDragEnd = [this] () {};
    LinkINButton.onClick = [this]() { LinkIn(); };
}

void ReverbTestAudioProcessorEditor::FxChannelLinked()
{
    *processor.ReverbParams.getRawParameterValue("fxverbbinding") = true;
    FxChannelLinkButton.setToggleState(*processor.ReverbParams.getRawParameterValue("fxverbbinding"), sendNotificationSync);
    LButton.setToggleState(false, dontSendNotification); RButton.setToggleState(false, dontSendNotification);
    LButton.setEnabled(false); RButton.setEnabled(false);
    SizeLValue.reset(); SizeRValue.reset(); WidthLValue.reset(); WidthRValue.reset(); DampingLValue.reset(); DampingRValue.reset(); DryLValue.reset();
    DryRValue.reset(); MixLValue.reset(); MixRValue.reset();
    SizeLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "sizel", SizeSlider);
    WidthLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "widthl", WidthSlider);
    DampingLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "dampingl", DampingSlider);
    DryLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "reverbdryl", DrySlider);
    MixLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "reverbmixl", WetSlider);
    SizeRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "sizer", SizeSlider);
    WidthRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "widthr", WidthSlider);
    DampingRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "dampingr", DampingSlider);
    DryRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "reverbdryr", DrySlider);
    MixRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "reverbmixr", WetSlider);
    DBG("Fx Channel Linked");
    VisibleSliders(); SliderColours();
    if (LRVerbSel == 0) {
        HiPassFilterSliderR.setValue(HiPassFilterSlider.getValue(), sendNotificationSync);
        LowPassFilterSliderR.setValue(LowPassFilterSlider.getValue(), sendNotificationSync);
        HiPassFilterSlider.onDragStart = [this] () { HiPassFilterSliderR.setValue(HiPassFilterSlider.getValue(), sendNotificationSync); };
        HiPassFilterSlider.onValueChange = [this] () { HiPassFilterSliderR.setValue(HiPassFilterSlider.getValue(), sendNotificationSync); };
        HiPassFilterSlider.onDragEnd = [this] () { HiPassFilterSliderR.setValue(HiPassFilterSlider.getValue(), sendNotificationSync); };
        LowPassFilterSlider.onDragStart = [this] () { LowPassFilterSliderR.setValue(LowPassFilterSlider.getValue(), sendNotificationSync); };
        LowPassFilterSlider.onValueChange = [this] () { LowPassFilterSliderR.setValue(LowPassFilterSlider.getValue(), sendNotificationSync); };
        LowPassFilterSlider.onDragEnd = [this] () { LowPassFilterSliderR.setValue(LowPassFilterSlider.getValue(), sendNotificationSync); };
    } else if (LRVerbSel == 1) {
        HiPassFilterSlider.setValue(HiPassFilterSliderR.getValue(), sendNotificationSync);
        LowPassFilterSlider.setValue(LowPassFilterSliderR.getValue(), sendNotificationSync);
        HiPassFilterSliderR.onDragStart = [this] () { HiPassFilterSlider.setValue(HiPassFilterSliderR.getValue(), sendNotificationSync); };
        HiPassFilterSliderR.onValueChange = [this] () { HiPassFilterSlider.setValue(HiPassFilterSliderR.getValue(), sendNotificationSync); };
        HiPassFilterSliderR.onDragEnd = [this] () { HiPassFilterSlider.setValue(HiPassFilterSliderR.getValue(), sendNotificationSync); };
        LowPassFilterSliderR.onDragStart = [this] () { LowPassFilterSlider.setValue(LowPassFilterSliderR.getValue(), sendNotificationSync); };
        LowPassFilterSliderR.onValueChange = [this] () { LowPassFilterSlider.setValue(LowPassFilterSliderR.getValue(), sendNotificationSync); };
        LowPassFilterSliderR.onDragEnd = [this] () { LowPassFilterSlider.setValue(LowPassFilterSliderR.getValue(), sendNotificationSync); };
    }
    FxChannelLinkButton.onClick = [this] (){ if (LRVerbSel == 0) { L(); } else if (LRVerbSel == 1){ R(); } };
}

void ReverbTestAudioProcessorEditor::L()
{
    *processor.ReverbParams.getRawParameterValue("fxverbbinding") = false;
    FxChannelLinkButton.setToggleState(*processor.ReverbParams.getRawParameterValue("fxverbbinding"), sendNotificationSync);
    if (LRVerbSel != 0) LRVerbSel = 0;
    DBG("L Channel Selected");
    LButton.setToggleState(true, sendNotificationSync); RButton.setToggleState(false, dontSendNotification);
    LButton.setEnabled(false); RButton.setEnabled(true);
    VisibleSliders(); SliderColours();
    SizeLValue.reset(); SizeRValue.reset(); WidthLValue.reset(); WidthRValue.reset(); DampingLValue.reset(); DampingRValue.reset(); DryLValue.reset();
```

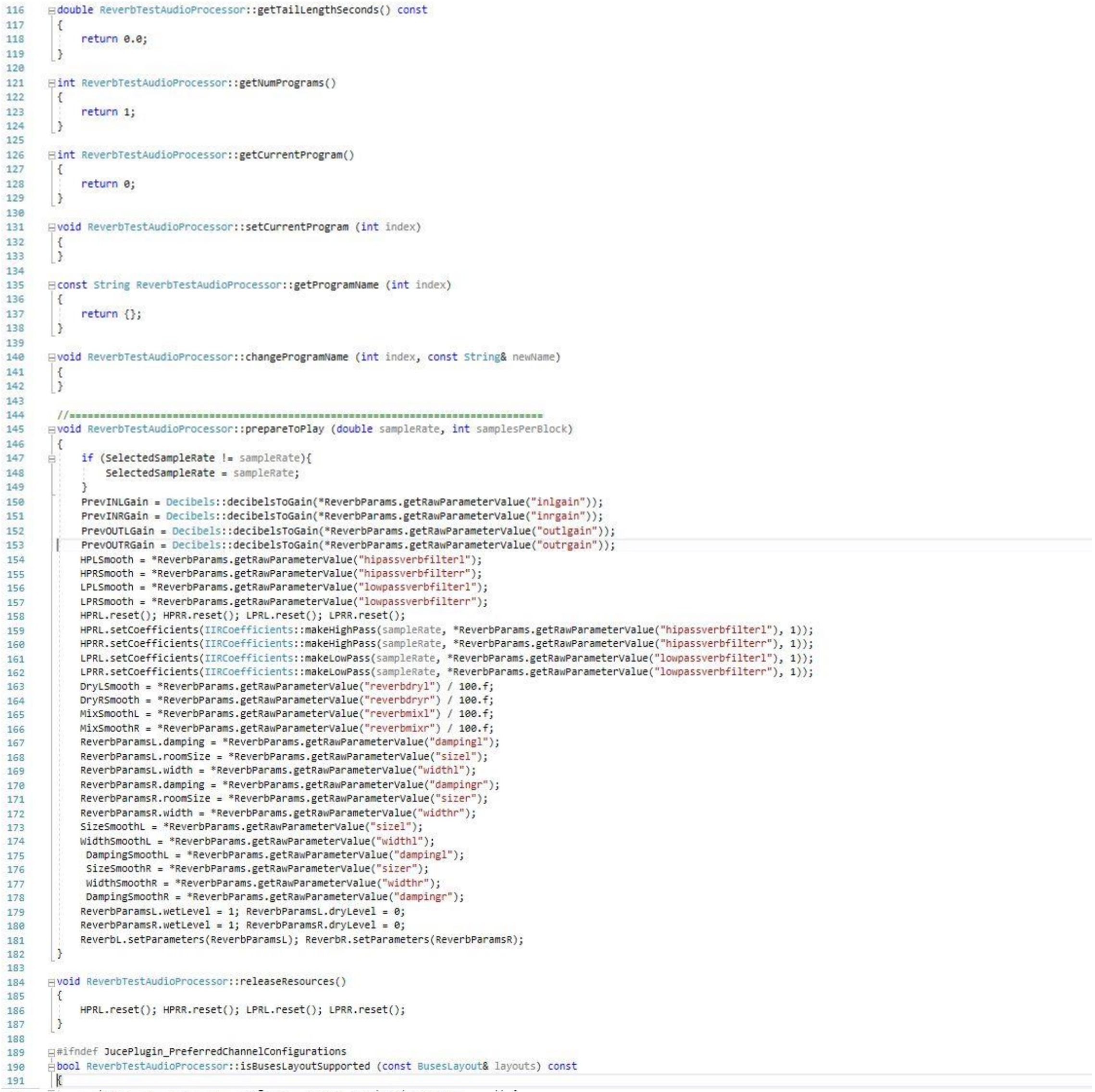
```
    DryRValue.reset(); MixLValue.reset(); MixRValue.reset();
    SizeLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "sizel", SizeSlider);
    WidthLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "widthl", WidthSlider);
    DampingLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "dampingl", DampingSlider);
    DryLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "reverbdryl", DrySlider);
    MixLValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "reverbmixl", WetSlider);
    HiPassFilterSlider.onDragStart = [this] () {}; HiPassFilterSlider.onValueChange = [this] () {}; HiPassFilterSlider.onDragEnd = [this] () {};
    LowPassFilterSlider.onDragStart = [this] () {}; LowPassFilterSlider.onValueChange = [this] () {}; LowPassFilterSlider.onDragEnd = [this] () {};
    FxChannelLinkButton.onClick = [this] () { FxChannelLinked(); };
    RButton.onClick = [this] () { R(); };
}

void ReverbTestAudioProcessorEditor::R()
{
    *processor.ReverbParams.getRawParameterValue("fxverbbinding") = false;
    FxChannelLinkButton.setToggleState(*processor.ReverbParams.getRawParameterValue("fxverbbinding"), sendNotificationSync);
    if (LRVerbSel != 1) LRVerbSel = 1;
    LButton.setToggleState(false, dontSendNotification); RButton.setToggleState(true, sendNotificationSync);
    LButton.setEnabled(true); RButton.setEnabled(false);
    DBG("R Channel Selected");
    VisibleSliders(); SliderColours();
    SizeLValue.reset(); SizeRValue.reset(); WidthLValue.reset(); WidthRValue.reset(); DampingLValue.reset(); DampingRValue.reset(); DryLValue.reset();
    DryRValue.reset(); MixLValue.reset(); MixRValue.reset();
    SizeRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "sizer", SizeSlider);
    WidthRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "widthr", WidthSlider);
    DampingRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "dampingr", DampingSlider);
    DryRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "reverbdryr", DrySlider);
    MixRValue = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(processor.ReverbParams, "reverbmixr", WetSlider);
    HiPassFilterSliderR.onDragStart = [this] () {}; HiPassFilterSliderR.onValueChange = [this] () {}; HiPassFilterSliderR.onDragEnd = [this] () {};
    LowPassFilterSliderR.onDragStart = [this] () {}; LowPassFilterSliderR.onValueChange = [this] () {}; LowPassFilterSliderR.onDragEnd = [this] () {};
    FxChannelLinkButton.onClick = [this] () { FxChannelLinked(); };
    LButton.onClick = [this] () { L(); };
}

void ReverbTestAudioProcessorEditor::LinkOut()
{
    *processor.ReverbParams.getRawParameterValue("outbinding") = true;
    LinkOUTButton.setToggleState(*processor.ReverbParams.getRawParameterValue("outbinding"), sendNotificationSync);
    DBG("Link Out");
    OUTLSlider.onDragStart = [this] () { OUTRSlider.setValue(OUTLSlider.getValue()); };
    OUTLSlider.onValueChange = [this] () { OUTRSlider.setValue(OUTLSlider.getValue()); };
    OUTLSlider.onDragEnd = [this] () { OUTRSlider.setValue(OUTLSlider.getValue()); };
    OUTRSlider.onDragStart = [this] () { OUTLSlider.setValue(OUTRSlider.getValue()); };
    OUTRSlider.onValueChange = [this] () { OUTLSlider.setValue(OUTRSlider.getValue()); };
    OUTRSlider.onDragEnd = [this] () { OUTLSlider.setValue(OUTRSlider.getValue()); };
    LinkOUTButton.onClick = [this]() { UnLinkOut(); };
}

void ReverbTestAudioProcessorEditor::UnLinkOut()
{
    *processor.ReverbParams.getRawParameterValue("outbinding") = false;
    LinkOUTButton.setToggleState(*processor.ReverbParams.getRawParameterValue("outbinding"), sendNotificationSync);
    DBG("UnLink Out");
    OUTLSlider.onDragStart = [this] () {};
    OUTLSlider.onValueChange = [this] () {};
    OUTLSlider.onDragEnd = [this] () {};
    OUTRSlider.onDragStart = [this] () {};
    OUTRSlider.onValueChange = [this] () {};
    OUTRSlider.onDragEnd = [this] () {};
    LinkOUTButton.onClick = [this]() { LinkOut(); };
}

void ReverbTestAudioProcessorEditor::VisibleSliders()
{
    if (LRVerbSel == 0) {
        HiPassFilterSlider.setVisible(true); LowPassFilterSlider.setVisible(true);
        HiPassFilterSliderR.setVisible(false); LowPassFilterSliderR.setVisible(false);
    } else if (LRVerbSel == 1){
        HiPassFilterSlider.setVisible(false); LowPassFilterSlider.setVisible(false);
        HiPassFilterSliderR.setVisible(true); LowPassFilterSliderR.setVisible(true);
    }
}

void ReverbTestAudioProcessorEditor::SliderColours()
{
    if (*processor.ReverbParams.getRawParameterValue("reverbstate") == true) {
        GeneralSliders.setColour(Slider::thumbColourId, Colours::darkgoldenrod); GeneralSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgoldenrodyellow);
        GeneralSliders.setColour(Slider::rotarySliderFillColourId, Colours::goldenrod);
        MixSliders.setColour(Slider::thumbColourId, Colours::darkgoldenrod); MixSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgoldenrodyellow);
        MixSliders.setColour(Slider::rotarySliderFillColourId, Colours::goldenrod);
        FilterSliders.setColour(Slider::thumbColourId, Colours::darkgoldenrod); FilterSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::lightgoldenrodyellow);
        FilterSliders.setColour(Slider::rotarySliderFillColourId, Colours::goldenrod);
        HiPassFilterSlider.setColour(Slider::textBoxTextColourId, Colours::goldenrod); HiPassFilterSlider.setColour(Slider::textBoxOutlineColourId, Colours::goldenrod);
        LowPassFilterSlider.setColour(Slider::textBoxTextColourId, Colours::goldenrod); LowPassFilterSlider.setColour(Slider::textBoxOutlineColourId, Colours::goldenrod);
        SizeSlider.setColour(Slider::textBoxTextColourId, Colours::goldenrod); SizeSlider.setColour(Slider::textBoxOutlineColourId, Colours::goldenrod);
        WidthSlider.setColour(Slider::textBoxTextColourId, Colours::goldenrod); WidthSlider.setColour(Slider::textBoxOutlineColourId, Colours::goldenrod);
        DampingSlider.setColour(Slider::textBoxTextColourId, Colours::goldenrod); DampingSlider.setColour(Slider::textBoxOutlineColourId, Colours::goldenrod);
        DrySlider.setColour(Slider::textBoxTextColourId, Colours::goldenrod); DrySlider.setColour(Slider::textBoxOutlineColourId, Colours::goldenrod);
        WetSlider.setColour(Slider::textBoxTextColourId, Colours::goldenrod); WetSlider.setColour(Slider::textBoxOutlineColourId, Colours::goldenrod);
        HiPassFilterSliderR.setColour(Slider::textBoxTextColourId, Colours::goldenrod); HiPassFilterSliderR.setColour(Slider::textBoxOutlineColourId, Colours::goldenrod);
        LowPassFilterSliderR.setColour(Slider::textBoxTextColourId, Colours::goldenrod); LowPassFilterSliderR.setColour(Slider::textBoxOutlineColourId, Colours::goldenrod);
    } else {
        GeneralSliders.setColour(Slider::thumbColourId, Colours::dimgrey); GeneralSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::grey);
        GeneralSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey);
        MixSliders.setColour(Slider::thumbColourId, Colours::dimgrey); MixSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::grey);
        MixSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey);
        FilterSliders.setColour(Slider::thumbColourId, Colours::dimgrey); FilterSliders.setColour(Slider::rotarySliderOutlineColourId, Colours::grey);
        FilterSliders.setColour(Slider::rotarySliderFillColourId, Colours::darkgrey);
        HiPassFilterSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); HiPassFilterSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        LowPassFilterSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); LowPassFilterSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        SizeSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); SizeSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        WidthSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); WidthSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        DampingSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); DampingSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        DrySlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); DrySlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        WetSlider.setColour(Slider::textBoxTextColourId, Colours::darkgrey); WetSlider.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        HiPassFilterSliderR.setColour(Slider::textBoxTextColourId, Colours::darkgrey); HiPassFilterSliderR.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
        LowPassFilterSliderR.setColour(Slider::textBoxTextColourId, Colours::darkgrey); LowPassFilterSliderR.setColour(Slider::textBoxOutlineColourId, Colours::darkgrey);
    }
}

void ReverbTestAudioProcessorEditor::paint (Graphics& g)
{
    Font Labels("Snell Roundhand", "bold", 30.0f);
    Font Title("Snell Roundhand", "bold", 40.0f);
```



```
    Font FilterSmallLabel("Snell Roundhand", "bold", 20.0f);
    Font SmallLabels("Snell Roundhand", "bold", 25.0f);
    g.fillAll(Colours::black);
    if (*processor.ReverbParams.getRawParameterValue("reverbstate") == true) { g.setColour(Colours::goldenrod); }
    else { g.setColour(Colours::darkgrey); }
    g.drawRoundedRectangle(0, 0, getWidth(), getHeight(), 10, jmin(getWidth(), getHeight()) / 30);
    RverbLabel.setFont(Title); INLabel.setFont(SmallLabels); OUTLabel.setFont(SmallLabels); HiPassFilterLabel.setFont(Labels);
    LowPassFilterLabel.setFont(Labels); SizeLabel.setFont(Labels); WidthLabel.setFont(Labels); DampingLabel.setFont(Labels); DryLabel.setFont(Labels);
    MixLabel.setFont(Labels); g.drawRoundedRectangle(0, 0, getWidth(), getHeight(), 10, jmin(getWidth(), getHeight()) / 30); repaint();
}
```

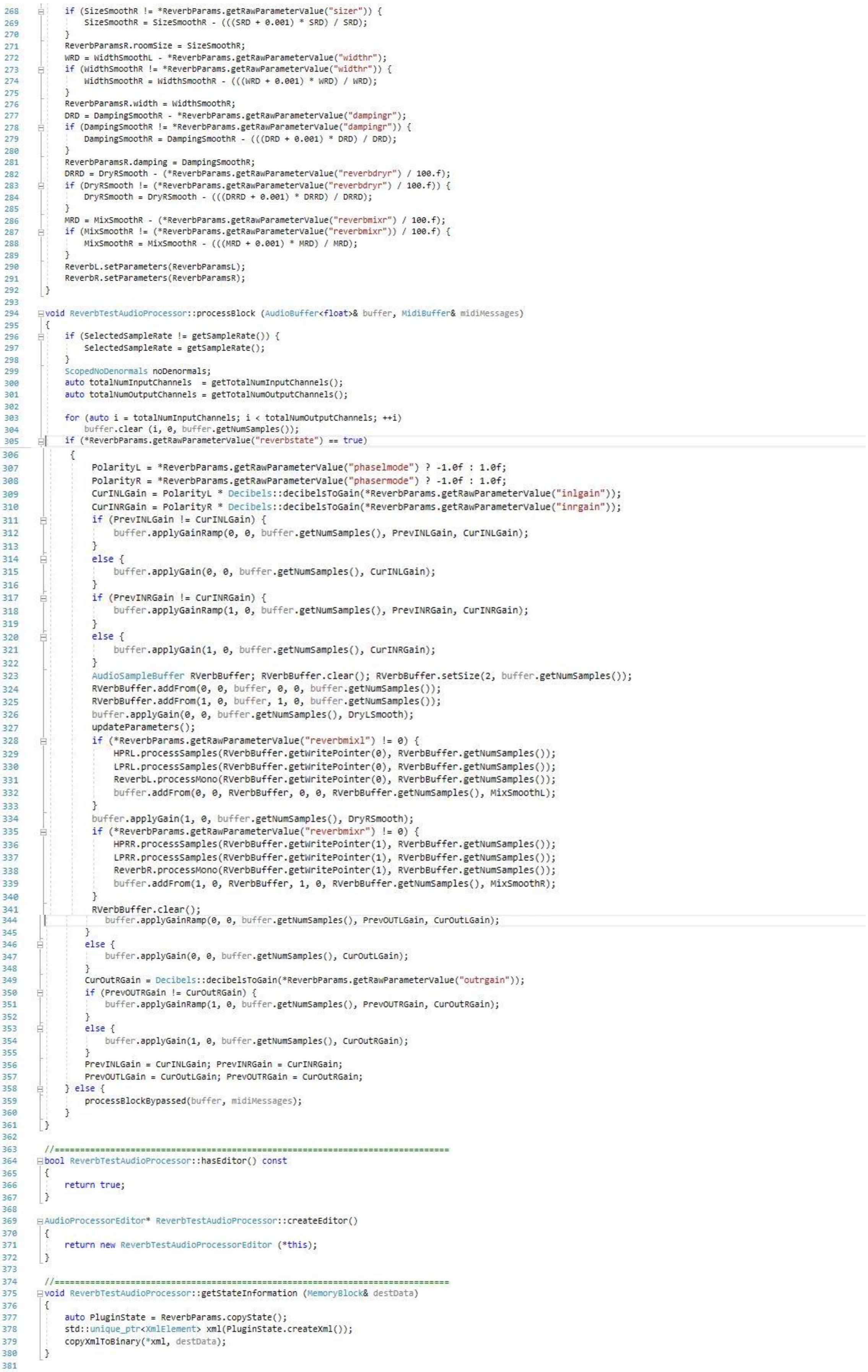
```
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/


#pragma once

#include "../JuceLibraryCode/JuceHeader.h"

//==============================================================================
/**
*/
class ReverbTestAudioProcessor  : public AudioProcessor
{
public:
    //==============================================================================
    ReverbTestAudioProcessor();
    ~ReverbTestAudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif

    void processBlock (AudioBuffer<float>&, MidiBuffer&) override;

    //==============================================================================
    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==============================================================================
    const String getName() const override;

    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;

    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const String getProgramName (int index) override;
    void changeProgramName (int index, const String& newName) override;

    //==============================================================================
    void getStateInformation (MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;
    void updateParameters();
    AudioProcessorValueTreeState ReverbParams;
    AudioProcessorValueTreeState::ParameterLayout createParameterLayout();

private:

    Reverb ReverbL, ReverbR;
    Reverb::Parameters ReverbParamsL, ReverbParamsR;
    float PrevINLGain, PrevINRGain, PrevOUTLGain, PrevOUTRGain, PolarityL, PolarityR, CurINLGain, CurINRGain, DryLSmooth,
        DryRSmooth, CurOutLGain, CurOutRGain, HPLSmooth, LPLSmooth, SelectedSampleRate, HPRSmooth, LPRSmooth, MixSmoothL,
        MixSmoothR, SizeSmoothL, WidthSmoothL, DampingSmoothL, SizeSmoothR, WidthSmoothR, DampingSmoothR;
    IIRFilter HPRL, HPRR, LPRL, LPRR;

    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (ReverbTestAudioProcessor)
};
```

```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
ReverbTestAudioProcessor::ReverbTestAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties().withInput("INPUT", AudioChannelSet::stereo(), true)
                                        .withOutput("OUTPUT", AudioChannelSet::stereo(), true)
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", AudioChannelSet::stereo(), true)
                     #endif
                       ), ReverbParams(*this, nullptr, "Parameters", createParameterLayout())
#endif
{
    HPRL.reset(); HPRR.reset(); LPRL.reset(); LPRR.reset();
    ReverbParamsL.wetLevel = 1.f; ReverbParamsL.dryLevel = 0.f;
     ReverbParamsR.wetLevel = 1.f; ReverbParamsR.dryLevel = 0.f;
}

ReverbTestAudioProcessor::~ReverbTestAudioProcessor()
{
    HPRL.reset(); HPRR.reset(); LPRL.reset(); LPRR.reset();
}

AudioProcessorValueTreeState::ParameterLayout ReverbTestAudioProcessor::createParameterLayout()
{
    std::vector<std::unique_ptr<RangedAudioParameter>> RverbParams;
    /*Parámetros Generales*/
    auto ReverbState = std::make_unique<AudioParameterBool>("reverbstate", "ON/OFF", true);
    auto INBinding = std::make_unique<AudioParameterBool>("inbinding", "Link In", true);
    auto OUTBinding = std::make_unique<AudioParameterBool>("outbinding", "Link Out", true);
    auto PhaseLMode = std::make_unique<AudioParameterBool>("phaselmode", "Polarity L", false);
    auto PhaseRMode = std::make_unique<AudioParameterBool>("phasermode", "Polarity R", false);
    auto FxVerbBinding = std::make_unique<AudioParameterBool>("fxverbbinding", "Fx Channel Link", true);
    /*Parametros de Ganancia de Entrada y de salida*/
    auto INLGainParamater = std::make_unique<AudioParameterFloat>("inlgain", "IN L Gain", -100.0f, 12.0f, 0.0f);
    auto INRGainParameter = std::make_unique<AudioParameterFloat>("inrgain", "IN R Gain", -100.0f, 12.0f, 0.0f);
    auto OUTLGainParameter = std::make_unique<AudioParameterFloat>("outlgain", "OUT L Gain", -100.0f, 12.0f, 0.0f);
    auto OUTRGainParameter = std::make_unique<AudioParameterFloat>("outrgain", "OUT R Gain", -100.0f, 12.0f, 0.0f);
    /*Parámetros de Filtros*/
    auto HipassVerbFilterLParameter = std::make_unique<AudioParameterFloat>("hipassverbfilterl", "HP Reverb Filter L", 20.f, 20000.f, 20.f);
    auto HipassVerbFilterRParameter = std::make_unique<AudioParameterFloat>("hipassverbfilterr", "HP Reverb Filter R", 20.f, 20000.f, 20.f);
    auto LowPassVerbFilterLParameter = std::make_unique<AudioParameterFloat>("lowpassverbfilterl", "LP Reverb Filter L", 20.f, 20000.f, 20000.f);
    auto LowPassVerbFilterRParameter = std::make_unique<AudioParameterFloat>("lowpassverbfilterr", "LP Reverb Filter R", 20.f, 20000.f, 20000.f);
    /*Parámetros para el efecto de Reverb*/
    auto ReverbSizeL = std::make_unique<AudioParameterFloat>("sizel", "Size L", 0.001f, 1.0f, 0.5f);//
    auto ReverbSizeR = std::make_unique<AudioParameterFloat>("sizer", "Size R", 0.001f, 1.0f, 0.5f);//
    auto ReverbWidthL = std::make_unique<AudioParameterFloat>("widthl", "Width L", 0.001f, 1.0f, 0.5f);//
    auto ReverbWetLParameter = std::make_unique<AudioParameterFloat>("reverbmixl", "Wet L", 0.0f, 100.0f, 50.0f);
    auto ReverbDryLParameter = std::make_unique<AudioParameterFloat>("reverbdryl", "Dry L", 0.f, 100.f, 50.0f);
    auto ReverbWidthR = std::make_unique<AudioParameterFloat>("widthr", "Width R", 0.001f, 1.0f, 0.5f);//
    auto DampingL = std::make_unique<AudioParameterFloat>("dampingl", "Damping L", 0.001f, 1.0f, 0.5f);//
    auto DampingR = std::make_unique<AudioParameterFloat>("dampingr", "Damping R", 0.001f, 1.0f, 0.5f);//
    auto ReverbWetRParameter = std::make_unique<AudioParameterFloat>("reverbmixr", "Wet R", 0.0f, 100.0f, 50.0f);
    auto ReverbDryRParameter = std::make_unique<AudioParameterFloat>("reverbdryr", "Dry R", 0.f, 100.f, 50.0f);
    /*Push Back Parameters*/
    RverbParams.push_back(std::move(ReverbState)); RverbParams.push_back(std::move(INBinding)); RverbParams.push_back(std::move(PhaseLMode));
    RverbParams.push_back(std::move(PhaseRMode));
    RverbParams.push_back(std::move(INLGainParamater)); RverbParams.push_back(std::move(INRGainParameter)); RverbParams.push_back(std::move(FxVerbBinding));
    RverbParams.push_back(std::move(HipassVerbFilterLParameter));
    RverbParams.push_back(std::move(HipassVerbFilterRParameter)); RverbParams.push_back(std::move(LowPassVerbFilterLParameter));
    RverbParams.push_back(std::move(ReverbDryRParameter)); RverbParams.push_back(std::move(ReverbWetLParameter));
    RverbParams.push_back(std::move(ReverbWetRParameter)); RverbParams.push_back(std::move(OUTBinding));
    RverbParams.push_back(std::move(OUTLGainParameter)); RverbParams.push_back(std::move(OUTRGainParameter));
    /*Cierre de vector de parámetros*/
    return { RverbParams.begin(), RverbParams.end() };
}
//==============================================================================
const String ReverbTestAudioProcessor::getName() const
{
    return JucePlugin_Name;
}

bool ReverbTestAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}

bool ReverbTestAudioProcessor::producesMidi() const
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
    return false;
   #endif
}

bool ReverbTestAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
   #else
    return false;
   #endif
}
```

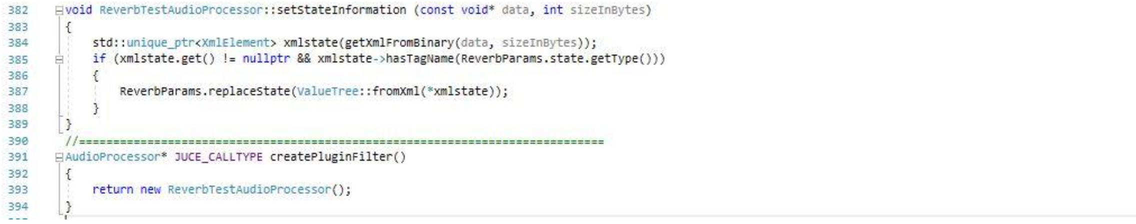


```cpp
double ReverbTestAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int ReverbTestAudioProcessor::getNumPrograms()
{
    return 1;
}

int ReverbTestAudioProcessor::getCurrentProgram()
{
    return 0;
}

void ReverbTestAudioProcessor::setCurrentProgram (int index)
{
}

const String ReverbTestAudioProcessor::getProgramName (int index)
{
    return {};
}

void ReverbTestAudioProcessor::changeProgramName (int index, const String& newName)
{
}

//==============================================================================
void ReverbTestAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    if (SelectedSampleRate != sampleRate){
        SelectedSampleRate = sampleRate;
    }
    PrevINLGain = Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("inlgain"));
    PrevINRGain = Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("inrgain"));
    PrevOUTLGain = Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("outlgain"));
    PrevOUTRGain = Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("outrgain"));
    HPLSmooth = *ReverbParams.getRawParameterValue("hipassverbfilterl");
    HPRSmooth = *ReverbParams.getRawParameterValue("hipassverbfilterr");
    LPLSmooth = *ReverbParams.getRawParameterValue("lowpassverbfilterl");
    LPRSmooth = *ReverbParams.getRawParameterValue("lowpassverbfilterr");
    HPRL.reset(); HPRR.reset(); LPRL.reset(); LPRR.reset();
    HPRL.setCoefficients(IIRCoefficients::makeHighPass(sampleRate, *ReverbParams.getRawParameterValue("hipassverbfilterl"), 1));
    HPRR.setCoefficients(IIRCoefficients::makeHighPass(sampleRate, *ReverbParams.getRawParameterValue("hipassverbfilterr"), 1));
    LPRL.setCoefficients(IIRCoefficients::makeLowPass(sampleRate, *ReverbParams.getRawParameterValue("lowpassverbfilterl"), 1));
    LPRR.setCoefficients(IIRCoefficients::makeLowPass(sampleRate, *ReverbParams.getRawParameterValue("lowpassverbfilterr"), 1));
    DryLSmooth = *ReverbParams.getRawParameterValue("reverbdryl") / 100.f;
    DryRSmooth = *ReverbParams.getRawParameterValue("reverbdryr") / 100.f;
    MixSmoothL = *ReverbParams.getRawParameterValue("reverbmixl") / 100.f;
    MixSmoothR = *ReverbParams.getRawParameterValue("reverbmixr") / 100.f;
    ReverbParamsL.damping = *ReverbParams.getRawParameterValue("dampingl");
    ReverbParamsL.roomSize = *ReverbParams.getRawParameterValue("sizel");
    ReverbParamsL.width = *ReverbParams.getRawParameterValue("widthl");
    ReverbParamsR.damping = *ReverbParams.getRawParameterValue("dampingr");
    ReverbParamsR.roomSize = *ReverbParams.getRawParameterValue("sizer");
    ReverbParamsR.width = *ReverbParams.getRawParameterValue("widthr");
    SizeSmoothL = *ReverbParams.getRawParameterValue("sizel");
    WidthSmoothL = *ReverbParams.getRawParameterValue("widthl");
     DampingSmoothL = *ReverbParams.getRawParameterValue("dampingl");
     SizeSmoothR = *ReverbParams.getRawParameterValue("sizer");
     WidthSmoothR = *ReverbParams.getRawParameterValue("widthr");
     DampingSmoothR = *ReverbParams.getRawParameterValue("dampingr");
    ReverbParamsL.wetLevel = 1; ReverbParamsL.dryLevel = 0;
    ReverbParamsR.wetLevel = 1; ReverbParamsR.dryLevel = 0;
    ReverbL.setParameters(ReverbParamsL); ReverbR.setParameters(ReverbParamsR);
}

void ReverbTestAudioProcessor::releaseResources()
{
    HPRL.reset(); HPRR.reset(); LPRL.reset(); LPRR.reset();
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool ReverbTestAudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
{
```

```cpp
        LPLSmooth = LPLSmooth - (((LPDifL + 0.001) * LPDifL) / LPDifL);
        LPRL.setCoefficients(IIRCoefficients::makeLowPass(SelectedSampleRate, LPLSmooth, 1));
    }
    else {
        LPRL.setCoefficients(IIRCoefficients::makeLowPass(SelectedSampleRate, *ReverbParams.getRawParameterValue("lowpassverbfilterl"), 1));
    }
    LPDifR = LPRSmooth - *ReverbParams.getRawParameterValue("lowpassverbfilterr");
    if (LPRSmooth != *ReverbParams.getRawParameterValue("lowpassverbfilterr")) {
        LPRSmooth = LPRSmooth - (((LPDifR + 0.001) * LPDifR) / LPDifR);
        LPRR.setCoefficients(IIRCoefficients::makeLowPass(SelectedSampleRate, LPRSmooth, 1));
    }
    else {
        LPRR.setCoefficients(IIRCoefficients::makeLowPass(SelectedSampleRate, *ReverbParams.getRawParameterValue("lowpassverbfilterr"), 1));
    }
    SLD = SizeSmoothL - *ReverbParams.getRawParameterValue("sizel");
    if (SizeSmoothL != *ReverbParams.getRawParameterValue("sizel")) {
        SizeSmoothL = SizeSmoothL - (((SLD + 0.001) * SLD) / SLD);
    }
    ReverbParamsL.roomSize = SizeSmoothL;
    WLD = WidthSmoothL - *ReverbParams.getRawParameterValue("widthl");
    if (WidthSmoothL != *ReverbParams.getRawParameterValue("widthl")) {
        WidthSmoothL = WidthSmoothL - (((WLD + 0.001) * WLD) / WLD);
    }
    ReverbParamsL.width = WidthSmoothL;
    DLD = DampingSmoothL - *ReverbParams.getRawParameterValue("dampingl");
    if (DampingSmoothL != *ReverbParams.getRawParameterValue("dampingl")) {
        DampingSmoothL = DampingSmoothL - (((DLD + 0.001) * DLD) / DLD);
    }
    ReverbParamsL.damping = DampingSmoothL;
    DRLD = DryLSmooth - (*ReverbParams.getRawParameterValue("reverbdryl") / 100.f);
    if (DryLSmooth != (*ReverbParams.getRawParameterValue("reverbdryl")) / 100.f) {
        DryLSmooth = DryLSmooth - (((DRLD + 0.001) * DRLD) / DRLD);
    }
    MLD = MixSmoothL - (*ReverbParams.getRawParameterValue("reverbmixl") / 100.f);
    if (MixSmoothL != (*ReverbParams.getRawParameterValue("reverbmixl")) / 100.f) {
        MixSmoothL = MixSmoothL - (((MLD + 0.001) * MLD) / MLD);
    }
    SRD = SizeSmoothR - *ReverbParams.getRawParameterValue("sizer");
```



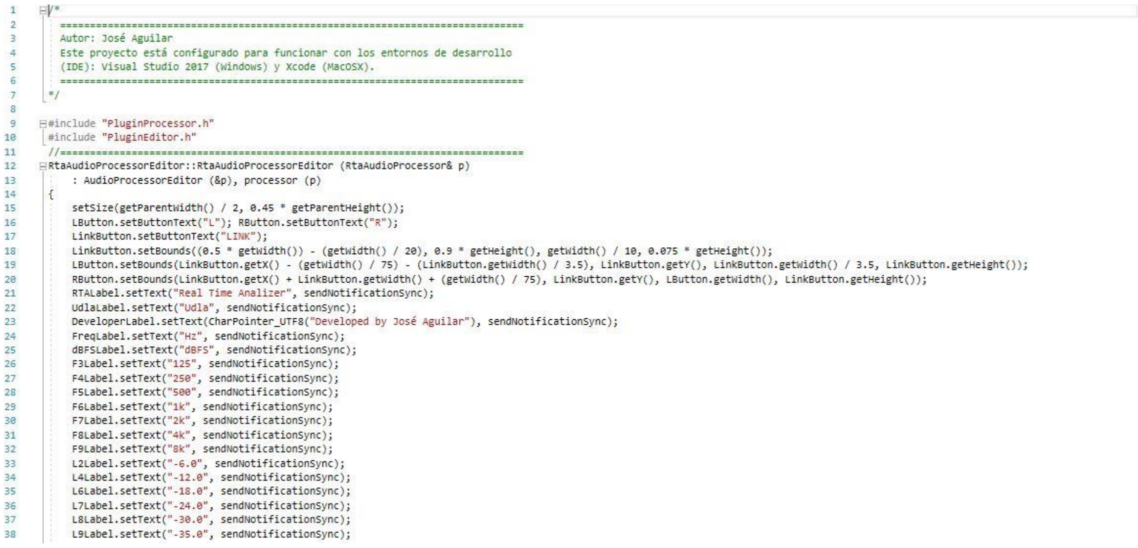
```cpp
    if (SizeSmoothR != *ReverbParams.getRawParameterValue("sizer")) {
        SizeSmoothR = SizeSmoothR - (((SRD + 0.001) * SRD) / SRD);
    }
    ReverbParamsR.roomSize = SizeSmoothR;
    WRD = WidthSmoothL - *ReverbParams.getRawParameterValue("widthr");
    if (WidthSmoothR != *ReverbParams.getRawParameterValue("widthr")) {
        WidthSmoothR = WidthSmoothR - (((WRD + 0.001) * WRD) / WRD);
    }
    ReverbParamsR.width = WidthSmoothR;
    DRD = DampingSmoothR - *ReverbParams.getRawParameterValue("dampingr");
    if (DampingSmoothR != *ReverbParams.getRawParameterValue("dampingr")) {
        DampingSmoothR = DampingSmoothR - (((DRD + 0.001) * DRD) / DRD);
    }
    ReverbParamsR.damping = DampingSmoothR;
    DRRD = DryRSmooth - (*ReverbParams.getRawParameterValue("reverbdryr") / 100.f);
    if (DryRSmooth != (*ReverbParams.getRawParameterValue("reverbdryr") / 100.f)) {
        DryRSmooth = DryRSmooth - (((DRRD + 0.001) * DRRD) / DRRD);
    }
    MRD = MixSmoothR - (*ReverbParams.getRawParameterValue("reverbmixr") / 100.f);
    if (MixSmoothR != (*ReverbParams.getRawParameterValue("reverbmixr")) / 100.f) {
        MixSmoothR = MixSmoothR - (((MRD + 0.001) * MRD) / MRD);
    }
    ReverbL.setParameters(ReverbParamsL);
    ReverbR.setParameters(ReverbParamsR);
}

void ReverbTestAudioProcessor::processBlock (AudioBuffer<float>& buffer, MidiBuffer& midiMessages)
{
    if (SelectedSampleRate != getSampleRate()) {
        SelectedSampleRate = getSampleRate();
    }
    ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());
    if (*ReverbParams.getRawParameterValue("reverbstate") == true)
    {
        PolarityL = *ReverbParams.getRawParameterValue("phaselmode") ? -1.0f : 1.0f;
        PolarityR = *ReverbParams.getRawParameterValue("phasermode") ? -1.0f : 1.0f;
        CurINLGain = PolarityL * Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("inlgain"));
        CurINRGain = PolarityR * Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("inrgain"));
        if (PrevINLGain != CurINLGain) {
            buffer.applyGainRamp(0, 0, buffer.getNumSamples(), PrevINLGain, CurINLGain);
        }
        else {
            buffer.applyGain(0, 0, buffer.getNumSamples(), CurINLGain);
        }
        if (PrevINRGain != CurINRGain) {
            buffer.applyGainRamp(1, 0, buffer.getNumSamples(), PrevINRGain, CurINRGain);
        }
        else {
            buffer.applyGain(1, 0, buffer.getNumSamples(), CurINRGain);
        }
        AudioSampleBuffer RVerbBuffer; RVerbBuffer.clear(); RVerbBuffer.setSize(2, buffer.getNumSamples());
        RVerbBuffer.addFrom(0, 0, buffer, 0, 0, buffer.getNumSamples());
        RVerbBuffer.addFrom(1, 0, buffer, 1, 0, buffer.getNumSamples());
        buffer.applyGain(0, 0, buffer.getNumSamples(), DryLSmooth);
        updateParameters();
        if (*ReverbParams.getRawParameterValue("reverbmixl") != 0) {
            HPRL.processSamples(RVerbBuffer.getWritePointer(0), RVerbBuffer.getNumSamples());
            LPRL.processSamples(RVerbBuffer.getWritePointer(0), RVerbBuffer.getNumSamples());
            ReverbL.processMono(RVerbBuffer.getWritePointer(0), RVerbBuffer.getNumSamples());
            buffer.addFrom(0, 0, RVerbBuffer, 0, 0, RVerbBuffer.getNumSamples(), MixSmoothL);
        }
        buffer.applyGain(1, 0, buffer.getNumSamples(), DryRSmooth);
        if (*ReverbParams.getRawParameterValue("reverbmixr") != 0) {
            HPRR.processSamples(RVerbBuffer.getWritePointer(1), RVerbBuffer.getNumSamples());
            LPRR.processSamples(RVerbBuffer.getWritePointer(1), RVerbBuffer.getNumSamples());
            ReverbR.processMono(RVerbBuffer.getWritePointer(1), RVerbBuffer.getNumSamples());
            buffer.addFrom(1, 0, RVerbBuffer, 1, 0, RVerbBuffer.getNumSamples(), MixSmoothR);
        }
        RVerbBuffer.clear();
            buffer.applyGainRamp(0, 0, buffer.getNumSamples(), PrevOUTLGain, CurOutLGain);
        }
        else {
            buffer.applyGain(0, 0, buffer.getNumSamples(), CurOutLGain);
        }
        CurOutRGain = Decibels::decibelsToGain(*ReverbParams.getRawParameterValue("outrgain"));
        if (PrevOUTRGain != CurOutRGain) {
            buffer.applyGainRamp(1, 0, buffer.getNumSamples(), PrevOUTRGain, CurOutRGain);
        }
        else {
            buffer.applyGain(1, 0, buffer.getNumSamples(), CurOutRGain);
        }
        PrevINLGain = CurINLGain; PrevINRGain = CurINRGain;
        PrevOUTLGain = CurOutLGain; PrevOUTRGain = CurOutRGain;
    } else {
        processBlockBypassed(buffer, midiMessages);
    }
}

//==============================================================================
bool ReverbTestAudioProcessor::hasEditor() const
{
    return true;
}

AudioProcessorEditor* ReverbTestAudioProcessor::createEditor()
{
    return new ReverbTestAudioProcessorEditor (*this);
}

//==============================================================================
void ReverbTestAudioProcessor::getStateInformation (MemoryBlock& destData)
{
    auto PluginState = ReverbParams.copyState();
    std::unique_ptr<XmlElement> xml(PluginState.createXml());
    copyXmlToBinary(*xml, destData);
}
```



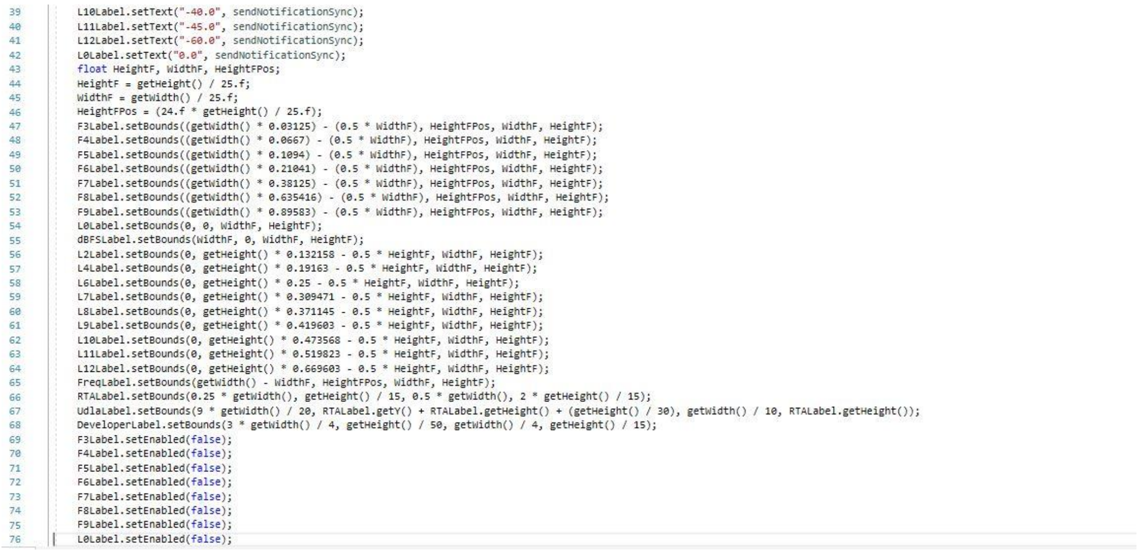
```cpp
void ReverbTestAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    std::unique_ptr<XmlElement> xmlstate(getXmlFromBinary(data, sizeInBytes));
    if (xmlstate.get() != nullptr && xmlstate->hasTagName(ReverbParams.state.getType()))
    {
        ReverbParams.replaceState(ValueTree::fromXml(*xmlstate));
    }
}
//==============================================================================
AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new ReverbTestAudioProcessor();
}
```

Anexo 5. Algoritmo del analizador de espectro

```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"
//==============================================================================
/**
*/
class RtaAudioProcessorEditor  : public AudioProcessorEditor,
                                 public Timer
{
public:
    RtaAudioProcessorEditor (RtaAudioProcessor&);
    ~RtaAudioProcessorEditor();

    //==============================================================================
    void buttons();
    void timerCallback() override;
    void paint (Graphics&) override;
    void DrawCurrentFrame(Graphics&g);
    void drawNextFrameOfSpectrum();
private:
    Label RTALabel, UdlaLabel, DeveloperLabel, F3Label, F4Label, F5Label, F6Label, F7Label, F8Label, F9Label, L2Label, L4Label, L6Label, L7Label, L8Label,
    L9Label, L10Label, L11Label, L12Label, FreqLabel, dBFSLabel, L0Label;
    TextButton LButton, RButton, LinkButton;
    Rectangle<float>          plotFrame;
    Rectangle<int>          brandingFrame;
    Path                    frequencyResponse;
    Path                    analyserPath;
    RtaAudioProcessor& processor;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RtaAudioProcessorEditor)
};
```

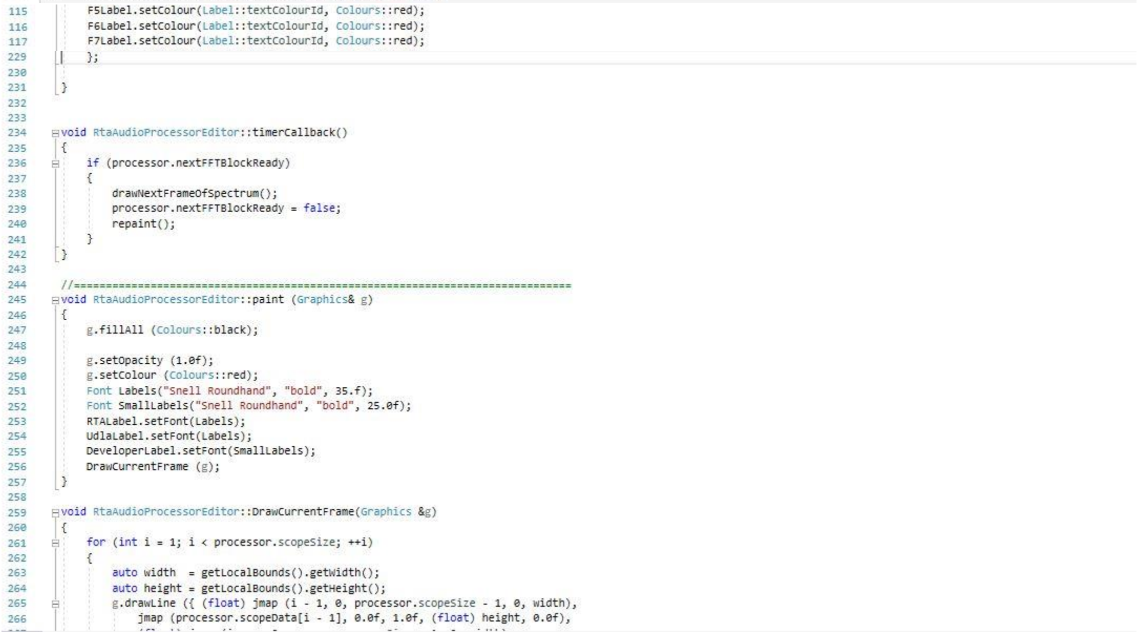


```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"
//==============================================================================
RtaAudioProcessorEditor::RtaAudioProcessorEditor (RtaAudioProcessor& p)
    : AudioProcessorEditor (&p), processor (p)
{
    setSize(getParentWidth() / 2, 0.45 * getParentHeight());
    LButton.setButtonText("L"); RButton.setButtonText("R");
    LinkButton.setButtonText("LINK");
    LinkButton.setBounds((0.5 * getWidth()) - (getWidth() / 20), 0.9 * getHeight(), getWidth() / 10, 0.075 * getHeight());
    LButton.setBounds(LinkButton.getX() - (getWidth() / 75) - (LinkButton.getWidth() / 3.5), LinkButton.getY(), LinkButton.getWidth() / 3.5, LinkButton.getHeight());
    RButton.setBounds(LinkButton.getX() + LinkButton.getWidth() + (getWidth() / 75), LinkButton.getY(), LButton.getWidth(), LinkButton.getHeight());
    RTALabel.setText("Real Time Analizer", sendNotificationSync);
    UdlaLabel.setText("Udla", sendNotificationSync);
    DeveloperLabel.setText(CharPointer_UTF8("Developed by José Aguilar"), sendNotificationSync);
    FreqLabel.setText("Hz", sendNotificationSync);
    dBFSLabel.setText("dBFS", sendNotificationSync);
    F3Label.setText("125", sendNotificationSync);
    F4Label.setText("250", sendNotificationSync);
    F5Label.setText("500", sendNotificationSync);
    F6Label.setText("1k", sendNotificationSync);
    F7Label.setText("2k", sendNotificationSync);
    F8Label.setText("4k", sendNotificationSync);
    F9Label.setText("8k", sendNotificationSync);
    L2Label.setText("-6.0", sendNotificationSync);
    L4Label.setText("-12.0", sendNotificationSync);
    L6Label.setText("-18.0", sendNotificationSync);
    L7Label.setText("-24.0", sendNotificationSync);
    L8Label.setText("-30.0", sendNotificationSync);
    L9Label.setText("-35.0", sendNotificationSync);
```



```cpp
    L10Label.setText("-40.0", sendNotificationSync);
    L11Label.setText("-45.0", sendNotificationSync);
    L12Label.setText("-60.0", sendNotificationSync);
    L0Label.setText("0.0", sendNotificationSync);
    float HeightF, WidthF, HeightFPos;
    HeightF = getHeight() / 25.f;
    WidthF = getWidth() / 25.f;
    HeightFPos = (24.f * getHeight() / 25.f);
    F3Label.setBounds((getWidth() * 0.03125) - (0.5 * WidthF), HeightFPos, WidthF, HeightF);
    F4Label.setBounds((getWidth() * 0.0667) - (0.5 * WidthF), HeightFPos, WidthF, HeightF);
    F5Label.setBounds((getWidth() * 0.1094) - (0.5 * WidthF), HeightFPos, WidthF, HeightF);
    F6Label.setBounds((getWidth() * 0.21041) - (0.5 * WidthF), HeightFPos, WidthF, HeightF);
    F7Label.setBounds((getWidth() * 0.38125) - (0.5 * WidthF), HeightFPos, WidthF, HeightF);
    F8Label.setBounds((getWidth() * 0.635416) - (0.5 * WidthF), HeightFPos, WidthF, HeightF);
    F9Label.setBounds((getWidth() * 0.89583) - (0.5 * WidthF), HeightFPos, WidthF, HeightF);
    L0Label.setBounds(0, 0, WidthF, HeightF);
    dBFSLabel.setBounds(WidthF, 0, WidthF, HeightF);
    L2Label.setBounds(0, getHeight() * 0.132158 - 0.5 * HeightF, WidthF, HeightF);
    L4Label.setBounds(0, getHeight() * 0.19163 - 0.5 * HeightF, WidthF, HeightF);
    L6Label.setBounds(0, getHeight() * 0.25 - 0.5 * HeightF, WidthF, HeightF);
    L7Label.setBounds(0, getHeight() * 0.309471 - 0.5 * HeightF, WidthF, HeightF);
    L8Label.setBounds(0, getHeight() * 0.371145 - 0.5 * HeightF, WidthF, HeightF);
    L9Label.setBounds(0, getHeight() * 0.419603 - 0.5 * HeightF, WidthF, HeightF);
    L10Label.setBounds(0, getHeight() * 0.473568 - 0.5 * HeightF, WidthF, HeightF);
    L11Label.setBounds(0, getHeight() * 0.519823 - 0.5 * HeightF, WidthF, HeightF);
    L12Label.setBounds(0, getHeight() * 0.669603 - 0.5 * HeightF, WidthF, HeightF);
    FreqLabel.setBounds(getWidth() - WidthF, HeightFPos, WidthF, HeightF);
    RTALabel.setBounds(0.25 * getWidth(), getHeight() / 15, 0.5 * getWidth(), 2 * getHeight() / 15);
    UdlaLabel.setBounds(9 * getWidth() / 20, RTALabel.getY() + RTALabel.getHeight() + (getHeight() / 30), getWidth() / 10, RTALabel.getHeight());
    DeveloperLabel.setBounds(3 * getWidth() / 4, getHeight() / 50, getWidth() / 4, getHeight() / 15);
    F3Label.setEnabled(false);
    F4Label.setEnabled(false);
    F5Label.setEnabled(false);
    F6Label.setEnabled(false);
    F7Label.setEnabled(false);
    F8Label.setEnabled(false);
    F9Label.setEnabled(false);
    L0Label.setEnabled(false);
```

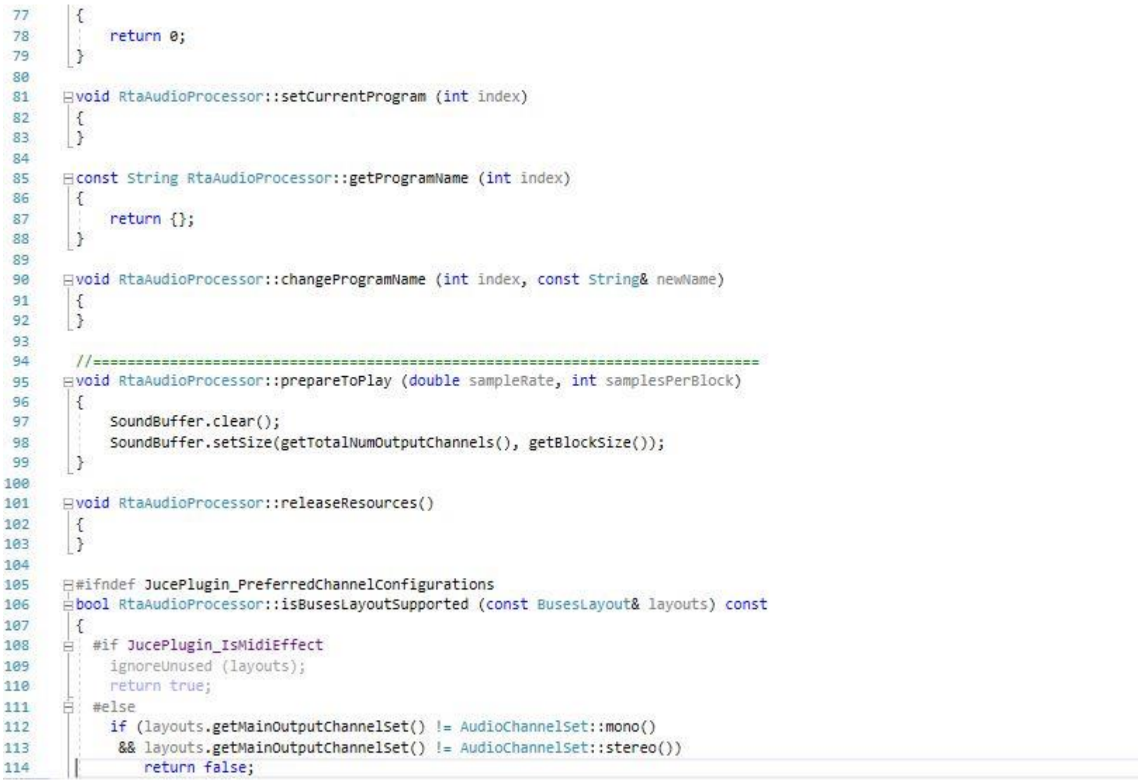
```cpp
    L2Label.setEnabled(false);
    L4Label.setEnabled(false);
    L6Label.setEnabled(false);
    L7Label.setEnabled(false);
    L8Label.setEnabled(false);
    L9Label.setEnabled(false);
    L10Label.setEnabled(false);
    L11Label.setEnabled(false);
    L12Label.setEnabled(false);
    FreqLabel.setEnabled(false);
    dBFSLabel.setEnabled(false);
    RTALabel.setJustificationType(Justification::Flags::centred);
    UdlaLabel.setJustificationType(Justification::Flags::centred);
    DeveloperLabel.setJustificationType(Justification::Flags::centred);
    F3Label.setJustificationType(Justification::Flags::centred);
    F4Label.setJustificationType(Justification::Flags::centred);
    F5Label.setJustificationType(Justification::Flags::centred);
    F6Label.setJustificationType(Justification::Flags::centred);
    F7Label.setJustificationType(Justification::Flags::centred);
    F8Label.setJustificationType(Justification::Flags::centred);
    F9Label.setJustificationType(Justification::Flags::centred);
    L0Label.setJustificationType(Justification::Flags::centred);
    L2Label.setJustificationType(Justification::Flags::centred);
    L4Label.setJustificationType(Justification::Flags::centred);
    L6Label.setJustificationType(Justification::Flags::centred);
    L7Label.setJustificationType(Justification::Flags::centred);
    L8Label.setJustificationType(Justification::Flags::centred);
    L9Label.setJustificationType(Justification::Flags::centred);
    L10Label.setJustificationType(Justification::Flags::centred);
    L11Label.setJustificationType(Justification::Flags::centred);
    L12Label.setJustificationType(Justification::Flags::centred);
    FreqLabel.setJustificationType(Justification::Flags::centred);
    dBFSLabel.setJustificationType(Justification::Flags::centred);
    RTALabel.setColour(Label::textColourId, Colours::red);
    UdlaLabel.setColour(Label::textColourId, Colours::red);
    DeveloperLabel.setColour(Label::textColourId, Colours::red);
    F3Label.setColour(Label::textColourId, Colours::red);
    F4Label.setColour(Label::textColourId, Colours::red);
```



```cpp
    F5Label.setColour(Label::textColourId, Colours::red);
    F6Label.setColour(Label::textColourId, Colours::red);
    F7Label.setColour(Label::textColourId, Colours::red);
    };

}


void RtaAudioProcessorEditor::timerCallback()
{
    if (processor.nextFFTBlockReady)
    {
        drawNextFrameOfSpectrum();
        processor.nextFFTBlockReady = false;
        repaint();
    }
}

//==============================================================================
void RtaAudioProcessorEditor::paint (Graphics& g)
{
    g.fillAll (Colours::black);

    g.setOpacity (1.0f);
    g.setColour (Colours::red);
    Font Labels("Snell Roundhand", "bold", 35.f);
    Font SmallLabels("Snell Roundhand", "bold", 25.0f);
    RTALabel.setFont(Labels);
    UdlaLabel.setFont(Labels);
    DeveloperLabel.setFont(SmallLabels);
    DrawCurrentFrame (g);
}

void RtaAudioProcessorEditor::DrawCurrentFrame(Graphics &g)
{
    for (int i = 1; i < processor.scopeSize; ++i)
    {
        auto width  = getLocalBounds().getWidth();
        auto height = getLocalBounds().getHeight();
        g.drawLine ({ (float) jmap (i - 1, 0, processor.scopeSize - 1, 0, width),
            jmap (processor.scopeData[i - 1], 0.0f, 1.0f, (float) height, 0.0f),
```

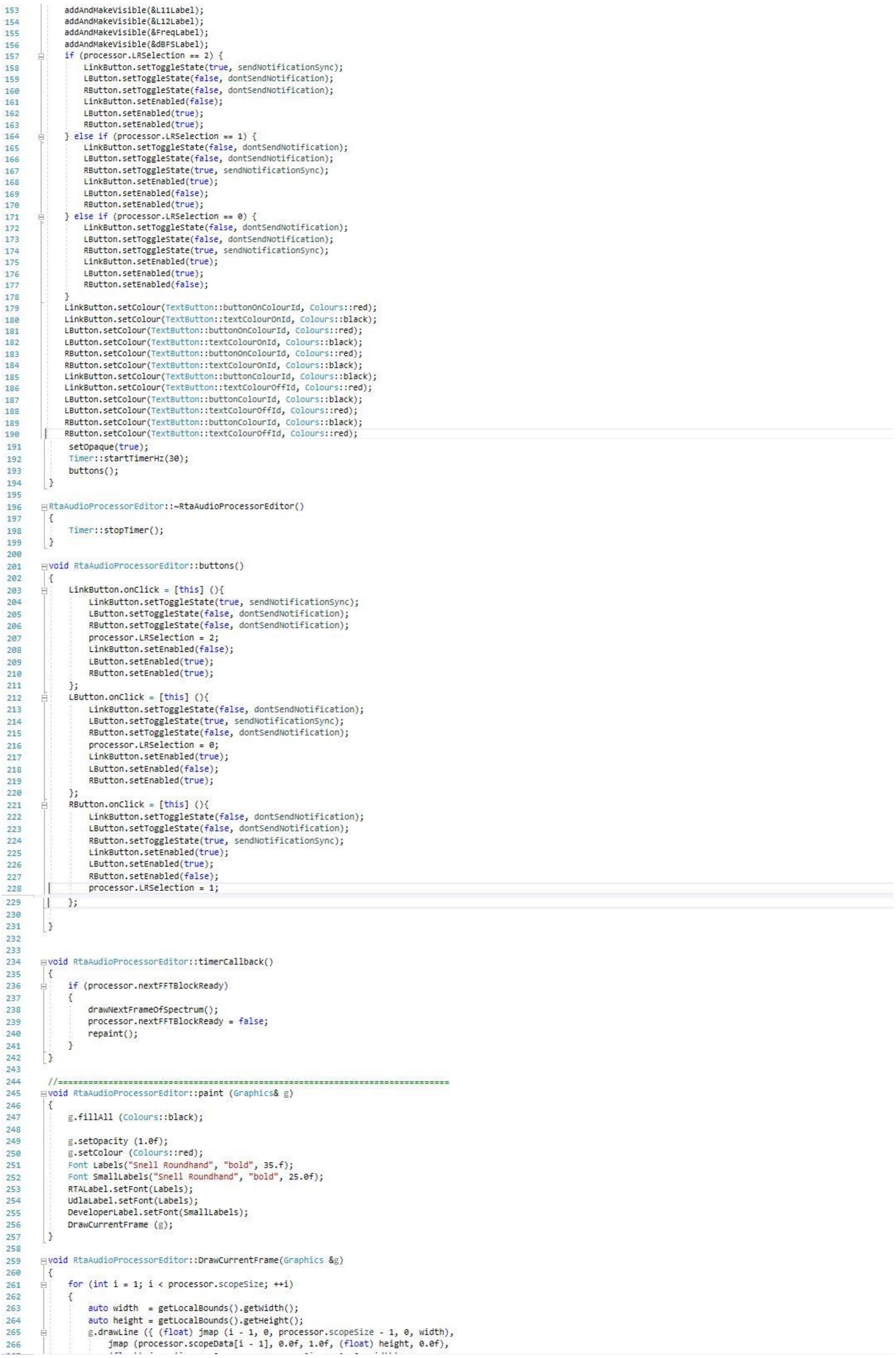
```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
//==============================================================================
/**
*/
class RtaAudioProcessor  : public AudioProcessor
{
public:
    //==============================================================================
    RtaAudioProcessor();
    ~RtaAudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif

    void processBlock (AudioBuffer<float>&, MidiBuffer&) override;

    //==============================================================================
    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;
    //==============================================================================
    const String getName() const override;

    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;


    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const String getProgramName (int index) override;
    void changeProgramName (int index, const String& newName) override;
    //==============================================================================
    void getStateInformation (MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;
    void getBlockofAudio(AudioBuffer<float>&Buffer);
    void pushNextSample(float sample);
    int LRSelection { 2 };

    enum
    {
        fftOrder  = 11,
        fftSize   = 1 << fftOrder,
        scopeSize = 512
    };
    dsp::FFT forwardFFT { 11 };
    dsp::WindowingFunction<float> window { fftSize, dsp::WindowingFunction<float>::hann };
    float fifo [fftSize];
    float fftData [2 * fftSize];
    int fifoIndex = 0;
    bool nextFFTBlockReady = false;
    float scopeData [scopeSize];

private:
    AudioBuffer<float> SoundBuffer;
    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RtaAudioProcessor)
};
```

```cpp
/*
  ===============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ===============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"
//===============================================================================
RtaAudioProcessor::RtaAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties().withInput("INPUT", AudioChannelSet::stereo(), true)
                                        .withOutput("OUTPUT", AudioChannelSet::stereo(), true)
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", AudioChannelSet::stereo(), true)
                     #endif
                       )
#endif
{
    SoundBuffer.clear();
}

RtaAudioProcessor::~RtaAudioProcessor()
{
    SoundBuffer.clear();
}

//===============================================================================
const String RtaAudioProcessor::getName() const
{
    return JucePlugin_Name;
}

bool RtaAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}

bool RtaAudioProcessor::producesMidi() const
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
    return false;
   #endif
}

bool RtaAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
   #else
    return false;
   #endif
}

double RtaAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int RtaAudioProcessor::getNumPrograms()
{
    return 1;
}

int RtaAudioProcessor::getCurrentProgram()
```



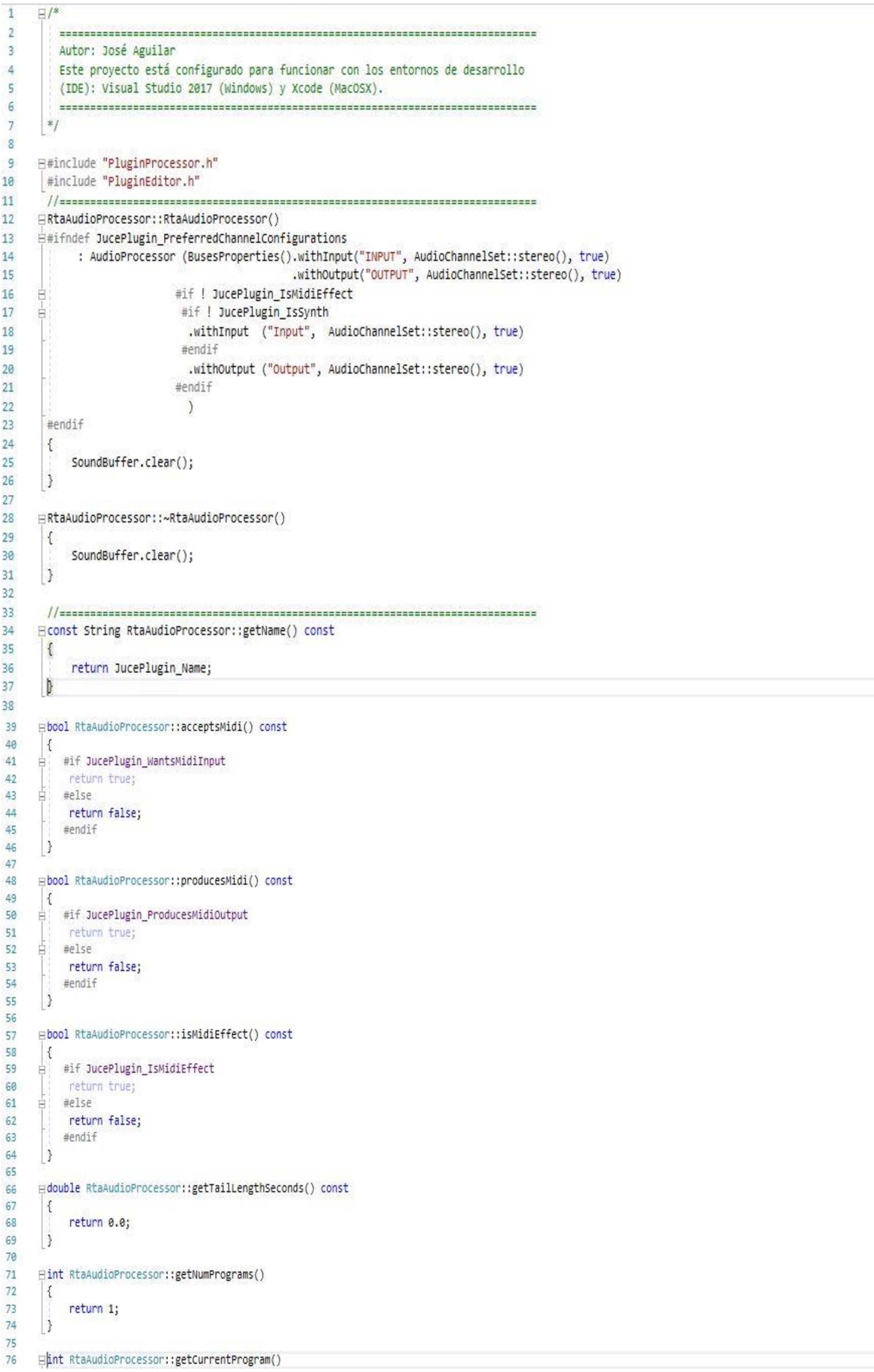
```
{
    return 0;
}

void RtaAudioProcessor::setCurrentProgram (int index)
{
}

const String RtaAudioProcessor::getProgramName (int index)
{
    return {};
}

void RtaAudioProcessor::changeProgramName (int index, const String& newName)
{
}

//==============================================================================
void RtaAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    SoundBuffer.clear();
    SoundBuffer.setSize(getTotalNumOutputChannels(), getBlockSize());
}

void RtaAudioProcessor::releaseResources()
{
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool RtaAudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    ignoreUnused (layouts);
    return true;
  #else
    if (layouts.getMainOutputChannelSet() != AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != AudioChannelSet::stereo())
        return false;
```

```
    F5Label.setColour(Label::textColourId, Colours::red);
    F6Label.setColour(Label::textColourId, Colours::red);
    F7Label.setColour(Label::textColourId, Colours::red);
    F8Label.setColour(Label::textColourId, Colours::red);
    F9Label.setColour(Label::textColourId, Colours::red);
    L0Label.setColour(Label::textColourId, Colours::red);
    L2Label.setColour(Label::textColourId, Colours::red);
    L4Label.setColour(Label::textColourId, Colours::red);
    L6Label.setColour(Label::textColourId, Colours::red);
    L7Label.setColour(Label::textColourId, Colours::red);
    L8Label.setColour(Label::textColourId, Colours::red);
    L9Label.setColour(Label::textColourId, Colours::red);
    L10Label.setColour(Label::textColourId, Colours::red);
    L11Label.setColour(Label::textColourId, Colours::red);
    L12Label.setColour(Label::textColourId, Colours::red);
    FreqLabel.setColour(Label::textColourId, Colours::red);
    dBFSLabel.setColour(Label::textColourId, Colours::red);
    addAndMakeVisible(&LButton);
    addAndMakeVisible(&LinkButton);
    addAndMakeVisible(&RButton);
    addAndMakeVisible(&RTALabel);
    addAndMakeVisible(&UdlaLabel);
    addAndMakeVisible(&DeveloperLabel);
    addAndMakeVisible(&F3Label);
    addAndMakeVisible(&F4Label);
    addAndMakeVisible(&F5Label);
    addAndMakeVisible(&F6Label);
    addAndMakeVisible(&F7Label);
    addAndMakeVisible(&F8Label);
    addAndMakeVisible(&F9Label);
    addAndMakeVisible(&L0Label);
    addAndMakeVisible(&L2Label);
    addAndMakeVisible(&L4Label);
    addAndMakeVisible(&L6Label);
    addAndMakeVisible(&L7Label);
    addAndMakeVisible(&L8Label);
    addAndMakeVisible(&L9Label);
    addAndMakeVisible(&L10Label);
```



```
    addAndMakeVisible(&L11Label);
    addAndMakeVisible(&L12Label);
    addAndMakeVisible(&FreqLabel);
    addAndMakeVisible(&dBFSLabel);
    if (processor.LRSelection == 2) {
        LinkButton.setToggleState(true, sendNotificationSync);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(false, dontSendNotification);
        LinkButton.setEnabled(false);
        LButton.setEnabled(true);
        RButton.setEnabled(true);
    } else if (processor.LRSelection == 1) {
        LinkButton.setToggleState(false, dontSendNotification);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(true, sendNotificationSync);
        LinkButton.setEnabled(true);
        LButton.setEnabled(false);
        RButton.setEnabled(true);
    } else if (processor.LRSelection == 0) {
        LinkButton.setToggleState(false, dontSendNotification);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(true, sendNotificationSync);
        LinkButton.setEnabled(true);
        LButton.setEnabled(true);
        RButton.setEnabled(false);
    }
    LinkButton.setColour(TextButton::buttonOnColourId, Colours::red);
    LinkButton.setColour(TextButton::textColourOnId, Colours::black);
    LButton.setColour(TextButton::buttonOnColourId, Colours::red);
    LButton.setColour(TextButton::textColourOnId, Colours::black);
    RButton.setColour(TextButton::buttonOnColourId, Colours::red);
    RButton.setColour(TextButton::textColourOnId, Colours::black);
    LinkButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkButton.setColour(TextButton::textColourOffId, Colours::red);
    LButton.setColour(TextButton::buttonColourId, Colours::black);
    LButton.setColour(TextButton::textColourOffId, Colours::red);
    RButton.setColour(TextButton::buttonColourId, Colours::black);
    RButton.setColour(TextButton::textColourOffId, Colours::red);
    setOpaque(true);
    Timer::startTimerHz(30);
    buttons();
}

RtaAudioProcessorEditor::~RtaAudioProcessorEditor()
{
    Timer::stopTimer();
}

void RtaAudioProcessorEditor::buttons()
{
    LinkButton.onClick = [this] (){
        LinkButton.setToggleState(true, sendNotificationSync);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(false, dontSendNotification);
        processor.LRSelection = 2;
        LinkButton.setEnabled(false);
        LButton.setEnabled(true);
        RButton.setEnabled(true);
    };
    LButton.onClick = [this] (){
        LinkButton.setToggleState(false, dontSendNotification);
        LButton.setToggleState(true, sendNotificationSync);
        RButton.setToggleState(false, dontSendNotification);
        processor.LRSelection = 0;
        LinkButton.setEnabled(true);
        LButton.setEnabled(false);
        RButton.setEnabled(true);
    };
    RButton.onClick = [this] (){
        LinkButton.setToggleState(false, dontSendNotification);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(true, sendNotificationSync);
        LinkButton.setEnabled(true);
        LButton.setEnabled(true);
        RButton.setEnabled(false);
        processor.LRSelection = 1;
    };

}


void RtaAudioProcessorEditor::timerCallback()
{
    if (processor.nextFFTBlockReady)
    {
        drawNextFrameOfSpectrum();
        processor.nextFFTBlockReady = false;
        repaint();
    }
}

//==============================================================================
void RtaAudioProcessorEditor::paint (Graphics& g)
{
    g.fillAll (Colours::black);

    g.setOpacity (1.0f);
    g.setColour (Colours::red);
    Font Labels("Snell Roundhand", "bold", 35.f);
    Font SmallLabels("Snell Roundhand", "bold", 25.0f);
    RTALabel.setFont(Labels);
    UdlaLabel.setFont(Labels);
    DeveloperLabel.setFont(SmallLabels);
    DrawCurrentFrame (g);
}

void RtaAudioProcessorEditor::DrawCurrentFrame(Graphics &g)
{
    for (int i = 1; i < processor.scopeSize; ++i)
    {
        auto width  = getLocalBounds().getWidth();
        auto height = getLocalBounds().getHeight();
        g.drawLine ({ (float) jmap (i - 1, 0, processor.scopeSize - 1, 0, width),
            jmap (processor.scopeData[i - 1], 0.0f, 1.0f, (float) height, 0.0f),
```

```cpp
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
//==============================================================================
/**
*/
class RtaAudioProcessor  : public AudioProcessor
{
public:
    //==============================================================================
    RtaAudioProcessor();
    ~RtaAudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif

    void processBlock (AudioBuffer<float>&, MidiBuffer&) override;

    //==============================================================================
    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;
    //==============================================================================
    const String getName() const override;

    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;


    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const String getProgramName (int index) override;
    void changeProgramName (int index, const String& newName) override;
    //==============================================================================
    void getStateInformation (MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;
    void getBlockofAudio(AudioBuffer<float>&Buffer);
    void pushNextSample(float sample);
    int LRSelection { 2 };

    enum
    {
        fftOrder  = 11,
        fftSize   = 1 << fftOrder,
        scopeSize = 512
    };
    dsp::FFT forwardFFT { 11 };
    dsp::WindowingFunction<float> window { fftSize, dsp::WindowingFunction<float>::hann };
    float fifo [fftSize];
    float fftData [2 * fftSize];
    int fifoIndex = 0;
    bool nextFFTBlockReady = false;
    float scopeData [scopeSize];

private:
    AudioBuffer<float> SoundBuffer;
    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RtaAudioProcessor)
};
```



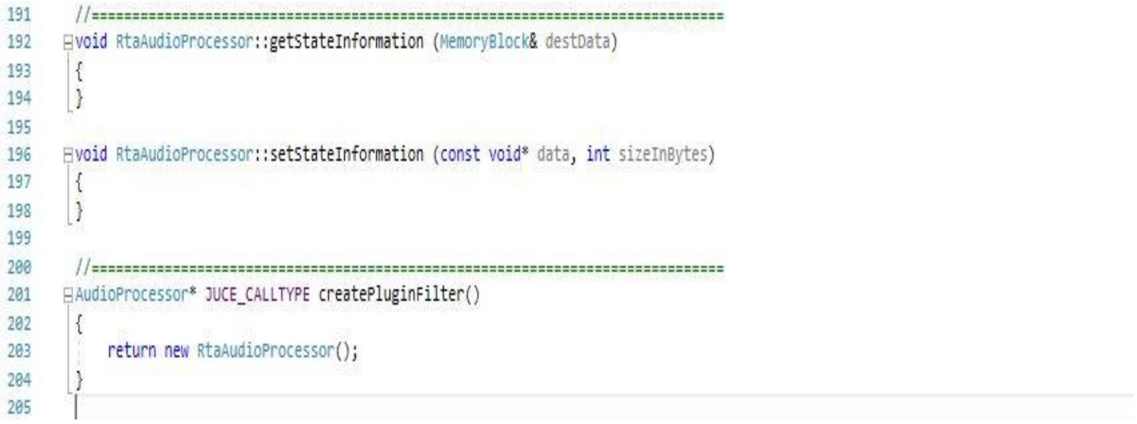
```
/*
  ==============================================================================
  Autor: José Aguilar
  Este proyecto está configurado para funcionar con los entornos de desarrollo
  (IDE): Visual Studio 2017 (Windows) y Xcode (MacOSX).
  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"
//==============================================================================
RtaAudioProcessor::RtaAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties().withInput("INPUT", AudioChannelSet::stereo(), true)
                                        .withOutput("OUTPUT", AudioChannelSet::stereo(), true)
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", AudioChannelSet::stereo(), true)
                     #endif
                       )
#endif
{
    SoundBuffer.clear();
}

RtaAudioProcessor::~RtaAudioProcessor()
{
    SoundBuffer.clear();
}

//==============================================================================
const String RtaAudioProcessor::getName() const
{
    return JucePlugin_Name;
}

bool RtaAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}

bool RtaAudioProcessor::producesMidi() const
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
    return false;
   #endif
}

bool RtaAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
   #else
    return false;
   #endif
}

double RtaAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int RtaAudioProcessor::getNumPrograms()
{
    return 1;
}

int RtaAudioProcessor::getCurrentProgram()
```

```
    L2Label.setEnabled(false);
    L4Label.setEnabled(false);
    L6Label.setEnabled(false);
    L7Label.setEnabled(false);
    L8Label.setEnabled(false);
    L9Label.setEnabled(false);
    L10Label.setEnabled(false);
    L11Label.setEnabled(false);
    L12Label.setEnabled(false);
    FreqLabel.setEnabled(false);
    dBFSLabel.setEnabled(false);
    RTALabel.setJustificationType(Justification::Flags::centred);
    UdlaLabel.setJustificationType(Justification::Flags::centred);
    DeveloperLabel.setJustificationType(Justification::Flags::centred);
    F3Label.setJustificationType(Justification::Flags::centred);
    F4Label.setJustificationType(Justification::Flags::centred);
    F5Label.setJustificationType(Justification::Flags::centred);
    F6Label.setJustificationType(Justification::Flags::centred);
    F7Label.setJustificationType(Justification::Flags::centred);
    F8Label.setJustificationType(Justification::Flags::centred);
    F9Label.setJustificationType(Justification::Flags::centred);
    L0Label.setJustificationType(Justification::Flags::centred);
    L2Label.setJustificationType(Justification::Flags::centred);
    L4Label.setJustificationType(Justification::Flags::centred);
    L6Label.setJustificationType(Justification::Flags::centred);
    L7Label.setJustificationType(Justification::Flags::centred);
    L8Label.setJustificationType(Justification::Flags::centred);
    L9Label.setJustificationType(Justification::Flags::centred);
    L10Label.setJustificationType(Justification::Flags::centred);
    L11Label.setJustificationType(Justification::Flags::centred);
    L12Label.setJustificationType(Justification::Flags::centred);
    FreqLabel.setJustificationType(Justification::Flags::centred);
    dBFSLabel.setJustificationType(Justification::Flags::centred);
    RTALabel.setColour(Label::textColourId, Colours::red);
    UdlaLabel.setColour(Label::textColourId, Colours::red);
    DeveloperLabel.setColour(Label::textColourId, Colours::red);
    F3Label.setColour(Label::textColourId, Colours::red);
    F4Label.setColour(Label::textColourId, Colours::red);
```

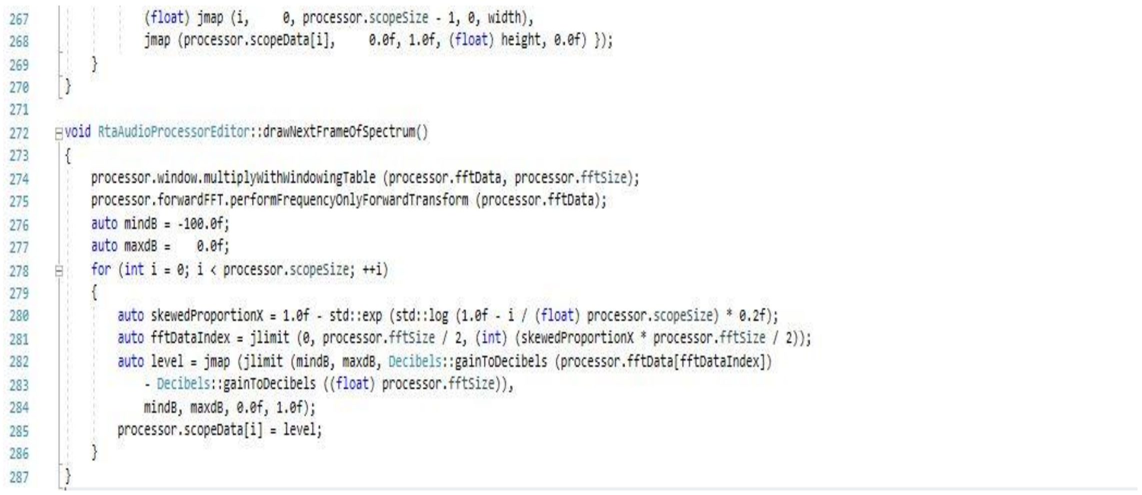
```
  #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() != layouts.getMainInputChannelSet())
        return false;
  #endif

    return true;
 #endif
}
#endif

void RtaAudioProcessor::processBlock (AudioBuffer<float>& buffer, MidiBuffer& midiMessages)
{
    ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();
    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());
    SoundBuffer.makeCopyOf(buffer);
    getBlockofAudio(SoundBuffer);
}

void RtaAudioProcessor::getBlockofAudio(AudioBuffer<float>&Buffer)
{
    if (Buffer.getNumChannels() > 0) {
        if (LRSelection == 2) {
            auto* ChannelDataL = Buffer.getReadPointer(0);
            auto* ChannelDataR = Buffer.getReadPointer(1);
            for (int sample = 0; sample<Buffer.getNumSamples(); sample++) {
                if (ChannelDataL [sample] != 0.f && ChannelDataR [sample] == 0.f) {
                    pushNextSample(ChannelDataL [sample]);
                } else if (ChannelDataR [sample] != 0.f && ChannelDataL [sample] == 0.f){
                    pushNextSample(ChannelDataR [sample]);
                } else {
                    auto ChannelSum = Decibels::decibelsToGain(20 * log(ChannelDataL [sample] + ChannelDataR [sample]));
                    pushNextSample(ChannelSum);
                }
            }
        } else if (LRSelection == 0) {
```

```
    addAndMakeVisible(&L11Label);
    addAndMakeVisible(&L12Label);
    addAndMakeVisible(&FreqLabel);
    addAndMakeVisible(&dBFSLabel);
    if (processor.LRSelection == 2) {
        LinkButton.setToggleState(true, sendNotificationSync);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(false, dontSendNotification);
        LinkButton.setEnabled(false);
        LButton.setEnabled(true);
        RButton.setEnabled(true);
    } else if (processor.LRSelection == 1) {
        LinkButton.setToggleState(false, dontSendNotification);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(true, sendNotificationSync);
        LinkButton.setEnabled(true);
        LButton.setEnabled(false);
        RButton.setEnabled(true);
    } else if (processor.LRSelection == 0) {
        LinkButton.setToggleState(false, dontSendNotification);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(true, sendNotificationSync);
        LinkButton.setEnabled(true);
        LButton.setEnabled(true);
        RButton.setEnabled(false);
    }
    LinkButton.setColour(TextButton::buttonOnColourId, Colours::red);
    LinkButton.setColour(TextButton::textColourOnId, Colours::black);
    LButton.setColour(TextButton::buttonOnColourId, Colours::red);
    LButton.setColour(TextButton::textColourOnId, Colours::black);
    RButton.setColour(TextButton::buttonOnColourId, Colours::red);
    RButton.setColour(TextButton::textColourOnId, Colours::black);
    LinkButton.setColour(TextButton::buttonColourId, Colours::black);
    LinkButton.setColour(TextButton::textColourOffId, Colours::red);
    LButton.setColour(TextButton::buttonColourId, Colours::black);
    LButton.setColour(TextButton::textColourOffId, Colours::red);
    RButton.setColour(TextButton::buttonColourId, Colours::black);
    RButton.setColour(TextButton::textColourOffId, Colours::red);
```

```
    setOpaque(true);
    Timer::startTimerHz(30);
    buttons();
}

RtaAudioProcessorEditor::~RtaAudioProcessorEditor()
{
    Timer::stopTimer();
}

void RtaAudioProcessorEditor::buttons()
{
    LinkButton.onClick = [this] (){
        LinkButton.setToggleState(true, sendNotificationSync);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(false, dontSendNotification);
        processor.LRSelection = 2;
        LinkButton.setEnabled(false);
        LButton.setEnabled(true);
        RButton.setEnabled(true);
    };
    LButton.onClick = [this] (){
        LinkButton.setToggleState(false, dontSendNotification);
        LButton.setToggleState(true, sendNotificationSync);
        RButton.setToggleState(false, dontSendNotification);
        processor.LRSelection = 0;
        LinkButton.setEnabled(true);
        LButton.setEnabled(false);
        RButton.setEnabled(true);
    };
    RButton.onClick = [this] (){
        LinkButton.setToggleState(false, dontSendNotification);
        LButton.setToggleState(false, dontSendNotification);
        RButton.setToggleState(true, sendNotificationSync);
        LinkButton.setEnabled(true);
        LButton.setEnabled(true);
        RButton.setEnabled(false);
        processor.LRSelection = 1;
```

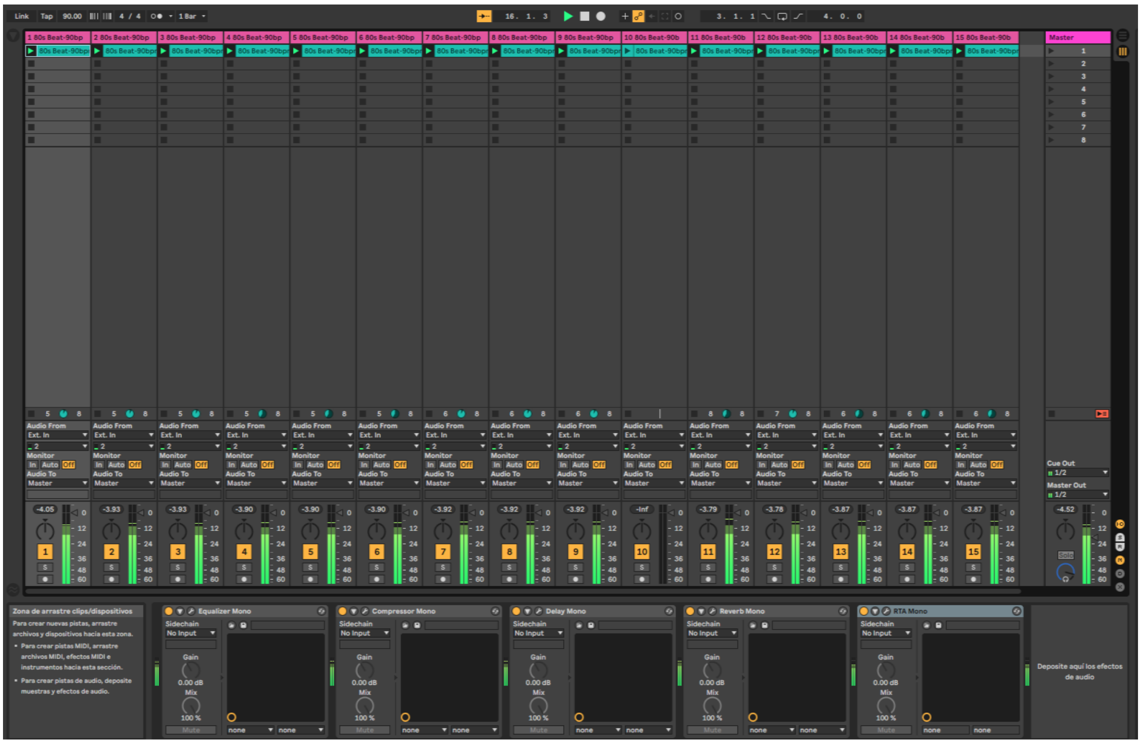


```
//==============================================================================
void RtaAudioProcessor::getStateInformation (MemoryBlock& destData)
{
}

void RtaAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
}

//==============================================================================
AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new RtaAudioProcessor();
}

```



```
                (float) jmap (i,     0, processor.scopeSize - 1, 0, width),
                jmap (processor.scopeData[i],     0.0f, 1.0f, (float) height, 0.0f) });
    }
}

void RtaAudioProcessorEditor::drawNextFrameOfSpectrum()
{
    processor.window.multiplyWithWindowingTable (processor.fftData, processor.fftSize);
    processor.forwardFFT.performFrequencyOnlyForwardTransform (processor.fftData);
    auto mindB = -100.0f;
    auto maxdB =    0.0f;
    for (int i = 0; i < processor.scopeSize; ++i)
    {
        auto skewedProportionX = 1.0f - std::exp (std::log (1.0f - i / (float) processor.scopeSize) * 0.2f);
        auto fftDataIndex = jlimit (0, processor.fftSize / 2, (int) (skewedProportionX * processor.fftSize / 2));
        auto level = jmap (jlimit (mindB, maxdB, Decibels::gainToDecibels (processor.fftData[fftDataIndex])
            - Decibels::gainToDecibels ((float) processor.fftSize)),
            mindB, maxdB, 0.0f, 1.0f);
        processor.scopeData[i] = level;
    }
}
```

Anexo 6. Funcionamiento en el software Ableton Live Suite 10

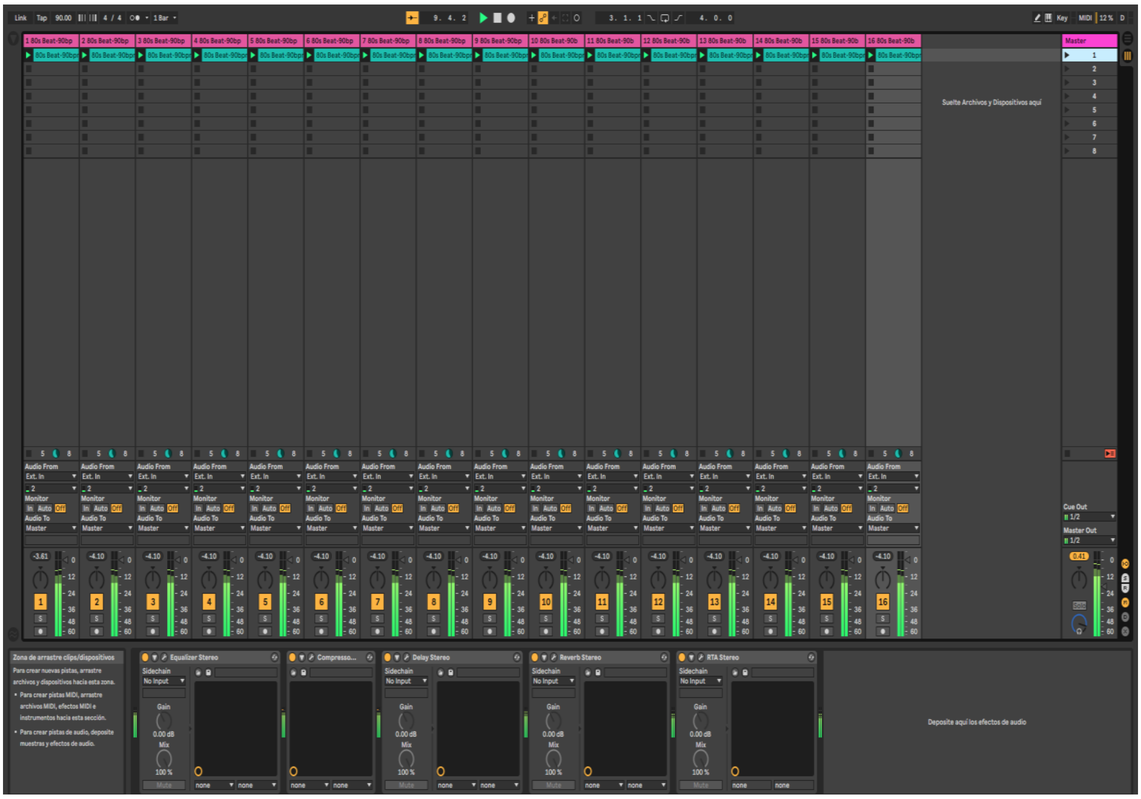


Zona de arrastre clips/dispositivos

Para crear nuevas pistas, arrastre archivos y dispositivos hacia esta zona.

- Para crear pistas MIDI, arrastre archivos MIDI, efectos MIDI e instrumentos hacia esta sección.
- Para crear pistas de audio, deposite muestras y efectos de audio.

Deposite aquí los efectos de audio



Suelte Archivos y Dispositivos aquí

Zona de arrastre clips/dispositivos

Para crear nuevas pistas, arrastre archivos y dispositivos hacia esta zona.

- Para crear pistas MIDI, arrastre archivos MIDI, efectos MIDI e instrumentos hacia esta sección.
- Para crear pistas de audio, deposite muestras y efectos de audio.

Deposite aquí los efectos de audio


Equalizer Mono/1-80s Beat-90bpm
Hi Pass 6dB/Oct
Equalizer
Low Pass 6dB/Oct
20.00 Hz
20000.00 Hz
Bass
Middle
Treble
0.00 dB
0.00dB
In
125 q:3
700 q:3
4kHz q:3
Out
Delay Mono/1-80s Beat-90bpm
Delay
Time
Feedback
Mix
1000.00 ms
50.00 %
SYNC
Compressor
10.00 ms
30.00 ms
Freshold
Ratio
Mix
5.00:1
Reverb Mono/1-80s Beat-90bpm
Reverb
Size
Width
Damping
Dry
Wet
50.00 %
RTA Mono/1-80s Beat-90bpm
Developed by José Aguilar
Real Time Analizer
Udla
Deposite aquí los efectos de audio
Guardar y exportar están desactivados.
¿Por qué?


Equalizer Stereo/1-80s Beat-90bpm
Hi Pass
Equalizer
Low Pass
L
LINK
R
20.00 Hz
20000.00 Hz
Bass
Middle
Treble
3.14dB
In
-7.20dB
Out
-2.94 dB
9.95 dB
-5.07 dB
260 Hz q:2
780 Hz q:3
8kHz q:3
Compressor Stereo/1-80s Beat-90bpm
Attack
Compressor
Release
5.00 ms
314.74 ms
Knee
Freshold
Ratio
Mix
-5.30dB
-1.64dB
15.00
-18.65 dB
5.00:1
100.00 %
Delay Stereo/1-80s Beat-90bpm
Delay
Time
Feedback
Mix
4.37dB
-6.94dB
1/4
50.00 %
SYNC
Reverb Stereo/1-80s Beat-90bpm
Hi Pass
Reverb
Low Pass
3.18dB
20.00 Hz
20000.00 Hz
Size
Width
Damping
Dry
0.50
73.45 %
Creación de patrones de batería
RTA Stereo/1-80s Beat-90bpm
Developed by José Aguilar
Real Time Analizer
Udla
Mostrar todas las lecciones incorporadas.

Zona de arrastre clips/dispositivos

Para crear nuevas pistas, arrastre archivos y dispositivos hacia esta zona.

- Para crear pistas MIDI, arrastre archivos MIDI, efectos MIDI e instrumentos hacia esta sección.
- Para crear pistas de audio, deposite muestras y efectos de audio.

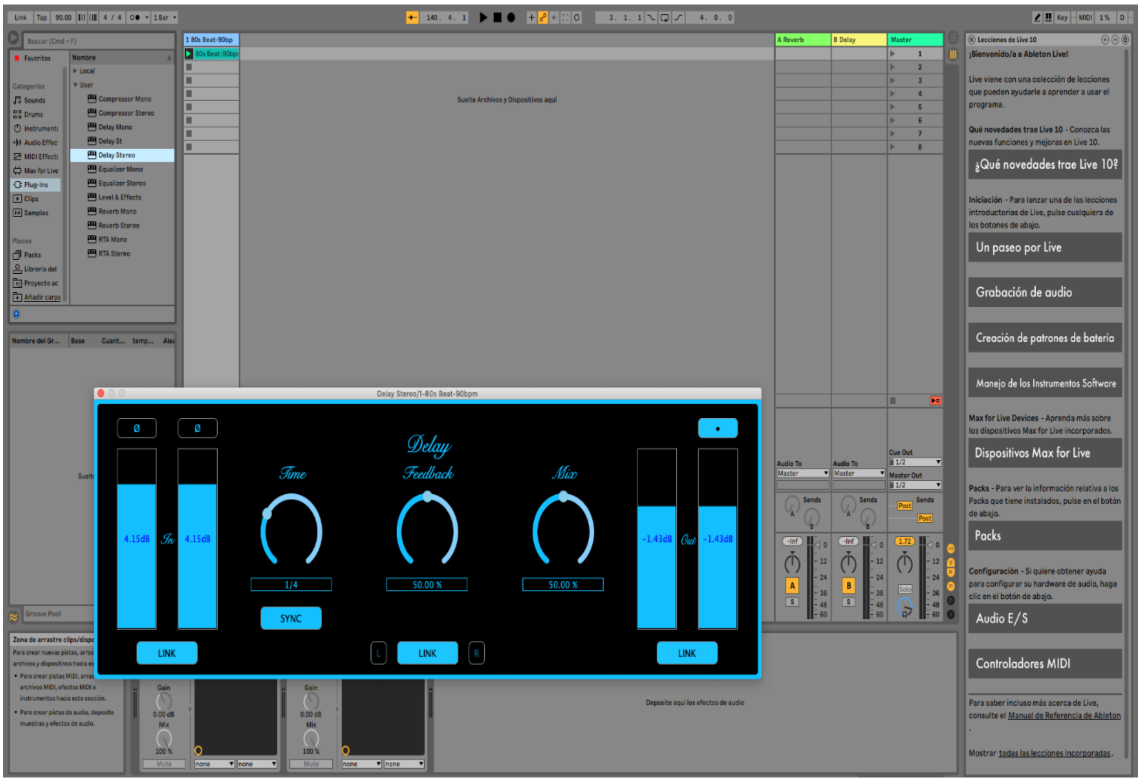
Zona de arrastre clips/dispositivos

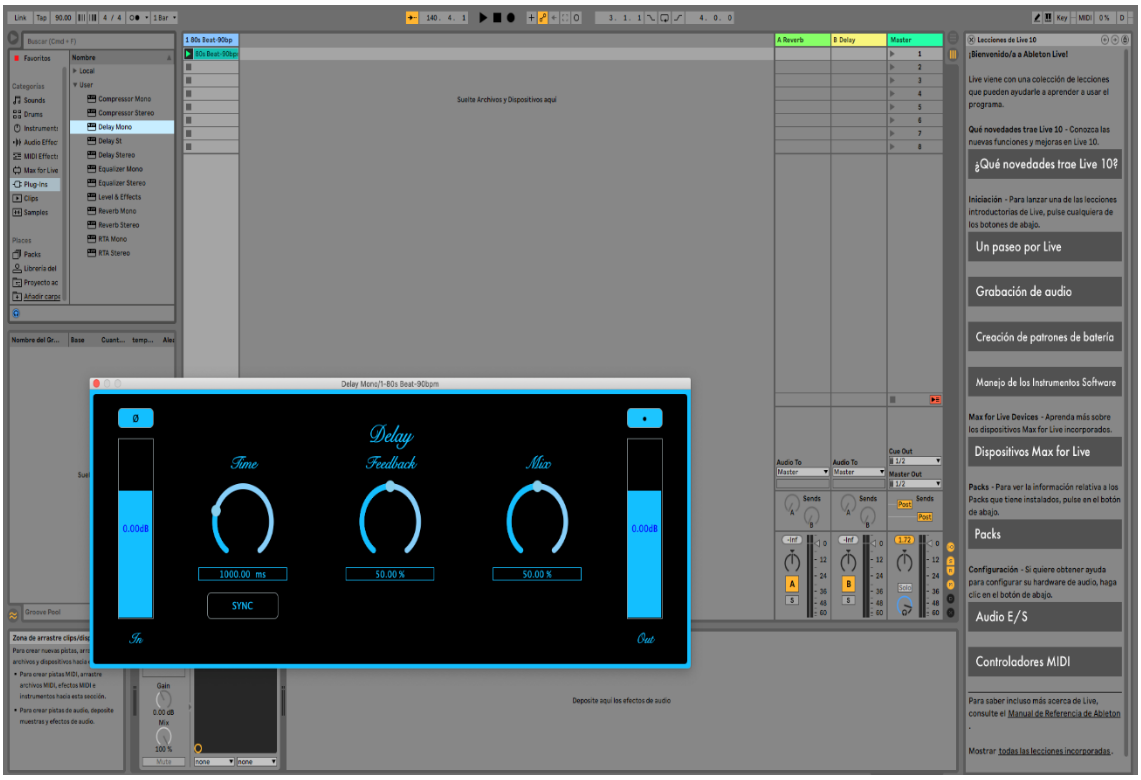
Para crear nuevas pistas, arrastre archivos y dispositivos hacia esta zona.

- Para crear pistas MIDI, arrastre archivos MIDI, efectos MIDI e instrumentos hacia esta sección.
- Para crear pistas de audio, deposite muestras y efectos de audio.


Compressor Stereo/1-80s Beat-90bpm
Compressor
Attack
5.00 ms
Release
10.00 ms
Treshold
0.00 dB
Ratio
5.00:1
Mix
100.00 %
In
Out
0.00dB
LINK
L
R
Suelte Archivos y Dispositivos aquí
Equalizer Stereo
Compresso...
Delay Stereo
Reverb Stereo
RTA Stereo
Deposite aquí los efectos de audio
Zona de arrastre clips/dispositivos

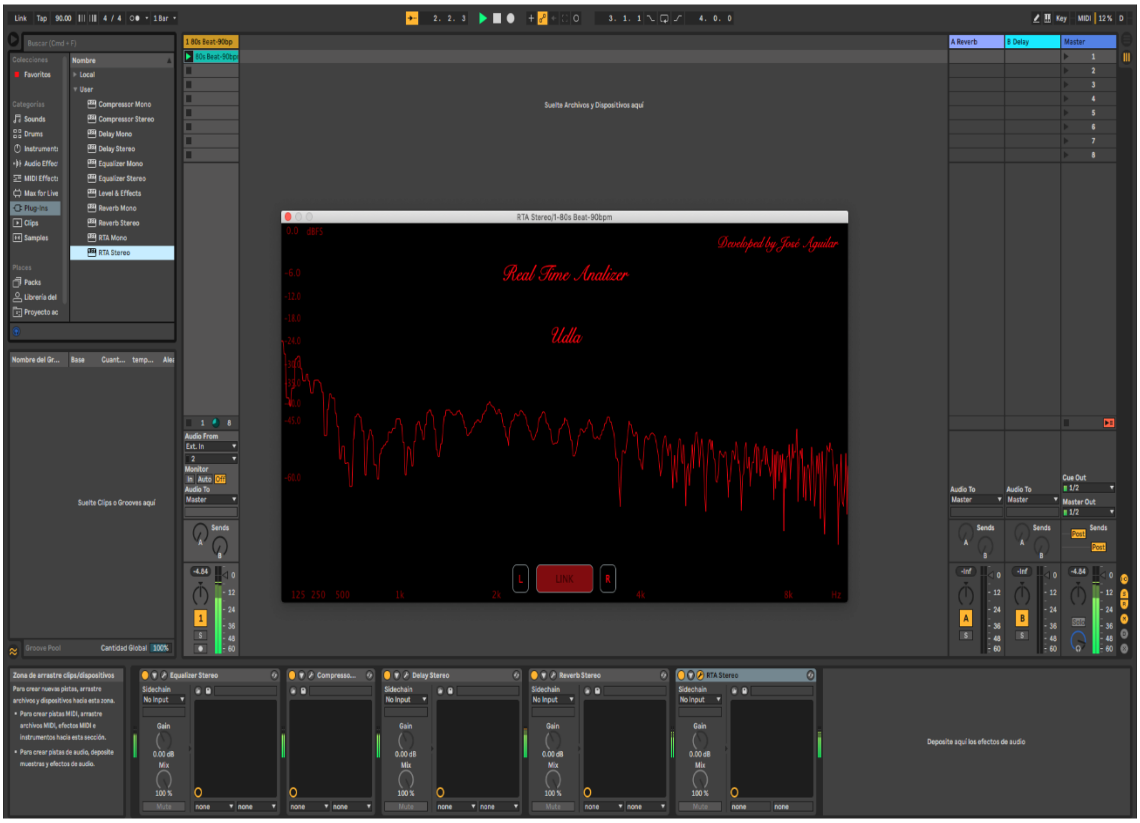

Compressor Mono/1-80s Beat-90bpm
Compressor
Attack
10.00 ms
Release
30.00 ms
Treshold
0.00 dB
Ratio
5.00:1
Mix
100.00 %
In
Out
0.00dB
Suelte Archivos y Dispositivos aquí
Suelte Clips o Grooves aquí
Equalizer Mono
Compressor Mono
Delay Mono
Reverb Mono
RTA Mono
Deposite aquí los efectos de audio
Zona de arrastre de mezclador


Delay Stereo/1-80s Beat-90bpm
Delay
Feedback
Time
Mix
In
Out
4.15dB
4.15dB
-1.43dB
-1.43dB
1/4
50.00 %
50.00 %
SYNC
LINK
LINK
LINK
L
R


Delay Mono/1-80s Beat-90bpm
Delay
Feedback
Time
Mix
In
Out
0.00dB
0.00dB
1000.00 ms
50.00 %
50.00 %
SYNC


Reverb Stereo/1-80s Beat-90bpm
ø
ø
Hi Pass 6dB/Oct
Reverb
Low Pass 6dB/Oct
20.00 Hz
20000.00 Hz
0.00dB
In
0.00dB
Size
Width
Damping
Dry
Wet
0.50
0.50
0.50
50.00 %
50.00 %
0.00dB
Out
0.00dB
LINK
L
LINK
R
LINK
Compressor Mono
Compressor Stereo
Delay Mono
Delay Stereo
Equalizer Mono
Equalizer Stereo
Level & Effects
Reverb Mono
Reverb Stereo
RTA Mono
RTA Stereo
Suelte Archivos y Dispositivos aquí
Suelte Clips o Grooves aquí
Deposite aquí los efectos de audio
Equalizer Stereo
Delay Stereo
Reverb Stereo
RTA Stereo


Reverb Mono/1-80s Beat-90bpm
ø
Hi Pass 6dB/Oct
Reverb
Low Pass 6dB/Oct
20.00 Hz
20000.00 Hz
0.00dB
Size
Width
Damping
Dry
Wet
0.50
0.50
0.50
50.00 %
50.00 %
0.00dB
In
Out
Compressor Mono
Compressor Stereo
Delay Mono
Delay Stereo
Equalizer Mono
Equalizer Stereo
Level & Effects
Reverb Mono
Reverb Stereo
RTA Mono
RTA Stereo
Suelte Archivos y Dispositivos aquí
Suelte Clips o Grooves aquí
Deposite aquí los efectos de audio
Equalizer Mono
Compressor Mono
Delay Mono
Reverb Mono
RTA Mono



Suelte Archivos y Dispositivos aquí

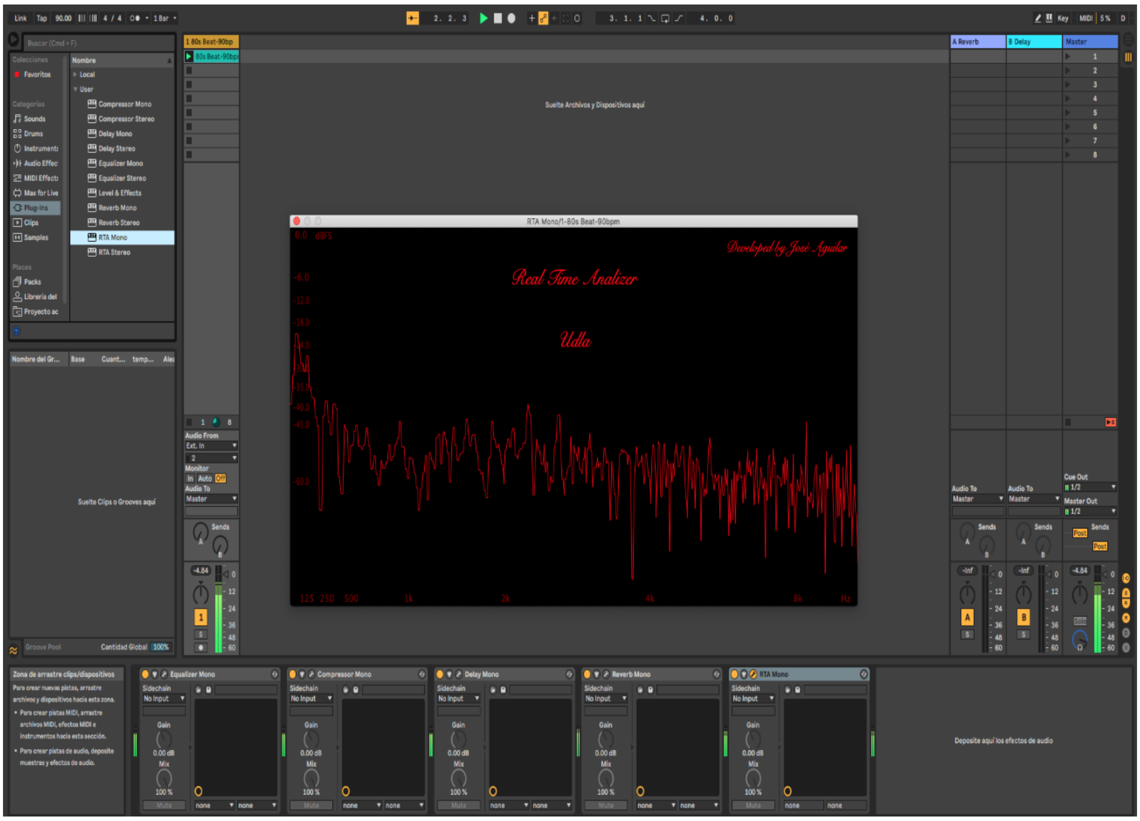
Zona de arrastre clips/dispositivos

Para crear nuevas pistas, arrastre archivos y dispositivos hacia esta zona.

- Para crear pistas MIDI, arrastre archivos MIDI, efectos MIDI e instrumentos hacia esta sección.
- Para crear pistas de audio, deposite muestras y efectos de audio.

Suelte Clips o Grooves aquí

Deposite aquí los efectos de audio



Suelte Archivos y Dispositivos aquí

Zona de arrastre clips/dispositivos

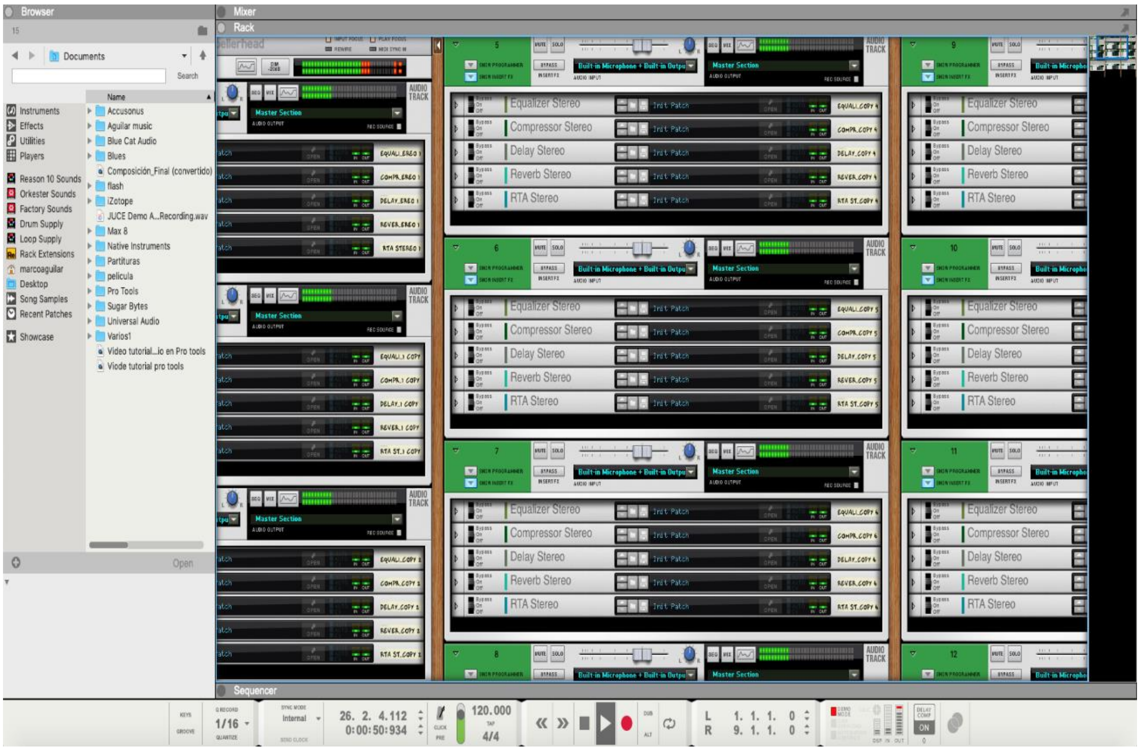
Para crear nuevas pistas, arrastre archivos y dispositivos hacia esta zona.

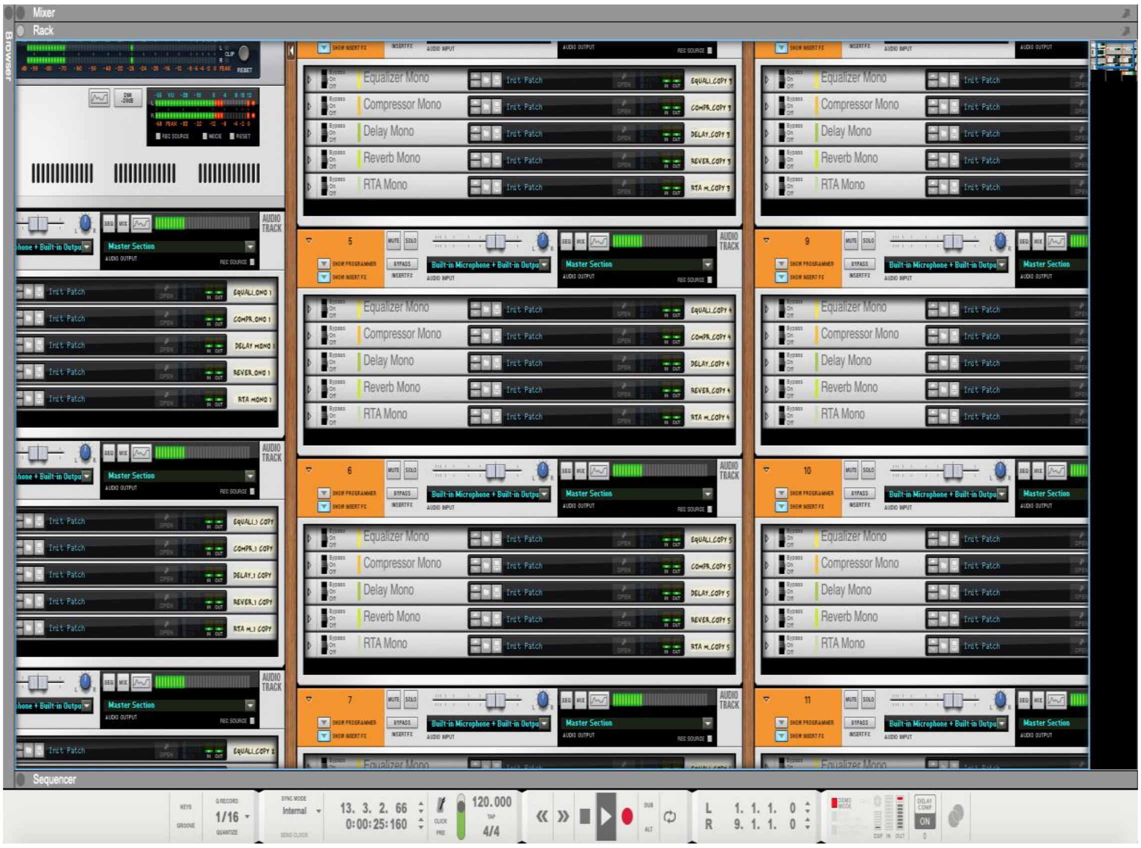
- Para crear pistas MIDI, arrastre archivos MIDI, efectos MIDI e instrumentos hacia esta sección.
- Para crear pistas de audio, deposite muestras y efectos de audio.

Suelte Clips o Grooves aquí

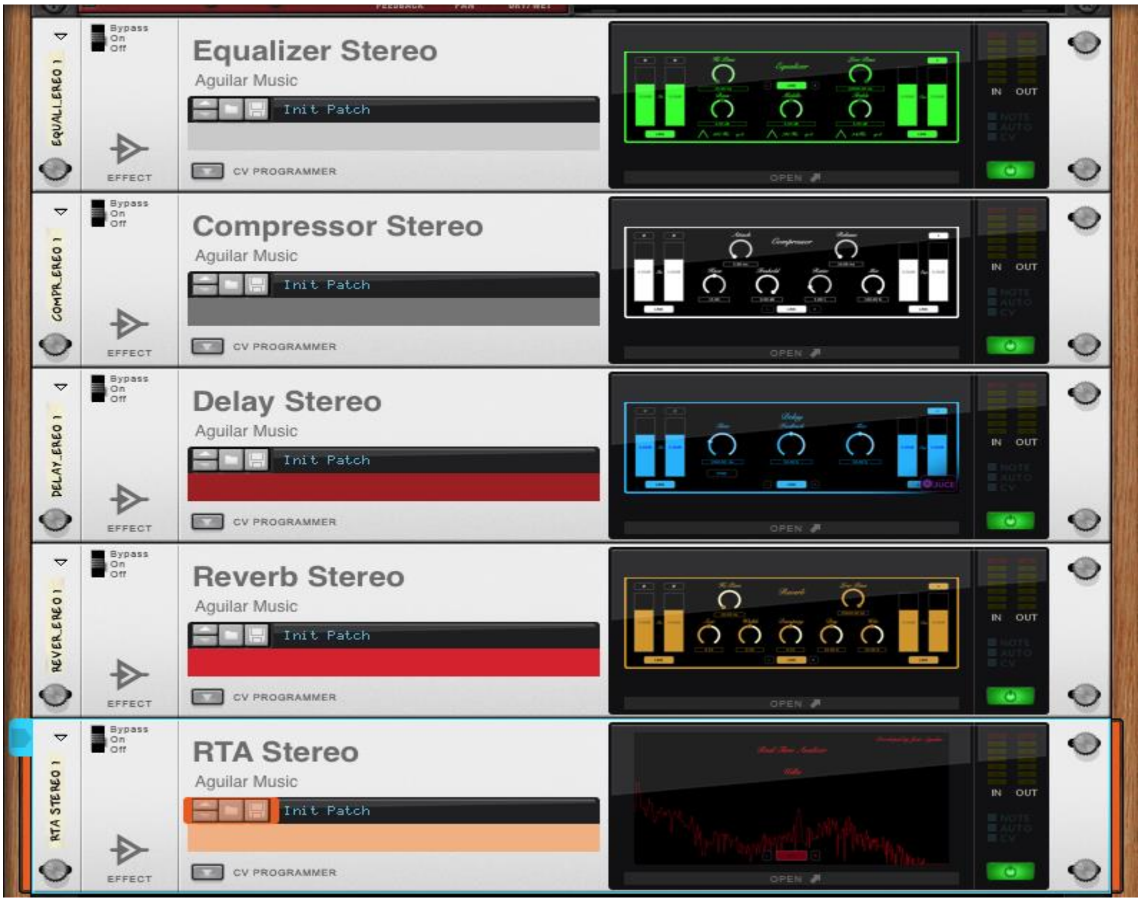
Deposite aquí los efectos de audio

Anexo 7. Funcionamiento en el software Reason 10.

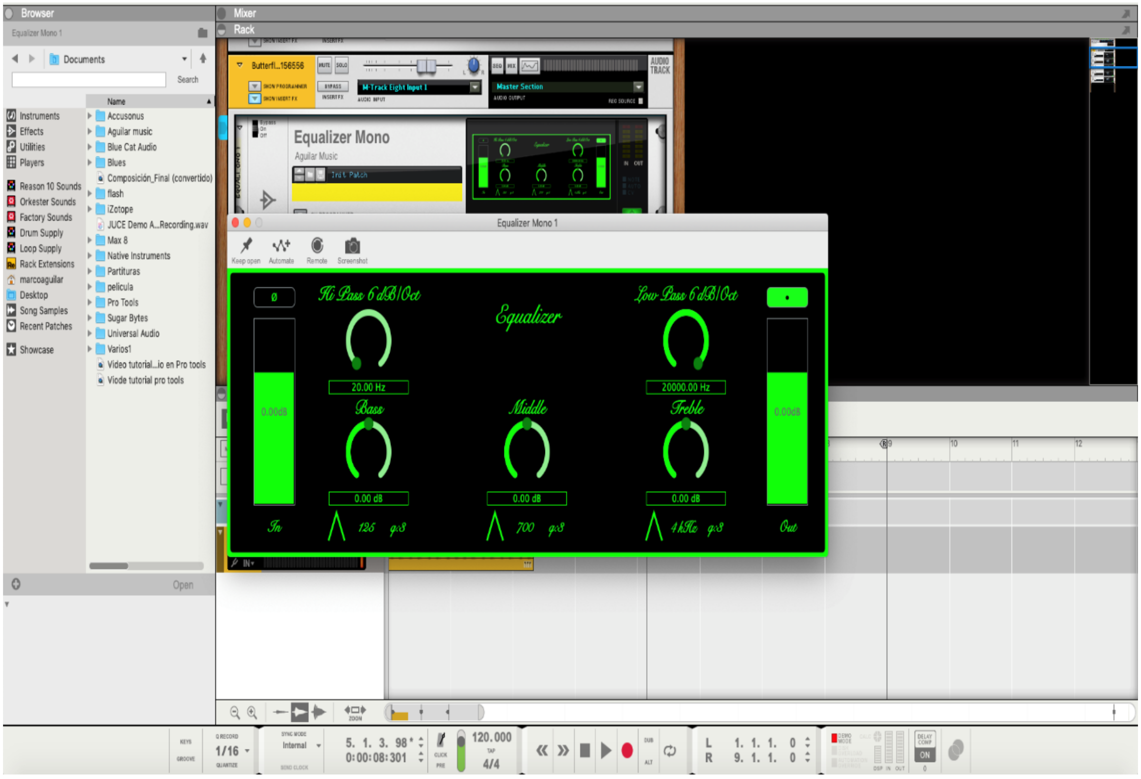




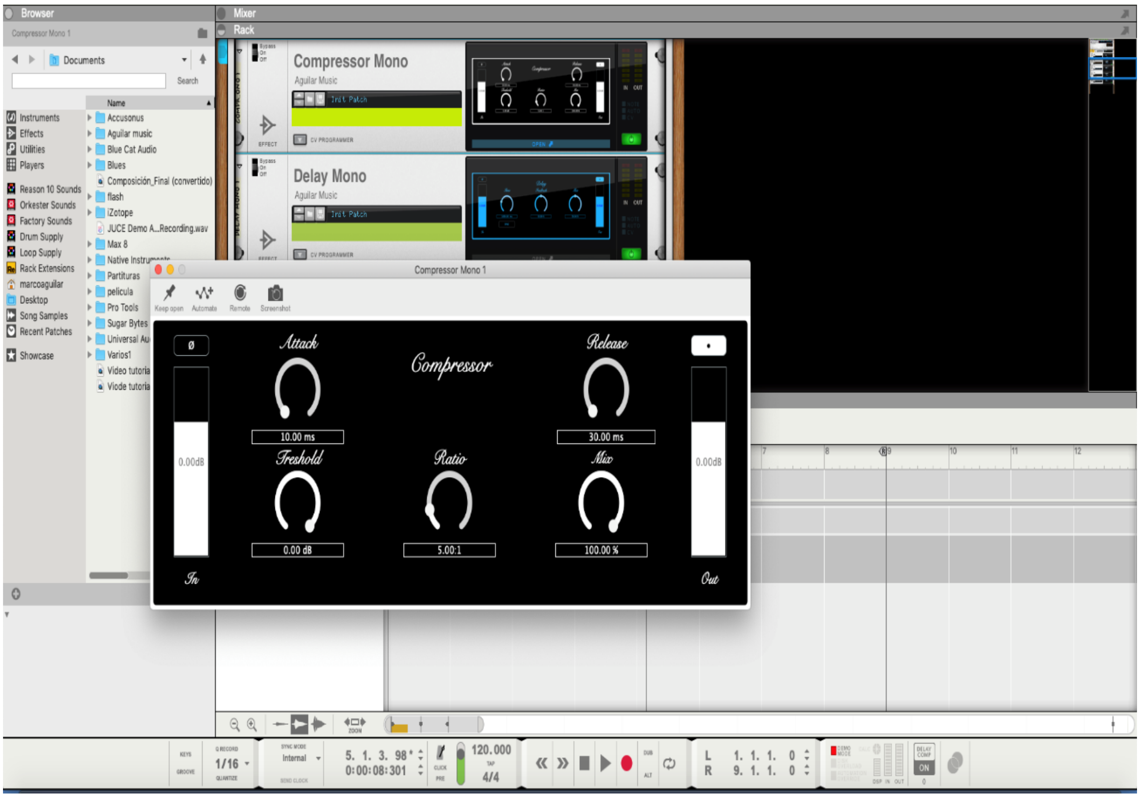

EQUALI..ONO 1
Bypass
On
Off
EFFECT
Equalizer Mono
Aguilar Music
Init Patch
CV PROGRAMMER
OPEN
IN
OUT
COMPR..ONO 1
Compressor Mono
Aguilar Music
Init Patch
DELAY MONO 1
Delay Mono
Aguilar Music
Init Patch
REVER..ONO 1
Reverb Mono
Aguilar Music
Init Patch
RTA MONO 1
RTA Mono
Aguilar Music
Init Patch


EQUALI..EREO 1
Bypass
On
Off
EFFECT
Equalizer Stereo
Aguilar Music
Init Patch
CV PROGRAMMER
OPEN
IN
OUT
COMPR..EREO 1
Compressor Stereo
Aguilar Music
Init Patch
DELAY..EREO 1
Delay Stereo
Aguilar Music
Init Patch
REVER..EREO 1
Reverb Stereo
Aguilar Music
Init Patch
RTA STEREO 1
RTA Stereo
Aguilar Music
Init Patch

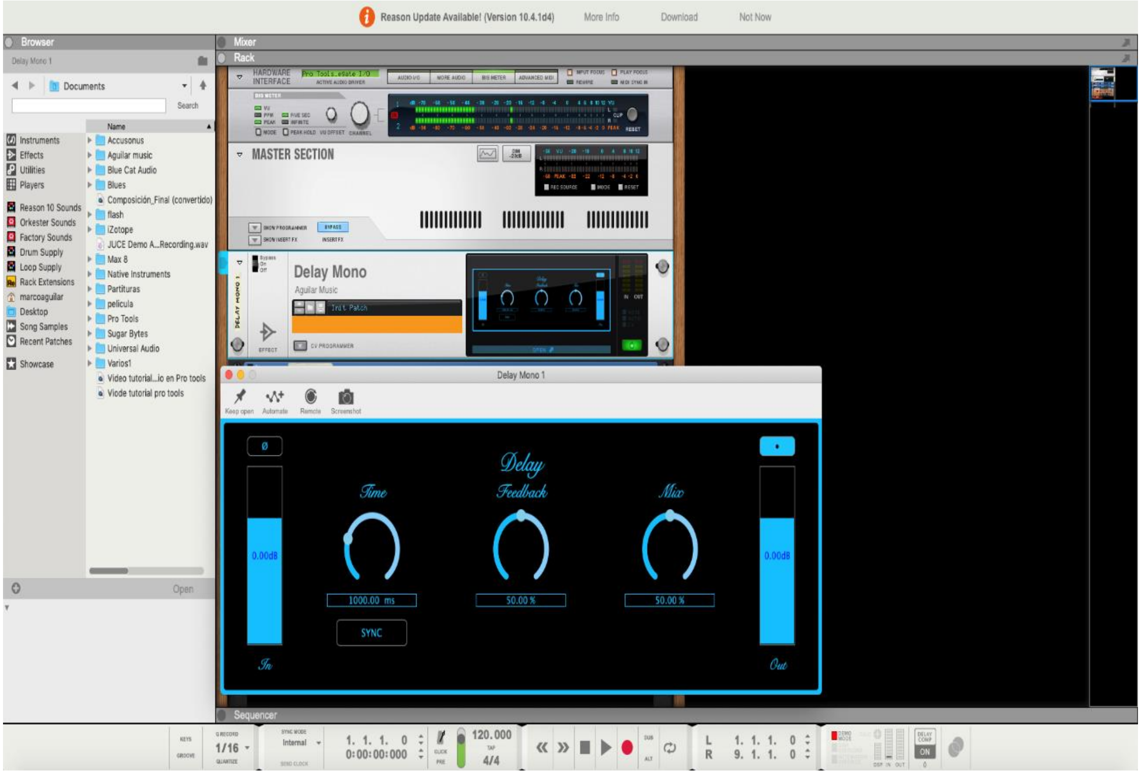

Equalizer Stereo 1
Equalizer Stereo
Aguilar Music
Init Patch
Keep open
Automate
Remote
Screenshot
Hi Pass 6dB/Oct
Equalizer
Low Pass 6dB/Oct
Ø
Ø
20.00 Hz
LINK
20000.00 Hz
Bass
Middle
Treble
In
Out
0.00dB
0.00 dB
0.00 dB
0.00 dB
LINK
125 Hz q:3
700 Hz q:3
4 kHz q:3
LINK
120.000
4/4


Equalizer Mono 1
Equalizer Mono
Aguilar Music
Init Patch
Keep open
Automate
Remote
Screenshot
Hi Pass 6dB/Oct
Equalizer
Low Pass 6dB/Oct
Ø
20.00 Hz
20000.00 Hz
Bass
Middle
Treble
0.00dB
0.00 dB
0.00 dB
0.00 dB
In
125 q:3
700 q:3
4 kHz q:3
Out
120.000
4/4

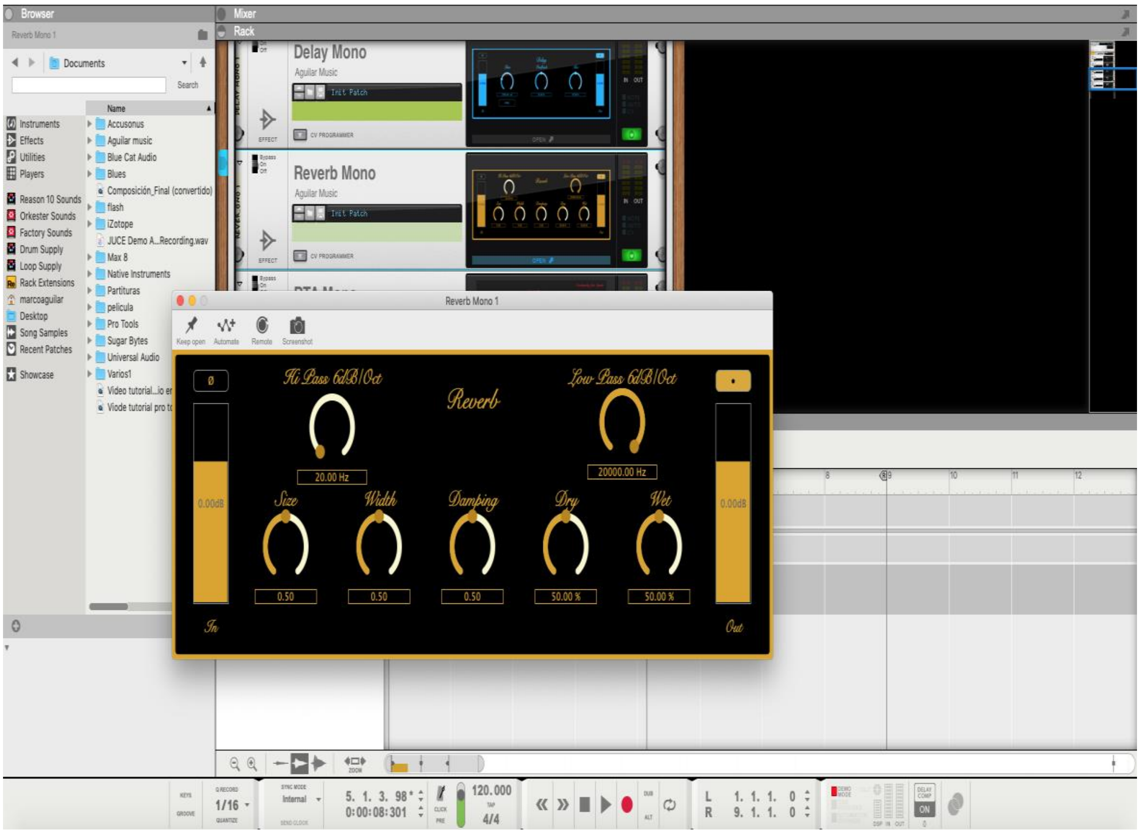

Browser
Mixer
Rack
Compressor Stereo 1
Documents
Compressor Stereo
Aguilar Music
Init Patch
Attack
Release
Compressor
5.00 ms
10.00 ms
Treshold
Ratio
Mix
0.00 dB
5.00:1
100.00 %
In
Out
0.00dB
LINK
L
R
Keep open
Automate
Remote
Screenshot
120.000
4/4
1/16

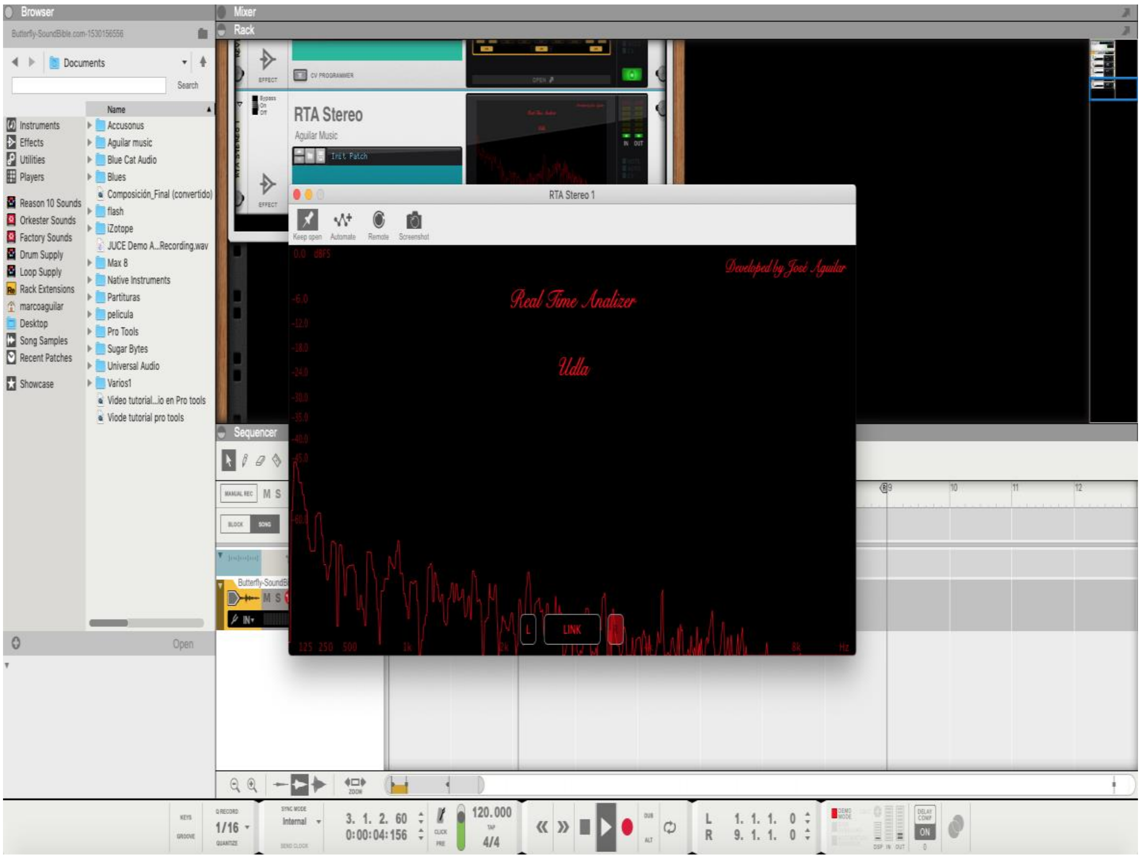

Browser
Mixer
Rack
Compressor Mono 1
Documents
Compressor Mono
Delay Mono
Aguilar Music
Init Patch
Attack
Release
Compressor
10.00 ms
30.00 ms
Treshold
Ratio
Mix
0.00 dB
5.00:1
100.00 %
In
Out
0.00dB
Keep open
Automate
Remote
Screenshot
120.000
4/4
1/16

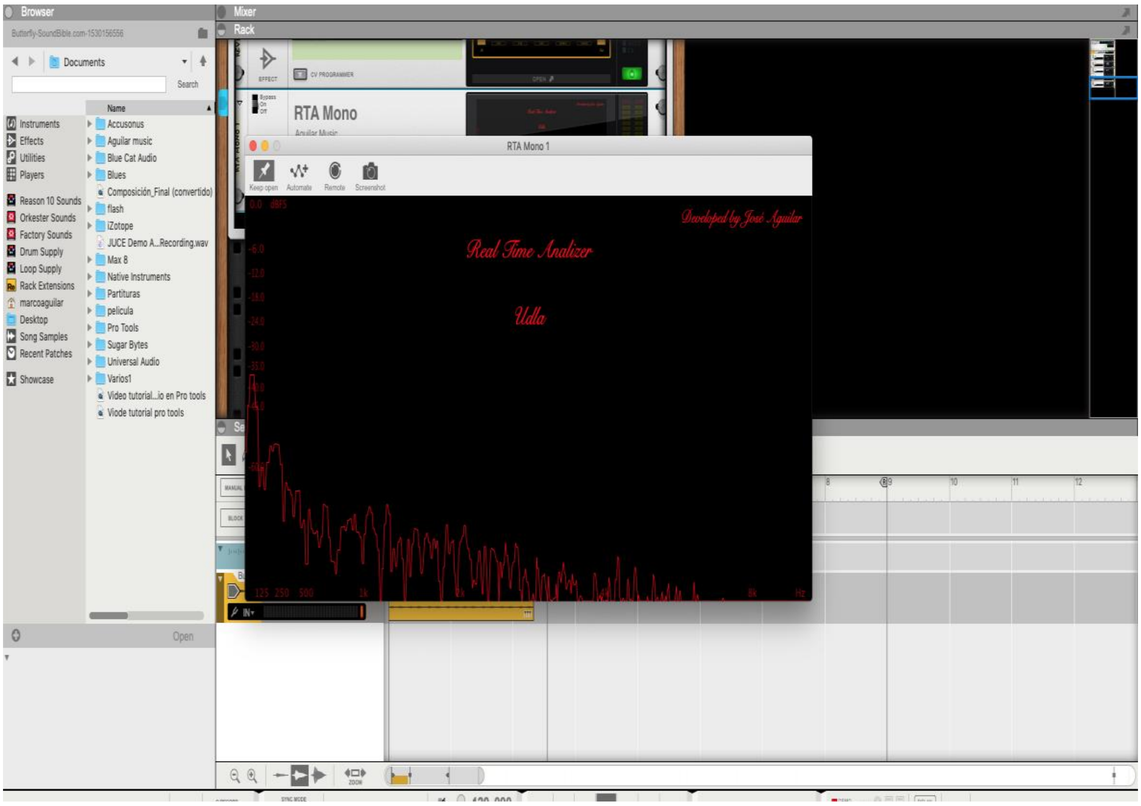

Reason Update Available! (Version 10.4.1d4)
More Info
Download
Not Now
Browser
Browse patches for: Delay Stereo 1
Documents
Search
Name
Instruments
Effects
Utilities
Players
Reason 10 Sounds
Orkester Sounds
Factory Sounds
Drum Supply
Loop Supply
Rack Extensions
marcoaguilar
Desktop
Song Samples
Recent Patches
Showcase
Accusonus
Aguilar music
Blue Cat Audio
Blues
flash
iZotope
Max 8
Native Instruments
Partituras
pelicula
Pro Tools
Sugar Bytes
Universal Audio
Varios1
Mixer
Rack
HARDWARE INTERFACE
MASTER SECTION
Delay Stereo
Aguilar Music
Init Patch
Delay Stereo 1
Keep open
Automate
Remote
Screenshot
Delay
Time
Feedback
Mix
In
Out
0.00dB
1000.00 ms
50.00 %
50.00 %
SYNC
LINK
L
LINK
R
LINK
Sequencer
1/16
Internal
1. 1. 1. 0
0:00:00:000
120.000
4/4
L
R
1. 1. 1. 0
9. 1. 1. 0
ON


Reason Update Available! (Version 10.4.1d4)
More Info
Download
Not Now
Browser
Delay Mono 1
Documents
Search
Name
Instruments
Effects
Utilities
Players
Reason 10 Sounds
Orkester Sounds
Factory Sounds
Drum Supply
Loop Supply
Rack Extensions
marcoaguilar
Desktop
Song Samples
Recent Patches
Showcase
Accusonus
Aguilar music
Blue Cat Audio
Blues
Composición_Final (convertido)
flash
iZotope
JUCE Demo A...Recording.wav
Max 8
Native Instruments
Partituras
pelicula
Pro Tools
Sugar Bytes
Universal Audio
Varios1
Video tutorial...io en Pro tools
Viode tutorial pro tools
Open
Mixer
Rack
HARDWARE INTERFACE
MASTER SECTION
Delay Mono
Aguilar Music
Init Patch
Delay Mono 1
Keep open
Automate
Remote
Screenshot
Delay
Time
Feedback
Mix
0.00dB
1000.00 ms
50.00 %
50.00 %
SYNC
In
Out
Sequencer
1/16
Internal
1. 1. 1. 0
0:00:00:000
120.000
4/4
L
R
1. 1. 1. 0
9. 1. 1. 0
ON

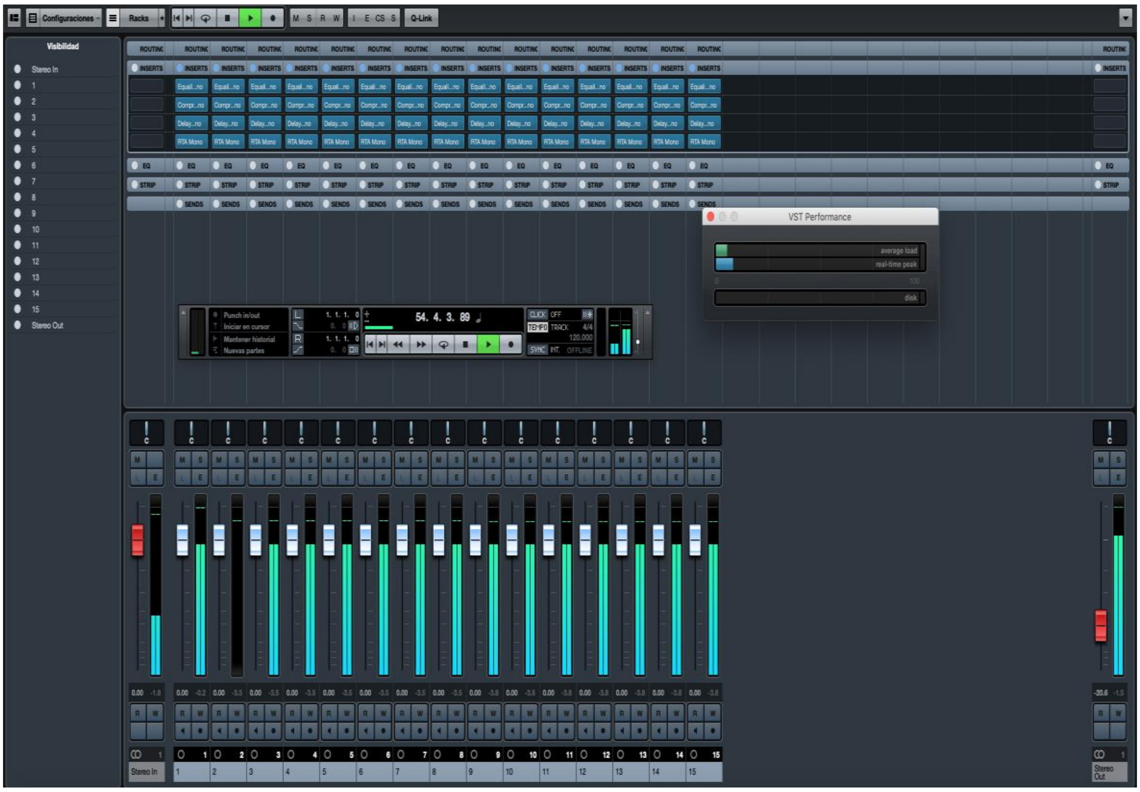

Browser
Mixer
Rack
Reverb Stereo
Aguilar Music
Init Patch
Reverb Stereo 1
Keep open
Automate
Remote
Screenshot
Hi Pass 6dB/Oct
Reverb
Low Pass 6dB/Oct
20.00 Hz
20000.00 Hz
In
Out
0.00dB
Size
Width
Damping
Dry
Wet
0.50
50.00 %
LINK
L
R
Instruments
Effects
Utilities
Players
Reason 10 Sounds
Orkester Sounds
Factory Sounds
Drum Supply
Loop Supply
Rack Extensions
marcoaguilar
Desktop
Song Samples
Recent Patches
Showcase
Accusonus
Aguilar music
Blue Cat Audio
Blues
Composición_Final (convertido)
flash
iZotope
JUCE Demo A...Recording.wav
Max 8
Native Instruments
Partituras
pelicula
Pro Tools
Sugar Bytes
Universal Audio
Varios1
Video tutorial...io en Pro tools
Viode tutorial pro tools
Open
1/16
Internal
3. 3. 3. 17
0:00:05:259
120.000
4/4
1. 1. 1. 0
9. 1. 1. 0
ON

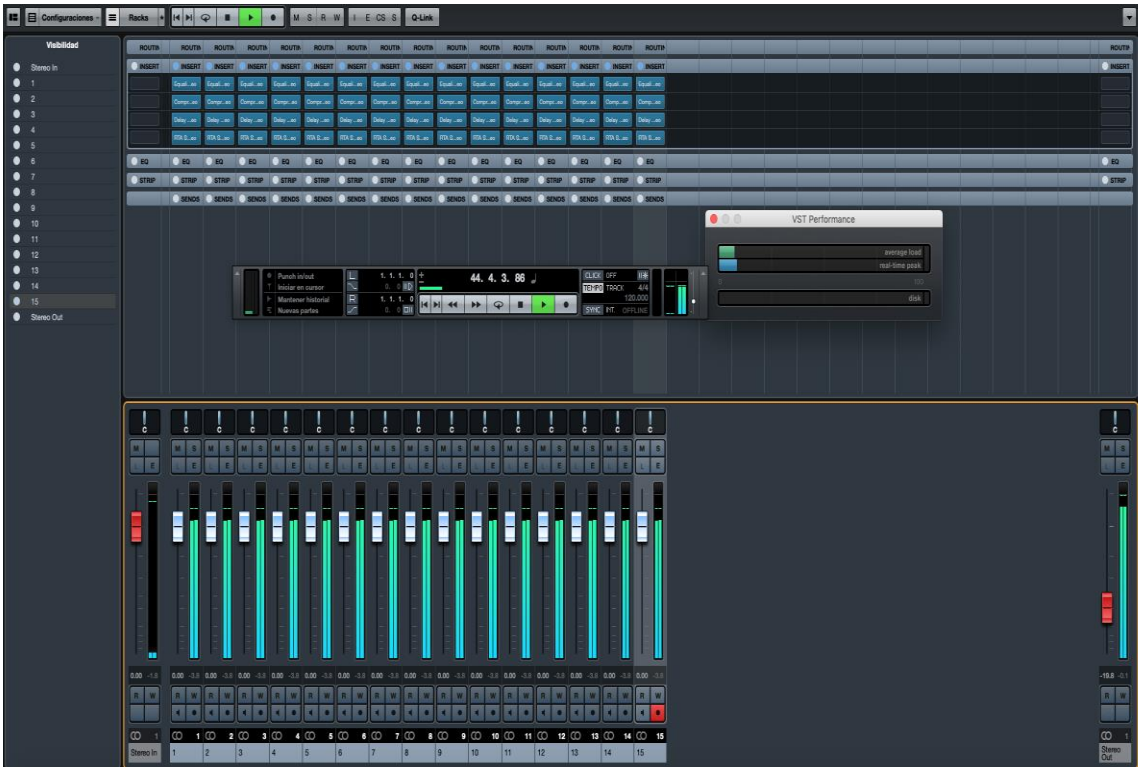

Browser
Mixer
Rack
Delay Mono
Aguilar Music
Init Patch
Reverb Mono
Aguilar Music
Init Patch
Reverb Mono 1
Keep open
Automate
Remote
Screenshot
Hi Pass 6dB/Oct
Reverb
Low Pass 6dB/Oct
20.00 Hz
20000.00 Hz
0.00dB
Size
Width
Damping
Dry
Wet
0.50
50.00 %
In
Out
1/16
Internal
5. 1. 3. 98
0:00:08:301
120.000
4/4
1. 1. 1. 0
9. 1. 1. 0
ON


Browser
Mixer
Rack
Documents
Search
Instruments
Effects
Utilities
Players
Reason 10 Sounds
Orkester Sounds
Factory Sounds
Drum Supply
Loop Supply
Rack Extensions
marcoaguilar
Desktop
Song Samples
Recent Patches
Showcase
Name
Accusonus
Aguilar music
Blue Cat Audio
Blues
Composición_Final (convertido)
flash
iZotope
JUCE Demo A...Recording.wav
Max 8
Native Instruments
Partituras
pelicula
Pro Tools
Sugar Bytes
Universal Audio
Varios1
Video tutorial...io en Pro tools
Viode tutorial pro tools
Open
RTA Stereo
Aguilar Music
Init Patch
RTA Stereo 1
Keep open
Automate
Remote
Screenshot
Developed by José Aguilar
Real Time Analizer
Udla
L
LINK
R
125
250
500
1k
8k
Hz
Sequencer
9
10
11
12
1/16
Internal
3. 1. 2. 60
0:00:04:156
120.000
4/4
L 1. 1. 1. 0
R 9. 1. 1. 0
ON

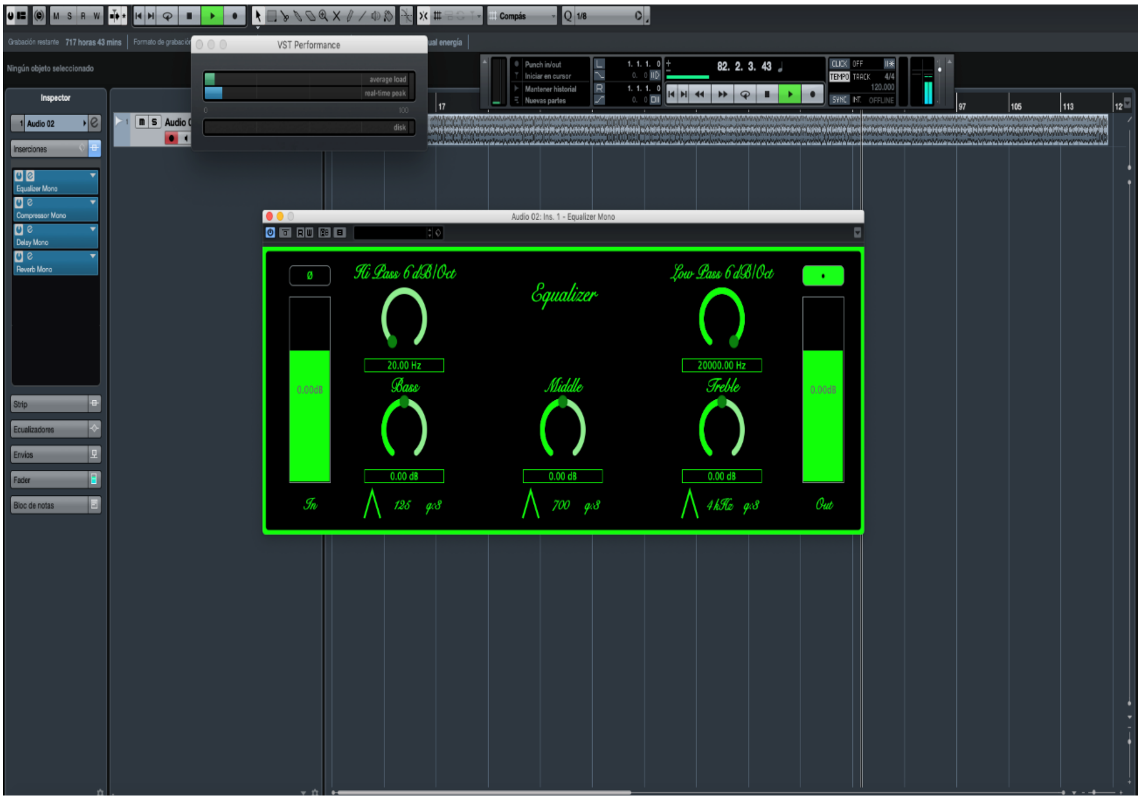

Browser
Mixer
Rack
Documents
Search
Instruments
Effects
Utilities
Players
Reason 10 Sounds
Orkester Sounds
Factory Sounds
Drum Supply
Loop Supply
Rack Extensions
marcoaguilar
Desktop
Song Samples
Recent Patches
Showcase
Name
Accusonus
Aguilar music
Blue Cat Audio
Blues
Composición_Final (convertido)
flash
iZotope
JUCE Demo A...Recording.wav
Max 8
Native Instruments
Partituras
pelicula
Pro Tools
Sugar Bytes
Universal Audio
Varios1
Video tutorial...io en Pro tools
Viode tutorial pro tools
Open
RTA Mono
Aguilar Music
RTA Mono 1
Keep open
Automate
Remote
Screenshot
Developed by José Aguilar
Real Time Analizer
Udla
125
250
500
1k
8k
Hz
8
9
10
11
12

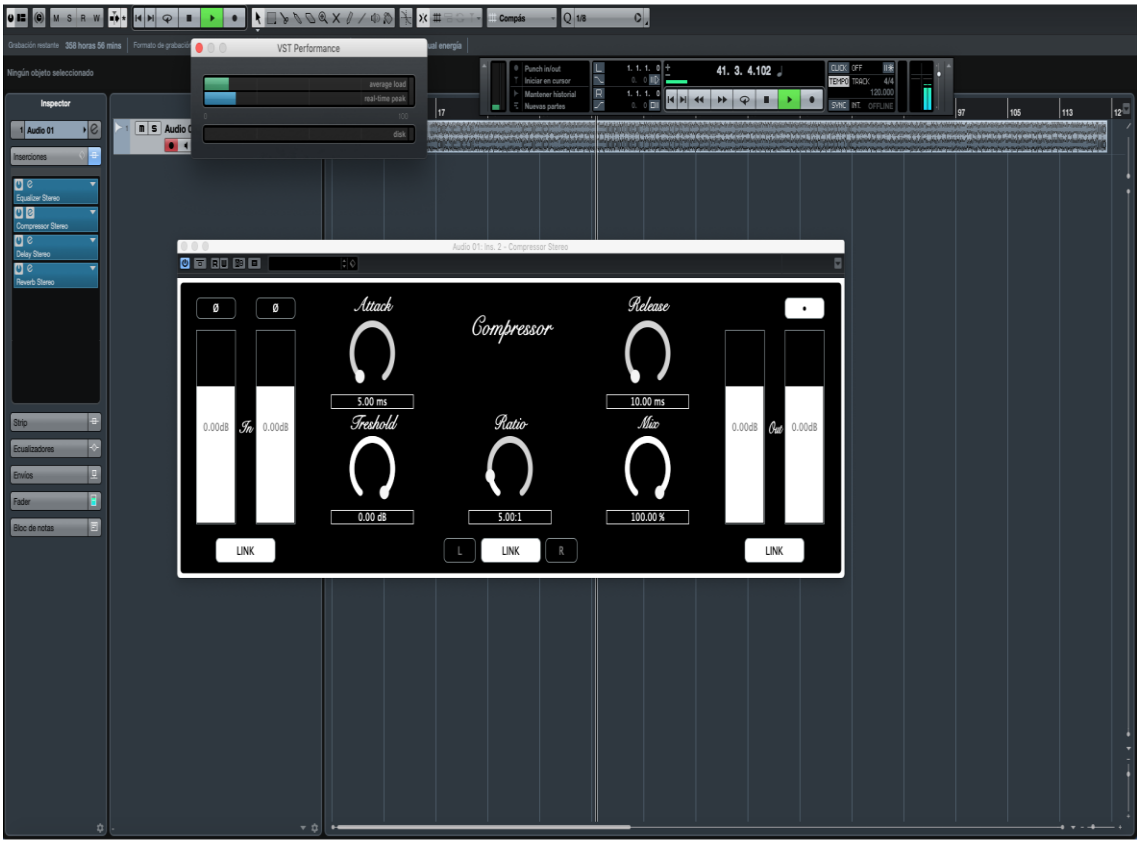
Anexo 8. Funcionamiento en el software Cubase 8.






VST Performance
average load
real-time peak
disk
Ningún objeto seleccionado
Inspector
Audio 01
Inserciones
Equalizer Stereo
Compressor Stereo
Delay Stereo
Reverb Stereo
Strip
Ecualizadores
Envíos
Fader
Bloc de notas
Audio 01: Ins. 1 - Equalizer Stereo
Hi Pass 6 dB/Oct
Equalizer
Low Pass 6 dB/Oct
20.00 Hz
20000.00 Hz
L
LINK
R
Bass
Middle
Treble
0.00dB
In
Out
0.00 dB
LINK
125 Hz q:3
700 Hz q:3
4 kHz q:3


VST Performance
average load
real-time peak
disk
Ningún objeto seleccionado
Inspector
Audio 02
Inserciones
Equalizer Mono
Compressor Mono
Delay Mono
Reverb Mono
Strip
Ecualizadores
Envíos
Fader
Bloc de notas
Audio 02: Ins. 1 - Equalizer Mono
Hi Pass 6 dB/Oct
Equalizer
Low Pass 6 dB/Oct
20.00 Hz
20000.00 Hz
Bass
Middle
Treble
0.00dB
0.00 dB
In
125 q:3
700 q:3
4 kHz q:3
Out



Audio 01: Ins. 2 - Compressor Stereo

Attack 5.00 ms

Release 10.00 ms

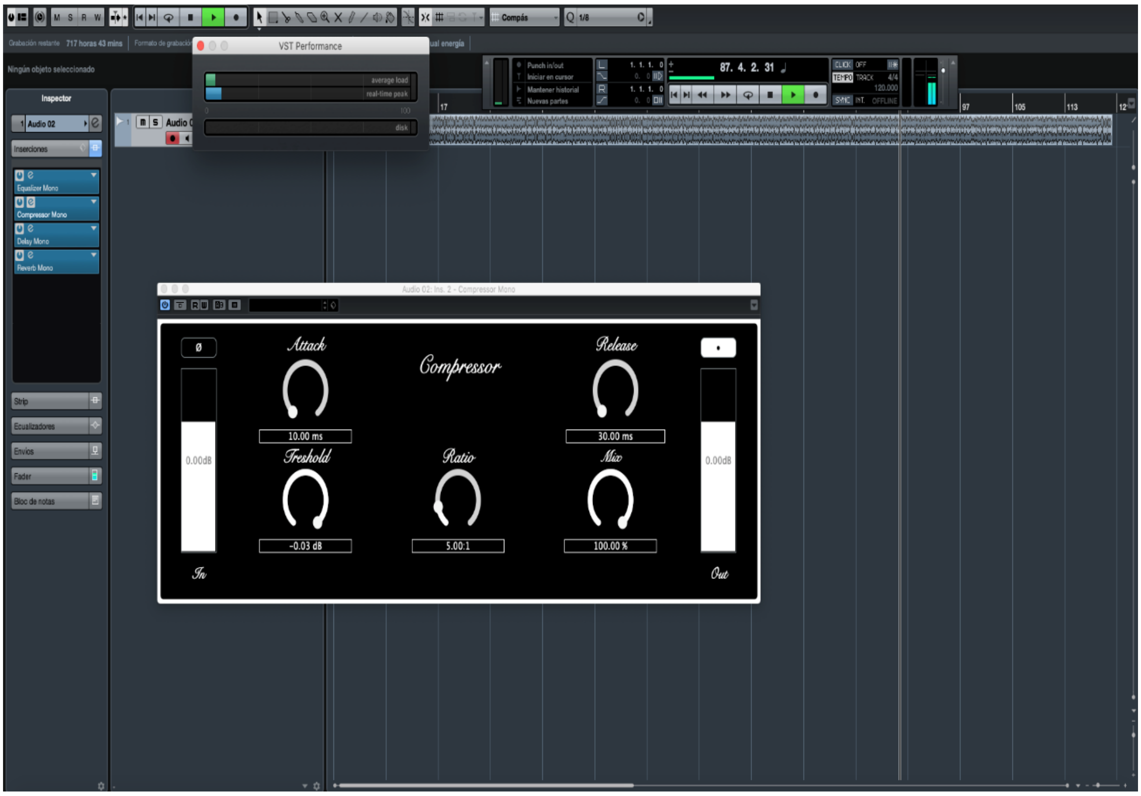
Treshold 0.00 dB

Ratio 5.00:1

Mix 100.00 %

In 0.00dB 0.00dB

Out 0.00dB 0.00dB



Audio 02: Ins. 2 - Compressor Mono

Attack 10.00 ms

Release 30.00 ms

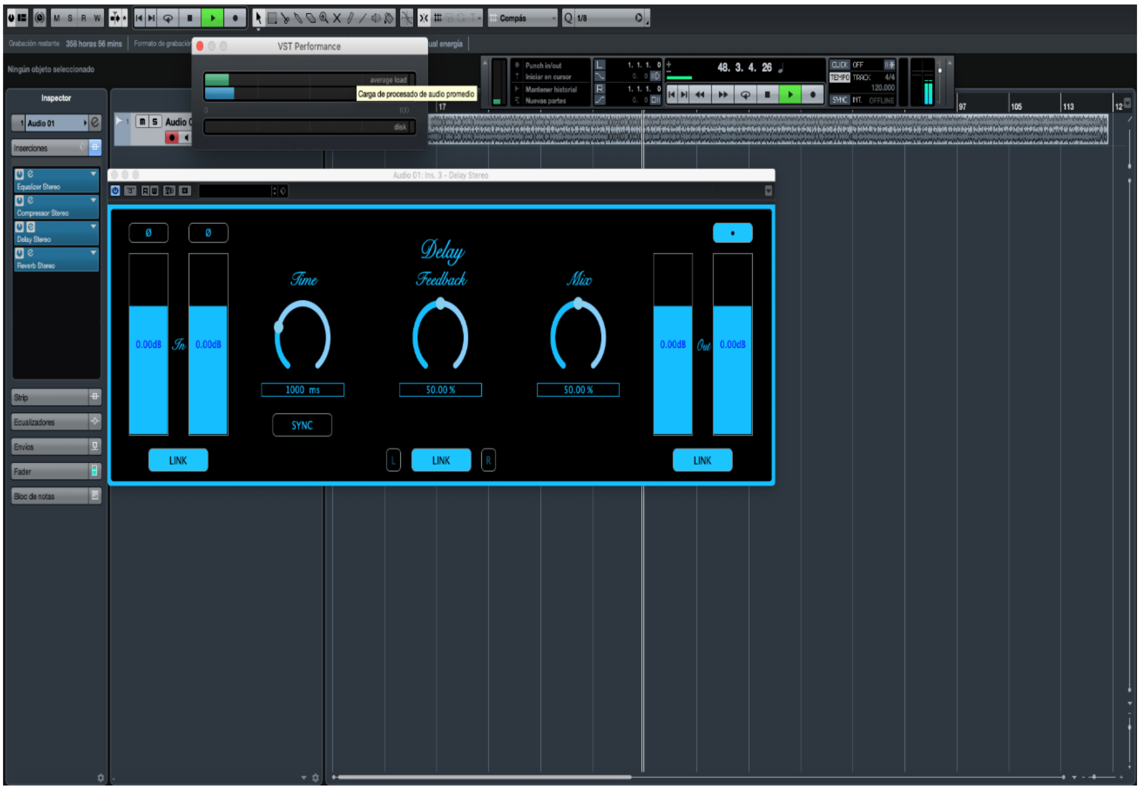
Treshold -0.03 dB

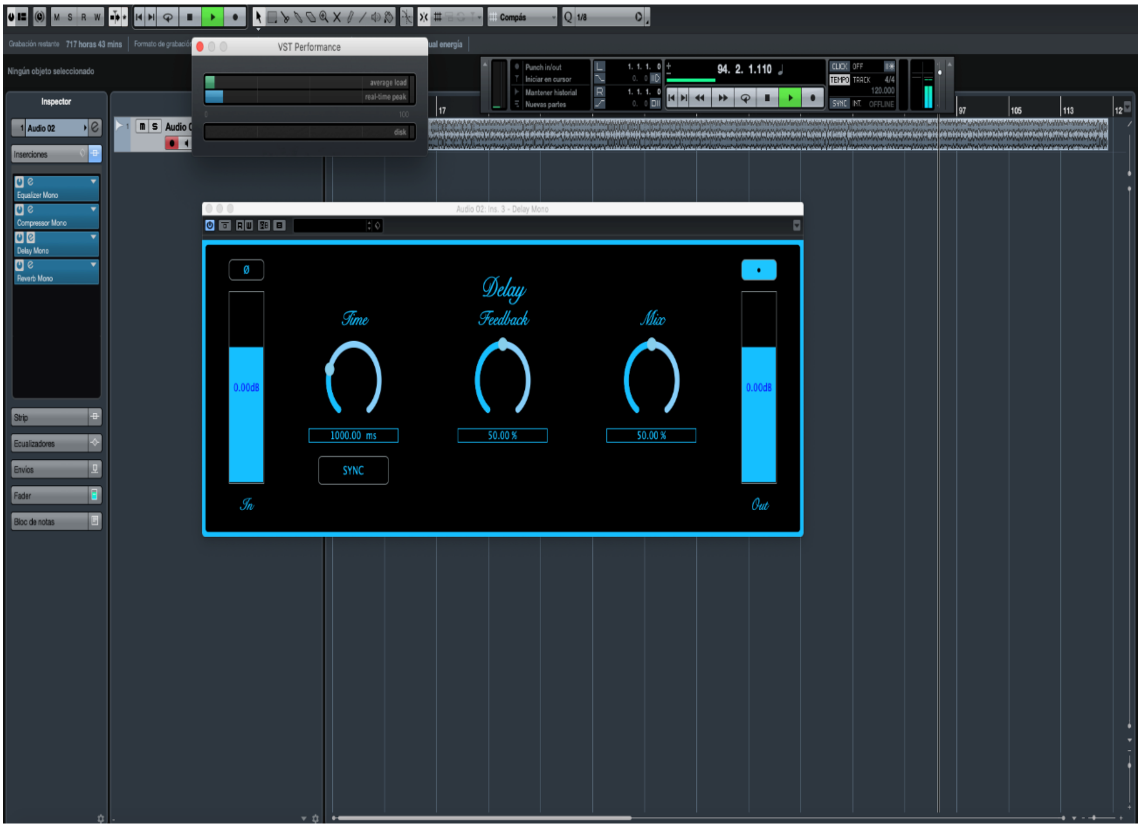
Ratio 5.00:1

Mix 100.00 %

In 0.00dB

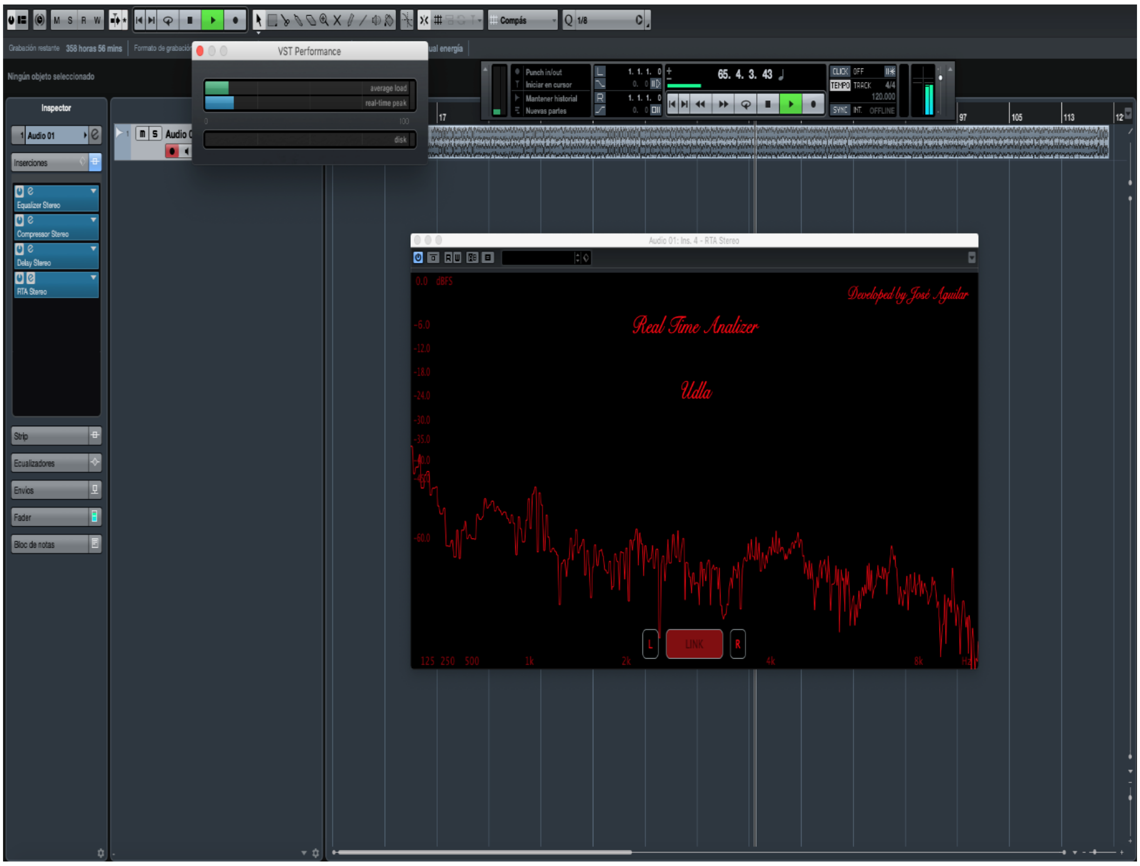
Out 0.00dB

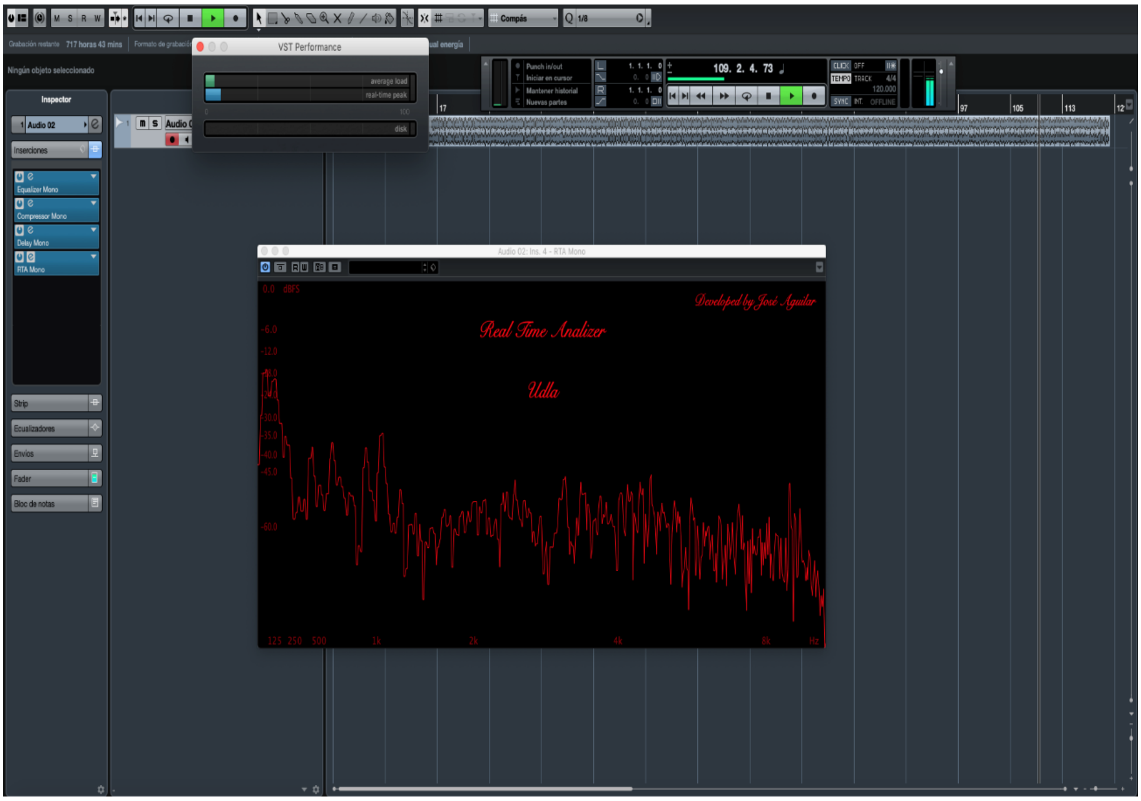

VST Performance
average load
Carga de procesado de audio promedio
disk
Inspector
Audio 01
Inserciones
Equalizer Stereo
Compresor Stereo
Delay Stereo
Reverb Stereo
Strip
Ecualizadores
Envíos
Fader
Bloc de notas
Audio 01: Ins. 3 - Delay Stereo
Delay
Time
Feedback
Mix
0.00dB
In
Out
1000 ms
50.00 %
50.00 %
SYNC
LINK
L
LINK
R
LINK


VST Performance
average load
real-time peak
disk
Inspector
Audio 02
Inserciones
Equalizer Mono
Compresor Mono
Delay Mono
Reverb Mono
Strip
Ecualizadores
Envíos
Fader
Bloc de notas
Audio 02: Ins. 3 - Delay Mono
Delay
Time
Feedback
Mix
0.00dB
1000.00 ms
50.00 %
50.00 %
SYNC
In
Out

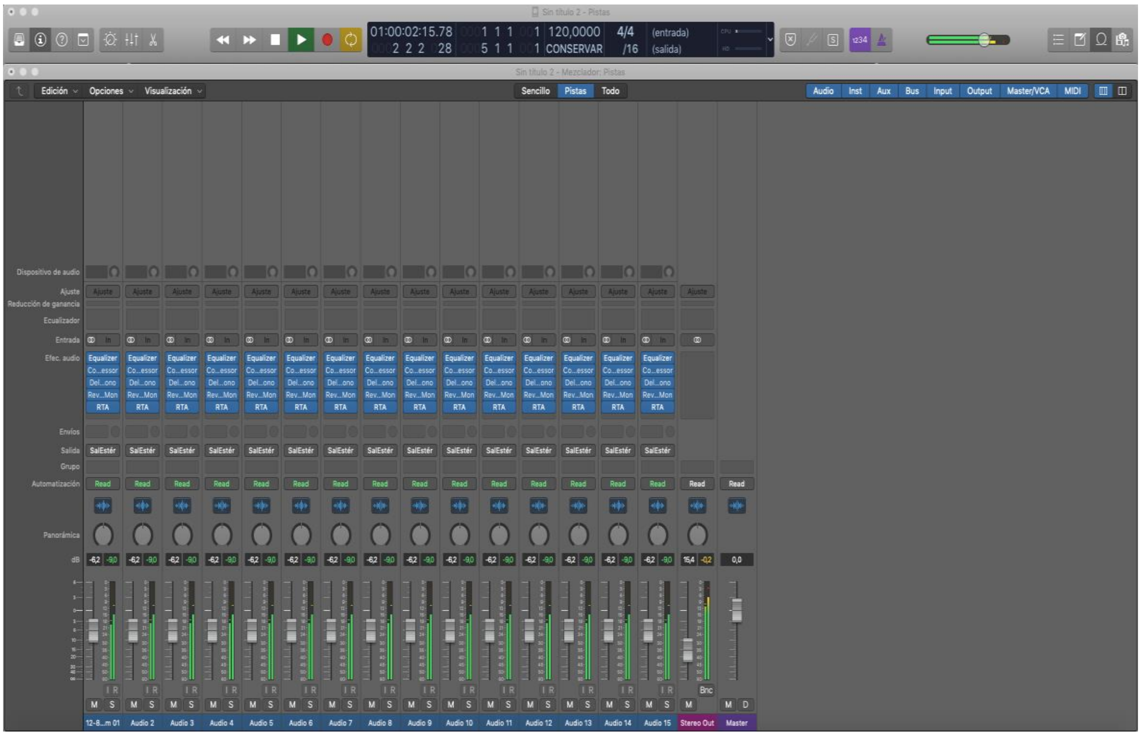

M S R W
Compás
Q 1/8
Grabación restante 358 horas 56 mins
Formato de grabación
VST Performance
average load
real-time peak
disk
Ningún objeto seleccionado
Inspector
Audio 01
Inserciones
Equalizer Stereo
Compressor Stereo
Delay Stereo
Reverb Stereo
Strip
Ecualizadores
Envíos
Fader
Bloc de notas
Punch in/out
Iniciar en cursor
Mantener historial
Nuevas partes
CLICK OFF
TEMPO TRACK 4/4
120.000
SYNC INT. OFFLINE
17
97
105
113
Audio 01: Ins. 4 - Reverb Stereo
Reverb
Hi Pass 6dB/Oct
20.00 Hz
Low Pass 6dB/Oct
20000.00 Hz
In
0.00dB
0.00dB
Size
0.50
Width
0.50
Damping
0.50
Dry
50.00 %
Wet
50.00 %
Out
0.00dB
0.00dB
LINK
L
LINK
R
LINK

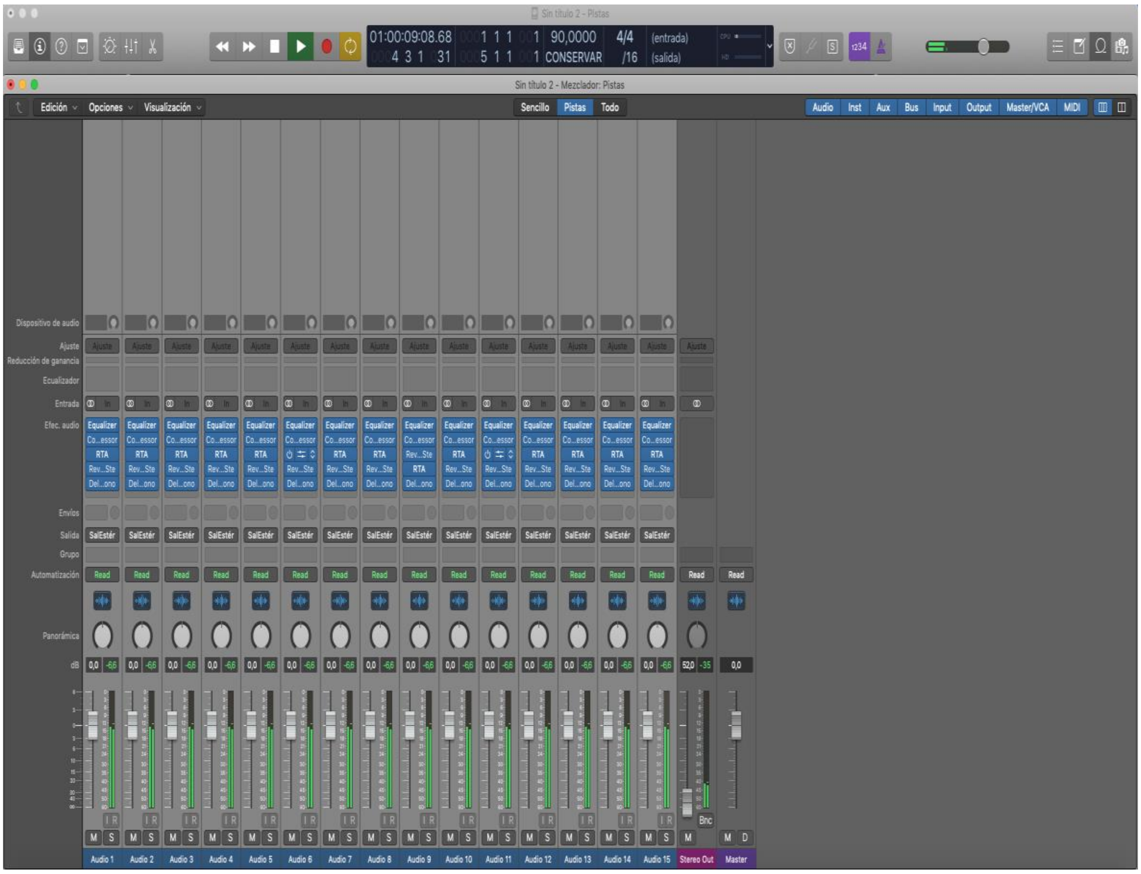

M S R W
Compás
Q 1/8
Grabación restante 717 horas 43 mins
Formato de grabación
VST Performance
average load
real-time peak
disk
Ningún objeto seleccionado
Inspector
Audio 02
Inserciones
Equalizer Mono
Compressor Mono
Delay Mono
Reverb Mono
Strip
Ecualizadores
Envíos
Fader
Bloc de notas
Punch in/out
Iniciar en cursor
Mantener historial
Nuevas partes
100. 3. 2. 30
CLICK OFF
TEMPO TRACK 4/4
120.000
SYNC INT. OFFLINE
17
97
105
113
Audio 02: Ins. 4 - Reverb Mono
Reverb
Hi Pass 6dB/Oct
20.00 Hz
Low Pass 6dB/Oct
20000.00 Hz
0.00dB
Size
0.50
Width
0.50
Damping
0.50
Dry
50.00 %
Wet
50.00 %
0.00dB
In
Out


VST Performance
average load
real-time peak
disk
Inspector
Audio 01
Inserciones
Equalizer Stereo
Compressor Stereo
Delay Stereo
RTA Stereo
Strip
Ecualizadores
Envíos
Fader
Bloc de notas
Audio 01: Ins. 4 - RTA Stereo
Developed by José Aguilar
Real Time Analizer
Udla
0.0 dBFS
-6.0
-12.0
-18.0
-24.0
-30.0
-60.0
L
LNK
R
125 250 500
1k
2k
4k
8k
Hz


VST Performance
average load
real-time peak
disk
Inspector
Audio 02
Inserciones
Equalizer Mono
Compressor Mono
Delay Mono
RTA Mono
Strip
Ecualizadores
Envíos
Fader
Bloc de notas
Audio 02: Ins. 4 - RTA Mono
Developed by José Aguilar
Real Time Analizer
Udla
0.0 dBFS
-6.0
-12.0
-60.0
125 250 500
1k
2k
4k
8k
Hz

Anexo 9. Funcionamiento en el software Logic Pro X.



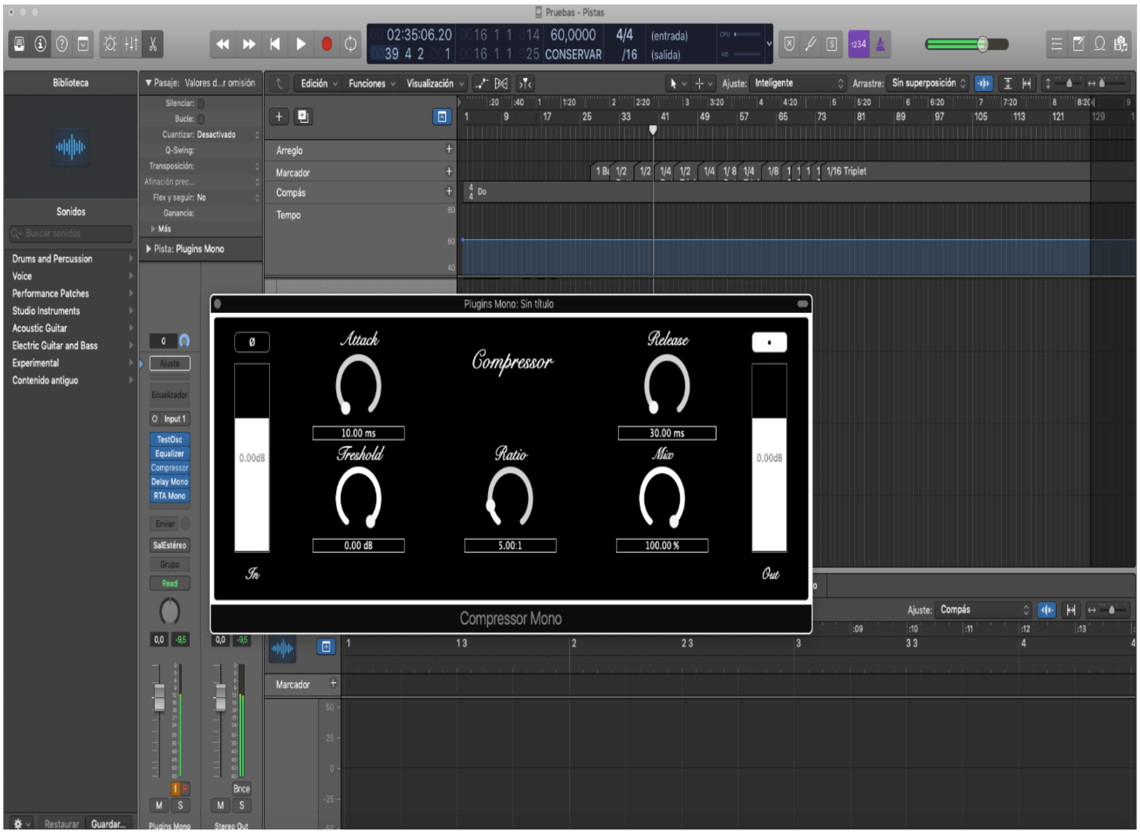



Pruebas - Pistas
Plugins Estéreo
Hi Pass 6 dB/Oct
Equalizer
Low Pass 6 dB/Oct
20.00 Hz
LINK
20000.00 Hz
L
R
0.00dB
In
Out
Bass
Middle
Treble
0.00 dB
LINK
125 Hz
q:3
700 Hz
q:3
4 kHz
q:3
Equalizer Stereo
Plugins Estéreo
Stereo Out


Pruebas - Pistas
Plugins Mono: Sin título
Hi Pass 6 dB/Oct
Equalizer
Low Pass 6 dB/Oct
20.00 Hz
20000.00 Hz
0.00dB
Bass
Middle
Treble
0.00 dB
In
125
q:3
700
q:3
4 kHz
q:3
Out
Equalizer Mono
Plugins Mono
Stereo Out

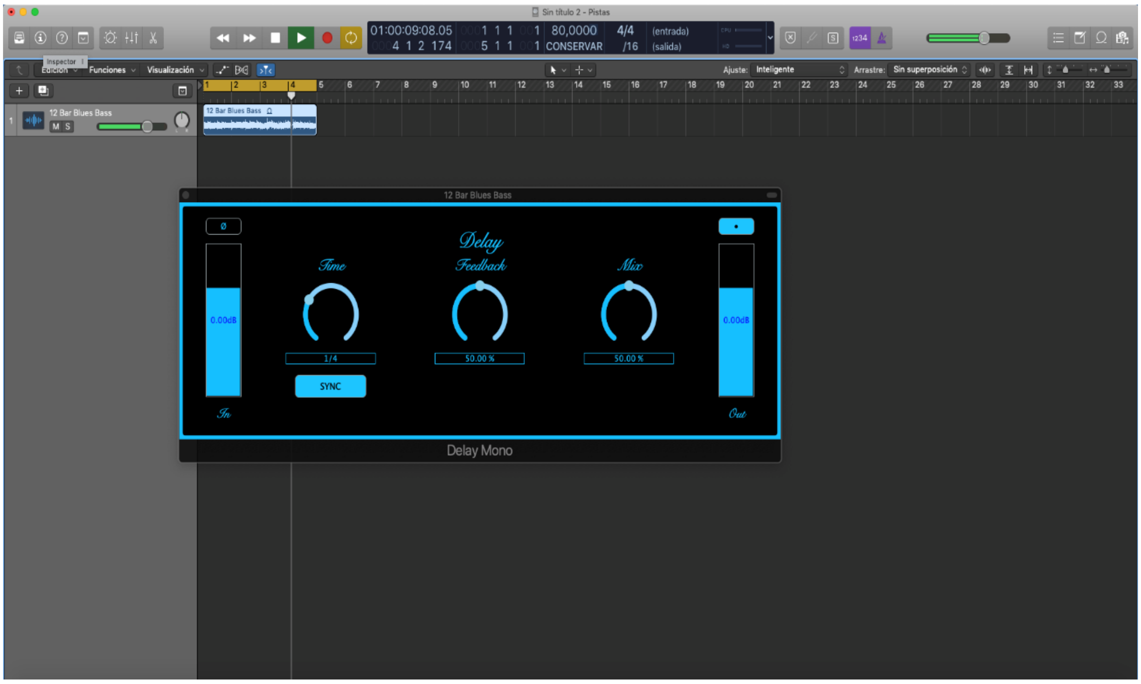
Plugins Estéreo

Attack: 5.00 ms
Release: 10.00 ms
Treshold: 0.00 dB
Ratio: 5.00:1
Mix: 100.00 %
In: 0.00dB 0.00dB
Out: 0.00dB 0.00dB

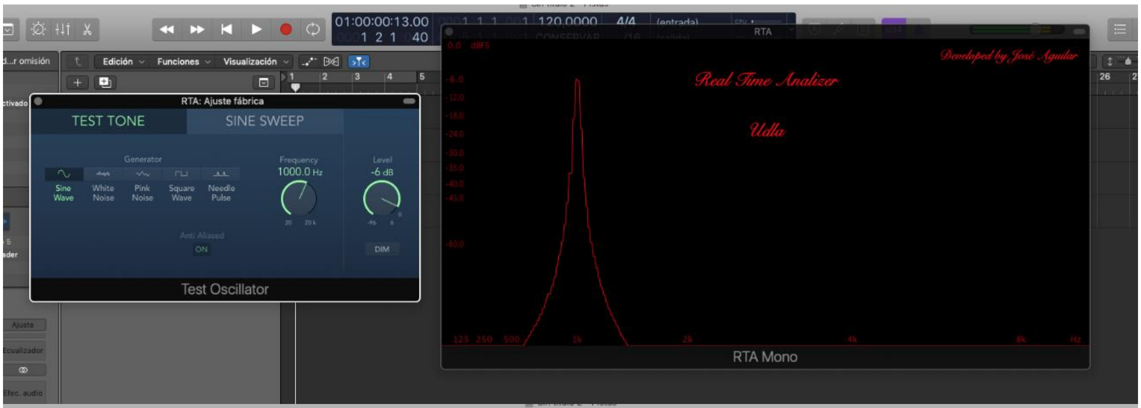


Plugins Mono: Sin título

Attack: 10.00 ms
Release: 30.00 ms
Treshold: 0.00 dB
Ratio: 5.00:1
Mix: 100.00 %
In: 0.00dB
Out: 0.00dB


Sin título 2 - Pistas
01:00:09:08.05
4 1 2 174
80,0000
CONSERVAR
4/4
/16
(entrada)
(salida)
Inspector
Edición
Funciones
Visualización
Ajuste:
Inteligente
Arrastre:
Sin superposición
12 Bar Blues Bass
M S
Delay
Time
Feedback
Mix
In
Out
0.00dB
0.00dB
1/4
50.00 %
50.00 %
SYNC
Delay Mono

Anexo 10. Pruebas de analizador de escala de nivel y frecuencia de RTA.



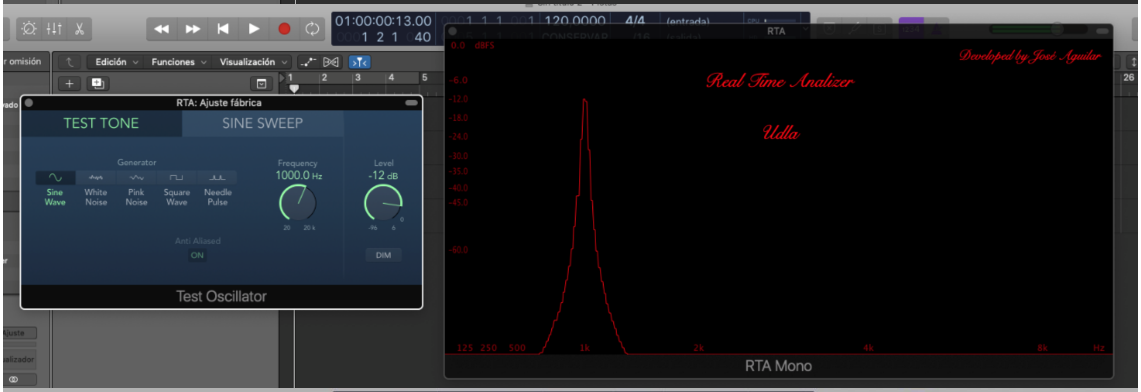
RTA: Ajuste fábrica

Generator: Sine Wave

Frequency: 1000.0 Hz

Level: -6 dB

Anti Aliased: ON



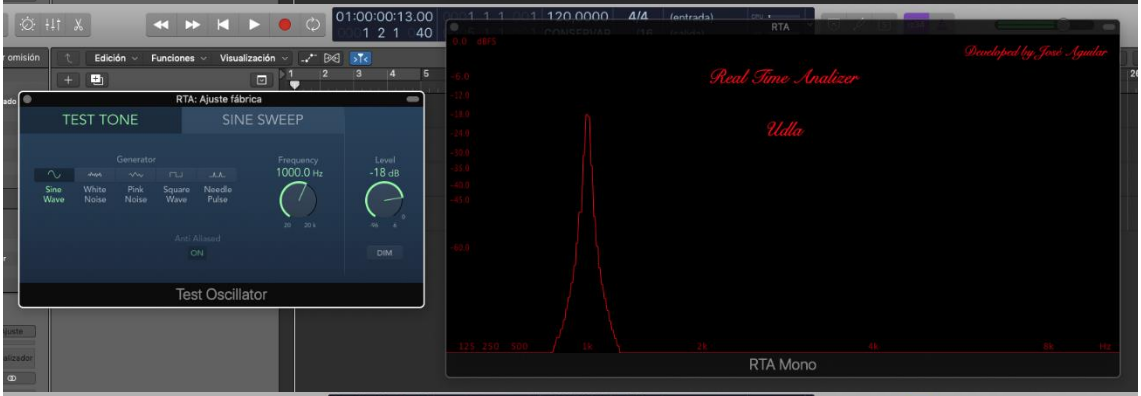
RTA: Ajuste fábrica

Generator: Sine Wave

Frequency: 1000.0 Hz

Level: -12 dB

Anti Aliased: ON



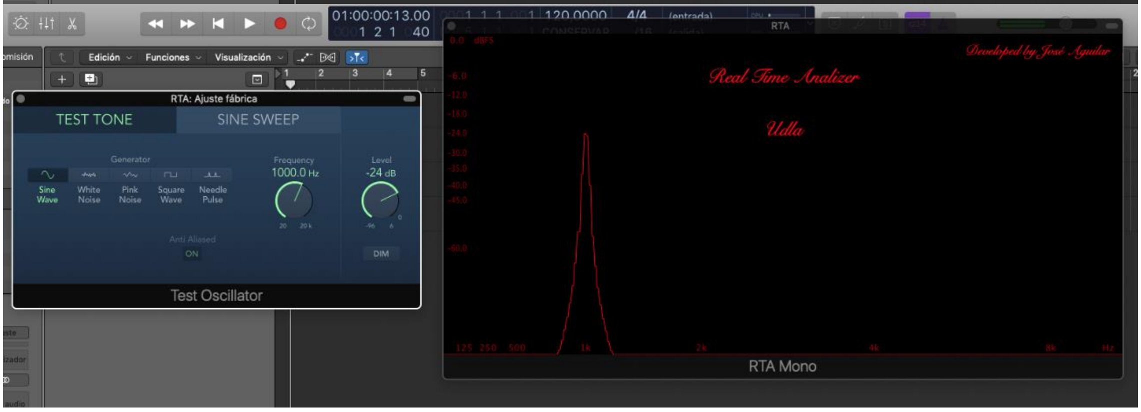
RTA: Ajuste fábrica

Generator: Sine Wave

Frequency: 1000.0 Hz

Level: -18 dB

Anti Aliased: ON



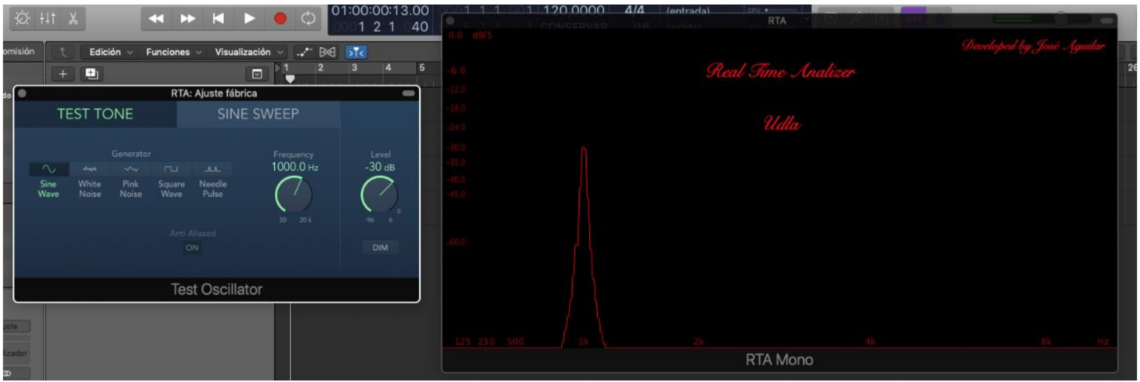
RTA: Ajuste fábrica

Generator: Sine Wave

Frequency: 1000.0 Hz

Level: -24 dB

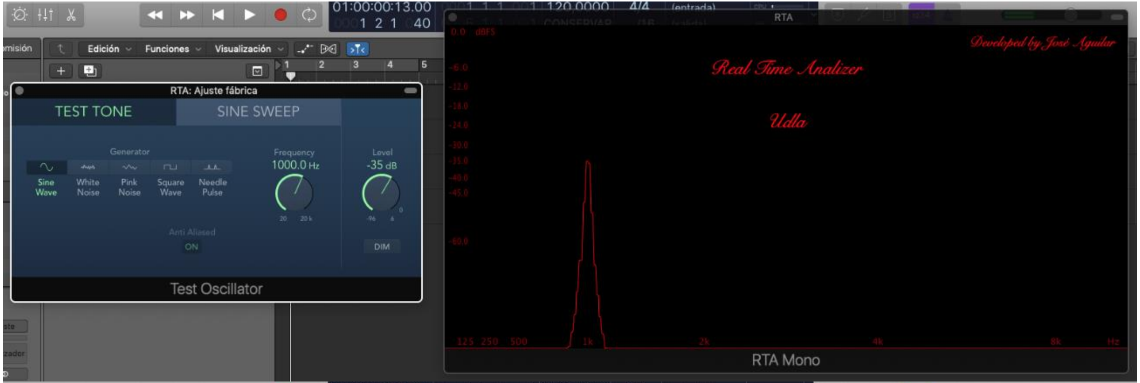
Anti Aliased: ON


RTA: Ajuste fábrica

Frequency 1000.0 Hz

Level -30 dB

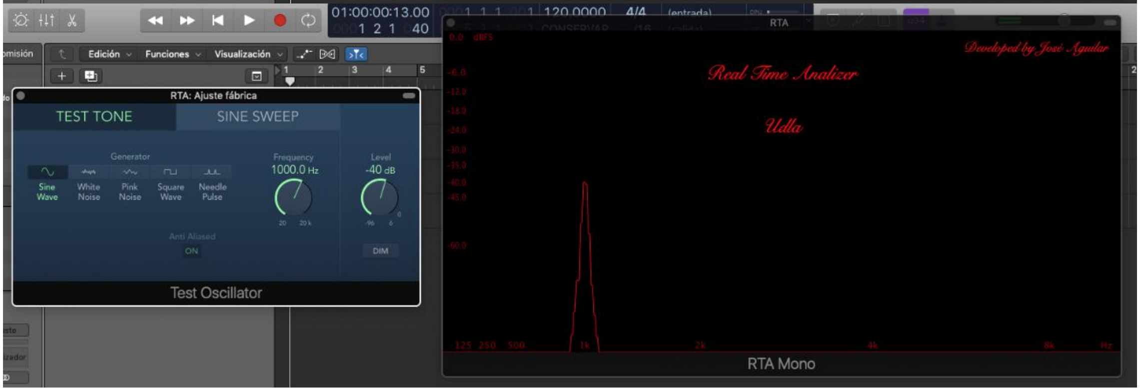
Anti Aliased ON


RTA: Ajuste fábrica

Frequency 1000.0 Hz

Level -35 dB

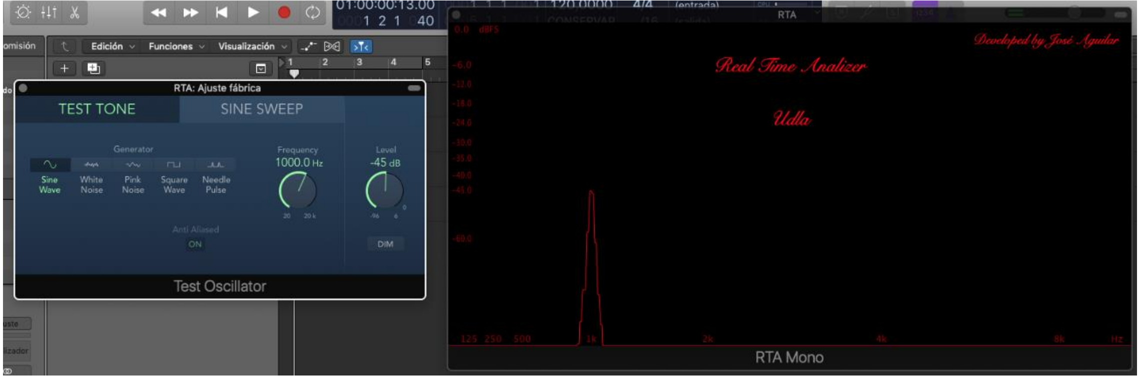
Anti Aliased ON


RTA: Ajuste fábrica

Frequency 1000.0 Hz

Level -40 dB

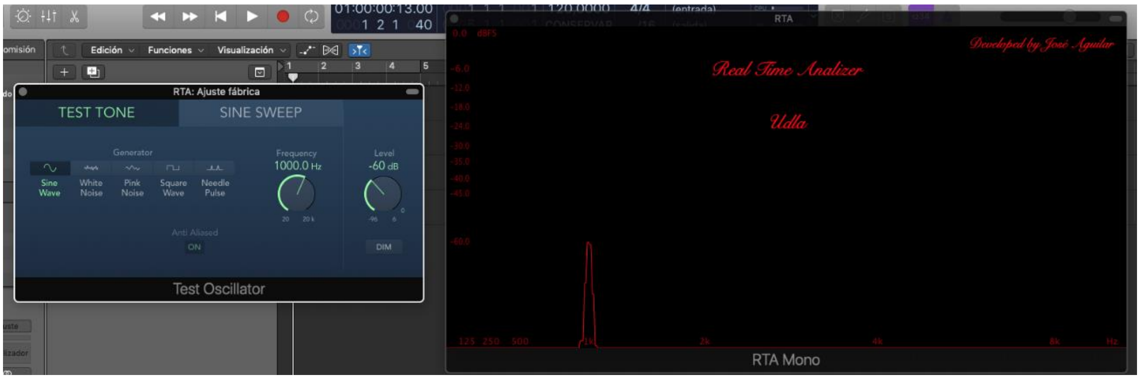
Anti Aliased ON


RTA: Ajuste fábrica

Frequency 1000.0 Hz

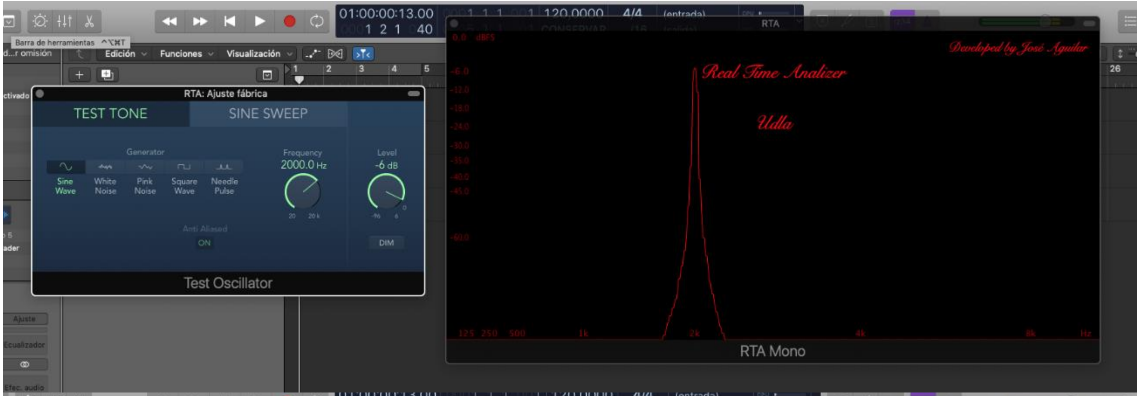
Level -45 dB

Anti Aliased ON



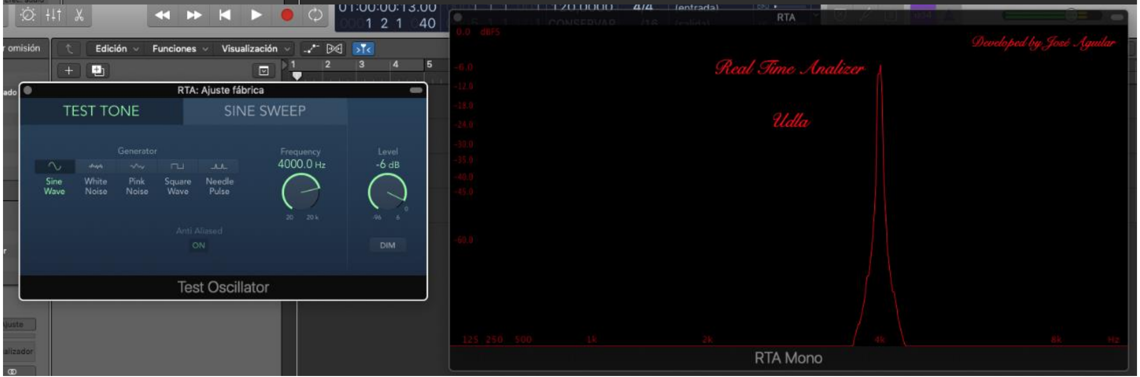
Frequency 1000.0 Hz

Level -60 dB



Frequency 2000.0 Hz

Level -6 dB



Frequency 4000.0 Hz

Level -6 dB

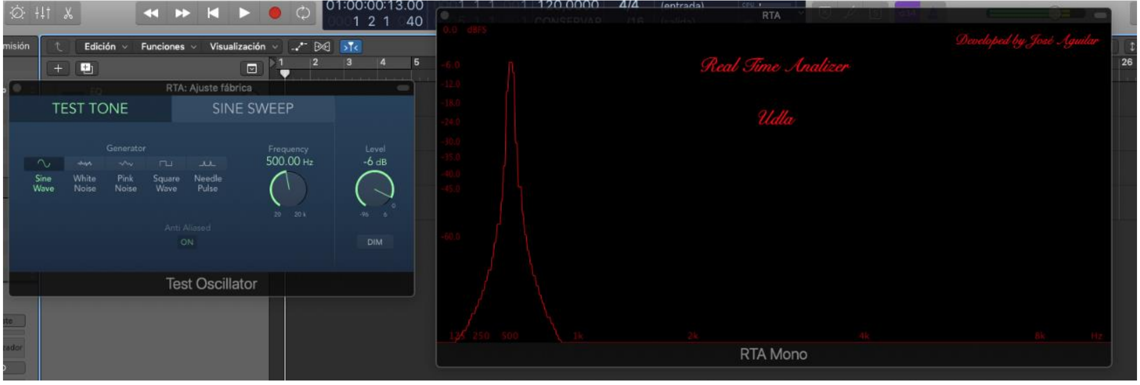
Frequency 125.00 Hz

Level -6 dB

Frequency 250.00 Hz

Level -6 dB

Anti Aliased ON



Frequency 500.00 Hz

Level -6 dB

Anti Aliased ON

Anexo 11. Módulos utilizados en la prueba de comparación de compresores

Anexo 12. Módulos utilizados en la prueba de comparación del efecto de *Delay*.

Anexo 13. Detalle de las pruebas de procesamiento

Ableton Live Windows										
Prueba de 1 canal de audio con una señal sonora y ejecución de los 5 módulos										
Plugins Mono										
Tamaño del Buffer de procesamiento (n ° de muestras)	**128**		**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**									
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	9,3%	15,0%	8,0%	14,6%	7,7%	13,0%	6,0%	11,0%	5,3%	9,8%
48000	10,6%	17,2%	9,3%	15,9%	8,2%	13,9%	6,8%	12,0%	6,0%	10,4%
88200	13,0%	20,0%	11,4%	18,6%	10,1%	16,9%	9,0%	14,5%	7,3%	12,6%
96000	14,6%	22,0%	13,2%	20,1%	11,9%	18,3%	10,2%	16,2%	8,9%	13,8%
192000	19,4%	27,1%	17,0%	25,5%	15,8%	23,4%	13,7%	21,2%	11,0%	18,5%
Plugins Estéreo										
44100	10.5%	17,1%	9,7%	15,0%	8,5%	14,3%	7,2%	13,2%	6,0%	11,4%
48000	12,7%	19,2%	11,2%	17,8%	10,8%	16,0%	9,1%	15,0%	7,4%	13,8%
88200	15,1%	21,4%	13,4%	20,1%	13,2%	18,4%	11,9%	17,3%	9,7%	16,3%
96000	18,3%	23,7%	16,2%	23,0%	15,0%	20,9%	14,0%	19,6%	11,5%	19,4%
192000	21,6%	28,0%	19,1%	26,4%	17,6%	24,8%	16,2%	22,4%	13,0%	20,9%

Ableton Live MacOS										
Prueba de 1 canal de audio con una señal sonora y ejecución de los 5 módulos										
Plugins Mono										
Tamaño del Buffer de procesamiento (n ° de muestras)	**128**		**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**									
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	8,5%	14,9%	7,3%	13,8%	7,8%	12,1%	6,0%	11,2%	5,1%	9,0%
48000	9,9%	17,0%	9,0%	15,2%	8,7%	13,6%	7,1%	12,4%	6,3%	10,1%
88200	12,7%	19,5%	10,5%	17,9%	9,8%	15,4%	8,7%	14,5%	7,7%	11,9%
96000	14,0%	21,7%	12,8%	19,5%	11,0%	18,0%	10,0%	15,9%	8,7%	13,0%
192000	17,8%	25,7%	16,3%	24,6%	14,2%	22,6%	12,5%	19,7%	10,2%	17,4%
Plugins Estéreo										
44100	10,0%	16,5%	9,5%	14,9%	8,0%	13,8%	6,5%	12,8%	6,2%	10,7%
48000	11,8%	18,6%	10,5%	16,7%	9,9%	15,5%	8,7%	14,9%	7,6%	12,7%
88200	14,3%	20,7%	12,8%	19,3%	12,6%	17,7%	10,5%	17,0%	9,0%	15,5%
96000	17,1%	22,1%	15,7%	21,9%	14,4%	20,1%	12,9%	19,1%	11,2%	18,3%
192000	20,3%	26,9%	18,6%	24,2%	17,0%	23,3%	15,7%	22,0%	12,9%	21,0%

Reason 10 Windows										
Prueba de 1 canal de audio con una señal sonora y ejecución de los 5 módulos										
Plugins Mono										
Tamaño del Buffer de procesamiento (n ° de muestras)	128		256		512		1024		2048	
Frecuencia de muestreo	Procesamiento Máximo									
	Interno	CPU	Interno	CPU	Interno	CPU	Interno	CPU	Interno	CPU
44100	8,0%	14,0%	7,3%	13,3%	6,5%	12,3%	6,1%	11,0%	5,0%	9,3%
48000	9,8%	16,7%	8,8%	14,8%	8,2%	14,0%	7,1%	12,3%	6,2%	10,5%
88200	12,6%	18,9%	10,5%	17,3%	9,4%	15,7%	8,5%	14,0%	7,1%	11,9%
96000	14,0%	21,7%	12,9%	19,5%	12,0%	17,7%	10,5%	15,7%	9,0%	12,6%
192000	17,9%	24,8%	17,0%	24,2%	14,6%	22,7%	12,4%	20,3%	10,8%	16,9%
Plugins Estéreo										
44100	9,8%	16,3%	9,3%	14,6%	8,1%	13,8%	6,7%	12,8%	6,0%	11,5%
48000	11,9%	18,6%	11,0%	16,9%	11,0%	15,3%	8,2%	15,1%	7,7%	12,9%
88200	14,0%	20,3%	12,9%	19,3%	12,9%	17,7%	10,0%	16,6%	10,1%	15,9%
96000	17,2%	22,8%	15,3%	22,4%	15,1%	19,2%	13,4%	19,3%	11,4%	18,3%
192000	20,9%	26,9%	18,8%	25,0%	16,9%	22,9%	15,7%	22,0%	13,2%	21,4%

Reason 10 MacOS										
Prueba de 1 canal de audio con una señal sonora y ejecución de los 5 módulos										
Plugins Mono										
Tamaño del Buffer de procesamiento (n ° de muestras)	128		256		512		1024		2048	
Frecuencia de muestreo	Procesamiento Máximo									
	Interno	CPU	Interno	CPU	Interno	CPU	Interno	CPU	Interno	CPU
44100	8,7%	14,6%	7,8%	14,3%	7,4%	13,3%	6,3%	10,8%	5,0%	10,5%
48000	10,1%	17,0%	9,1%	15,6%	8,8%	15,1%	7,4%	13,0%	6,9%	11,9%
88200	12,9%	19,3%	11,2%	18,7%	10,0%	16,8%	8,8%	14,8%	8,1%	13,0%
96000	14,3%	22,0%	13,4%	21,2%	11,7%	18,7%	11,5%	16,3%	10,1%	15,4%
192000	18,0%	26,2%	18,9%	25,9%	15,1%	23,8%	14,0%	21,2%	11,9%	18,0%
Plugins Estéreo										
44100	10,3%	17,1%	10,0%	15,0%	8,8%	14,4%	6,8%	13,4%	6,3%	12,3%
48000	11,3%	19,4%	11,4%	17,1%	11,7%	16,1%	8,8%	15,0%	8,0%	13,8%
88200	13,9%	21,0%	13,9%	19,9%	13,8%	18,1%	10,9%	17,1%	9,9%	16,7%
96000	16,8%	24,3%	16,7%	23,2%	16,0%	19,8%	13,6%	20,2%	12,1%	19,6%
192000	19,0%	27,1%	19,3%	25,5%	17,7%	24,1%	16,3%	23,0%	14,2%	22,3%

Cubase 8 Windows										
Prueba de 1 canal de audio con una señal sonora y ejecución de los 5 módulos										
Plugins Mono										
Tamaño del Buffer de procesamiento (n ° de muestras)	128		256		512		1024		2048	
Frecuencia de muestreo	Procesamiento Máximo									
	Interno	CPU	Interno	CPU	Interno	CPU	Interno	CPU	Interno	CPU
44100	6,8%	12,1%	5,9%	11,5%	5,2%	10,4%	5,0%	9,8%	4,1%	7,3%
48000	8,2%	14,7%	7,2%	12,8%	6,8%	12,6%	6,5%	11,0%	5,6%	9,2%
88200	10,3%	16,3%	9,1%	14,9%	8,3%	13,9%	7,7%	12,9%	6,8%	10,5%
96000	12,8%	19,7%	11,4%	17,3%	10,7%	15,7%	9,2%	14,3%	9,0%	12,0%
192000	15,7%	23,0%	13,6%	22,5%	12,9%	18,6%	11,6%	17,8%	9,9%	14,7%
Plugins Estéreo										
44100	7,2%	12,7%	6,4%	12,5%	6,0%	11,2%	5,7%	10,3%	5,1%	8,4%
48000	8,5%	15,0%	7,9%	13,9%	7,4%	13,7%	7,6%	11,7%	6,2%	10,2%
88200	9,9%	17,2%	9,9%	15,7%	9,2%	15,1%	8,8%	13,3%	7,8%	11,8%
96000	11,3%	20,3%	12,2%	18,2%	11,1%	16,3%	10,1%	15,2%	8,9%	13,4%
192000	13,7%	23,8%	14,6%	23,1%	13,6%	19,4%	12,4%	18,4%	10,5%	16,0%

Cubase 8 MacOS										
Prueba de 1 canal de audio con una señal sonora y ejecución de los 5 módulos										
Plugins Mono										
Tamaño del Buffer de procesamiento (n ° de muestras)	128		256		512		1024		2048	
Frecuencia de muestreo	Procesamiento Máximo									
	Interno	CPU	Interno	CPU	Interno	CPU	Interno	CPU	Interno	CPU
44100	5,7%	11,7%	5,8%	10,6%	4,8%	9,6%	4,5%	9,2%	3,8%	6,8%
48000	7,5%	13,2%	6,8%	11,6%	5,9%	11,4%	6,3%	10,5%	4,7%	8,7%
88200	8,9%	15,4%	8,4%	13,4%	7,7%	12,7%	7,1%	11,7%	6,0%	10,0%
96000	10,4%	18,8%	10,5%	16,9%	9,6%	14,3%	8,8%	13,6%	8,3%	11,4%
192000	13,7%	22,1%	12,7%	21,8%	11,5%	17,6%	11,1%	17,0%	9,4%	13,6%
Plugins Estéreo										
44100	6,3%	12,5%	6,7%	11,5%	5,8%	10,8%	5,3%	10,1%	4,2%	7,4%
48000	7,9%	14,0%	7,9%	12,8%	6,8%	12,5%	6,8%	11,4%	5,7%	9,5%
88200	9,5%	16,8%	9,9%	14,3%	8,3%	13,4%	7,9%	12,6%	7,1%	10,9%
96000	11,8%	19,1%	11,4%	17,9%	9,9%	15,2%	9,4%	14,1%	9,5%	11,9%
192000	14,2%	23,4%	13,7%	22,5%	12,3%	18,6%	12,0%	17,8%	10,5%	14,3%

Logic Pro X MacOS

Prueba de 1 canal de audio con una señal sonora y ejecución de los 5 módulos

Plugins Mono

Tamaño del Buffer de procesamiento (n ° de muestras)	**128**		**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**									
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	7,4%	13,0%	6,7%	12,3%	6,0%	11,5%	6,2%	10,7%	5,0%	8,1%
48000	9,1%	15,4%	8,1%	13,5%	7,4%	13,5%	7,7%	12,1%	6,7%	10,3%
88200	10,6%	17,3%	10,0%	15,4%	9,2%	15,1%	8,9%	13,7%	7,9%	11,5%
96000	13,6%	20,5%	12,2%	18,6%	11,2%	16,9%	10,2%	15,5%	10,3%	12,9%
192000	16,2%	24,2%	14,6%	23,5%	13,7%	19,4%	12,6%	18,6%	11,9%	15,2%
Plugins Estéreo										
44100	8,1%	13,5%	7,2%	13,8%	7,3%	12,4%	6,9%	11,4%	6,5%	9,5%
48000	9,3%	16,1%	8,5%	14,8%	8,5%	14,2%	8,1%	13,1%	7,2%	11,5%
88200	10,2%	18,0%	10,3%	16,6%	10,3%	16,0%	9,5%	14,9%	8,9%	12,9%
96000	12,5%	21,4%	13,4%	19,1%	12,4%	17,7%	11,2%	17,3%	10,1%	14,6%
192000	14,0%	24,3%	15,6%	23,0%	14,2%	20,3%	13,6%	19,7%	11,8%	17,1%

Ableton Live Windows

Plugins Mono

Prueba de 15 canales con los 5 módulos (75 instancias de *plugins*)

Tamaño del buffer (muestras)	**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**							
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	20,0%	36,5%	19,3%	32,5%	15,0%	27,5%	13,3%	24,5%
48000	23,3%	39,8%	20,4%	34,8%	17,0%	30,0%	15,0%	26,0%
88200	28,5%	46,5%	25,3%	42,3%	22,5%	36,3%	18,3%	31,5%
96000	33,0%	50,3%	29,8%	45,8%	25,5%	40,5%	22,3%	34,5%
192000	42,5%	63,8%	39,5%	58,5%	34,3%	53,0%	27,5%	46,3%
Plugins Estéreo								
44100	24,3%	37,5%	21,3%	35,8%	18,0%	33,0%	15,0%	28,5%
48000	28,0%	44,5%	27,0%	40,0%	22,8%	37,5%	18,5%	34,5%
88200	33,5%	50,3%	33,0%	46,0%	29,8%	43,3%	24,3%	40,8%
96000	40,5%	57,5%	37,5%	52,3%	35,0%	49,0%	28,8%	48,5%
192000	47,8%	66,0%	44,0%	62,0%	40,5%	56,0%	32,5%	52,3%

Ableton Live MacOS								
Plugins Mono								
Prueba de 15 canales con los 5 módulos (75 instancias de *plugins*)								
Tamaño del buffer (muestras)	**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**							
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	18,3%	34,5%	19,5%	30,3%	15,0%	28,0%	12,8%	22,5%
48000	22,5%	38,0%	21,8%	34,0%	17,8%	31,0%	15,8%	25,3%
88200	26,3%	44,8%	24,5%	38,5%	21,8%	36,3%	19,3%	29,8%
96000	32,0%	48,8%	27,5%	45,0%	25,0%	39,8%	21,8%	32,5%
192000	40,8%	61,5%	35,5%	56,5%	31,3%	49,3%	25,5%	43,5%
Plugins Estéreo								
44100	23,8%	37,3%	20,0%	34,5%	16,3%	32,0%	15,5%	26,8%
48000	26,3%	41,8%	24,8%	38,8%	21,8%	37,3%	19,0%	31,8%
88200	32,0%	48,3%	31,5%	44,3%	26,3%	42,5%	22,5%	38,8%
96000	39,3%	54,8%	36,0%	50,3%	32,3%	47,8%	28,0%	45,8%
192000	46,5%	60,5%	42,5%	58,3%	39,3%	55,0%	32,3%	52,5%

Reason 10 Windows								
Plugins Mono								
Prueba de 15 canales con los 5 módulos (75 instancias de *plugins*)								
Tamaño del buffer (muestras)	**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**							
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	18,3%	33,3%	16,3%	30,8%	15,3%	27,5%	12,5%	23,3%
48000	22,0%	37,0%	20,4%	35,0%	17,8%	30,8%	15,5%	26,3%
88200	26,3%	43,3%	23,5%	39,3%	21,3%	35,0%	17,8%	29,8%
96000	32,3%	48,8%	30,0%	44,3%	26,3%	39,3%	22,5%	31,5%
192000	42,5%	60,5%	36,5%	56,8%	31,0%	50,8%	27,0%	42,3%
Plugins Estéreo								
44100	23,3%	36,5%	20,3%	34,5%	16,8%	32,0%	15,0%	28,8%
48000	27,5%	42,3%	27,5%	38,3%	20,5%	37,8%	19,3%	32,3%
88200	32,3%	48,3%	32,3%	44,3%	25,0%	41,5%	25,3%	39,8%
96000	38,3%	56,0%	37,8%	48,0%	33,5%	48,3%	28,5%	45,8%
192000	47,0%	62,5%	42,3%	57,3%	39,3%	55,0%	33,0%	53,5%

Reason 10 MacOS								
Plugins Mono								
Prueba de 15 canales con los 5 módulos (75 instancias de *plugins*)								
Tamaño del buffer (muestras)	**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**							
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	19,5%	35,8%	18,5%	33,3%	15,8%	27,0%	12,5%	26,3%
48000	22,8%	39,0%	22,0%	37,8%	18,5%	32,5%	17,3%	29,8%
88200	28,0%	46,8%	25,0%	42,0%	22,0%	37,0%	20,3%	32,5%
96000	33,5%	53,0%	29,3%	46,8%	28,8%	40,8%	25,3%	38,5%
192000	47,3%	64,8%	37,8%	59,5%	35,0%	53,0%	29,8%	45,0%
Plugins Estéreo								
44100	25,0%	37,5%	22,0%	36,0%	17,0%	33,5%	15,8%	30,8%
48000	28,5%	42,8%	29,3%	40,3%	22,0%	37,5%	20,0%	34,5%
88200	34,8%	49,8%	34,5%	45,3%	27,3%	42,8%	24,8%	41,8%
96000	41,8%	58,0%	40,0%	49,5%	34,0%	50,5%	30,3%	49,0%
192000	48,3%	63,8%	44,3%	60,3%	40,8%	57,5%	35,5%	55,8%

Cubase 8 Windows								
Plugins Mono								
Prueba de 15 canales con los 5 módulos (75 instancias de *plugins*)								
Tamaño del buffer (muestras)	**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**							
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	14,8%	28,8%	13,0%	26,0%	12,5%	24,5%	10,3%	18,3%
48000	18,0%	32,0%	17,0%	31,5%	16,3%	27,5%	14,0%	23,0%
88200	22,8%	37,3%	20,8%	34,8%	19,3%	32,3%	17,0%	26,3%
96000	28,5%	43,3%	26,8%	39,3%	23,0%	35,8%	22,5%	30,0%
192000	34,0%	56,3%	32,3%	46,5%	29,0%	44,5%	24,8%	36,8%
Plugins Estéreo								
44100	16,0%	31,3%	15,0%	28,0%	14,3%	25,8%	12,8%	21,0%
48000	19,8%	34,8%	18,5%	34,3%	19,0%	29,3%	15,5%	25,5%
88200	24,8%	39,3%	23,0%	37,8%	22,0%	33,3%	19,5%	29,5%
96000	30,5%	45,5%	27,8%	40,8%	25,3%	38,0%	22,3%	33,5%
192000	36,5%	57,8%	34,0%	48,5%	31,0%	46,0%	26,3%	40,0%

Cubase 8 MacOS								
Plugins Mono								
Prueba de 15 canales con los 5 módulos (75 instancias de *plugins*)								
Tamaño del buffer (muestras)	**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**							
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	14,5%	26,5%	12,0%	24,0%	11,3%	23,0%	9,5%	17,0%
48000	17,0%	29,0%	14,8%	28,5%	15,8%	26,3%	11,8%	21,8%
88200	21,0%	33,5%	19,3%	31,8%	17,8%	29,3%	15,0%	25,0%
96000	26,3%	42,3%	24,0%	35,8%	22,0%	34,0%	20,8%	28,5%
192000	31,8%	54,5%	28,8%	44,0%	27,8%	42,5%	23,5%	34,0%
Plugins Estéreo								
44100	16,8%	28,8%	14,5%	27,0%	13,3%	25,3%	10,5%	18,5%
48000	19,8%	32,0%	17,0%	31,3%	17,0%	28,5%	14,3%	23,8%
88200	24,8%	35,8%	20,8%	33,5%	19,8%	31,5%	17,8%	27,3%
96000	28,5%	44,8%	24,8%	38,0%	23,5%	35,3%	23,8%	29,8%
192000	34,3%	56,3%	30,8%	46,5%	30,0%	44,5%	26,3%	35,8%

Logic Pro X MacOS								
Plugins Mono								
Prueba de 15 canales con los 5 módulos (75 instancias de *plugins*)								
Tamaño del buffer (muestras)	**256**		**512**		**1024**		**2048**	
Frecuencia de muestreo	**Procesamiento Máximo**							
	Interno	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**	**Interno**	**CPU**
44100	16,8%	30,8%	15,0%	28,8%	15,5%	26,8%	12,5%	20,3%
48000	20,3%	33,8%	18,5%	33,8%	19,3%	30,3%	16,8%	25,8%
88200	25,0%	38,5%	23,0%	37,8%	22,3%	34,3%	19,8%	28,8%
96000	30,5%	46,5%	28,0%	42,3%	25,5%	38,8%	25,8%	32,3%
192000	36,5%	58,8%	34,3%	48,5%	31,5%	46,5%	29,8%	38,0%
Plugins Estéreo								
44100	18,0%	34,5%	18,3%	31,0%	17,3%	28,5%	16,3%	23,8%
48000	21,3%	37,0%	21,3%	35,5%	20,3%	32,8%	18,0%	28,8%
88200	25,8%	41,5%	25,8%	40,0%	23,8%	37,3%	22,3%	32,3%
96000	33,5%	47,8%	31,0%	44,3%	28,0%	43,3%	25,3%	36,5%
192000	39,0%	57,5%	35,5%	50,8%	34,0%	49,3%	29,5%	42,8%