ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: Normalizador de sonoridad de audio digital para tramas AES/EBU utilizando algoritmos de medida de sonoridad basados en la norma ITU-R BS.1770-4

AUTOR: Manuel Pablo Sánchez Hernández

TITULACIÓN: Grado en Ingeniería Electrónica de Comunicaciones

DIRECTOR: Javier María Hernández Muñoz

TUTOR: Matías J. Garrido González

DEPARTAMENTO: Ingeniería Telemática y Electrónica

VºBº

Miembros del Tribunal Calificador:

PRESIDENTE: Francisco Prieto Castrillo

TUTOR: Matías J. Garrido González

SECRETARIO: Pedro J. Lobo Perea

Fecha de lectura:

Calificación:

El Secretario,

Resumen.

En la actualidad son muchas y muy distintas las fuentes de audio y video que se transmiten a través de los medios de comunicación. Durante muchos años el ajuste de nivel de las señales de audio de estas fuentes lo realizaban técnicos de sonido basándose en su experiencia y percepción subjetiva, ayudándose de medidores de nivel de tipo vúmetro o picómetro. Este método proporciona en general resultados muy satisfactorios cuando se dispone de personal cualificado y con experiencia. Sin embargo, cuando el número de programas que se produce es grande y las fuentes que se utilizan para producir estos programas son muy diversas, el coste de realizar este proceso resulta muy elevado. Conscientes de este hecho, los organismos normalizadores han desarrollado algoritmos para medir y ajustar el nivel de las señales de audio en base a un parámetro denominado sonoridad y lo han plasmado en normas como la ITU-R BS.1770-4 y la EBU R 128. El interés de este nuevo parámetro radica en que satisface a la vez dos condiciones: se puede calcular de forma automática y refleja bastante bien la percepción subjetiva que el oyente tiene de una señal de audio.

El objeto de este proyecto consiste en el diseño de un equipo que mida la sonoridad de una señal de audio estéreo utilizando los algoritmos que se describen en estas normas y ajuste su nivel al valor objetivo de sonoridad que se le indique. Las señales de audio sobre la que se realiza el proceso son señales de audio digital según el estándar AES3 también conocido como AES/EBU.

Desde un punto de vista formativo, el proyecto se ha abordado como si se tratase de un proyecto profesional, en el cual ha sido necesario estudiar y comprender las normas que se tenían que cumplir e implementar una solución al problema utilizando tecnologías de diseño digital, diseño analógico y diseño de software para sistemas empotrados.

Abstract.

Nowadays, there are many different audio and video sources transmitted on media. On the near past, adjust of signal audio level was done by professional people trusting in their work experience and subjective perception, using some level meters. With this method they could reach very good results. However, the cost was too high when a TV company had many different programs. Knowing that problem, normalizer organisms have developed algorithms to measure and adjust audio level, with a new parameter called *Loudness,* writing some standards like ITU-R BS.1770-4 or EBU R 128. This parameter is so useful because it can be calculated automatically and is equivalent to the subjective perception of an audio signal of the listener.

The goal of this project is to design an electronic system that could measure loudness of a stereo audio signal using algorithms of these standards and to adjust a chosen target loudness. The digital audio standard used is known as AES3 or AES/EBU.

As a learning point of view, the project has been thought as a professional project, so it has been needed to understand and study some necessary standards and to build a solution using digital design, analog design and software design technologies.

Índice de contenidos.

1 Introducción 1

2 Marco tecnológico 3

3 Especificaciones 5

4 Acrónimos 7

5 Solución 9

5.1 Firmware 9

5.1.1 Norma AES/EBU 9

5.1.2 Diseño estructural 11

5.1.3 PLL 13

5.1.4 Decodificador 14

5.1.5 Procesador de audio 15

5.1.6 Interfaz I2S 16

5.1.7 Interfaz SPI 16

5.1.8 Codificador 18

5.2 Hardware 18

5.2.1 Microcontrolador 18

5.2.2 PLL 19

5.2.3 Multiplicador de frecuencia 22

5.2.4 FPGA 23

5.2.5 Drivers de I/O 23

5.3 Software 23

5.3.1 Interfaz I2S 25

5.3.2 Interfaz SPI 27

5.3.3 Algoritmo de cálculo de sonoridad 28

5.3.4 Algoritmo de cálculo de nivel de pico 33

5.3.5 Algoritmo de cálculo de ganancia 34

5.4 Sistema completo 35

6 Resultados 37

6.1 Firmware 37

6.2 Hardware 41

6.3 Software 44

6.4 Sistema completo 47

7 Presupuesto 51

8 Conclusiones 53
9 Bibliografía.. 55
10 Anexos 57
10.1 Código VHDL... 57
10.2 Algoritmos Software 57
10.2.1 Función principal 57
10.2.2 Funciones auxiliares 63
10.3 Depuración 68

1 Introducción

Un sistema normalizador de audio aplica una ganancia instantánea variable a una señal de audio digital con el fin de conseguir un nivel de sonoridad objetivo constante. Un sistema de este tipo, uno de los muchos tipos de sistemas de control automático de ganancia (CAG), implica una serie de procesos basados en métodos de cálculo integral con el fin de obtener medidas de sonoridad, el parámetro que se emplea para ajustar el nivel del audio.

Albalá Ingenieros es una empresa puntera en el diseño e implementación de sistemas electrónicos de televisión profesional. El problema que aborda este proyecto tiene por objeto cubrir una de las necesidades de los clientes de esta empresa.

Hoy en día, una cadena de televisión difunde contenido de muchas fuentes distintas. Estas fuentes pueden haber realizado las grabaciones con niveles de audio distintos. Este hecho puede provocar el desagrado del usuario, puesto que puede variar ampliamente la percepción sonora real con respecto a la que él ha decidido fijar. Es por tanto una necesidad el diseño y desarrollo de una tecnología capaz de nivelar la sonoridad automáticamente, liberando tanto al espectador como a la cadena de televisión de este trabajo. Aunque la empresa ya dispone de otro producto que realiza esta función, el algoritmo se realiza dentro de una FPGA de alto coste. Este proyecto busca una evolución de diseño, de tal manera que se pueda usar una FPGA menos potente y por tanto más barata, realizando la mayoría de los cálculos y tareas en un microcontrolador.

A lo largo de esta memoria se resume el diseño de un sistema electrónico completo, desde las especificaciones iniciales, pasando por el proceso de diseño, hasta la finalización de las pruebas de validez y documentación.

2 Marco tecnológico

La norma AES/EBU [1] caracteriza una interfaz de comunicación de audio digital comúnmente utilizada en televisión y audio profesional. La implementación de un sistema que cumpla los requisitos de esta interfaz y que además permita el tratamiento de las señales de audio, requiere de conocimientos en tres ámbitos básicos de las telecomunicaciones: Firmware, software y hardware.

La placa *HAM2000C01* [2] de Albalá Ingenieros, sobre la que se ha realizado el proyecto, constituye la parte de hardware del sistema. El firmware incluye la parte de lógica digital que funciona dentro de una FPGA, en la que se implementan, entre otras cosas, los circuitos de codificación y decodificación de señales según la norma AES/EBU. Finalmente, en la parte de software se desarrolla un programa para un microcontrolador, el cual ejecuta los algoritmos de tratamiento digital de señal.

Se habla del tratamiento de señales ya que ahí reside el objetivo principal del proyecto, la normalización de señales digitales de audio con el fin de atenuar o aumentar la sonoridad utilizando algoritmos de medida basados en la recomendación ITU-R BS.1770-4 [3] y siguiendo los pasos marcados en la norma EBU Tech. 3341 [4].

Dicha norma define los pasos que se deben de realizar sobre una señal de audio para medir su sonoridad. Esta medida permitirá después ajustar la ganancia que el sistema aplica a la señal para que la sonoridad que entrega sea la que se desea.

Por tanto, implementación de sistemas basados en FPGA a alta frecuencia, cálculos intensivos en microcontrolador y desarrollo de hardware son las bases de este proyecto y así se realizará su división.

3 Especificaciones

Un normalizador de audio puede diseñarse según numerosas normas y especificaciones para el cálculo de la sonoridad y normalización de señal. En este caso, el sistema cumple los requisitos de medición de sonoridad marcados por la recomendación ITU-R BS.1770-4 a través de técnicas reflejadas en la norma EBU Tech. 3341.

Todas las especificaciones del sistema quedan definidas a continuación:

- El sistema debe de poder transmitir y recibir señal de audio digital en formato AES/EBU, a una frecuencia de 48 kHz. Las tramas de audio constan de 32 bits, 24 de los cuales portan la señal de audio propiamente dicha. Del resto, 4 bits son necesarios para la sincronización de las recepciones y transmisiones, quedando 4 bits libres para paridad, canal de estado y otras utilidades.
- El sistema debe de poder realizar cálculos de sonoridad cada 100 ms de tres tipos: instantánea (400 ms), a corto plazo (3 s) e integral (variable hasta un máximo de 210 s). El método de cálculo de dicha sonoridad debe de cumplir las especificaciones marcadas en la recomendación ITU-R BS.1770-4.
- El sistema debe de poder modificar la ganancia de ambos canales por separado con el fin de atenuar o amplificar la señal de audio que ofrece a su salida.
- Las muestras de audio deben de ser transmitidas al microcontrolador para la realización de las medidas de sonoridad. Para ello se hace indispensable la implementación de una interfaz de comunicación I2S entre FPGA y microcontrolador.
- El sistema debe de poder controlarse y monitorizarse desde la aplicación *Annette* de la empresa Albalá Ingenieros a través de una estructura de control y estado que ha de implementarse haciendo uso de una interfaz SPI.
- El reloj del sistema debe de sincronizarse con el reloj de los datos que se reciben a través de la interfaz AES/EBU para lo cual es necesaria la implementación de un PLL.
- El sistema debe de implementar un mecanismo para seleccionar la fuente de audio que ofrecerá a su salida, entre la entrada modificada en ganancia o la señal de audio procesada en el microcontrolador.
- El diseño se ejecutará en la tarjeta *HAM2000C01* de la empresa, quedando su configuración grabada en una memoria flash para su funcionamiento de forma autónoma.

4 Acrónimos

FPGA	Field Programmable Gate Arrays
AES	Audio Engineering Society
EBU	European Broadcasting Union
ITU	International Telecommunication Union
VHDL	Very High Speed Integrated Circuit Hardware Description Language
MSB	Most Significant Bit
PLL	Phase Lock Loop
PHC	Phase Comparator
FIFO	First Input First Output
LED	Light Emiter Diode
DMA	Direct Memory Access
SERCOM	Serial Communication
VCO	Voltage Controlled Oscilator
LF	Loop Filter
dBFS	dB relativos al fondo de escala
DSP	Digital Signal Processor
CAG	Sistema de Control Automático de Ganancia

5 Solución

Puesto que el proyecto constituye un producto funcional y vendible para la empresa, el diseño se ha modularizado con el fin de reducir la dificultad y facilitar la evolución de futuras versiones. Por tanto, ha quedado dividido en tres bloques principales (Firmware, Software y Hardware).

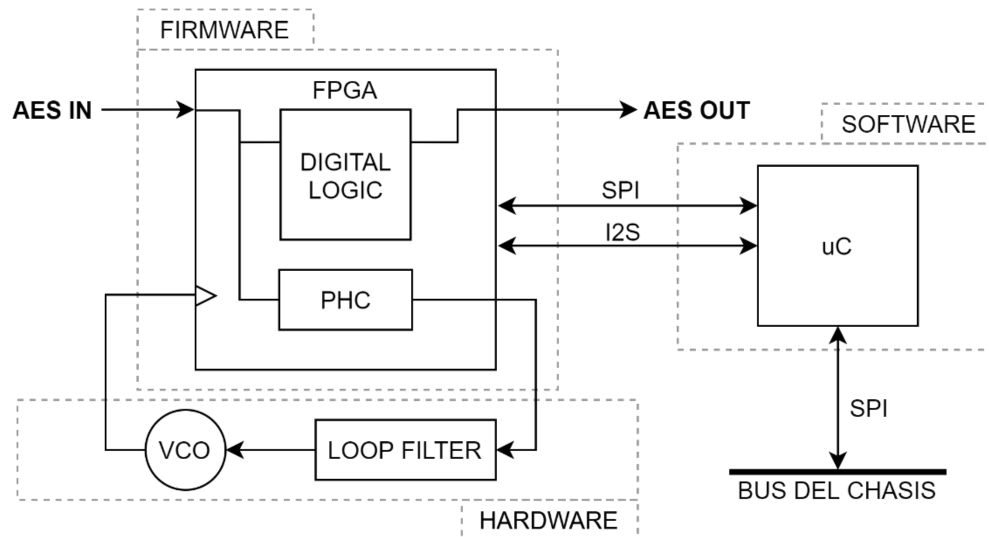

Fig. 1. Esquema general del normalizador de audio

5.1 Firmware

El firmware en VHDL abarca todas las especificaciones de diseño relacionadas con la norma AES/EBU y comunicación con el microcontrolador.

5.1.1 Norma AES/EBU

La norma AES/EBU define una interfaz de comunicación de audio digital profesional con longitud de palabra variable y bits reservados para información auxiliar. Está constituida por un único canal físico de comunicación, si bien es posible escoger entre la transmisión monocanal (*mono*) y multicanal (*stereo*), a través de mecanismos de multiplexación temporal.

La transmisión de datos se divide en bloques de 192 *frames*, dentro de los cuales hay dos *subframes*. Cada *subframe* se divide en 32 instantes de tiempo o time slots enumerados del 0 al 31, que corresponden a cada uno de los bits que se envían.

Los 4 primeros bits, del 0 al 3, están reservados para los preámbulos, X correspondiente al canal 1, Y correspondiente al canal 2 y Z como preámbulo de comienzo de bloque, sustituyendo al preámbulo X.

Los bits del 4 al 27 constituyen la muestra de audio, que puede variar entre 20 y 24 bits, siendo el MSB (*Most Significant Bit*) siempre el bit 27. Cabe resaltar que la codificación de la muestra de audio se realiza en complemento a 2. En este caso particular, se ha realizado el diseño para muestras de audio de 24 bits, por lo que la señal puede tomar valores entre (2^{23} – 1) y -2^{23}.

Los bits restantes hasta el 31 se corresponden con el bit de validez, usuario, canal de estado y paridad. En el caso del canal de estado, han de transmitirse 24 registros de 8 bits, que, a un bit por *frame*, dan los 192 *frames* necesarios para la transmisión o recepción de un bloque completo.

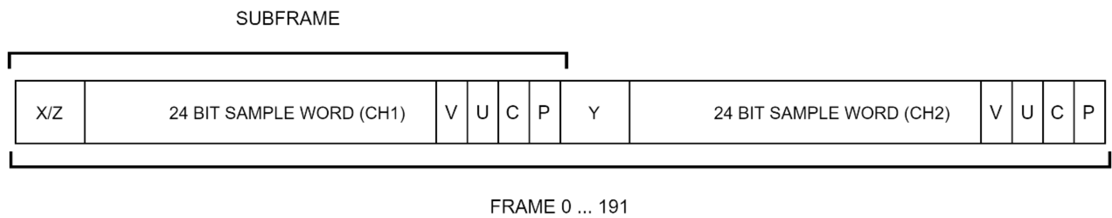

Fig. 2. Estructura de un frame AES completo

La codificación de los datos se realiza en código bifase. Este método de codificación serie se caracteriza por realizar cambios de nivel cada ciclo o dos de reloj, según corresponda a un 1 o 0 lógico. Esta codificación se viola en los preámbulos, que tienen unas formas de onda únicas, ya que es la única manera de garantizar que los patrones de nivel nunca van a repetirse. De esta manera, cada vez que se recibe un preámbulo Z se sincroniza el canal de estado, se reconoce un nuevo bloque y se puede identificar cada uno de sus bits. De la misma manera, cada vez que se recibe una violación del código correspondiente al preámbulo X o Y, se identifican los canales 1 y 2.

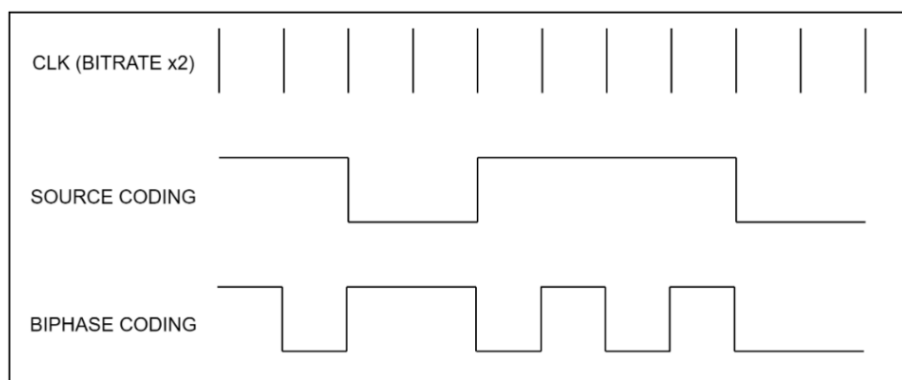

Fig. 3. Código bifase

El canal de estado incluye 192 bits de datos, que aportan información adicional a la de audio. Cada bit tiene asignado un único time slot, por lo que resulta necesario detectar un preámbulo Z para poder sincronizar y detectar correctamente esta información.

Key:

a	use of channel status block	**j**	indication of alignment level
b	linear PCM identification	**k**	channel number
c	audio signal pre-emphasis	**l**	channel number
d	lock indication	**m**	multichannel mode number
e	sampling frequency	**n**	multichanel mode
f	channel mode	**o**	digital audio reference signal
g	user bits management	**p**	reserved but undefined
h	use of auxiliary sample bits	**q**	sampling frequency
i	source word length	**r**	sampling frequency scaling flag
		s	reserved but undefined

Fig. 4. Información adicional del canal de estado (fuente: [1])

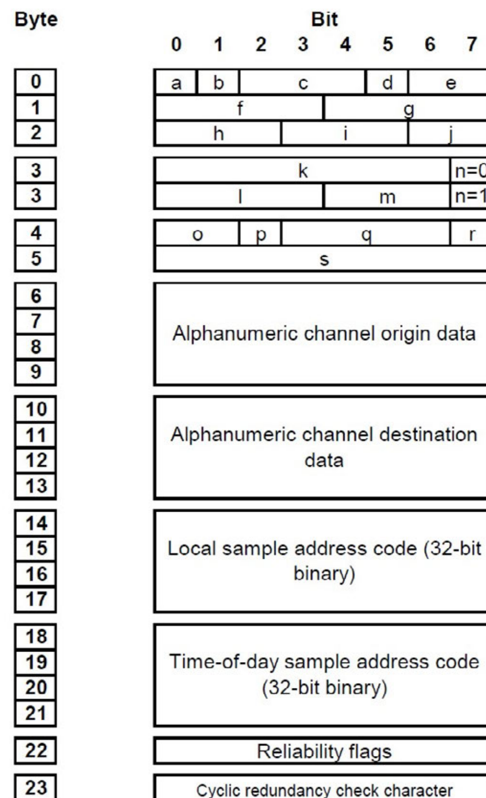

Fig. 5. Estructura del canal de estado (fuente: [1])

Por tanto, la realización de un sistema que transmita y reciba señal de audio en AES/EBU, identifique el canal de estado y modifique las muestras ofrecidas a la salida requiere de la descomposición del problema en una serie de bloques que se abordaran independientemente pero que han de trabajar de manera síncrona.

5.1.2 Diseño estructural

Previamente al proceso de diseño estructural, es importante enumerar y clasificar las especificaciones del firmware:

- Todos los bloques deben de ser sensibles únicamente a la señal de reloj del sistema de 49.192 MHz.
- El sistema ha de ser capaz de manejar señales de audio AES/EBU de 48 kHz. Puesto que la codificación se realiza en bifase, el *bitrate* ha de ser de 3.072 MHz y el

baudrate de 6.114 MHz. Debe de transmitir con el mismo formato y a la misma frecuencia.
- El canal de estado recibido en la entrada debe de propagarse a la salida, o sustituirse si así se desea.
- El reloj del sistema ha de sincronizarse con el reloj de datos de la señal de audio de entrada.
- El diseño debe de disponer de mecanismos de comunicación con otros sistemas externos a la FPGA.
- El sistema debe de disponer de mecanismos de control y chequeo de estado.
- La ganancia aplicada debe estar acotada entre ±12 dB con una resolución de al menos 0.25 dB.
- Todo el sistema trabajara de manera síncrona, por lo que las señales de sincronización serán comunes a todos los bloques.

Tras el análisis de las especificaciones y posterior al proceso de diseño estructural, el firmware queda dividido en 6 bloques principales:

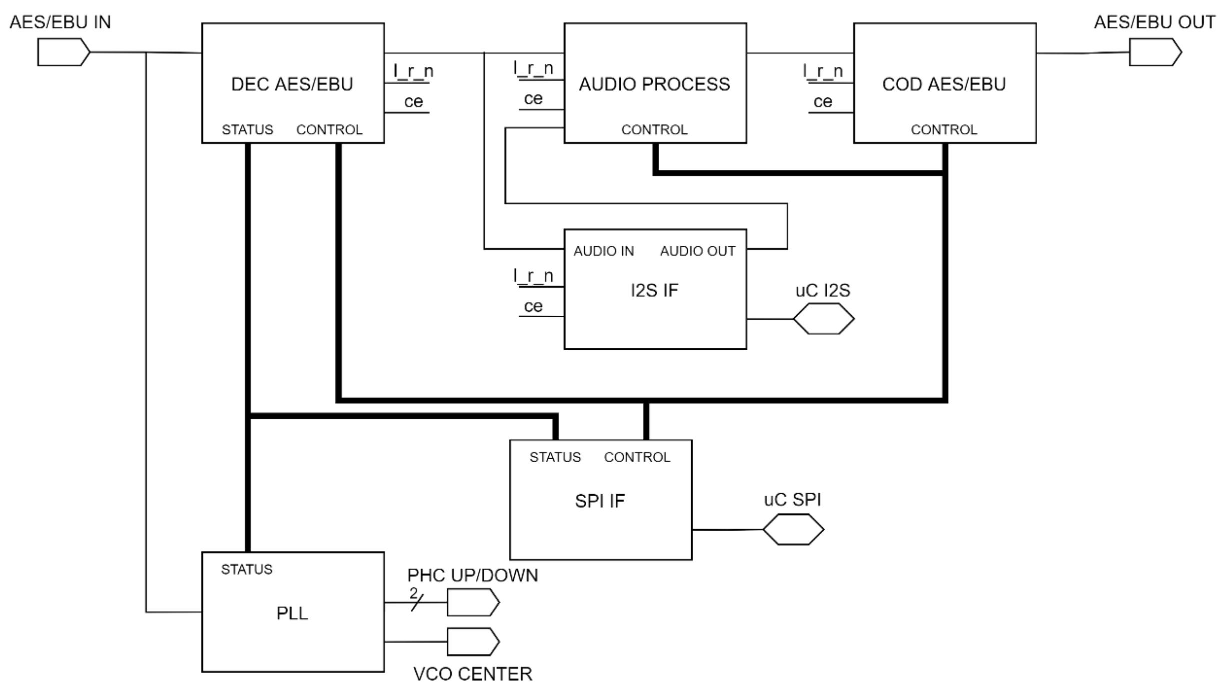

Fig. 6. Diseño estructural del firmware

Como mecanismo de control y supervisión, se han diseñado dos estructuras de control y estado en forma de bus de datos que comunican los bloques del sistema con el exterior de la FPGA. Es una técnica muy útil a la hora de simplificar grandes cantidades de señales en sistemas de media y gran envergadura.

5.1.3 PLL

El bloque PLL recibe la señal directamente del canal de transmisión. Esta señal codificada en bifase es una señal generada con un reloj externo al sistema. Para recuperar el reloj y que la señal de salida esté enganchada con la entrada es necesario utilizar un PLL.

Un PLL (*Phase Locked Loop*) es un sistema ampliamente usado en telecomunicaciones que, entre sus múltiples usos, sirve para enganchar sistemas con señales de frecuencias distintas o no síncronas. Un PLL consta, como mínimo, de los siguientes bloques:

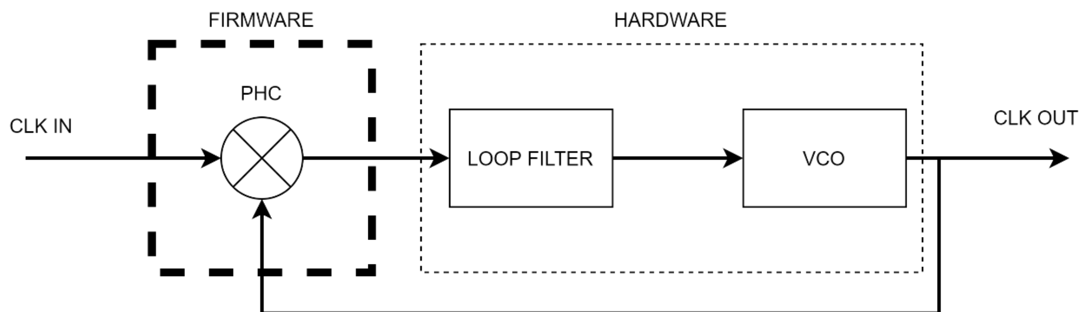

Fig. 7. Estructura de un PLL

En primer lugar se encuentra el detector de fase, cuya salida corresponde con la entrada del módulo *loop filter*. Finalmente se incorpora a la salida de este un oscilador controlado por tensión cuya salida será la señal de reloj del sistema.

En este apartado se va a realizar especial énfasis en el detector de fase (primer módulo de la Figura 7), puesto que es el que se implementa en el firmware. El resto de los componentes del PLL están implementados en el hardware, por lo que se abordarán en el apartado correspondiente.

Para el diseño del PHC (*Phase comparator*) se ha recurrido al documento *Challenges in the design of high-speed clock and data recovery circuits* [5], en el que se explican y valoran numerosos tipos de detectores de fase y sus características. Primando los detectores con respuesta lineal por su fácil integración con el resto del PLL, se ha escogido un detector de tipo *Hogge*.

El detector de fase de *Hogge* genera a su salida dos señales. Una de ellas correspondiente al retraso de la señal muestreada con respecto al reloj del sistema y otra correspondiente al periodo de reloj del sistema. Idealmente, la relación entre ambas ofrece una respuesta lineal del desfase de las señales implicadas, sin embargo, en la práctica, esta respuesta no es completamente lineal debido a efectos de segundo orden.

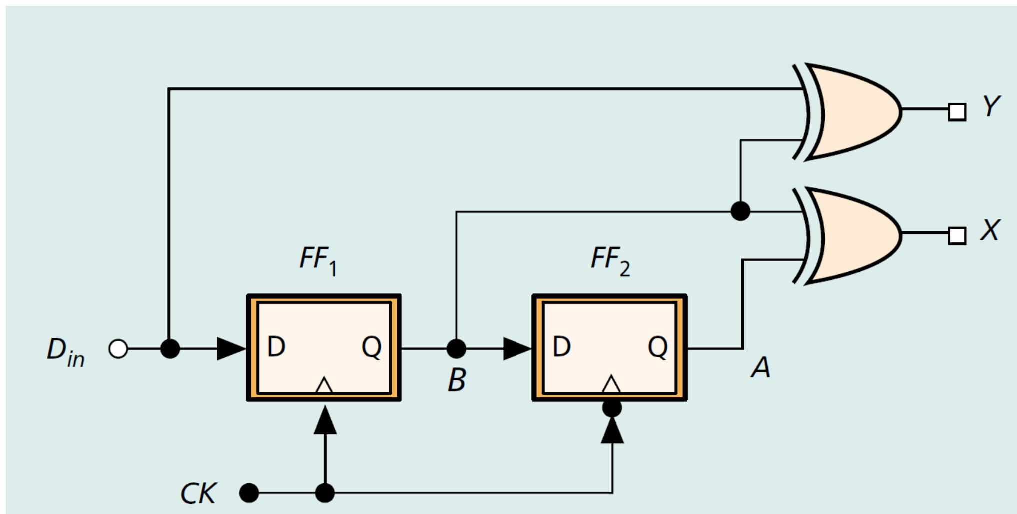

Fig. 8. Detector de fase de tipo Hogge (fuente: [5])

5.1.4 Decodificador

El decodificador es un módulo que recibe la trama AES/EBU, realiza la decodificación del código bifase, realiza la alineación a nivel de frame, subframe o canal y bit y deserializa los datos recibidos. A su salida entrega los 32 bits de datos, una señal que identifica el canal al que pertenecen y un *clock enable* para registrarlos en el siguiente módulo. Estas dos últimas señales son comunes al resto de los módulos que están en el camino de la señal y se emplean para transferir los datos de audio entre ellos. Además, el módulo cuenta con las dos estructuras de datos de estado y control mencionadas anteriormente.

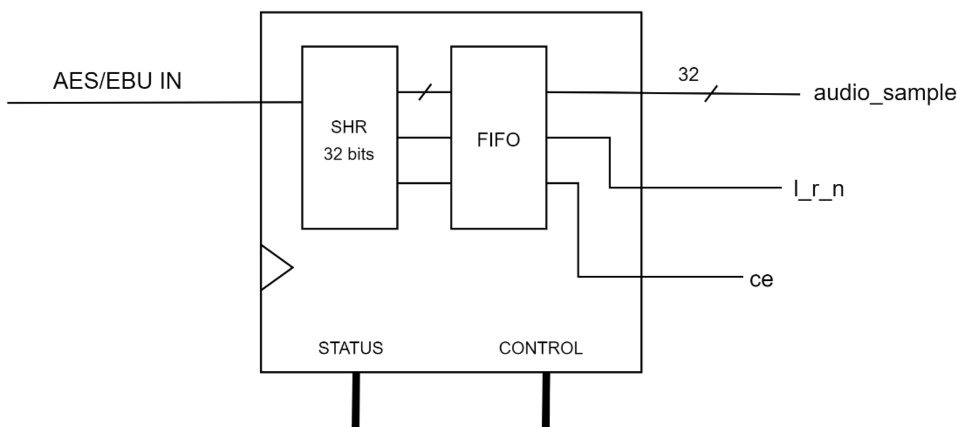

Fig. 9. Diagrama de bloques simplificado de un decodificador AES

La primera fase del decodificador consiste en un registro de desplazamiento con dos fines. El primero de ellos es prevenir la metaestabilidad en los datos que se reciben. El segundo es detectar flancos de subida o bajada e identificar los posibles preámbulos.

Síncronamente al flanco de subida de la señal de entrada, se generan dos pulsos correspondientes al *bitrate* y al *baudrate*. Estas señales son las encargadas de habilitar la escritura en el registro de desplazamiento. Como se ha explicado anteriormente, los preámbulos tienen una forma de onda única, por lo que, comparando este registro de

desplazamiento con las formas de onda particulares de cada preámbulo, se puede ubicar el instante temporal o *time slot* en el que se encuentra la transmisión.

El canal de estado se transmite bit a bit en cada *frame*, por lo que se ha implementado una memoria 24x8 para almacenar los datos. Dicha memoria se incrementa automáticamente y se sincroniza con el preámbulo Z. Estos datos son accesibles a través de la estructura de estado para su gestión por la interfaz de usuario.

Para independizar el flujo de señal del proceso de recepción, este módulo genera dos señales, una para el *clock enable* de las palabras de 32 bits recibidas (ce) y otra indica el canal (l_n_r) al que pertenecen y una memoria tipo FIFO que actúa como buffer para que las muestras que entrega el módulo sean síncronas con las dos señales anteriores.

Finalmente, las muestras de audio son trasmitidas al procesador de audio y a la interfaz I2S, cuya funcionalidad se explica a continuación.

5.1.5 Procesador de audio

Las muestras de audio en paralelo junto con la señal que indica el canal y el *clock enable* son recibidas por el procesador de audio. Este dispone de dos fuentes de señal distintas conmutadas a través de un multiplexor. En función a la fuente escogida, la señal que se procesa es la señal de audio proveniente directamente desde el decodificador o la señal recibida a través de la interfaz I2S, tratada previamente en el microcontrolador.

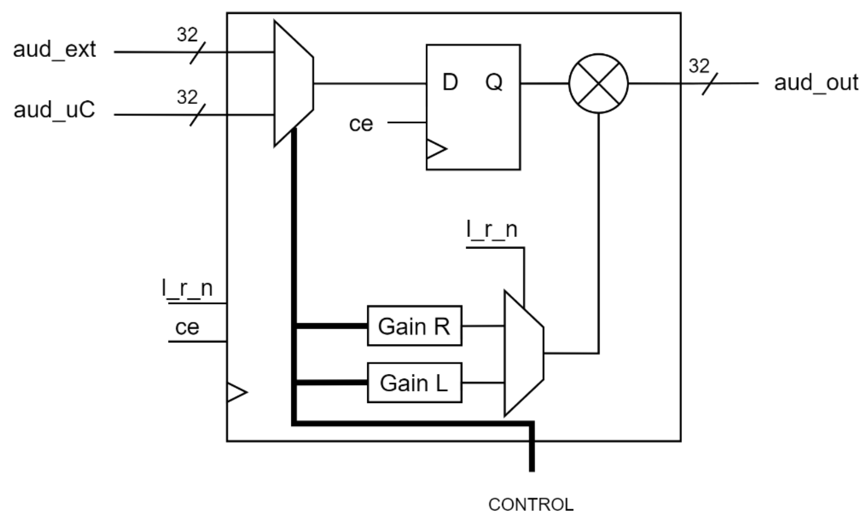

Fig. 10. Diagrama de bloques simplificado de un procesador de audio

El procesador de audio dispone de dos registros de 16 bits para la ganancia de cada canal. Finalmente, a su salida se obtiene la señal escogida a la entrada multiplicada por la ganancia. Además, el diseño dispone de mecanismos de control de desbordamiento, saturando la señal si se sobrepasa el nivel máximo. Los datos de salida son síncronos con la señal que indica el canal y con el *clock enable* y, por tanto, no es necesario implementarlas en la señal de salida. Cabe destacar que tanto los registros de ganancia como la selección de la fuente de señal se manejan a través de la estructura de control.

5.1.6 Interfaz I2S

Para la realización del cálculo de sonoridad, es necesaria una interfaz de comunicación entre microcontrolador y FPGA y así poder realizar transmisiones y recepciones de datos de audio. En este caso se ha optado por la implementación de una interfaz I2S, diseñada en efecto para transmisión de audio digital.

La interfaz I2S consta de cinco líneas: un reloj múltiplo de la frecuencia de muestreo, una señal de alineado de canal, una señal de alineado de bit y dos líneas de datos, una para cada sentido. Las tres primeras líneas son las que se emplean para sincronizar los datos que se transmiten en ambos sentidos y son generadas por la FPGA dado que es la que dispone de un reloj enganchado a los datos que se reciben.

La implementación de la interfaz no es complicada. Se instancian dos registros de desplazamiento que se desplazan en el flanco de subida o bajada de la señal de sincronización de bit, según sean de lectura o escritura. Cuando se completa la transmisión de un canal, los datos del registro de lectura se envían al procesador de audio y se cargan los nuevos valores en el registro de escritura.

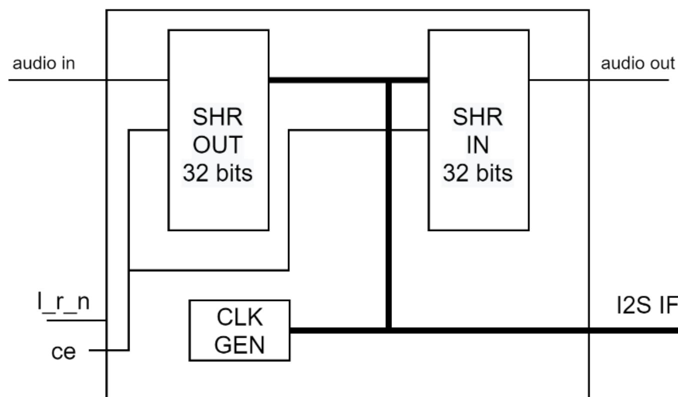

Fig. 11. Diagrama de bloques simplificado de una interfaz I2S bidireccional

5.1.7 Interfaz SPI

El sistema debe de poder monitorizarse a través de la aplicación *Annette* de *Albalá Ingenieros*. Esta aplicación está diseñada para comunicarse directamente con el microcontrolador de la placa *HAM2000*, por lo que resulta necesario incluir una interfaz de comunicación para el manejo de las estructuras de datos implementadas en la FPGA.

Por tanto, se ha diseñado una interfaz SPI, con la que se transmiten al microcontrolador los datos de estado, y se recibe de este toda la información de control necesaria. Para ello, aparte de un esclavo SPI, el módulo de la interfaz incluye una máquina de estados para el manejo de dichas estructuras de manera rápida y sencilla.

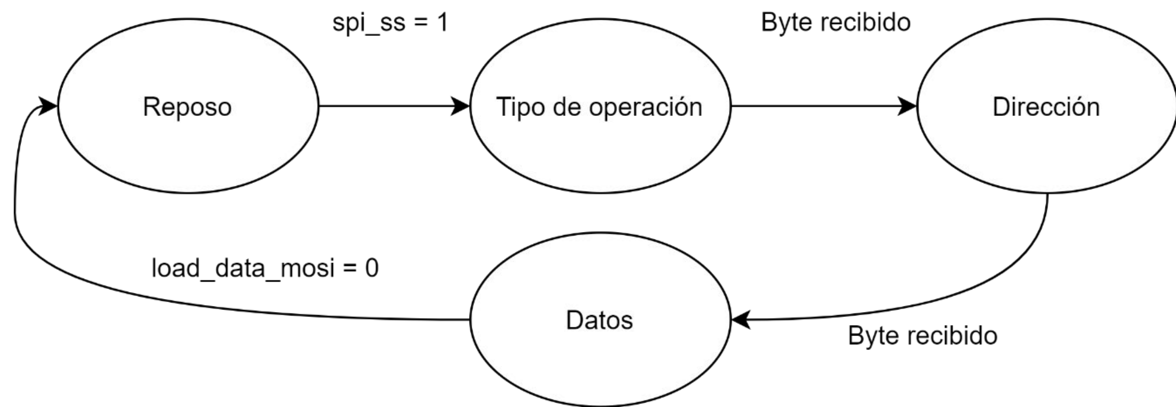

Fig. 12. Autómata de la interfaz SPI

Normalmente la máquina de estados se encuentra en *reposo*. Al detectar el inicio de una transferencia SPI pasa a estado *Tipo de operación*. Tras recibir un byte completo, pasa a estado *Dirección*, en el que vuelve a leer un byte completo. Finalmente pasa a estado *Datos*, donde se quedará hasta que termine la transferencia SPI, para regresar posteriormente a reposo.

Para entender la máquina de estados y las transferencias SPI efectuadas es importante conocer el árbol de registros de las estructuras de control y estado.

CONTROL
WRITE STATUS CHANNEL
ADRESS (7...0)
DATA MOSI (7...0)
BACKPLANE LEDS(1...0)
AUDIO SOURCE SELECTOR
GAIN R (15...0)
GAIN L (15...0)

STATUS
DATA MISO (7...0)
FPGA VERSION(7...0)
NO SIGNAL
UNLOCKED
PARITY ERROR

Fig. 13. Estructuras de control y estado del sistema

A través de la estructura de control se pueden realizar operaciones de escritura, al contrario que la estructura de estado, la cual es únicamente interfaz de lectura.

El primer byte que se recibe en una transferencia SPI, correspondiente al tipo de operación, indica si se va a leer o escribir, es decir, la estructura a la que se quiere apuntar. El segundo, correspondiente al estado *Dirección*, apunta al registro al que se quiere apuntar dentro de la estructura seleccionada. Finalmente, el resto de bytes transmitidos corresponden a los datos que se leen o escriben.

5.1.8 Codificador

El último bloque del sistema es el codificador de AES/EBU. Este módulo es el complementario del decodificador. Recibe los datos en paralelo, la señal que indica el canal al que pertenecen y un *clock enable* y realiza la serialización, la inserción de preámbulos y la codificación bifase.

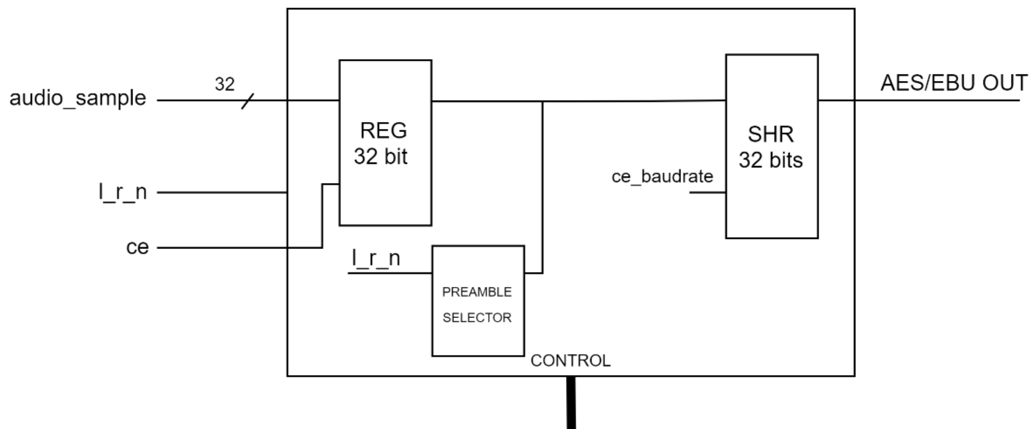

Fig.14. Diagrama de bloques simplificado de un codificador AES

Dispone de un registro de desplazamiento que se carga en paralelo y se desplaza para la codificación en bifase y la transmisión de cada uno de los bits. El canal de estado se introduce directamente desde una memoria 24x8 que se puede escribir a través de la estructura de control. Finalmente se calcula el bit de paridad y añade a la trama.

5.2 Hardware

La tarjeta *HAM2000C01* se incluye dentro de la serie de productos TL2000 de la empresa. Dicha tarjeta dispone de una serie de periféricos para la comunicación con el exterior. En la parte frontal dispone de dos LEDs RGB de notificación y un conector SUB-D de 25 pines para la recepción y transmisión de 8 canales de audio digital. La parte trasera de la placa incorpora un bus de datos para la comunicación con otros módulos del chasis y con la aplicación *Annette* a través de un driver SPI. Adicionalmente y para depuración de diseño, la placa dispone de un puerto serie y de distintos pines de test.

La placa ha sido diseñada para que realice otras funciones además de la de normalizador de audio y dispone de algunas partes del hardware que no se usan. En este documento únicamente se describen las partes de este que se emplean en el normalizador.

5.2.1 Microcontrolador

El microcontrolador SAM E53J19A del fabricante *Microchip* ejecuta todo el software del normalizador. Las características y funcionalidades relevantes con relación al sistema son:

- Núcleo Cortex M4F de 32 bits
- Reloj de sistema de 120 MHz
- Interfaz DMAC
- Interfaz SPI

- Interfaz I2S
- Interfaz SERCOM
- Timers de hasta 24 bits
- Watchdog timer
- 64 pines, 51 de ellos de I/O
- Memoria flash de 512 kB
- Temperatura de trabajo: -40 ºC a +85 ºC

5.2.2 PLL

Como ya se ha explicado anteriormente, un PLL consta de un comparador de fases, un filtro y un oscilador controlado en tensión. En la sección del firmware se ha descrito la parte relativa al comparador de fases que se ha elegido. En este apartado se describen los otros dos bloques y los cálculos realizados para el diseño del PLL.

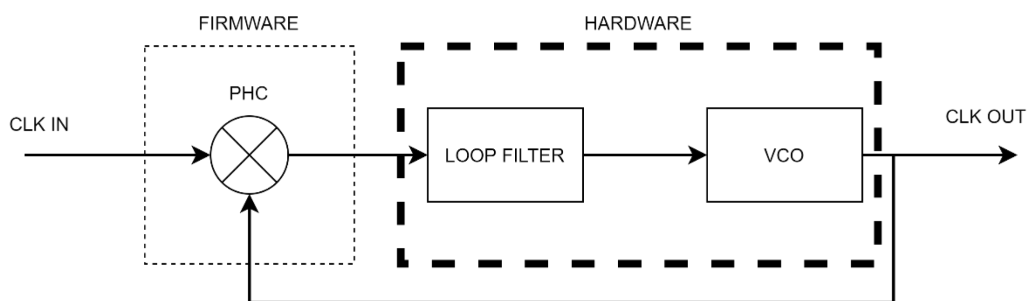

Fig. 15. Estructura de un PLL

La salida del PHC implementada en el firmware del diseño corresponde con la entrada del primero de los bloques hardware del PLL.

El bloque denominado *loop filter* tiene dos funciones: Se encarga de realizar un filtrado paso bajo de la señal recibida del PHC y adapta el nivel con el que se controla el VCO.

La placa HAM2000C01 cuenta con un oscilador controlado en tensión de 27 MHz ± 10 ppm. El margen de control de la tensión de este oscilador es de 0 a 3,3 V. La salida de este oscilador se pasa por un circuito integrado que contiene un divisor fraccionario que permite obtener una frecuencia de 49,152 MHz, que dentro de la FPGA se divide por 8 mediante un divisor de frecuencia para que coincida con los 6,144 Megabaudios de la señal de audio digital. Dado que el comparador de fases funciona a 6,144 MHz, de cara al diseño del PLL se pueden considerar el VCO de 27 MHz, el circuito integrado que multiplica la frecuencia y el divisor que se realiza dentro de la FPGA como si se tratase de un VCO de 6,144 MHz ± 10 ppm de 0 a 3,3V.

Para el diseño del comportamiento del PLL hay que tener en cuenta las características de cada uno de los módulos que intervienen en él. Todo este proceso de diseño se ha basado en las directrices del libro *Phaselock techniques* [6]. En dicho libro, se especifican una serie de procedimientos y parámetros para el diseño de un PLL.

Para realizar el diseño es necesario caracterizar las funciones de transferencia del comparador de fases y del VCO, especificar la función de transferencia que debe tener el PLL y

finalmente calcular los valores de los componentes del filtro paso bajo que se necesitan para obtener esa respuesta. Para ello:

- La función de transferencia del comparador de fases está caracterizada por su factor de ganancia k_d:

  El firmware implementa un detector de fase de tipo *Hooge*, que relaciona de manera lineal la tensión a su salida entre 0 y 3.3 V con la diferencia de fase entre la entrada y la salida del PLL.

  Por tanto, para el cálculo del factor de ganancia de PHC (K_d):

$$k_d = \frac{\Delta V}{\Delta \varphi} = \frac{3.3\ V}{2\pi\ rad} \approx 0.52\ \left[{}^{V}/_{rad}\right] \tag{1}$$

- La función de transferencia del VCO está caracterizada por su factor de ganancia k_o:

  De acuerdo con lo que se ha comentado anteriormente los parámetros que hay que emplear para calcular el factor de ganancia del VCO son la frecuencia nominal del oscilador 6,144 MHz y margen de control de la frecuencia 10 ppm (~60 Hz) de 0 a 3,3 V. Con estos datos el factor de ganancia del VCO se puede calcular de la siguiente manera:

$$k_o = \frac{\Delta \omega}{\Delta V} = \frac{2\pi \Delta f}{\Delta V} = \frac{240\pi\ rad/s}{3.3\ V} \approx 229\ [{}^{rad/s}/_{V}] \tag{2}$$

Un PLL realizado con un filtro de primer orden en el bucle de realimentación da lugar a una función de transferencia de segundo orden que queda completamente caracterizada por dos parámetros, su frecuencia natural y el factor de amortiguamiento. A continuación se describe cómo se han elegido estos dos parámetros para obtener la respuesta del PLL deseada.

La función de transferencia de un PLL de segundo orden tiene el siguiente aspecto:

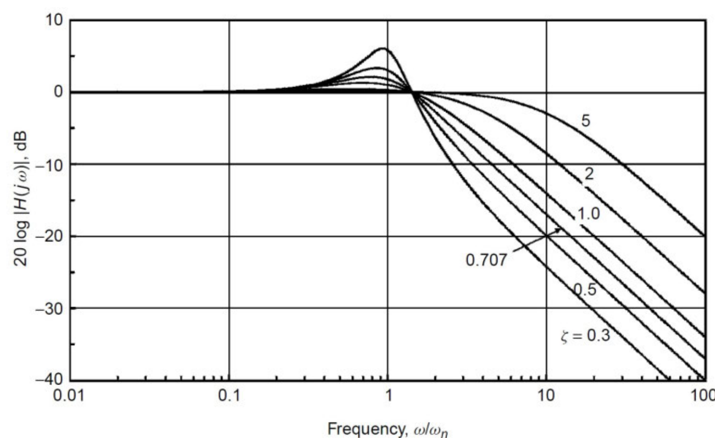

Fig. 16. Respuesta en frecuencia de un PLL de segundo orden en función al factor de amortiguamiento (fuente: [6])

Y se caracteriza por su frecuencia natural y su factor de amortiguamiento:

- Frecuencia natural:
  La frecuencia natural está relacionada con la frecuencia de corte del filtro paso bajo que constituye el PLL para las componentes de *jitter* de la señal. Por este motivo, para la elección de esta variable hay que tener en cuenta las especificaciones de *jitter* de la norma AES/EBU.
  El gálibo que propone la norma para la función de transferencia de *jitter* es el siguiente:

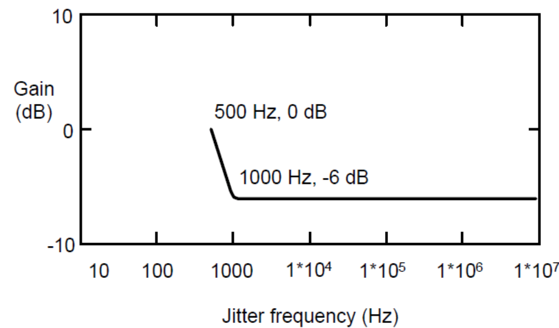

Fig. 17. Función de transferencia de jitter según la norma AES (fuente: [1])

  Para cumplir de manera holgada la especificación, se opta por una frecuencia natural de una décima parte de la especificación. Por tanto:

$$f_n = \frac{1000\ Hz}{10} = 100\ [Hz] \tag{3}$$

$$\omega_n = 2\pi f_n = 200\pi \left[{}^{rad}/_{s}\right] \tag{4}$$

- Factor de amortiguamiento:
  El factor de amortiguamiento caracteriza la respuesta transitoria del PLL. Un factor de amortiguamiento muy alto puede provocar sobreamortiguamiento en la respuesta transitoria y una amplificación del *jitter* de la señal. Por contra, un factor demasiado reducido puede provocar una respuesta subamortiguada y dar lugar a una respuesta transitoria muy lenta.
  Por tanto, siguiendo la recomendación de diseño [6]:

$$\zeta = 0.707 \tag{5}$$

Finalmente, para el filtro de bucle se va a utilizar a siguiente configuración, que constituye un filtro paso bajo activo de un integrador ideal:

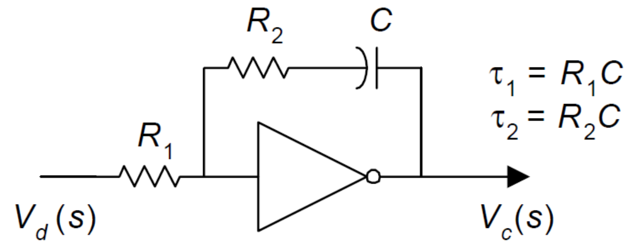

Fig. 18. Filtro de primer orden para la etapa loop filter (fuente: [6])

Cuya función de transferencia es:

$$F(s) = -\frac{s\tau_2 + 1}{s\tau_1} \tag{6}$$

Finalmente, la función de transferencia en función de los distintos componentes que forman el PLL queda como sigue:

$$\omega_n = \sqrt{\frac{K_d K_o}{\tau_1}}$$

$$\zeta = \frac{\tau_2}{2}\omega_n \tag{7}$$

$$H(s) = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

Sustituyendo los valores de los factores de ganancia del comparador de fases y del VCO y los parámetros de diseño del PLL en estas ecuaciones se obtienen los siguientes valores para los componentes.

Tabla 1. Valores de los componentes del filtro

C	R_1	R_2
1 uF	0.3 kΩ	2.2 kΩ

5.2.3 Multiplicador de frecuencia

La señal de reloj de la FPGA se genera externamente a esta, a partir de un oscilador de 27 MHz. Para obtener una frecuencia múltiplo de la frecuencia de muestreo del audio que se recibe se utiliza un circuito integrado ICS661 que es un multiplicador de frecuencia configurable diseñado específicamente para este fin. La interfaz de programación del chip consiste en la puesta a 0 o a 1 de sus 4 entradas digitales conectadas al microcontrolador. Por tanto, es muy sencillo cambiar la frecuencia de funcionamiento del firmware a través de software.

5.2.4 FPGA

La FPGA montada en la placa es una *Spartan 3E* del fabricante *Xilinx*.

Las características principales de esta FPGA son:

- 100 pins
- Temperatura de trabajo: 0 ºC a +80 ºC
- 33.192 celdas lógicas
- Multiplicadores 18x18
- 8 DCMs
- 648 Kbits RAM

5.2.5 Drivers de I/O

La norma AES/EBU marca que la transmisión y recepción ha de hacerse a través de cables de tensión balanceada. Para adecuar la señal tanto de entrada como de salida, se dispone de unos drivers configurables por software, de tal manera que se pueden seleccionar las 8 interfaces de audio como entradas y salidas independientes.

5.3 Software

El software del normalizador es el encargado de realizar el cálculo de la sonoridad, nivel de pico y ganancia. Se ha partido de un esqueleto de programa proporcionado por la empresa que incluye todo el código encargado de la comunicación con el chasis sobre el que va montada la placa y la transmisión y recepción de datos entre esta y la aplicación web de control. De esta manera, el desarrollo del software restante incluye:

- Recepción y transmisión de muestras de audio a través de I2S
- Recepción y transmisión de datos de estado y control a través de SPI
- Cálculo de sonoridad
- Cálculo de nivel de pico
- Cálculo de ganancia aplicada

Antes de codificar, se ha diseñado un algoritmo que cumpla las especificaciones citadas:

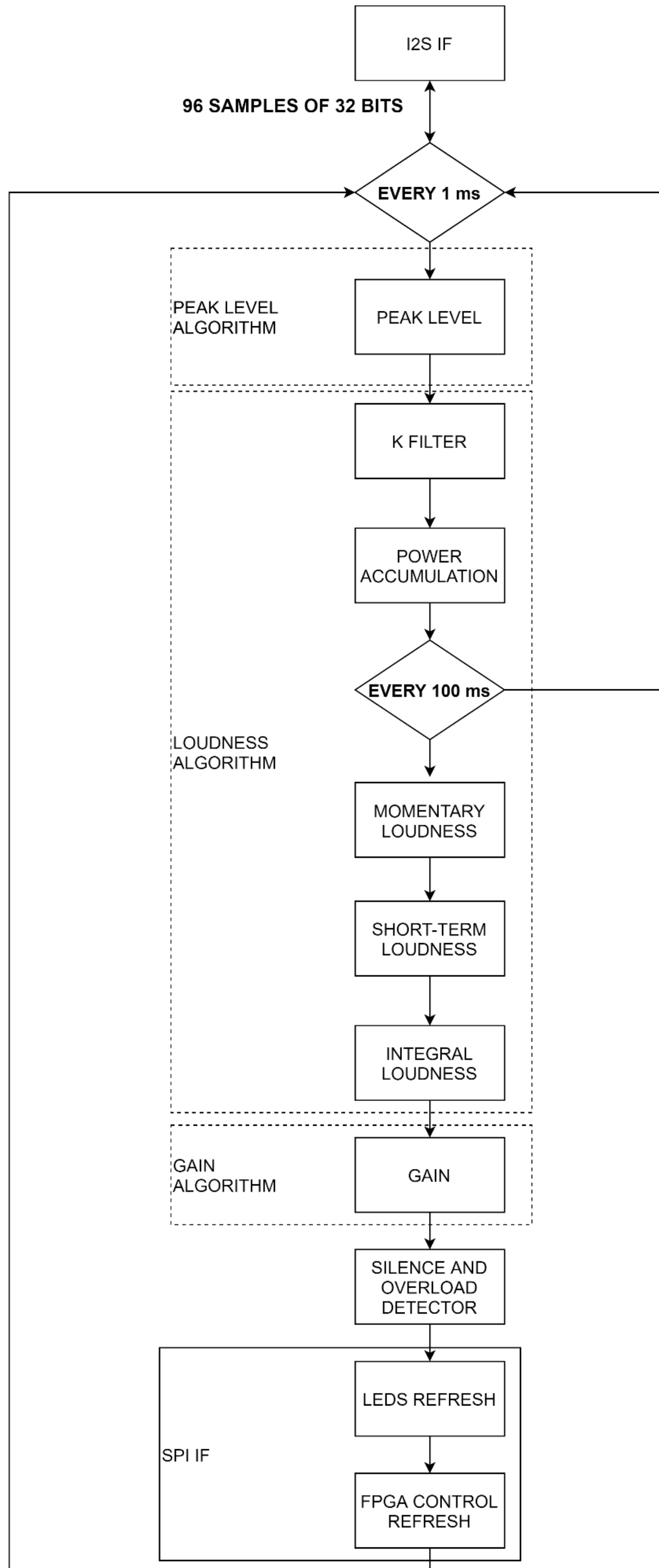

Fig. 19. Algoritmo del software de la aplicación

Para la implementación de ambos drivers de comunicaciones (I2S y SPI) se ha recurrido a las librerías que pone a disposición *Atmel* (actualmente *Microchip*), el fabricante del microcontrolador.

5.3.1 Interfaz I2S

La interfaz I2S que se instancia automáticamente a través de las librerías no funciona correctamente a la hora de trabajar como esclavo. Por ello, ha habido que modificar, con ayuda de los ingenieros de la empresa, dicha interfaz.

Para hacerlo funcionar ha sido necesario cambiar el orden en el que se inicializan los distintos elementos del I2S en el *driver*. Además, ha sido necesario cambiar algunos parámetros de las librerías, en concreto en el fichero hpl_i2s_config.h, donde hay que modificar el parámetro CONF_I2S_0_CLKSEL_1 para que tenga el valor 0 y tome el reloj del Clock Unit 0.

La interfaz I2S funciona con DMA. Cuando se ha recibido un bloque de N muestras de audio por esta interfaz salta la interrupción del canal que se ha asignado a la recepción de datos por I2S. Esta interrupción se utiliza para inicializar los descriptores de las siguientes transferencias de DMA. Esto debe de hacerse de la manera más rápida posible. Para conseguirlo se ha mantenido para esta interrupción la prioridad más alta (0) que es la que tiene por defecto y se han bajado las de los otros periféricos (SPI_BUS (1), TIMER0 (2), DEBUG_UART (2), I2C (2)). Esto permite que el NVIC (Nested Vectored Interrupt Controler) priorize la rutina de atención a interrupciones del I2S aunque esté atendiendo otra.

Los datos que se reciben o se leen se encuentran en la dirección del puntero de memoria asignado y cada vez que salta la interrupción del DMA, se ejecuta su rutina de atención a interrupciones que lee y escribe en dicha memoria.

En el código se distinguen dos bufferes distintos. A la vez que se leen los valores recibidos en uno de ellos, se escribe en el otro para su posterior transmisión. En la siguiente interrupción, estos bufferes intercambiaran los papeles, automatizando y agilizando lecturas y escrituras a través de esta interfaz.

```
// DMA CH 1 call back function
static void I2S_RX_dma_complete_callback(struct _dma_resource
*resource)
{
	uint8_t l_r_n_ok = TRUE;
	uint8_t i2sc_rx_dma_channel_number = 1;
	uint8_t i2sc_tx_dma_channel_number = 0;

	#if DEBUG_I2S == TRUE
	//gpio_set_pin_level(RESET_ASRC_N,true);
	gpio_set_pin_level(TEST0,true);
	#endif

	if (gpio_get_pin_level(I2S_FS0)==0) {
		l_r_n_ok = FALSE;
	} else {
		rx_aud = TRUE;
	};
	if (buffer_in_use==0) {
		buffer_in_use = 1;
		_dma_set_source_address(i2sc_rx_dma_channel_number, (const
		void *) &I2S->RXDATA.reg);
		_dma_set_destination_address(i2sc_rx_dma_channel_number,
		(const void *)i2sc_rx1_buf);
		_dma_set_data_amount(i2sc_rx_dma_channel_number,
		I2S_AUD_BUF_LEN);
		_dma_set_source_address(i2sc_tx_dma_channel_number, (const
		void *)i2sc_tx1_buf);
		_dma_set_destination_address(i2sc_tx_dma_channel_number,
		(const void *) &I2S->TXDATA.reg);
		_dma_set_data_amount(i2sc_tx_dma_channel_number,
		I2S_AUD_BUF_LEN);
		rx_buf=i2sc_rx0_buf;
		tx_buf=i2sc_tx0_buf;
	} else {
		buffer_in_use = 0;
		_dma_set_source_address(i2sc_rx_dma_channel_number, (const
		void *) &I2S->RXDATA.reg);
		_dma_set_destination_address(i2sc_rx_dma_channel_number,
		(const void *)i2sc_rx0_buf);
		_dma_set_data_amount(i2sc_rx_dma_channel_number,
		I2S_AUD_BUF_LEN);
		_dma_set_source_address(i2sc_tx_dma_channel_number, (const
		void *)i2sc_tx0_buf);
		_dma_set_destination_address(i2sc_tx_dma_channel_number,
		(const void *) &I2S->TXDATA.reg);
		_dma_set_data_amount(i2sc_tx_dma_channel_number,
		I2S_AUD_BUF_LEN);
		rx_buf=i2sc_rx1_buf;
		tx_buf=i2sc_tx1_buf;
	}
	if (l_r_n_ok == FALSE) {
		do {} while (gpio_get_pin_level(I2S_FS0)==0);
	}
	do {} while (gpio_get_pin_level(I2S_FS0));
	_dma_enable_transaction(i2sc_rx_dma_channel_number, false);
	_dma_enable_transaction(i2sc_tx_dma_channel_number, false);
}
```

Función Auxiliar 1. Rutina de atención a interrupciones por DMA

5.3.2 Interfaz SPI

La instanciación del driver SPI se realiza de manera correcta con las librerías disponibles. Como la comunicación a través de SPI es menos densa en datos y se realiza menos veces por segundo que las transferencias I2S, no ha sido necesario incrementar el nivel de prioridad de este driver.

Para realizar una correcta modularización del código, se han desarrollado dos funciones, de escritura y lectura de datos, a las que se recurre siempre que es necesario leer o actualizar algún valor de las estructuras de datos de la FPGA:

```
/*
 * Name:                SPI_FPFA_read_op
 * Description:         Reads n_data_bytes from the specified fpga area
 */

#define WR_OP                       0x00
#define RD_OP                       0x80

void SPI_FPGA_read_op(uint8_t area, uint8_t *data_ptr, uint16_t
                        n_data_bytes) {
      uint8_t op_add[2] = {area|RD_OP, 0x00};
      struct io_descriptor *io;
      gpio_set_pin_level(SPI_FPGA_SS_N,false);
      spi_m_sync_get_io_descriptor(&SPI_FPGA, &io);
      io_write(io,op_add,2);
      io_read(io,data_ptr,n_data_bytes);
      gpio_set_pin_level(SPI_FPGA_SS_N,true);
}

/*
 * Name:                SPI_FPGA_write_op
 * Description:         Writes n_data_bytes to the specified fpga area
 */

void SPI_FPGA_write_op(uint8_t area, uint8_t *data_ptr, uint16_t
                        n_data_bytes) {

      uint8_t op_add[2] = {area, 0x00};
      struct io_descriptor *io;
      gpio_set_pin_level(SPI_FPGA_SS_N,false);
      spi_m_sync_get_io_descriptor(&SPI_FPGA, &io);
      io_write(io,op_add,2);
      io_write(io,data_ptr,n_data_bytes);
      gpio_set_pin_level(SPI_FPGA_SS_N,true);
}
```

Función Auxiliar 2. Escritura y lectura SPI

Como se puede observar, la línea de activación de comunicación ha de controlarse a través de GPIO, ya que el *driver* por defecto no contempla esta característica de la interfaz.

5.3.3 Algoritmo de cálculo de sonoridad

En cuanto al cálculo de sonoridad, la recomendación ITU-R BS.1770-4 propone un algoritmo y una serie de especificaciones para una correcta medida.

Este algoritmo se divide en cuatro fases fundamentales, siendo las dos primeras independientes para cada canal n. A la entrada (x_n) se esperan muestras de audio digitales de, en este caso particular, 32 bits. En el proyecto únicamente se han utilizado los canales L (*Left*) y R (*Right*):

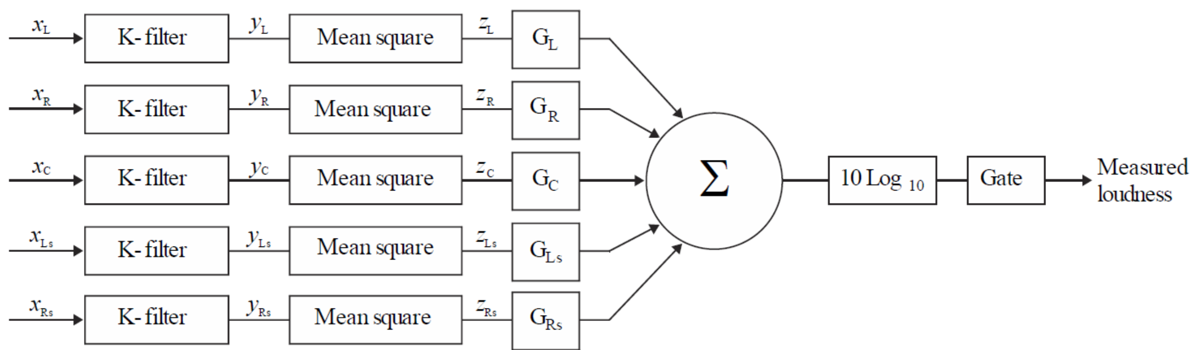

Fig. 20. Técnica de cálculo de sonoridad (fuente: [3])

- Filtro K:

Esta primera fase implementa un filtro IIR de segundo orden cuyos coeficientes están definidos en la recomendación. Su función es modelar la forma de la cabeza humana, con el fin de realizar la posterior medida lo más fiel posible a la percepción auditiva del ser humano.

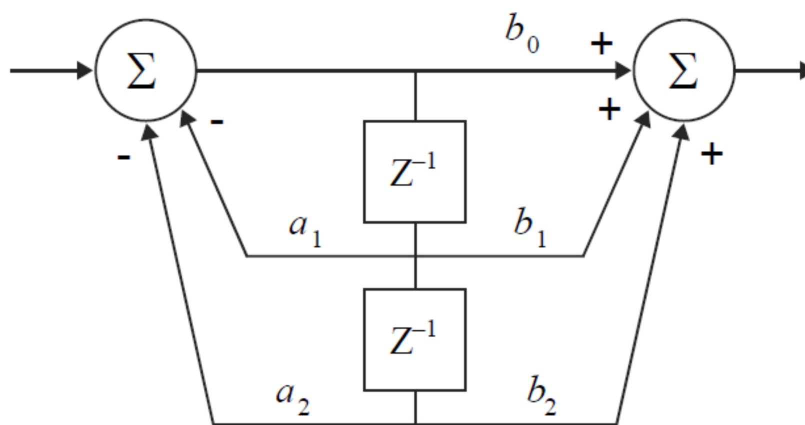

Fig. 21. Filtro de segundo orden para la primera fase del cálculo de sonoridad (fuente: [3])

Tabla 2. Coeficientes del filtro k

a_1	a_2	b_0	b_1	b_2
-1.69065929318241	0.7348077421585	1.53512485958697	-2.69169618940638	1.19839281085285

- *Mean square*:

La siguiente fase consiste en el cálculo de la potencia en función a la amplitud de cada canal. La potencia z de un canal i en un periodo T se calcula de la siguiente manera:

$$z_i = \frac{1}{T}\int_0^T y_i^2\, dt \tag{7}$$

- Sumatorio:

Las potencias de cada canal se suman para calcular la potencia total. En función del canal, se le aplica a esta potencia una ponderación determinada. En este caso particular, al disponer únicamente de dos canales, ambas ponderaciones serán de 1. Por tanto, la potencia total z será la suma de las potencias parciales z_i multiplicadas por su factor de ponderación G_i:

$$z = \sum_i G_i * z_i \tag{8}$$

- *Gatting*:

Se calcula la sonoridad intermedia L_K, a través del sumatorio de potencia calculado anteriormente, en unidades que se denominan LKFS:

$$L_K = -691 + 10log_{10}(z) \tag{9}$$

Finalmente, cabe la posibilidad de realizar un nuevo cálculo de la sonoridad, eliminando los niveles que no producen sensación sonora o la que producen es muy débil. Es por tanto un filtrado en el que solo se aceptan como válidas para el cálculo muestras que están por encima de un cierto umbral.

Este proceso se realiza en dos fases. La primera, desecha todas las muestras que estén por debajo de -70 LKFS. La función de esta primera fase es eliminar del cálculo componentes de audio tan débiles que se pueden considerar ruido y que falsearían la medida. Tras realizar de nuevo el cálculo de la sonoridad, se aplica un *gatting* relativo p, que, a diferencia de la primera fase, elimina del cálculo final todas las muestras por debajo de un nivel relativo al último cálculo realizado. Esto es:

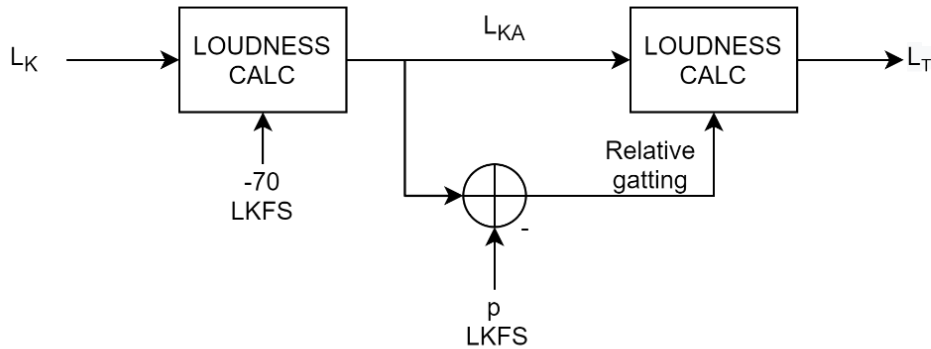

Fig. 22. Algoritmo de cálculo de sonoridad con gatting

El algoritmo se ha aplicado según a la norma EBU Tech. 3341, en la que se especifican una serie de parámetros como el periodo de integración de los cálculos.

Esta norma define 3 tipos de medida:

- Sonoridad momentánea (400 ms)
- Sonoridad a corto plazo (3 s)
- Sonoridad integral con *gatting* (ventana de tiempo variable)

La norma específica que, al menos, debe de existir un 75% de superposición entre los periodos de cálculo, lo que supone su realización cada 100 ms.

Para la implementación del algoritmo, se han utilizado las herramientas de CMSIS [7].

CMSIS engloba una serie de librerías y funcionalidades de uso gratuito para multitud de microcontroladores. Entre sus utilidades, dispone de unas librerías con funciones muy optimizadas para cálculos de DSP (*Digital Signal Processor*). Estas librerías han resultado de gran ayuda a lo largo del proceso de codificación.

Prestando atención al código, partiendo de la base de que las muestras de audio han sido guardadas correctamente por la interfaz I2S en el buffer de recepción, el primer proceso consiste en el filtrado de las muestras a través del filtro K. Para la implementación de este filtro se ha recurrido a la función *arm_biquad_cascade_df1_f32()* del paquete de DSP de CMSIS que implementa un filtro de segundo orden.

```
y[n] = b0 * x[n] + b1 * x[n-1] + b2 * x[n-2] + a1 * y[n-1] + a2 * y[n-2]
```

A Direct Form I algorithm is used with 5 coefficients and 4 state variables per stage.

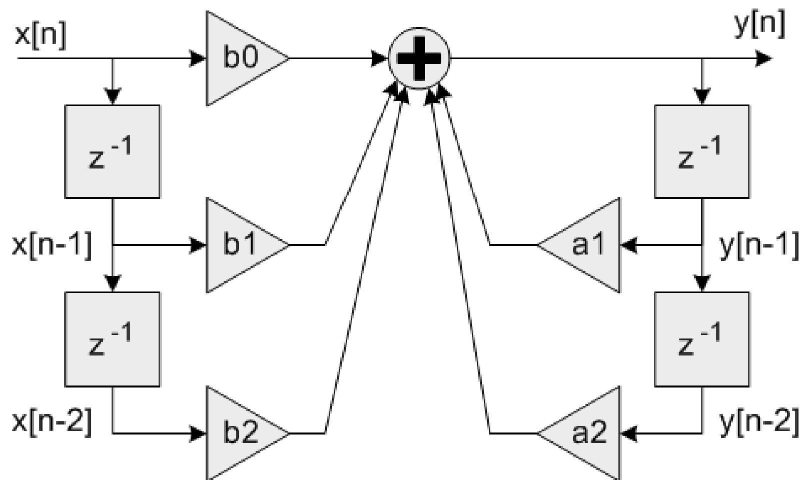

Fig. 23. Filtro de segundo orden que implementa la función arm_biquad_cascade_df1_f32 (fuente: [7])

En primer lugar, se realiza la instanciación de sendos filtros, uno para cada canal de audio, a través de la función de inicialización que tiene a disposición CMSIS. En los parámetros de la función se le indican los buffers de entrada y salida, coeficientes y numero de muestras a procesar. Todo ello se engloba dentro de la siguiente función:

```
/*
 * Name:           filter_init
 * Description:    Init CMSIS 2 order filters with the chosen parameters
 */
void filter_init(void){

      arm_biquad_cascade_df1_init_q31 (&kfilterR, numStages,
      secondOrderFilterCoefs, stateR, postshift);
      arm_biquad_cascade_df1_init_q31 (&kfilterL, numStages,
      secondOrderFilterCoefs, stateL, postshift);
}
```

Función Auxiliar 3. Inicialización de filtros K

Puesto que hay que realizar el cálculo de sonoridad cada 100 ms, se codifican las fases previas (filtrado y potencia) para que se ejecuten cada 1 ms y el resto del algoritmo cada 100 ms.

El seudocódigo de las dos primeras fases se muestra a continuación:

Cada 1 ms

Filtrar N muestras de cada canal a través de filtros de CMSIS

Calcular potencias a través de las funciones de CMSIS

Sumar potencias a los buffers de potencia de 100 ms

Repetir

Para terminar, cada 100 milisegundos se realiza un cálculo de sonoridad momentánea, sonoridad a corto plazo y sonoridad integral. Aquí reside uno de los problemas principales a los que ha habido que enfrentarse. Debido a la gran cantidad de cálculos que hay que realizar para obtener estos tres valores, el tiempo que necesita el procesador para realizar todo el algoritmo puede exceder el milisegundo. Al ocurrir esto, antes de acabar una medida tendrá que empezar a ejecutar de nuevo el algoritmo, lo que produce el mal funcionamiento de este. Para eso, dadas las características de la aplicación, se ha decidido dividir en *slots* de tiempo los distintos cálculos. Por tanto, se dispone de 100 *slots* de 1 ms para realizar los cálculos que se ejecutan cada 100 ms. Haciendo uso del osciloscopio y programando pines de test, se evalúan los tiempos de procesado de cada uno de los tres cálculos de sonoridad. Finalmente se separan, por un lado, la sonoridad momentánea y sonoridad a corto plazo y, por otro, la sonoridad integral en dos *slots* distintos.

El seudocódigo de este algoritmo se muestra a continuación:

Cada 1 ms

Si contador_100_ms = 1

Calcular potencia media de los buffers de potencias de 100 ms

Calcular sonoridad momentánea

Añadir potencia al buffer de 400 ms para calcular sonoridad integral

Calcular sonoridad integral

Actualizar valores en la interfaz de usuario

Si no, Si contador_100_ms = 2

Calcular sonoridad con gatting absoluto

Calcular nivel de gatting relativo

Si no, Si contador_100_ms = 3

Calcula sonoridad integral

Actualizar valores en la interfaz de usuario

Si no, Si contador_100_ms = 100

Reset de contador_100_ms

Fin Si

Incrementar contador_100_ms

Repetir

Para detalles de codificación, consultar el Anexo 10.2.

5.3.4 Algoritmo de cálculo de nivel de pico

Siguiendo con la lista de especificaciones del software, hay que desarrollar un algoritmo detector de pico. Este está constantemente comparando las amplitudes de la señal. Siempre que el valor a comparar sea mayor que el valor máximo almacenado, se actualiza. En caso contrario, la muestra pasa por un filtro de primer orden que atenúa su amplitud de tal manera que, en un periodo cercano a 1.7 segundos, esta haya disminuido en torno a 20 dB.

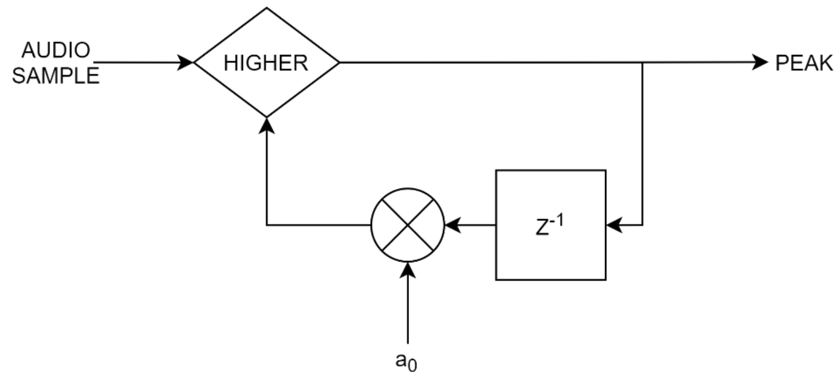

Fig. 24. Algoritmo de cálculo de nivel de pico

Tabla 3. Coeficiente del filtro de primer orden del detector de pico

a_0
0.8733

Para implementar este algoritmo, se ha creado una función a la que se llama cada vez que es necesario realizar el cálculo de nivel de pico. Debido a las imperfecciones del ojo humano y para no abrumar al usuario con demasiados cambios en la interfaz de usuario, este valor solo se actualizará cada 100 ms. El seudocódigo que caracteriza este algoritmo se muestra a continuación:

Cada 1 ms

Seguidor de nivel de pico

Cada 100 ms

Si nivel_pico > nivel actual

Actualizar valor en la interfaz de usuario

Si no

Aplicar filtro

Actualizar valor en la interfaz de usuario

Fin Si

Repetir

Para detalles de codificación, consultar el Anexo 10.2.

5.3.5 Algoritmo de cálculo de ganancia

Posterior al cálculo de sonoridad y el nivel de pico, ha de implementarse otro algoritmo que calcule la ganancia que hay que dar a cada canal para conseguir el nivel deseado. Para ello, hay que tener en cuenta que un constante cambio de ganancia instantáneo grande puede provocar una percepción auditiva desagradable en el oyente. Por tanto, se ha incluido a su salida un filtro de primer orden que suaviza la respuesta transitoria de la ganancia.

Como referencia, se propone que la respuesta al escalón del sistema alcance el 80% de la ganancia deseada en 5 segundos transcurridos. Así, el algoritmo de cálculo de ganancia con los coeficientes del filtro calculados queda de la siguiente manera:

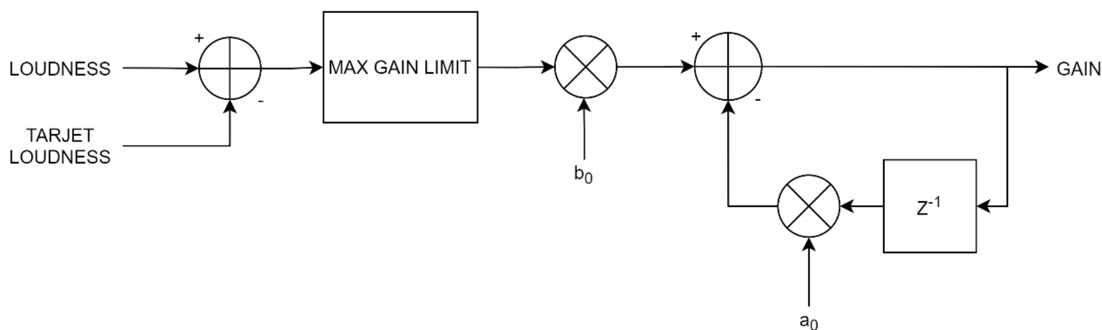

Fig. 25. Algoritmo de cálculo de ganancia

Tabla 4. Coeficientes del filtro de aplicación de ganancia

a_0	b_0
0.968	0.032

Se muestra a continuación el seudocódigo que caracteriza el algoritmo de cálculo de ganancia:

Cada 100 ms

Calcular ganancia necesaria

Si ganancia necesaria > ganancia máxima

Activar notificación de sobrecarga

Si no

Aplicar filtro de ganancia

Enviar ganancia a los registros del procesador de audio

Fin Si

Repetir

Para detalles de codificación, consultar el Anexo 10.2.

5.4 Sistema completo

Finalmente se integran los tres bloques en conjunto. Para automatizar la placa, tanto el firmware como el software se guardan en una memoria flash. De esta manera, a través del *bootloader* desarrollado por la empresa, la tarjeta arranca sola y programa la FPGA cada vez que se conecta al chasis.

Para la gestión y configuración de esta, se accede a una aplicación web, *Annette*, que se comunica con la tarjeta *HAM2000* a través de la tarjeta de comunicaciones del chasis.

En la interfaz que se presenta al usuario se pueden observar todos los campos de interés del normalizador, como son los tres tipos de sonoridad, niveles de pico de ambos canales, notificación de errores y parámetros configurables:

Fig. 26. Interfaz de usuario del normalizador de audio

Solución

6 Resultados

El sistema ha de someterse a una serie de pruebas para su validación. Dada la envergadura de este, al igual que en el apartado de diseño, se dividen las pruebas en 4 tipos en función del subsistema sobre el que se está centrando el test.

6.1 Firmware

Los entornos de desarrollo de diseño digital actuales disponen de herramientas muy potentes que permiten simular prácticamente cualquier eventualidad que se les proponga. En este caso se ha usado el simulador incluido en el programa de diseño de Xilinx como herramienta de depuración. A continuación, se muestra por separado el comportamiento de cada uno de los bloques del sistema con una breve explicación.

Antes de todo se generan las señales de entrada al sistema para testearlo. Por simplicidad y para la mejor depuración, en el canal 1 se generan muestras de audio de 24 bits con un contador incremental en cada *frame*. Por otro lado, en el canal 2 se introduce un silencio constante.

En primer lugar, se puede observar el funcionamiento del decodificador. Para ello, se capturan las formas de onda principales del módulo, en las que se visualiza el final de la recepción de un *subframe* y el comienzo del nuevo. Se puede ver como el decodificador detecta la violación del código bifase, comparando el registro de desplazamiento del que dispone a su entrada con la forma de onda de los preámbulos. En efecto, en este caso ha detectado un preámbulo X. Puesto que al detectar este preámbulo se inicia un contador sensible a la señal de *baudrate*, no es necesario detectar el preámbulo Y, ya que en todo momento se sabe en qué *slot* se tiempo se encuentra la comunicación. Además, se muestran las señales de dirección y datos del canal de estado, que se incluyen dentro de las señales de la estructura de control del sistema completo.

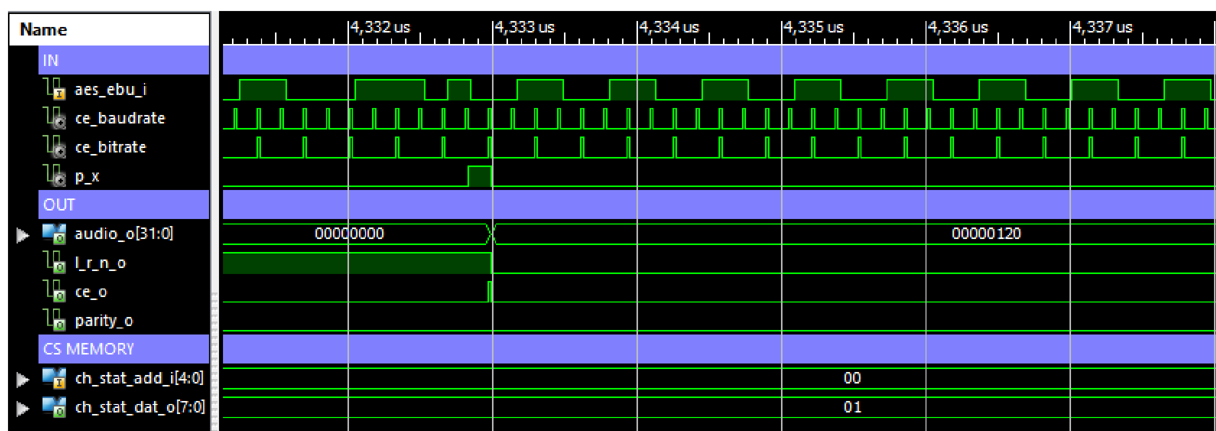

Fig. 27. Simulación del decodificador AES

En serie con el decodificador se encuentra la memoria FIFO de 8 posiciones que sincroniza y evita errores de señal debidos a posibles desincronizaciones del reloj con la entrada del sistema. En la figura 28 se puede analizar el funcionamiento de ésta. Las posiciones de memoria pares corresponden a muestras del canal 1 y las impares a muestras del canal 2. Por

tanto, se guardan 4 muestras de audio de cada canal. Se introduce un pequeño retardo, de 4 veces el periodo de *subframe*, que es despreciable en el funcionamiento del sistema.

Fig. 28. Simulación de la memoria FIFO

El codificador funciona de manera contraria que el decodificador. En la figura 29 se muestran las señales de entrada y salida del módulo, señales auxiliares como los pulsos de violación del código bifase para la construcción de un preámbulo y la relación entre el dato en binario a enviar y su correspondiente codificación en bifase.

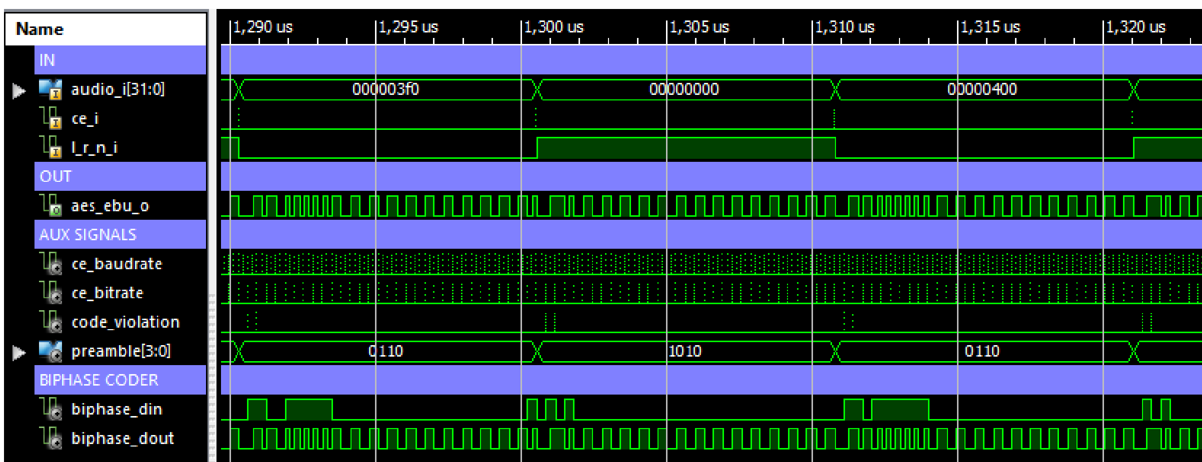

Fig. 29. Simulación del codificador AES

Después, se realiza una simulación del procesador de ganancia, con ganancias independientes para cada canal de 8 y 2. La señal de entrada sigue siendo la salida que ofrece el decodificador.

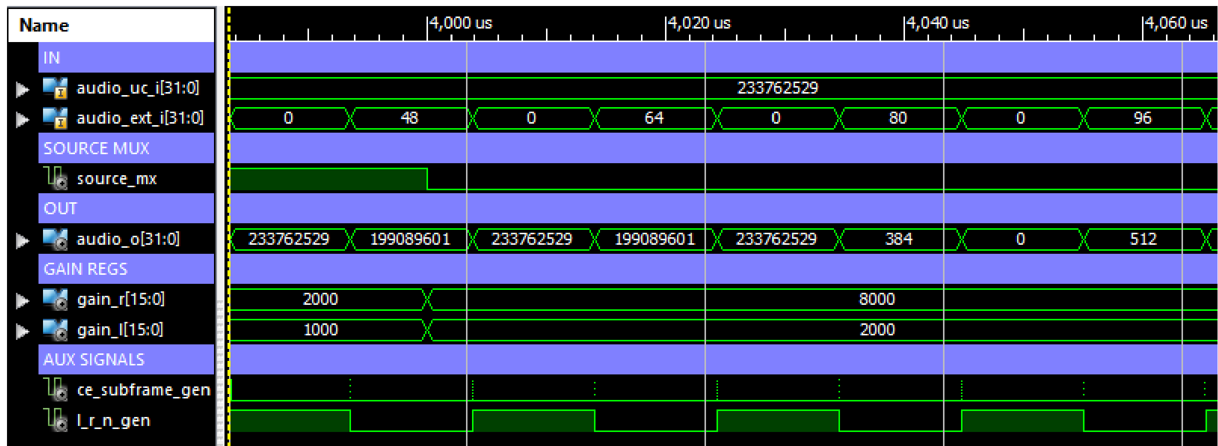

Fig. 30. Simulación del procesador de ganancia

En cuanto a la interfaz I2S, en la figura 31 se distinguen las líneas de entrada y salida de datos serie, con los registros de desplazamiento para capturar los datos al final de la imagen:

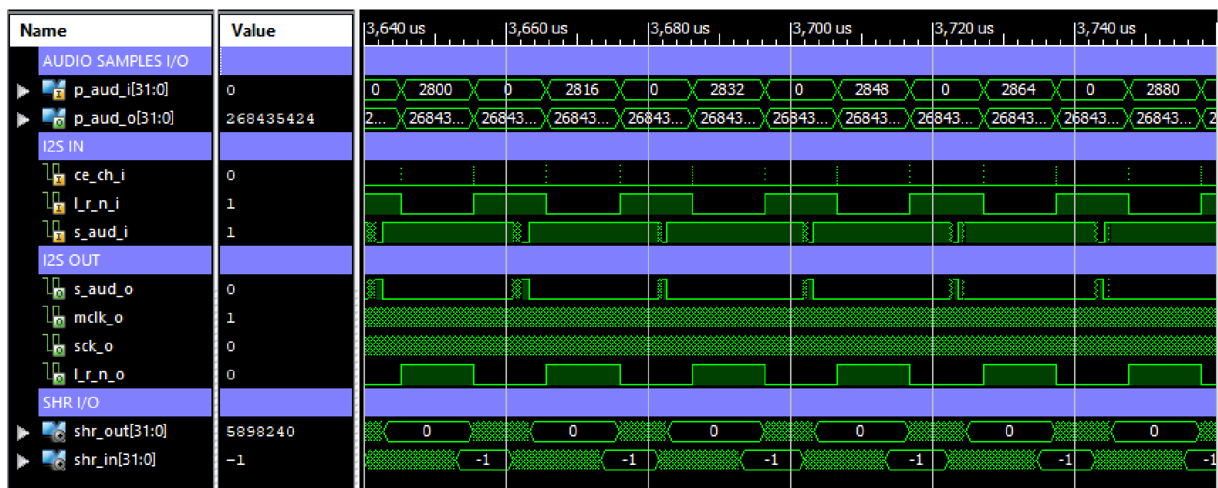

Fig. 31. Simulación de la interfaz I2S

De la misma manera, el funcionamiento de la interfaz SPI se muestra en la figura 32. En ella se pueden observar una transmisión completa, destacando el autómata diseñado para la transmisión y recepción de bloques de datos y los estados por los que pasa:

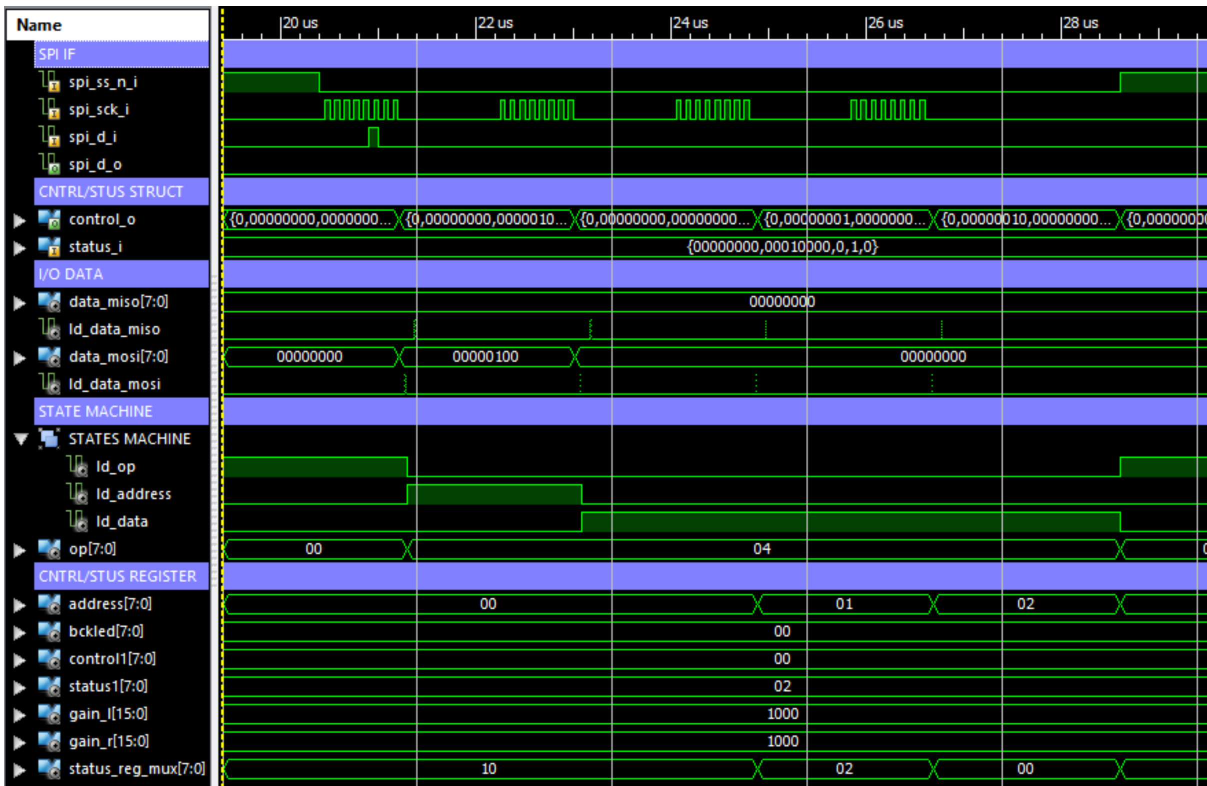

Fig. 32. Simulación de la interfaz SPI

Finalmente, la figura 33 muestra la respuesta del detector de fase de *Hogge* a una señal con *jitter*:

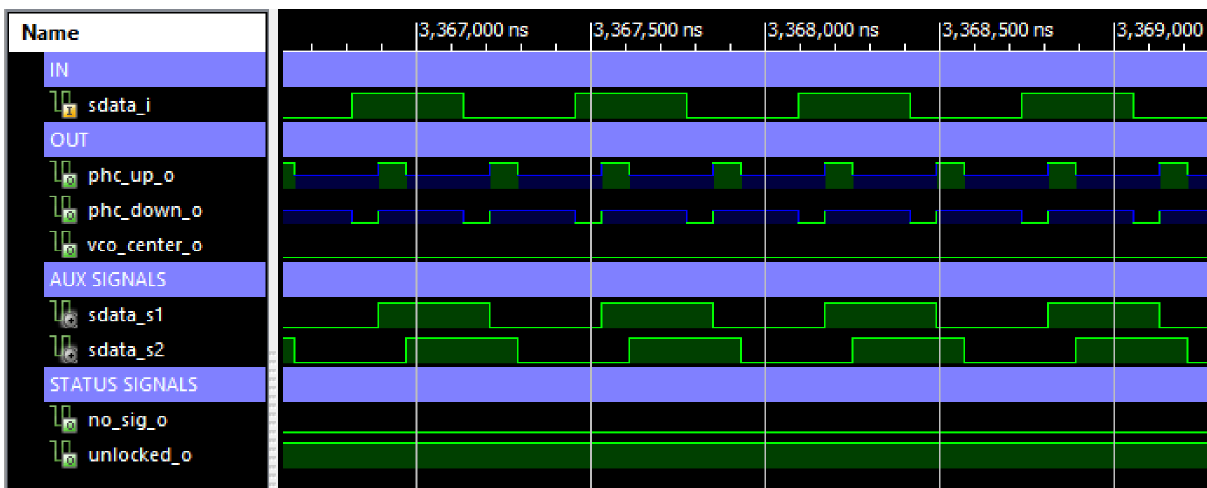

Fig. 33. Simulación del detector de fase de tipo Hogge

También es importante tener en cuenta la frecuencia máxima de trabajo del diseño y los recursos utilizados de la FPGA. En la figura 34 se observa el resumen de recursos utilizados que genera el entorno de desarrollo:

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	519	4,896	10%	
Number of 4 input LUTs	471	4,896	9%	
Number of occupied Slices	440	2,448	17%	
Number of Slices containing only related logic	440	440	100%	
Number of Slices containing unrelated logic	0	440	0%	
Total Number of 4 input LUTs	529	4,896	10%	
Number used as logic	348			
Number used as a route-thru	58			
Number used for Dual Port RAMs	116			
Number used as Shift registers	7			
Number of bonded IOBs	24	66	36%	
IOB Flip Flops	11			
Number of BUFGMUXs	2	24	8%	
Number of MULT18X18SIOs	2	12	16%	
Average Fanout of Non-Clock Nets	3.03			

Performance Summary			
Final Timing Score:	0 (Setup: 0, Hold: 0, Component Switching Limit: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Fig. 34. Resumen de recursos del firmware

A su vez, en la figura 35 se visualiza el máximo retardo calculado producido por la lógica del sistema, mediante el cual se calcula la frecuencia máxima de trabajo:

```
Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Clock to Setup on destination clock clk_49m152_i
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk_49m152_i   |   14.630|         |         |         |
---------------+---------+---------+---------+---------+
```

Fig. 35. Máximo retardo del sistema

Por lo tanto, la frecuencia máxima de trabajo del sistema es de 68.352 MHz, la cual cumple con holgura el requisito del sistema de 49.152 MHz.

6.2 Hardware

Todo el hardware ha sido diseñado por la empresa, a excepción del PLL. Por tanto, las pruebas de este apartado se centrarán en este módulo.

Los valores que se han elegido para los componentes son los que más se aproximan a los teóricos dentro de los valores normalizados para las tolerancias de los mismos. La diferencia entre los valores teóricos y los reales no ha supuesto un problema porque las especificaciones del PLL son muy relajadas. Para comprobar el correcto funcionamiento del PLL, se ha forzado en el sistema una pérdida de sincronismo, suprimiendo la fuente de señal, para volver a conectarla y así observar con el osciloscopio el proceso de enganche.

La figura 36 muestra la salida de la etapa *loop filter* del PLL, directamente conectada al VCO. Como se esperaba, al introducir de nuevo la fuente de señal, el PLL trata de engancharse a la frecuencia de la señal de entrada, realizando correcciones cada vez más pequeñas ofreciendo la respuesta esperada para el factor de amortiguamiento que se ha utilizado en los cálculos.

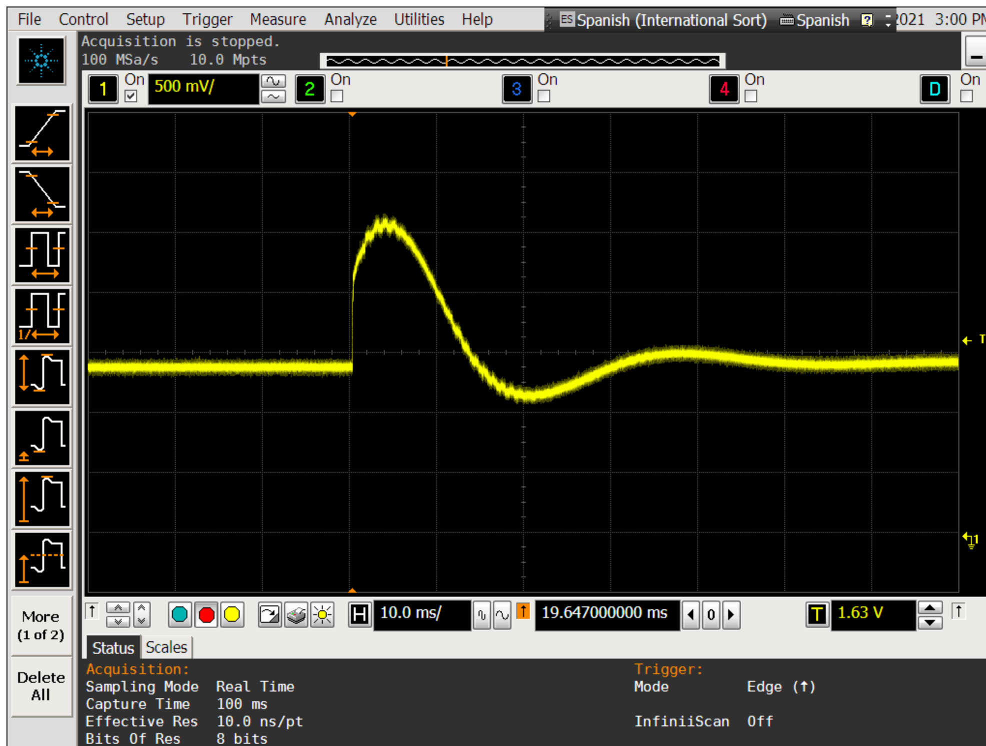

Fig. 36. Proceso de enganche de un PLL de segundo orden

Para terminar con la verificación del PLL, se ha generado en la entrada una señal con *jitter* a distintas frecuencias. Para observar el resultado, se muestran en el osciloscopio los diagramas de ojo de la entrada (traza superior) y la salida de la tarjeta (traza inferior), disparándose con la señal de salida.

En la figura 37 se observa la respuesta a una señal con *jitter* de 0.1 UI_p (Unidades de intervalo) a 2 Hz. A esta frecuencia tan baja la ganancia de la función de transferencia del PLL es de 1 y por lo tanto todo el *jitter* de la señal de entrada se propaga a la salida. Dado que el osciloscopio se está disparando con la señal de salida no se observa *jitter* en ninguna de las dos trazas.

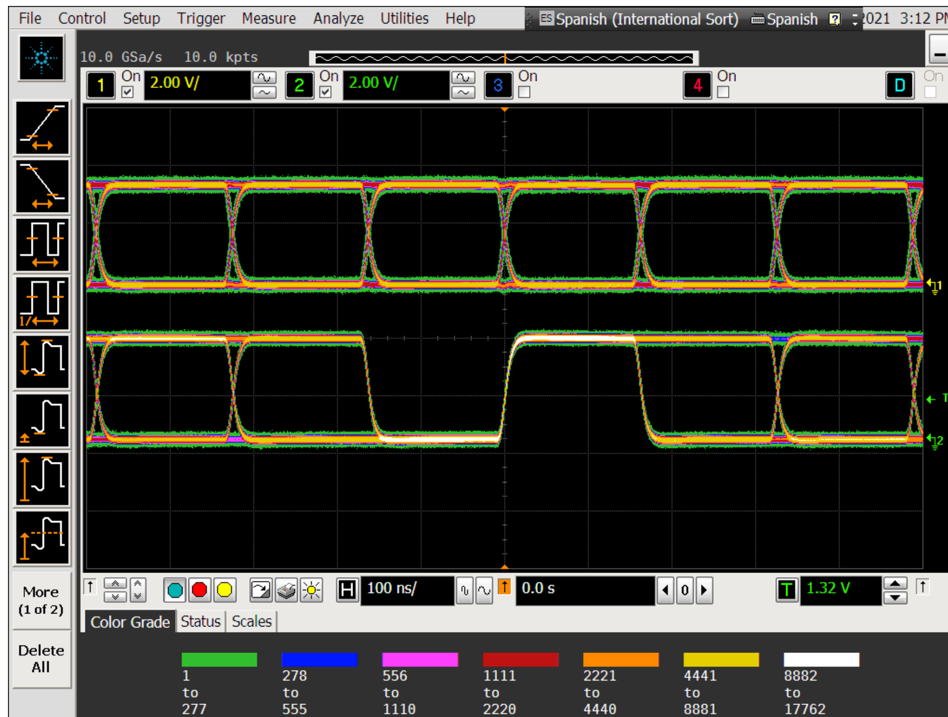

Fig. 37. Respuesta del PLL a una señal con jitter de 2 Hz

Sin embargo, si se aumenta de frecuencia de *jitter*, a 20 Hz, se empiezan a observar los efectos producidos por el filtro paso bajo del PLL. En la figura 38 se puede apreciar como al reducirse ligeramente el *jitter* en la señal de salida se empieza a notar el *jitter* con el que se recibe la señal de entrada.

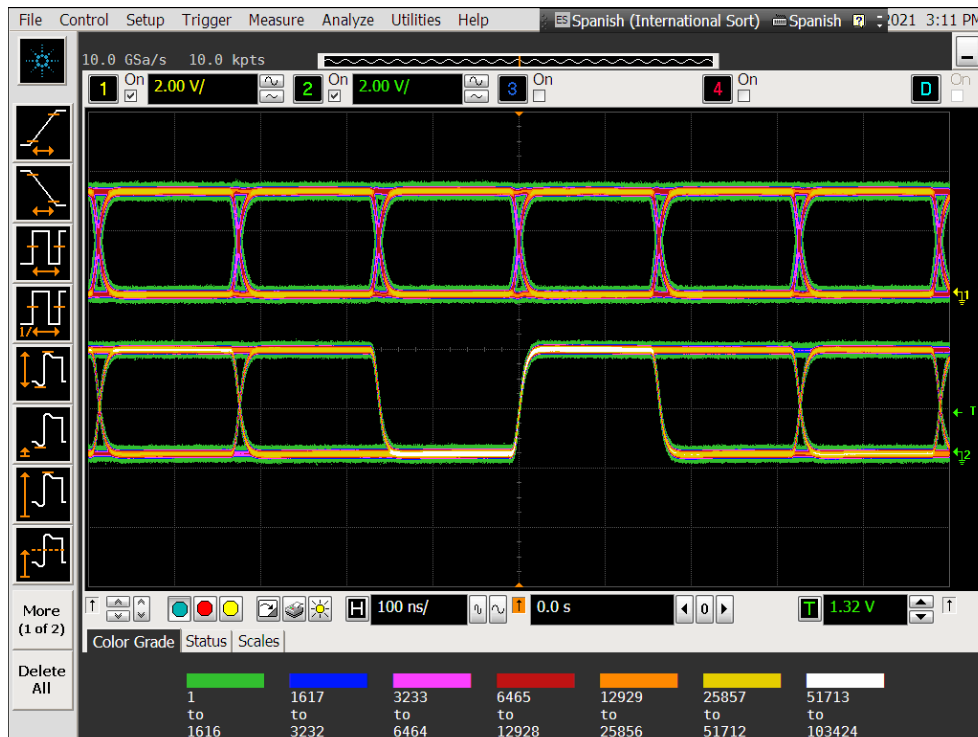

Fig. 38. Respuesta del PLL a una señal con jitter de 20 Hz

Finalmente, para comprobar la respuesta del filtro a frecuencias altas, se modifica la frecuencia de *jitter* a 1 kHz. En este caso el PLL filtra casi por completo el *jitter* con el que se recibe la señal de entrada que ahora ya se puede apreciar que es de 0.1 UI_p.

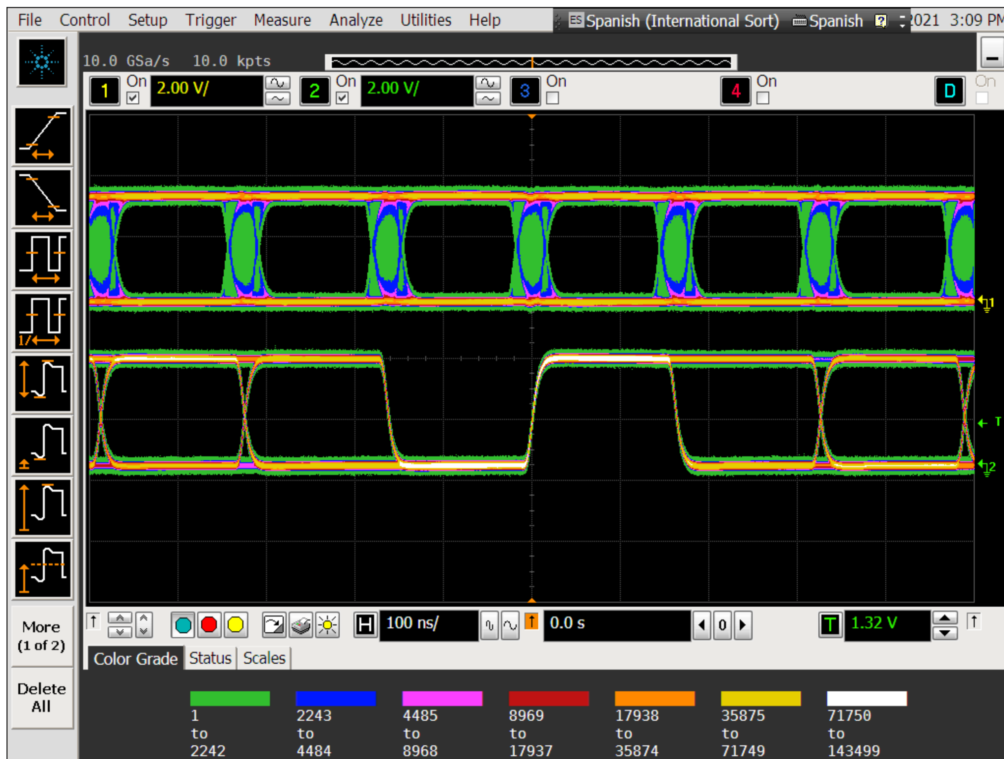

Fig. 39. Respuesta del PLL a una señal con jitter de 1 kHz

6.3 Software

Para la depuración del código, se incluye en el algoritmo del normalizador el envío de toda la información de cálculo relevante a través del puerto serie. De esta manera, los cálculos instantáneos quedan registrados y pueden guardarse para su análisis a través de Matlab.

Las pruebas comienzan con la validación del algoritmo de cálculo de sonoridad. Para ello, se ha implementado una función que genera una sinusoide de frecuencia y amplitud configurables. Esta sinusoide digital se introduce simulando los buffers de recepción por DMA y se analizan los distintos resultados obtenidos. Para todas las pruebas se ha escogido un periodo de cálculo de la sonoridad integral de 30 segundos.

En el primer escenario planteado se busca comparar la respuesta del algoritmo a un cambio en la entrada de una sinusoide de -23 dBFS de amplitud a otra de -13 dBFS. Tras ejecutar el script del anexo 10.4 con los valores capturados a través del puerto serie, se obtiene el siguiente resultado:

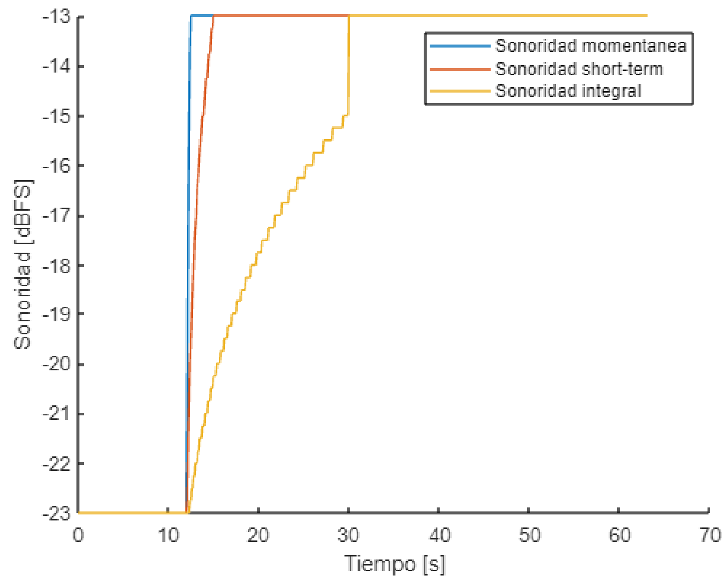

Fig. 40. Respuesta transitoria del algoritmo de cálculo de sonoridad a un cambio de amplitud a la entrada desde -23 dBFS hasta -13 dBFS

En segundo lugar, se realiza el experimento contrario, en el que se parte de una señal digital de -13 dBFS y se cambia su amplitud a -23 dBFS. Al igual que en el caso anterior, se presentan los resultados obtenidos:

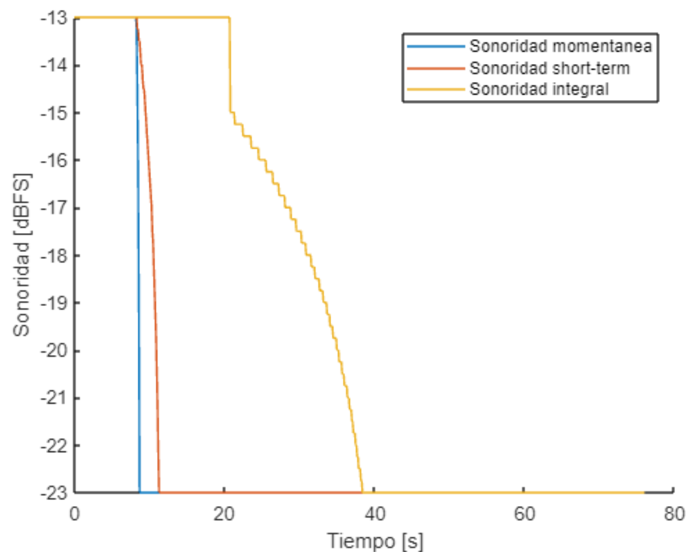

Fig. 41. Respuesta transitoria del algoritmo de cálculo de sonoridad a un cambio de amplitud a la entrada desde -13 dBFS hasta -23 dBFS

A continuación se verifica el algoritmo de control de ganancia. Para ello, de la misma manera que en el resto de las pruebas de software, se recrean una serie de escenarios y analizan los resultados.

En la figura 42 se distingue la variación de ganancia y su efecto en la señal de salida en función a la variación de sonoridad cuando se configura el normalizador para una sonoridad destino de -23 dBFS y una ganancia máxima de 12 dB. Como se esperaba, la ganancia varía de forma lineal aproximadamente, hasta que el valor de la sonoridad integral se hace

constante. A partir de ese momento se observa una respuesta exponencial negativa característica del filtro de primer orden diseñado para la aplicación de ganancia.

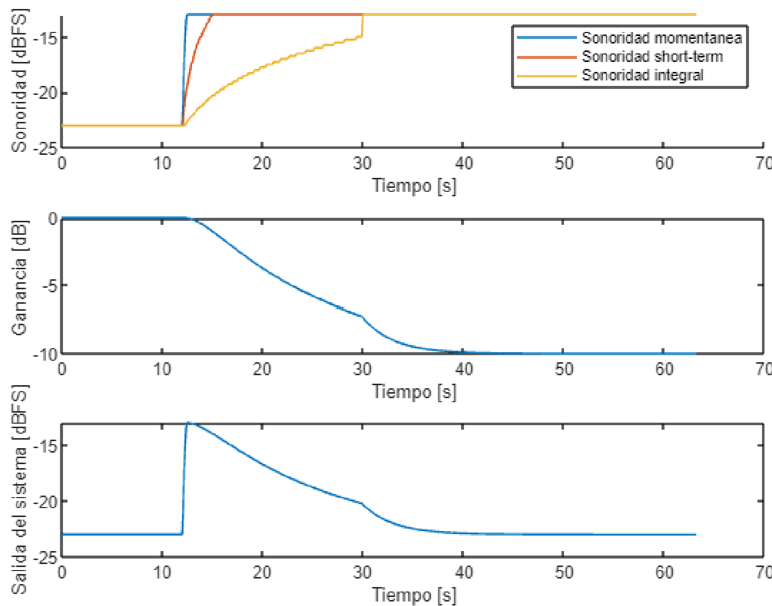

Fig. 42. Respuesta transitoria del algoritmo de cálculo de ganancia a un cambio de amplitud a la entrada desde -23 dBFS hasta -13 dBFS

De la misma manera, se repite el segundo escenario en el que la sonoridad desciende bruscamente. En este segundo supuesto la sonoridad deseada es de -13 dBFS y la ganancia máxima se mantiene en 12 dB. Se puede observar cómo, al bajar la sonoridad por debajo de la deseada, el algoritmo aplica ganancia para lograr esa sonoridad objetivo de nuevo.

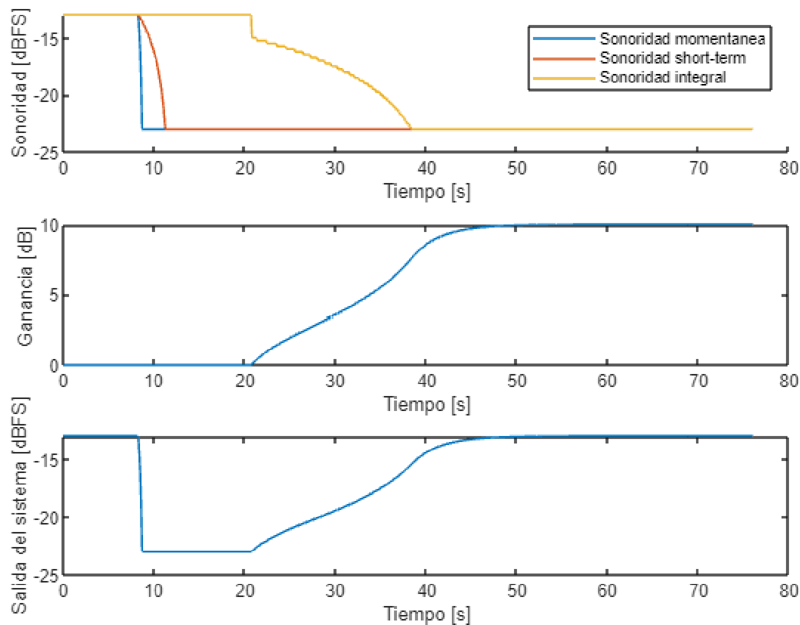

Fig. 43. Respuesta transitoria del algoritmo de cálculo de ganancia a un cambio de amplitud a la entrada desde -13 dBFS hasta -23 dBFS

Finalmente, se realiza una última prueba, en la que se verifica el funcionamiento del algoritmo de cálculo de ganancia cuando la ganancia necesaria es mayor de la que el sistema es capaz de dar.

En la figura 44 se puede ver el comportamiento del normalizador ante un desbordamiento de ganancia, en el que el nivel de sonoridad pasa de -13 dBFS a -23 dBFS, se fija una sonoridad objetivo de -13 dBFS y una ganancia máxima de 8 dB. El algoritmo actúa de manera correcta, ya que no debe de realizar ningún cambio de ganancia.

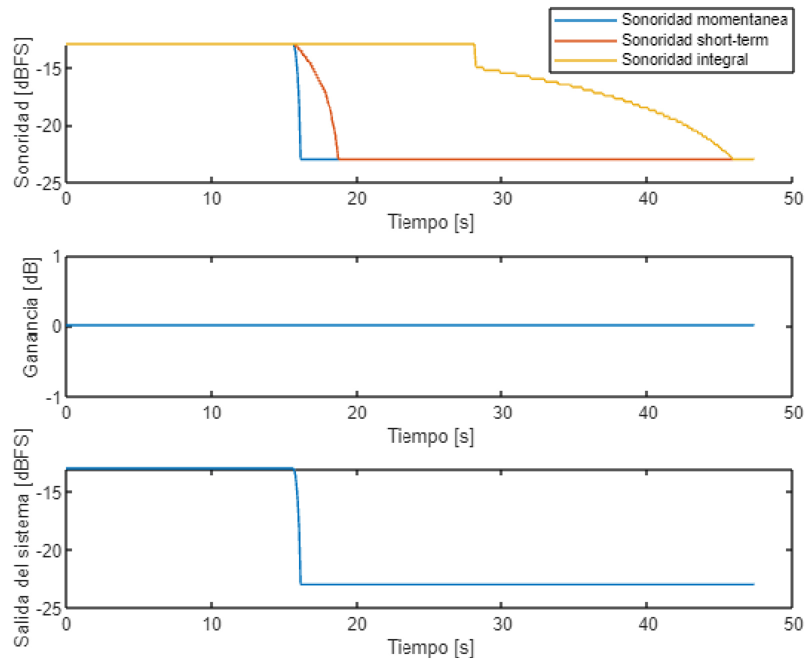

Fig. 44. Respuesta transitoria del algoritmo de cálculo de ganancia cuando satura la ganancia

6.4 Sistema completo

A la hora de validar el sistema en conjunto, se han capturado los datos de varios escenarios distintos y se han graficado en Matlab. A la vez, se han conectado a la entrada y salida del sistema dos DAC (Conversores Digital-Analógico) para demostrar el funcionamiento del diseño.

En primer lugar, se configura el normalizador con los siguientes parámetros:

Tabla 5. Parámetros de configuración para las pruebas del sistema completo

Sonoridad objetivo	Máxima ganancia
-23 LUFS	10 dB

Se configura el generador de señal para que ofrezca una señal de silencio, se espera un pequeño periodo de tiempo y se modifica la entrada por una señal de -23 dBFS. Según lo esperado, el algoritmo no debería de compensar el nivel inicial, puesto que la ganancia que debe aplicar al sistema es infinita y por tanto mayor que la ganancia máxima definida. Tras

realizar el cambio de nivel, el algoritmo tampoco debería de actuar, a excepción del transitorio de sonoridad integral entre los dos niveles, en el que se aplicará una pequeña ganancia que luego se corregirá y volverá a 0.

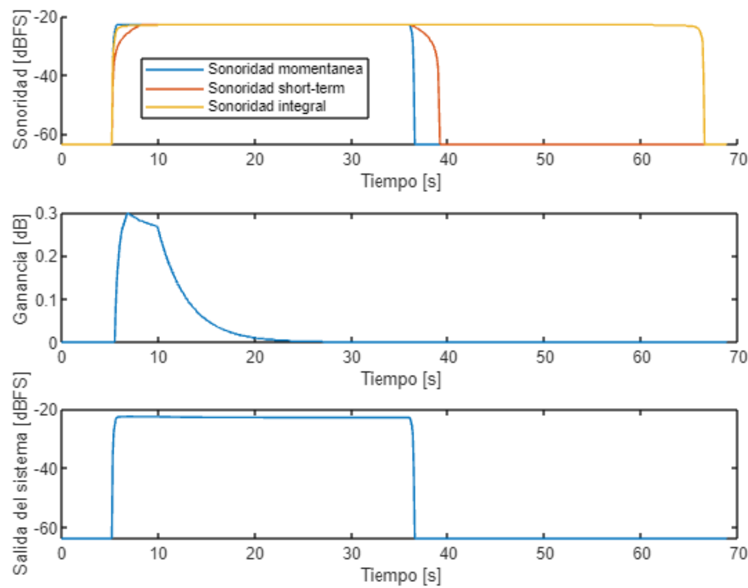

Fig. 45. Respuesta transitoria del sistema a un cambio de amplitud a la entrada desde mute hasta -23 dBFS

La siguiente prueba mantiene los parámetros iniciales, pero se realizan una serie de cambios de nivel a la entrada para contrastar los tiempos de corrección de ganancia. De esta manera, partiendo de una señal de silencio, se pasa a otra de -23 dBFS. A continuación, se modifica la amplitud de la señal a -27 dBFS, -18 dBFS, de nuevo -23 dBFS y vuelta a silencio. En la siguiente imagen podemos observar el resultado:

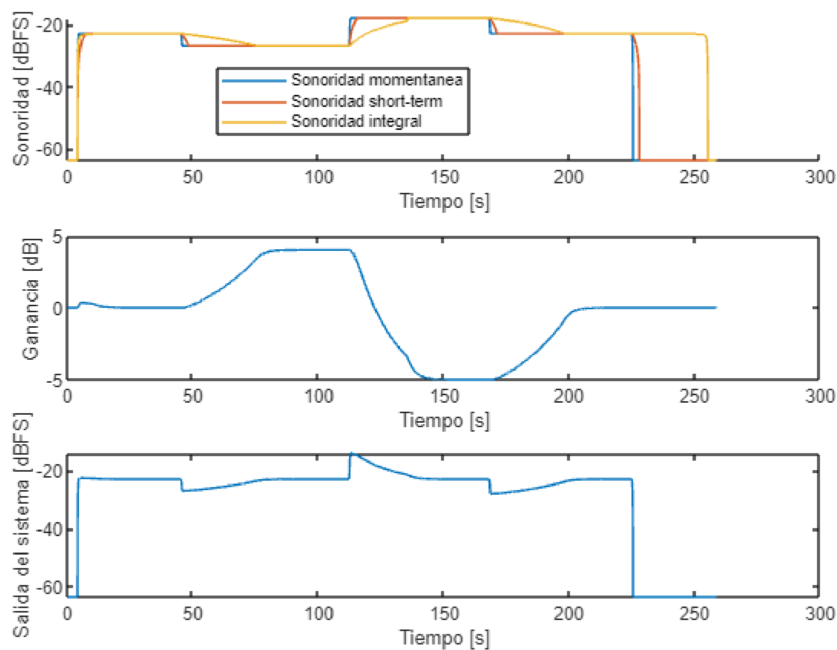

Fig. 46. Respuesta transitoria del sistema a distintos cambios de amplitud en la entrada

Para finalizar, se ha introducido un DAC a la entrada y la salida del normalizador. Modificando la amplitud a la entrada y monitorizando la salida del DAC se puede observar claramente el efecto producido por el normalizador.

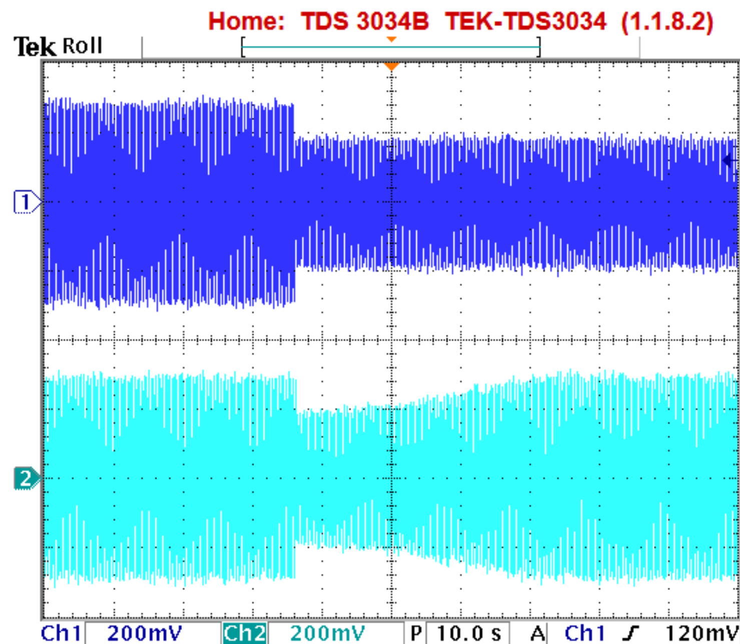

Fig. 47. Señal analógica reconstruida de un proceso de nivelación de sonoridad positiva

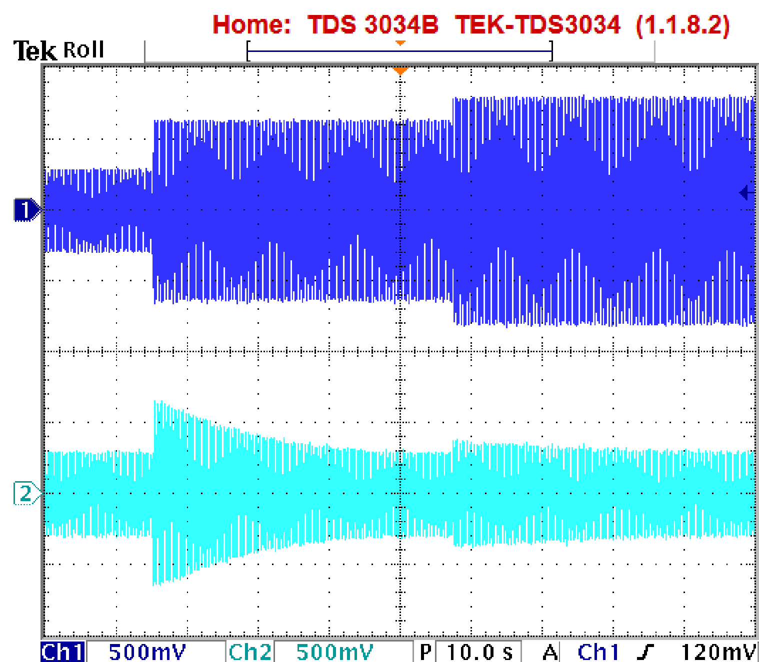

Fig. 48. Señal analógica reconstruida de un proceso de nivelación de sonoridad negativa

7 Presupuesto

El proyecto se ha enfocado como una subcontratación de ingeniero para realizar una tarea de diseño electrónico concreta y aislada en la empresa. Para ello, hay que tener en cuenta el precio por hora de ingeniero y el valor material de las herramientas utilizadas.

La tabla 6 incluye el coste material de los recursos necesarios para el proyecto, siendo el valor total el coste de amortización, poniendo como periodo de amortización 4 años:

Tabla 6. Costes materiales

Costes materiales			
Amortizados			
Herramienta	Precio	Periodo de uso	Coste de amortización
Osciloscopio	3000 €	4 meses	250 €
Generador de señal digital	2000 €		167 €
Depurador procesador	200 €		17 €
Depurador FPGA	200 €		17 €
Interfaz puerto serie	20 €		2 €
Computadora	500 €		42 €
Licencia académica MATLAB	500 €		42 €
Fungibles			
Cables	50 €	4 meses	50 €
Material de oficina	40 €		40 €
Electricidad	50 €		30 €
		Total:	655 €

Por otro lado, en la tabla 7 se detallan todas las tareas y horas dedicadas a precio de ingeniero junior:

Tabla 7. Costes de Ingeniero

Coste de Ingeniero				
Puesto	Tarea		Horas	Coste
Ingeniero Junior (15€/h)	Investigación	Estudio de normas y recomendaciones	20	300 €
		Documentación	20	300 €
	Diseño	Firmware	120	1800 €
		Hardware	20	300 €
		Software	100	1500 €
	Pruebas	Firmware	15	225 €
		Hardware	5	75 €
		Software	10	150 €
	Redacción	Realización de la memoria	60	900 €
			Total:	5550 €

Finalmente, el presupuesto total del proyecto es de **6.205 €**.

8 Conclusiones

Gracias a los conocimientos adquiridos a lo largo del grado, se ha podido hacer frente a un proyecto de este tipo. Con su realización, ya no solo se han afianzado los conceptos aprendidos, sino también se ha demostrado la capacidad de aprendizaje y las habilidades para solucionar problemas, características de un buen ingeniero.

La experiencia de poder realizar el proyecto de fin de grado en una empresa aumenta la velocidad de aprendizaje, ya que, aunque el proyecto se haya realizado de forma individual, el hecho de tener gente a la que poder preguntar cuando surgen inconvenientes ayuda mucho a la hora de depurar fallos, optimizar algoritmos y mejorar técnicas de diseño.

En cuanto al objetivo principal del proyecto, se ha intentado realizar de la manera más óptima posible, tratando de implementar un código legible y fácil de entender, favoreciendo así la evolución de futuras versiones del producto. El uso de herramientas informáticas para el diseño facilita en gran medida las tareas que ha habido que realizar, herramientas tales como entornos de desarrollo o aplicaciones de cálculo matemático para depuración y simulaciones.

Como en todo diseño electrónico, según se avanza con las tareas a realizar surgen nuevos imprevistos que pueden provocar un rediseño parcial o completo del proyecto. En este caso particular esto también ha ocurrido, siendo necesarias varias versiones del software y de las estructuras de comunicación con la FPGA.

Con este proyecto se ha demostrado la alta capacidad de integración de las distintas tecnologías de diseño electrónico punteras en la actualidad. El hecho de poder escoger distintos subsistemas en función al problema que hay que solventar y poder comunicarse entre ellos facilita y reduce el coste de diseño y del producto.

Para concluir esta memoria y como punto de vista personal, se ha obtenido una visión general de lo que supone la realización de un proyecto para solucionar un problema de ingeniería. Esto consiste en la división del problema principal en problemas cada vez más pequeños, de tal manera que sean de fácil comprensión y solución.

“Un problema no es fácil ni difícil, lo difícil es descomponerlo de la manera correcta para que todo resulte fácil”

9 Bibliografía

[1] Audio Engineering Society, Inc. *“AES3-2003”.*

[2] Albalá Ing, *“HAM2000”.*

[3] ITU-R 1, *“Algorithms to measure audio programme loudness and true-peak audio level”.*

[4] EBU, *“Loudness Metering: ‘EBU Mode’ metering so supplement loudness normalisation in accordance with EBU R128”.*

[5] RAZAVI, Behzad; *“Challenges in the Designo f High-Speed Clock and Data Recovery Circuits”.*

[6] GARDNER, Floyd M.; *“Phaselock Techniques Thrid Edition”.*

[7] “CMSIS” https://developer.arm.com/tools-and-software/embedded/cmsis (accesed May 30, 2021).

[8] “Spartan E3 dathasheet” https://www.xilinx.com/support/documentation/data_sheets/ds312.pdf (accesed May 30, 2021).

[9] “SAM E53 datasheet” https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D5x_E5x_Family_Data_Sheet_DS60001507G.pdf (accesed May 30, 2021).

Bibliografía

10 Anexos

10.1 Código VHDL

Debido a la extensión de los ficheros HDL, estos han quedado guardados en el CD del PFG. El directorio /FIRMWARE dispone de los siguientes ficheros:

Tabla 8. Estructura de ficheros del anexo digital del código VHDL

<table>
<tr><td rowspan="10">/FIRMWARE</td><td rowspan="3">/AES_EBU_IF</td><td>aes_ebu_if_in.vhd</td></tr>
<tr><td>audio_fifo.vhd</td></tr>
<tr><td>aes_ebu_if_out.vhd</td></tr>
<tr><td>/AUDIO_PROCESS</td><td>audio_process.vhd</td></tr>
<tr><td>/I2S</td><td>uc_i2s_if.vhd</td></tr>
<tr><td>/PLL</td><td>phc.vhd</td></tr>
<tr><td rowspan="3">/SPI</td><td>spi_shr.vhd</td></tr>
<tr><td>uc_pkg.vhd</td></tr>
<tr><td>uc_spi_if.vhd</td></tr>
<tr><td>ham2000_fpga.vhd</td><td></td></tr>
</table>

10.2 Algoritmos Software

10.2.1 Función principal

Todo el diseño del software se ha incluido en la función principal *audio_process()*. Esta función se ejecuta cada milisegundo y en su interior dispone de contadores periódicos que marcan el time slot en el que se encuentra el código y por tanto la rutina de código que debe de ejecutar. Debido a la envergadura de la función, se presenta primero la estructura general para pasar luego a cada una de las secciones de código:

```
/*
 * Name:    audio_process
 * Description: Implements the loudness, peak and gain algorithm.
 * This function is ejecuted every 1 ms, with variant
 * period because of the variant parameters involved.
 */
void audio_process(void){
      //Every 1ms
      /*Load audio simples*/
      ...(Rutina 1)
      /*2nd order Fiter*/
      ...(Rutina 2)
      /*Power of 1 ms samples (48 for each channel)*/
      ...(Rutina 3)
      /*Additive buffer of 100 samples*/
      ...(Rutina 4)
      //Every 100ms
      switch (tic100ms){
            case MOMENTARY_SHORT_TERM_SLOT:
            /*MOMENTARY AND SHORT-TERM LOUDNESS*/
            /*Peak calc*/
            /*Overload and Silence Calc*/
            ...(Rutina 5)
            break;
            case ABS_GAT_INTEGRAL_SLOT:
            /*LOUDNESS WITH ABSOLUTE THRESHOLD*/
            ...(Rutina 6)
            break;
            case REL_GAT_INTEGRAL_SLOT:
            /*LOUDNESS WITH RELATIVE THRESHOLD*/
            /*INTEGRAL LOUDNESS*/
            ...(Rutina 7)
            break;
            case GAIN_CALC_SLOT:
            /*GAIN CALC*/
            /*FPGA_DATA_REFRESH*/
            /*LED STATUS REFRESH*/
            ...(Rutina 8)
            break;
            case SERIAL_DEBUG:
            /*SERIAL COMMUNICATIONS*/
            ...(Rutina 9)
            break;
            case TIC100MS:
            /*INCREMENT COUNTERS*/
            ...(Rutina 10)
            break;
            default:
            break;
      }
      tic100ms++;
}
```

Función 1. Función principal del algoritmo

```
/*Load audio samples from internal audio generator or DMA and save the
peak value*/
if (control.param.aud_uc_ext_n==0) {
	j=0;
	for(i = 0; i<AUDIO_FILTER_SAMPLES; i++){
		sourceL[i] = rx_buf[j++];
		max_value_L(sourceL[i]);
		sourceR[i] = rx_buf[j++];
		max_value_R(sourceR[i]);
	}
	} else {
	j=0;
	for(i = 0; i<AUDIO_FILTER_SAMPLES; i++){
		sourceL[i] = gen_buf[j++];
		max_value_L(sourceL[i]);
		sourceR[i] = gen_buf[j++];
		max_value_R(sourceR[i]);
	}
	for(i = 0; i<2*AUDIO_FILTER_SAMPLES; i++){
		tx_buf[i] = gen_buf[i];
	}
}
```

Rutina 1. Recepción y transmisión por DMA según la fuente de señal escogida

```
/*2nd order Fiter*/
arm_biquad_cascade_df1_q31(&kfilterR, sourceR, destR,
AUDIO_FILTER_SAMPLES);
arm_biquad_cascade_df1_q31(&kfilterL, sourceL, destL,
AUDIO_FILTER_SAMPLES);
```

Rutina 2. Filtros de segundo orden de DSP CMSIS

```
/*Power of 1 ms samples (48 for each channel)*/
arm_power_q31 (destL, SAMPLES_1MS/2, &pow1ms_l);
arm_power_q31 (destR, SAMPLES_1MS/2, &pow1ms_r);
```

Rutina 3. Funciones de cálculo de potencia de DSP CMSIS

```
/*Additive buffer of 100 samples*/
acc_pow_100ms+=pow1ms_l;
acc_pow_100ms+=pow1ms_r;
```

Rutina 4. Acumuladores de potencia de 100 ms

```
/*MOMENTARY AND SHORT-TERM LOUDNESS*/
/*Peak calc*/
peak_value();

/*Overload and Silence Calc*/
ovld_sil_calc();

/*Mean of acc_pow_100ms*/
acc_pow_100ms /= (SAMPLES_1MS*TIC100MS*FIXED_POINT_ADJ_16_48_TO_1_31);
//96 samples * 100 ms
buff_pow_100ms[count_short_term]= (uint32_t) acc_pow_100ms;
acc_pow_100ms = 0;

/*Mean of MOMENTARY_PERIOD*/
tmp_power = 0;
for(i=0;i<MOMENTARY_PERIOD;i++) {
      if ((count_short_term-i)<0)
tmp_power+=buff_pow_100ms[count_short_term-i+SHORT_TERM_BUFFER_SIZE];
      else tmp_power+=buff_pow_100ms[count_short_term-i];
}
tmp_power /= MOMENTARY_PERIOD;
power_momentary = (uint32_t) tmp_power;
buff_pow_400ms[count_integral]=power_momentary;

/*Mean of SHORT_TERM_PERIOD*/
tmp_power = 0;
for(i=0;i<SHORT_TERM_PERIOD;i++) {
      tmp_power+=buff_pow_100ms[i];
}
tmp_power /= SHORT_TERM_PERIOD;
power_short_term=(uint32_t) tmp_power;

/*Momentary and Short term loudness calc in uint8_t format*/
loud_400ms = loudness_calc(power_momentary);
status.param.m_loudness = loud_400ms;

loud_3s = loudness_calc(power_short_term);
status.param.s_loudness = loud_3s;
```

Rutina 5. Cálculo de sonoridad momentánea y sonoridad a corto plazo

```
/*LOUDNESS WITH ABSOLUTE THRESHOLD*/
/*Load param of the loudness calc*/
integral_period = (control.param.int_period)*300;
relative_th_db = control.param.rel_gating;

/*Refresh and calculate valid samples*/
valid_samples = 0;
i=0;
tmp_power = 0;
if(count_integral >= (integral_period-1)){
	for(i = count_integral; i > (count_integral - integral_period);
	i--){
		if(buff_pow_400ms[i] > THRESHOLD_70DB){
			tmp_power+=buff_pow_400ms[i];
			valid_samples++;
		}
	}
}else{
	for(i = count_integral; i > -1; i--){
		if(buff_pow_400ms[i] > THRESHOLD_70DB){
			tmp_power+=buff_pow_400ms[i];
			valid_samples++;
		}
	}
	for(i = (INTEGRAL_BUFFER_SIZE-1); i > (INTEGRAL_BUFFER_SIZE -
	(integral_period - count_integral)); i--){
		if(buff_pow_400ms[i] > THRESHOLD_70DB){
			tmp_power+=buff_pow_400ms[i];
			valid_samples++;
		}
	}
}
if (valid_samples==0)tmp_power = 0;
else tmp_power /= valid_samples;
power_integral=(uint32_t) tmp_power;

/*Temporal loudness (only for debug)*/
tmp_loud_int=loudness_calc(power_integral);

/*Relative threshold calc*/
relative_threshold = relative_threshold_calc(relative_th_db,
power_integral);
```

Rutina 6. Cálculo de sonoridad integral con gatting absoluto

```
/*LOUDNESS WITH RELATIVE THRESHOLD*/
/*Refresh and calculate valid samples*/
tmp_power = 0;
valid_samples = 0;
if(count_integral >= (integral_period-1)){
	for(i = count_integral; i > (count_integral - integral_period);
	i--){
		if(buff_pow_400ms[i] > relative_threshold){
			tmp_power+=buff_pow_400ms[i];
			valid_samples++;
		}
	}
}else{
	for(i = count_integral; i > -1; i--){
		if(buff_pow_400ms[i] > relative_threshold){
			tmp_power+=buff_pow_400ms[i];
			valid_samples++;
		}
	}
	for(i = (INTEGRAL_BUFFER_SIZE-1); i > (INTEGRAL_BUFFER_SIZE -
	(integral_period - count_integral)); i--){
		if(buff_pow_400ms[i] > relative_threshold){
			tmp_power+=buff_pow_400ms[i];
			valid_samples++;
		}
	}
}
if (valid_samples==0)tmp_power = 0;
else tmp_power /= valid_samples;
power_integral=(uint32_t) tmp_power;

/*Integral loudness calc*/
loud_int=loudness_calc(power_integral);
status.param.i_loudness = loud_int;
```

Rutina 7. Cálculo de sonoridad integral con gatting relativo

```
/*GAIN CALC*/
gain_calc();
/*FPGA_DATA_REFRESH*/
write_fpga_control();
/*LED STATUS REFRESH*/
led_status_refresh();
```

Rutina 8. Cálculo de ganancia y actualización de parámetros

```
/*Only for debug*/
sprintf(console_msg,"%d %d %d %d;\r\n",loud_400ms,loud_3s,loud_int,
	(uint16_t)swap16((fpga_control.param.gain_r_be)));
USART_DEBUG_msg(console_msg);
```

Rutina 9. Comunicación serial de datos para depuración

```
/*Increment counters*/
count_short_term++;
if (count_short_term>=SHORT_TERM_BUFFER_SIZE) count_short_term=0;
count_integral++;
if (count_integral>=INTEGRAL_BUFFER_SIZE) count_integral=0;
tic100ms=0;
```

Rutina 10. Iteración de los contadores de la función

10.2.2 Funciones auxiliares

Una buena práctica de programación es dividir las tareas principales en funciones. De esta manera el código queda más legible y es más fácil depurar errores. A continuación se presentan las funciones auxiliares desarrolladas:

```
int32_t gen_buf[2*AUD_BUF_LEN];
/*
 * Name:          buffer_init
 * Description:   Create audio sine samples with amplitude and
 *                frequency variables.
*/
void buffer_init() {
      uint32_t i;
      double amplitude_ch1, amplitude_ch2;
      uint32_t frequency_ch1,frequency_ch2;
      if (control.param.level_ch1==0) amplitude_ch1=0;
      else amplitude_ch1 = (pow(2,30)-1)*pow(10,(((float)
            control.param.level_ch1)-255)/80);
      if (control.param.level_ch2==0) amplitude_ch2=0;
      else amplitude_ch2 = (pow(2,30)-1)*pow(10,(((float)
            control.param.level_ch2)-255)/80);
      frequency_ch1 = control.param.freq_ch1;
      frequency_ch2 = control.param.freq_ch2;
      for (i=0;i<AUD_BUF_LEN;i++) {
            gen_buf[2*i] = amplitude_ch1*
                  sin((2*3.14159265359*i*frequency_ch1)/AUD_BUF_LEN);
            gen_buf[2*i+1] = amplitude_ch2*
                  sin((2*3.14159265359*i*frequency_ch2)/AUD_BUF_LEN);
      }
}
```

Función Auxiliar 4. Generador de sinusoides digitales

```
/* Name:                        gain_calc
 * Description:                 Implements gain control algorithm. */
void gain_calc(void){
loud_target = TWOPOW8 - (control.param.target_loudness)*4;
max_gain = control.param.max_dB_adj*4;
if (max_gain == 0){
	if (control.param.independent_gain == 1){
		float gain_l = control.param.gain_ch2;
		float gain_r = control.param.gain_ch1;
		fpga_control.param.gain_r_be = swap16((uint16_t)
			round(powf(10,(gain_r/80))*powf(2,12)));
		fpga_control.param.gain_l_be = swap16((uint16_t)
			round(powf(10,(gain_l/80))*powf(2,12)));
	}else{
		float gain_r = control.param.gain_ch1;
		fpga_control.param.gain_r_be=fpga_control.param.gain_l_be =
			swap16((uint16_t)
			round(powf(10,(gain_r/80))*powf(2,12)));}
}else{
	if(loud_target>loud_int){
		needed_gain = (float) (loud_target - loud_int);
		if(needed_gain>=max_gain){
			status.param.gain_limit = TRUE;
		}else{
			status.param.gain_limit = FALSE;
			if((loud_int>loud_target-
			max_gain)&&(loud_3s>loud_target-max_gain)){
				if(needed_gain>max_gain)needed_gain=max_gain;
				needed_gain = needed_gain/4;
				gain=(1-B0_CONTS)*gain;
				gain+= B0_CONTS*needed_gain;
				fpga_control.param.gain_r_be =
				fpga_control.param.gain_l_be= swap16((uint16_t)
					round(powf(10,(gain/20))*powf(2,12)));
				status.param.applied_gain = (int8_t)
					round(gain*4);}}
	}else{
		needed_gain = (float) (loud_int - loud_target);
		needed_gain = - needed_gain;
		if(needed_gain<=-max_gain){
			status.param.gain_limit = TRUE;
		}else{
			status.param.gain_limit = FALSE;
			if((loud_int<loud_target+max_gain)
			&&(loud_3s<loud_target+max_gain)){
				if(needed_gain<-max_gain)neded_gain=-max_gain;
				needed_gain = needed_gain/4;
				gain=(1-B0_CONTS)*gain;
				gain+= B0_CONTS*needed_gain;
				fpga_control.param.gain_r_be =
				fpga_control.param.gain_l_be= swap16((uint16_t)
					round(powf(10,(gain/20))*powf(2,12)));
				status.param.applied_gain = (int8_t)
					round(gain*4);}}}
	}
}
```

Función Auxiliar 5. Algoritmo de cálculo de ganancia

```
/*
 * Name:                       loudness_calc
 * Description:                Calculate loudness in Annette format
 *                             ((mute)0-255(0dBFS)) with the power
 * Return:                     loudness in 0-255 format
 */
uint8_t loudness_calc(uint32_t power){
      double dpower = (double) power;
      if (power==0) return(0);
      else {
            dpower /= POWER_0DBFS;
            dpower = -0.691 + 6.0206 + 10*log10(dpower);
            dpower = 255 + dpower*4;
            if (dpower>255) return(255);
            else if (dpower<0) return(0);
            else return (uint8_t) round(dpower);
      }
}
```

Función Auxiliar 6. Cálculo de sonoridad en función de la potencia

```
/*
 * Name:                       ovld_sil_calc
 * Description:                Calculate and asign silience and overload
 *                             signals to status structure
 */
void ovld_sil_calc(void){

      //Silence calc
      if ((status.param.level_ch2 <= control.param.silence_thr)
      &&(status.param.level_ch1 <= control.param.silence_thr)){
            if(status.param.silence == 0) count_silence+=100;
      }
      else count_silence = 0;

      if ((count_silence >= ((control.param.silence_time)*1000))
      &&(status.param.silence == 0))
            status.param.silence = 1;
      else if((count_silence == 0)&&(status.param.silence == 1))
            status.param.silence = 0;

      //Overload calc
      if ((status.param.level_ch2 >= control.param.overload_thr)
      ||(status.param.level_ch1 >= control.param.overload_thr))
            status.param.overload = 1;
      else status.param.overload = 0;
}
```

Función Auxiliar 7. Detector de silencio y sobrecarga

```
/*
 * Name:                    peak_db
 * Description:             Calculate peak level in Annette format
 *                          ((mute)0-255(0dBFS)) with the power
 * Return:                  Peak level in 0-255 format
 */
uint8_t peak_db(uint32_t power){
	double dpower = (double) power;
	if (power==0) return(0);
	else {
		dpower /= pow(2,30);
		dpower = 20*log10(dpower);
		dpower = (255 + dpower*4);
		if (dpower>255) return(255);
		else if (dpower<0) return(0);
		else return (uint8_t) round(dpower);
	}
}
/*
 * Name:                          max_value_R
 * Description:                   Save peak level of R channel
 */
void max_value_R(int32_t audio_sample){
	//sample = abs(audio_sample);
	if(audio_sample < 0) sample = -audio_sample;
	else sample= audio_sample;
	if(audio_max_r < sample) audio_max_r = sample;
}
/*
 * Name:                          max_value_L
 * Description:                   Save peak level of L channel
 */
void max_value_L(int32_t audio_sample){
	//sample = abs(audio_sample);
	if(audio_sample < 0) sample = -audio_sample;
	else sample= audio_sample;
	if(audio_max_l < sample) audio_max_l = sample;
}
/*
 * Name:                          peak_value
 * Description:                   Asign peak levels in dB to status
 *                                structure
 */
void peak_value(void){
	//float fpeak;
	peak_val_r -= peak_val_r/8;
	if(peak_val_r < audio_max_r) peak_val_r = audio_max_r;
	status.param.level_ch2 = peak_db(peak_val_r);

	peak_val_l -= peak_val_l/8;
	if(peak_val_l< audio_max_l) peak_val_l = audio_max_l;
	status.param.level_ch1 = peak_db(peak_val_l);

	audio_max_r = 0;
	audio_max_l = 0;
}
```

Función Auxiliar 8. Cálculo de nivel de pico

```
/*
 * Name:          write_fpga_control
 * Description:   Writes control registers in FPGA
 */
void write_fpga_control(void)
{
      fpga_control.param.aud_uc_ext_n=control.param.aud_uc_ext_n;
      SPI_FPGA_write_op(REGISTER_AREA,fpga_control.array,
            N_FPGA_CONTROL);
}
```

Función Auxiliar 9. Escritura de estructura de control de la FPGA

10.3 Depuración

Por defecto, el programa desarrollado en el microcontrolador envía a través del puerto serie los cálculos instantáneos de sonoridad y ganancia. Guardando todos estos datos e introduciéndolos en un script desarrollado en Matlab, se pueden analizar anomalías o comportamientos erróneos del normalizador. El código del script realizado se presenta a continuación:

```
clear all;
clc;
%%[M(Momentary Loudness) S(Short Term Loudness) I(Integral Loudness)
G(Gain)]

y=0:0.1:%x_size;

x=[
% M0 S0 I0 G0
% M1 S1 I1 G1
% M2 S2 I2 G2
%     ...
];

figure();
subplot(3,1,1);
xlabel('Tiempo [s]');
ylabel('Sonoridad [dBFS]');
hold on;
plot(y,(x(:,1)-255)/4);
plot(y,(x(:,2)-255)/4);
plot(y,(x(:,3)-255)/4);
legend({'Sonoridad momentanea','Sonoridad short-term','Sonoridad
integral'});
hold off;

subplot(3,1,2);

plot(y,20*log10(x(:,4)/(2^12)));
xlabel('Tiempo [s]');
ylabel('   Ganancia [dB]');
subplot(3,1,3);

plot(y,((x(:,1)-255)/4)+20*log10(x(:,4)/(2^12)));
xlabel('Tiempo [s]');
ylabel('Salida del sistema [dBFS]');
```

Script 1. Representación gráfica de los valores instantáneos del normalizador