

**UNIVERSIDAD NACIONAL DE CÓRDOBA**

**FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y  
NATURALES**

**CÁTEDRA DE ELECTRÓNICA DIGITAL III**

---

**TRABAJO PRÁCTICO INTEGRADOR**

**CONSOLA DIGITAL DE SONIDOS**

**Grupo N° 14**

**Alumnos:** Blanco, Luciano Joaquin  
Tonini, Ciro Facundo

**Profesor:** Migliore, Emiliano Elvio

Año 2025

# Índice

<b>Términos y Definiciones</b>	<b>2</b>
<b>1. Proyecto:</b>	<b>3</b>
<b>2. Descripción del SEP</b>	<b>4</b>
<b>3. Desarrollo de Hardware</b>	<b>5</b>
3.1. Circuito del SSE de Amplificador de la señal . . . . .	5
3.2. Circuito del SSE de Etapa de aislación . . . . .	8
3.3. Circuito del SSE de Entrada de Datos . . . . .	9
<b>4. Desarrollo de Firmware</b>	<b>10</b>
4.1. Firmware del SSE de Entrada de Datos . . . . .	10
4.2. Firmware del SSE de Visualización de Datos y Señalización Acústica . . . . .	11
4.3. Firmware del SSE de Comunicación de Datos . . . . .	19
<b>5. Pruebas de Sistema</b>	<b>21</b>
<b>6. Conclusiones</b>	<b>23</b>
<b>7. Bibliografía y Referencias</b>	<b>24</b>
<b>8. Anexo</b>	<b>24</b>
8.1. Circuito Esquemático del SEP . . . . .	24
8.2. Firmware del SEP . . . . .	24
8.3. Lista de Materiales . . . . .	29
8.4. Hojas de Datos . . . . .	29

## Términos y Definiciones

**TPI:** Trabajo Práctico Integrador.

**SEP:** Sistema Electrónico Programable.

**SSE:** Subsistema Electrónico.

**SSEP:** Subsistema Electrónico Programable.

# 1. Proyecto:

El presente proyecto consiste en el desarrollo de una **Consola de Mezcla Digital**, basada en la aplicación de la placa de desarrollo “**LPCXpresso 1769 ARM Cortex-M3**” como microcontrolador, destinada al **procesamiento digital de audio**, la **reproducción de señales**, el **control de volumen** y aplicación de efectos.

**SEP: Consola electrónica de sonidos**

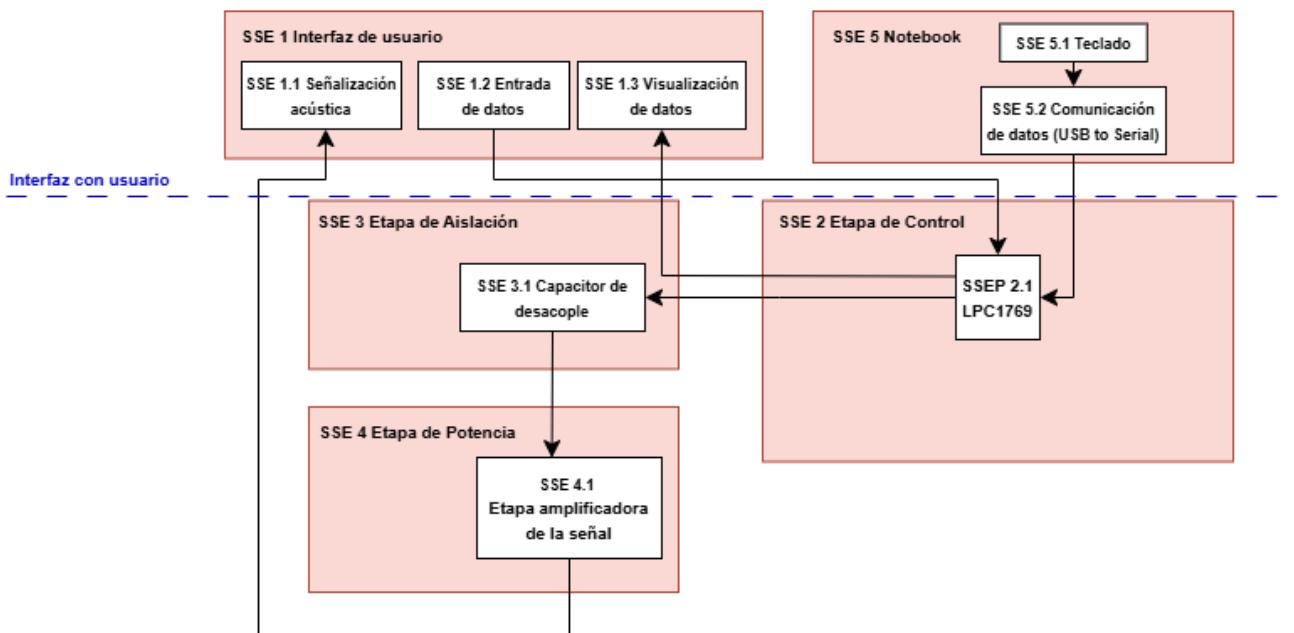


Figura 1: Diagrama general del Sistema Electrónico Programable (SEP): Consola electrónica de sonidos.

## 2. Descripción del SEP

Controlada por el teclado de la misma notebook que alimenta la placa, mediante comunicación UART se envían los datos en serie vía USB (**SSE 5.2**) de la tecla presionada (**SSE 5.1**). Esta activará uno de los samples guardados en **FLASH** y se reproducirán a través de un woofer (**SSE 1.1**).

Con el fin de observar el comportamiento de las señales reproducidas, así como su interacción al aplicar diferentes efectos en tiempo real, se empleará un osciloscopio (**SSE 1.3**).

La consola contará además con un amplificador de audio (**SSE 4.1**) conectado a la salida del **DAC**, permitiendo acondicionar la señal para ser percibida por el usuario. Para la protección del sistema se incorporará un capacitor de desacople (**SSE 3.1**).

El control de volumen se realizará mediante un potenciómetro (**SSE 1.2**) conectado al conversor analógico-digital (**ADC**). El valor digital obtenido de dicha conversión será interpretado por el sistema para ajustar el nivel de salida de audio. Además de esta aplicación para ADC usaremos otras perillas o botones para controlar distintos efectos del sistema.

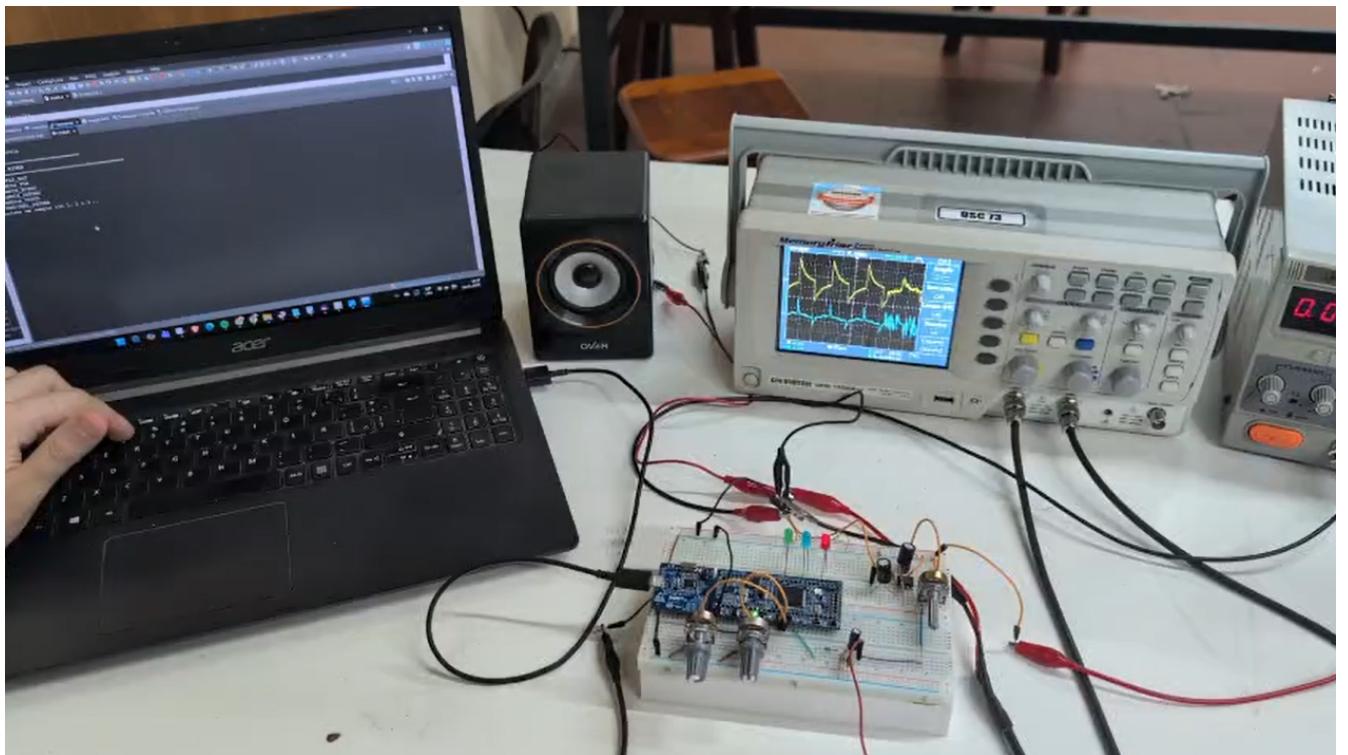


Figura 2: Presentación del TPI

### 3. Desarrollo de Hardware

#### 3.1. Circuito del SSE de Amplificador de la señal

Para el bloque en cuestión, utilizamos el amplificador de potencia LM386 diseñado para aplicaciones de bajo voltaje como es nuestro caso, utilizamos el circuito propuesto por el fabricante para una ganancia de 20 veces:

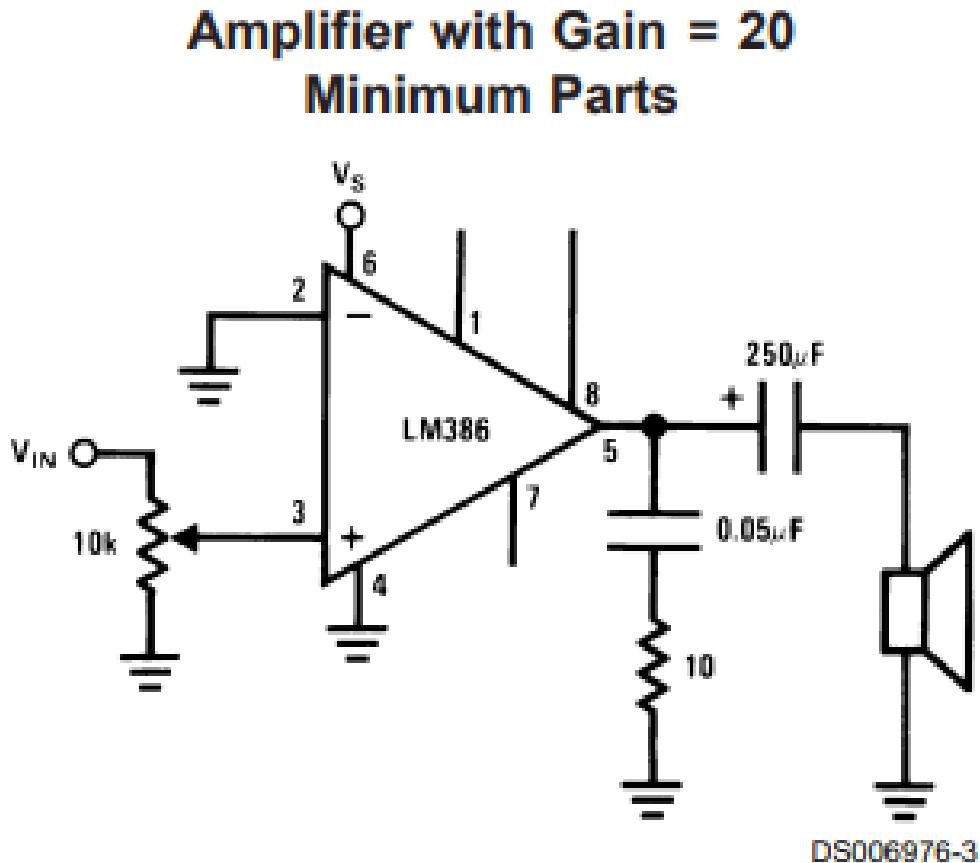


Figura 3: Imagen extraída del datasheet LM386N

En esta configuración el amplificador posee un ancho de banda de 300kHz (más que suficiente para el rango audible), agregamos capacitores de 100uF y 0.1uF entre GND y Vs (pines 4 y 6) para eliminar los picos altos y bajos provenientes de la alimentación. Esto mejora la estabilidad del amplificador eliminando ruido.

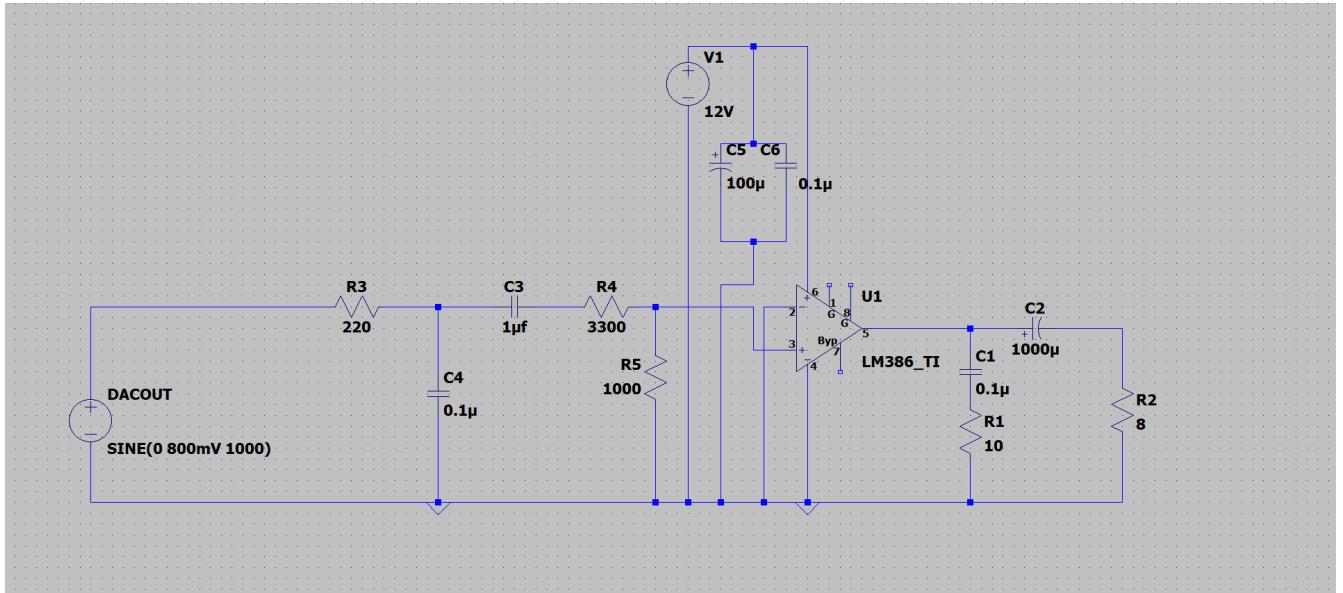


Figura 4: Circuito etapa Amplificadora realizado en LTspice

Simulamos el circuito final, comprobamos ganancia de 20 veces y baja distorsión en la señal de salida.

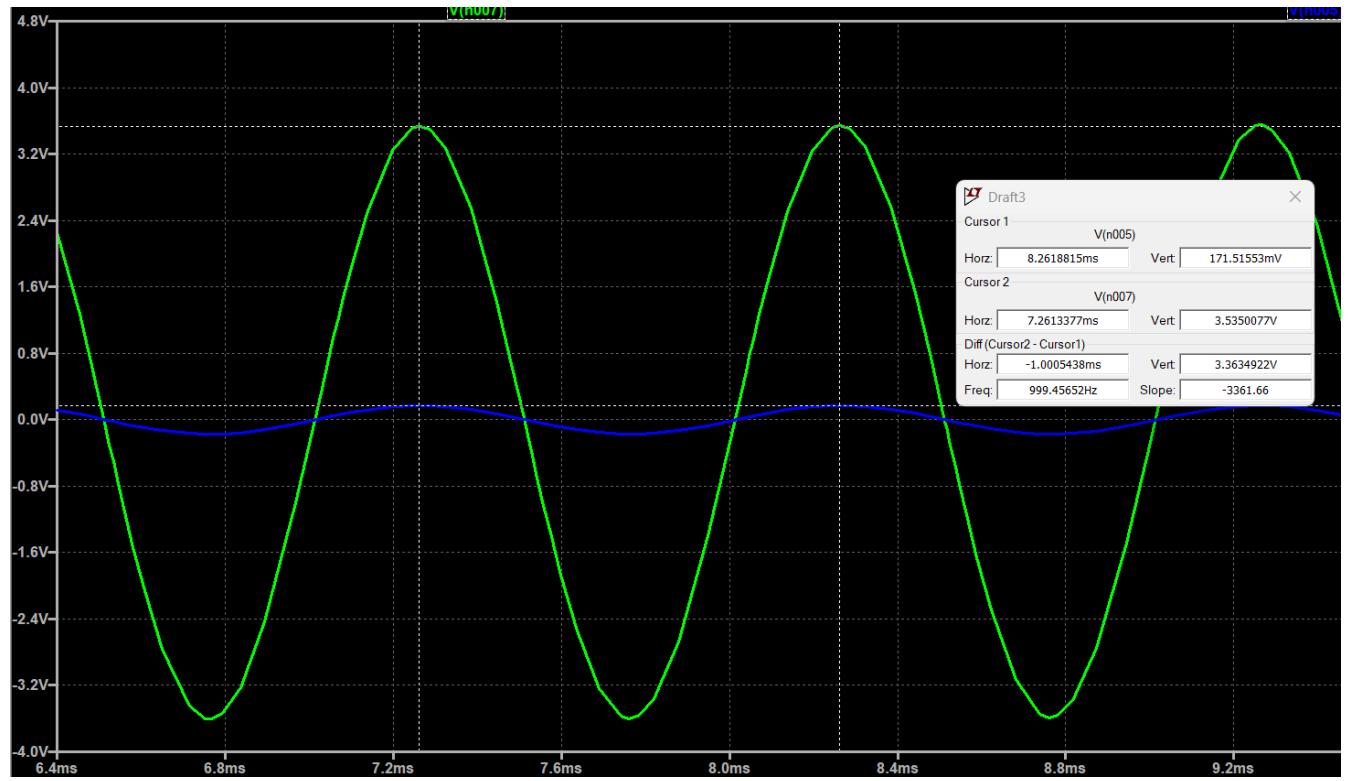


Figura 5: Señal de entradas vs salida en simulación de LTspice

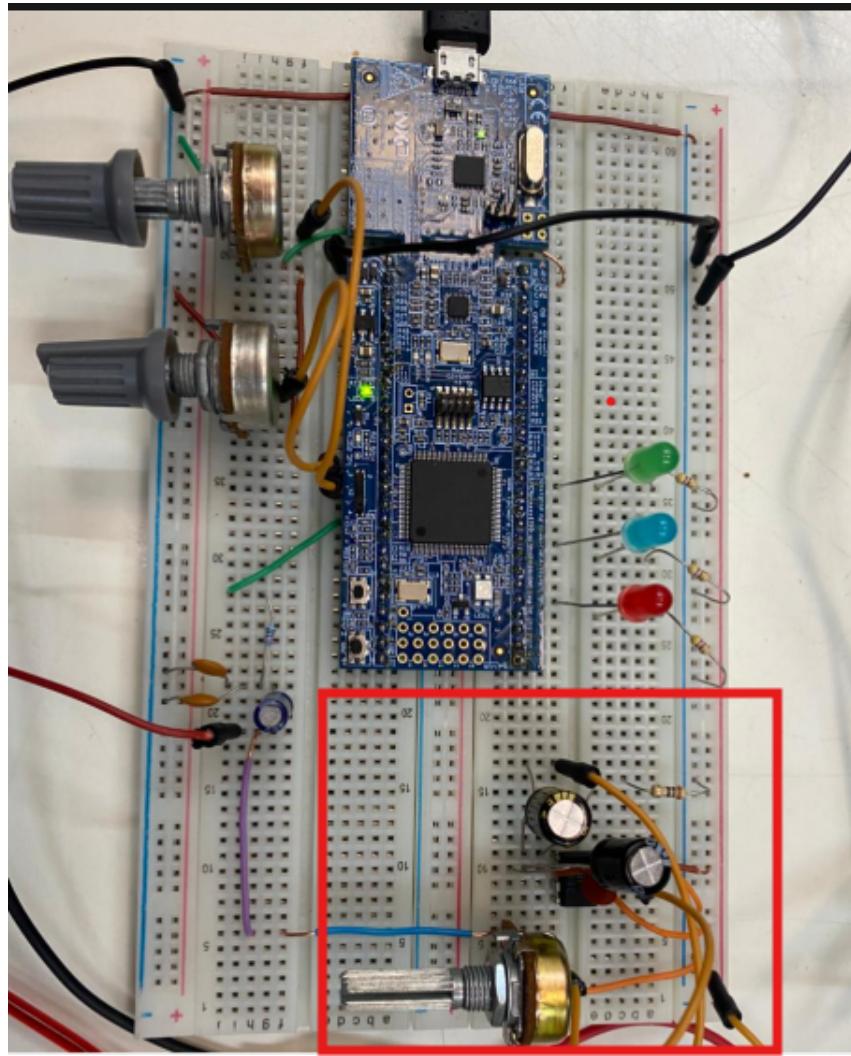


Figura 6: Circuito etapa Amplificadora

### 3.2. Circuito del SSE de Etapa de aislación

Para acondicionar la señal que sale del DAC (P0[26]) aplicamos un filtro RC para eliminar la señal visible (interferencia) generada como consecuencia del tiempo de establecimiento de las muestras en la salida.

Además, se agregó un capacitor de  $1\ \mu\text{F}$  para el desacople de continua de la señal para su posterior tratamiento en la etapa de amplificación.

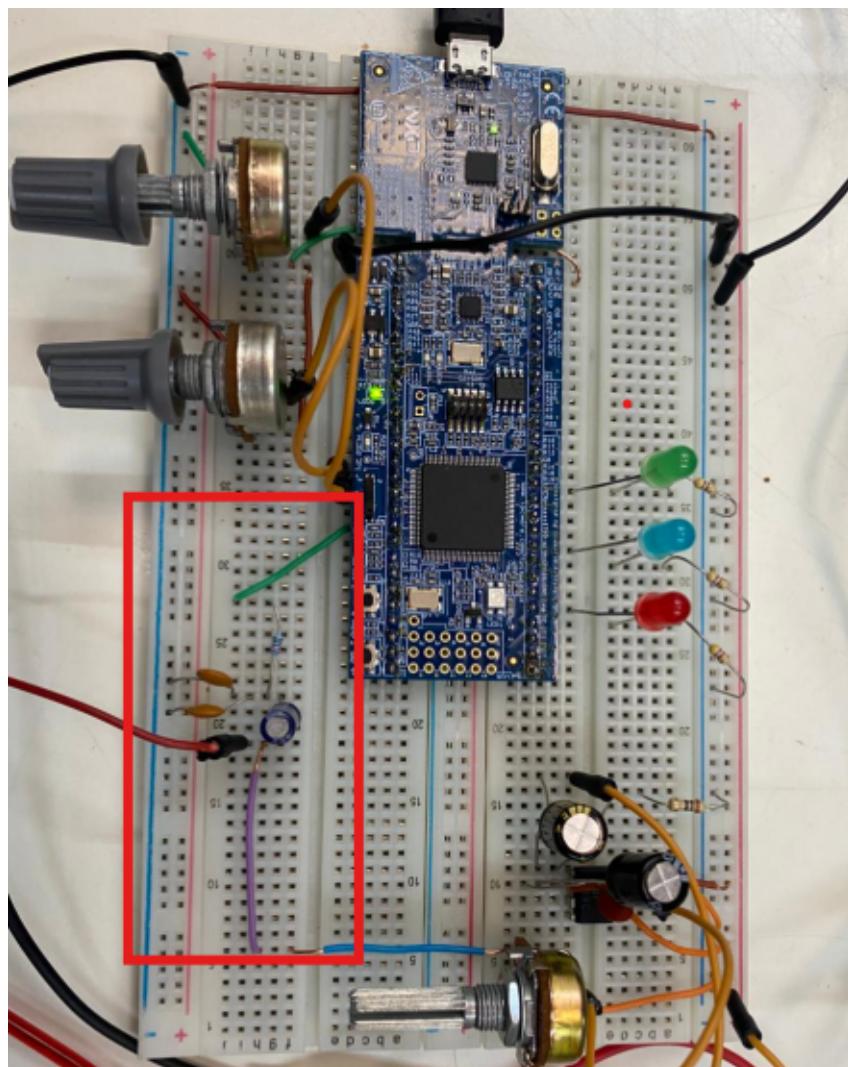


Figura 7: Circuito etapa Amplificadora

### 3.3. Circuito del SSE de Entrada de Datos

Consiste en dos potenciómetros referenciados a 3,3 V y 0 V provenientes de la alimentación del microcontrolador. Éstos están conectados a los pines P0.23 (ADC0.0) y P0.24 (ADC0.1), que corresponden a los canales 0 y 1 del ADC.

La configuración de ambos canales y del ADC en general se detallan en la sección 4.1 *Firmware del SSE de Entrada de Datos*, así como la función que cumplen las señales digitalizadas de ambos potenciómetros.

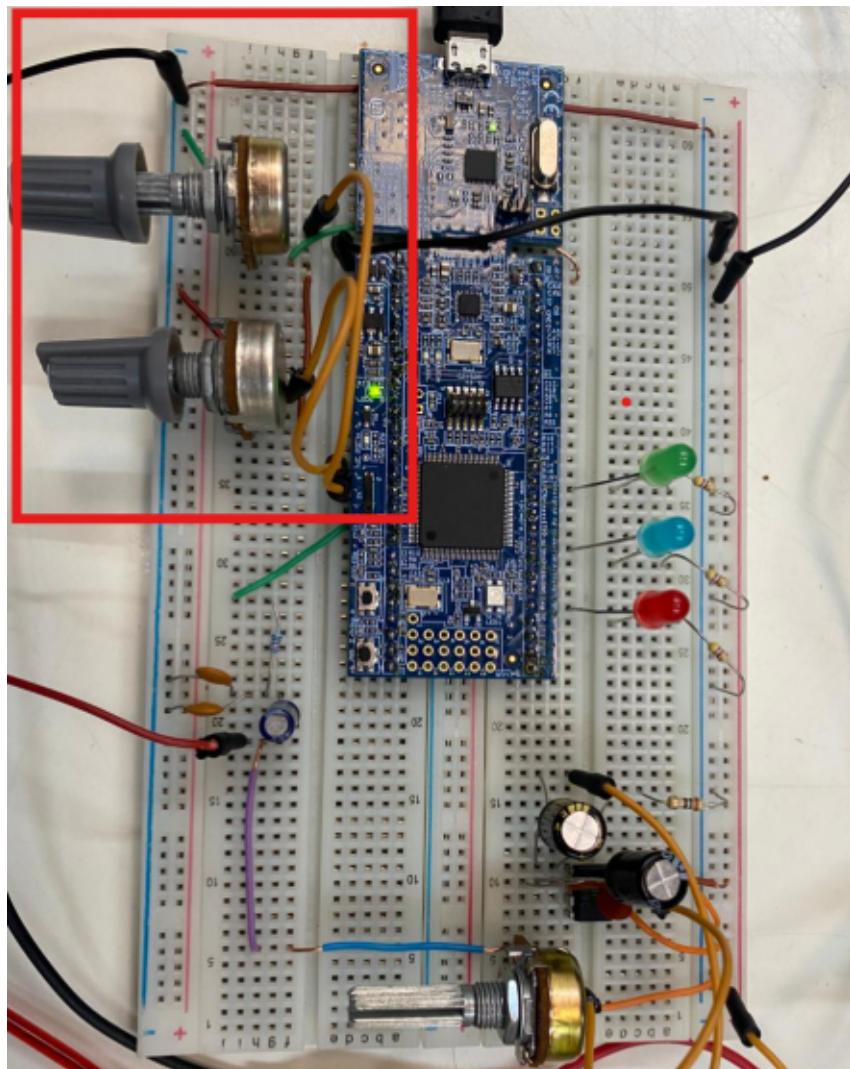


Figura 8: Circuito etapa Amplificadora

## 4. Desarrollo de Firmware

### 4.1. Firmware del SSE de Entrada de Datos

Este bloque permite al usuario modificar la señal que está siendo reproducida, los distintos dispositivos que permiten esta acción están especificados en la Sección *Desarrollo de Hardware — Circuito del SSE de Entrada de Datos*.

Inicializamos el ADC con una frecuencia de muestreo de 200 kHz (máxima posible). Alimentamos al periférico (AD0CR ON) y se le asigna una frecuencia de clock correspondiente a 1/8 de la frecuencia del core:

$$F_{\text{CLKADC}} = \frac{F_{\text{CORE}}}{8} = \frac{100 \text{ MHz}}{8} = 12,5 \text{ MHz}$$

Habilitamos el canal 0 (AD0.0), correspondiente a las muestras provenientes del potenciómetro regulador de volumen, previamente configurado en GPIO (ver Código Fuente — Archivo `main.c`, función `cfgGPIO()`), y deshabilitamos la interrupción por fin de conversión para dicho canal. Las interrupciones son deshabilitadas ya que al estar en modo ráfaga (a una frecuencia de muestreo suficiente) el ADC va a estar convirtiendo los dos canales constantemente y las conversiones de los canales estarán listas cada vez que las necesitemos sin necesidad de atender una interrupción de ADC, .

```
128 void cfgADC(void) {
129     ADC_Init(LPC_ADC, 200000);           // Frecuencia de muestreo del ADC = 200 kHz
130     ADC_ChannelCmd(LPC_ADC, 0, ENABLE); // Canal 0 (P0.23)
131     ADC_IntConfig(LPC_ADC, ADC_ADINTEN0, DISABLE);
132
133
134     ADC_ChannelCmd(LPC_ADC, 1, ENABLE); // Canal 1 (P0.23)
135     ADC_IntConfig(LPC_ADC, ADC_ADINTEN1, DISABLE);
136
137     ADC_BurstCmd(LPC_ADC, ENABLE);
138
139 }
```

Figura 9: Fragmento de código de la configuración del periférico ADC.

Al estar en modo BURST, el ADC convierte constantemente para tener los datos actualizados del volumen o del efecto correspondiente a la misma frecuencia con la que se dispara una muestra de un SAMPLE determinado ya que capturamos el dato cada vez que se ingresa al handler del TIMERO donde se ejecuta la mayor parte del procesamiento de audio, detallado más adelante en la Sección *Firmware del SSE de Visualización de Datos*. Del mismo modo se obtienen los datos del resto de canales.

## 4.2. Firmware del SSE de Visualización de Datos y Señalización Acústica

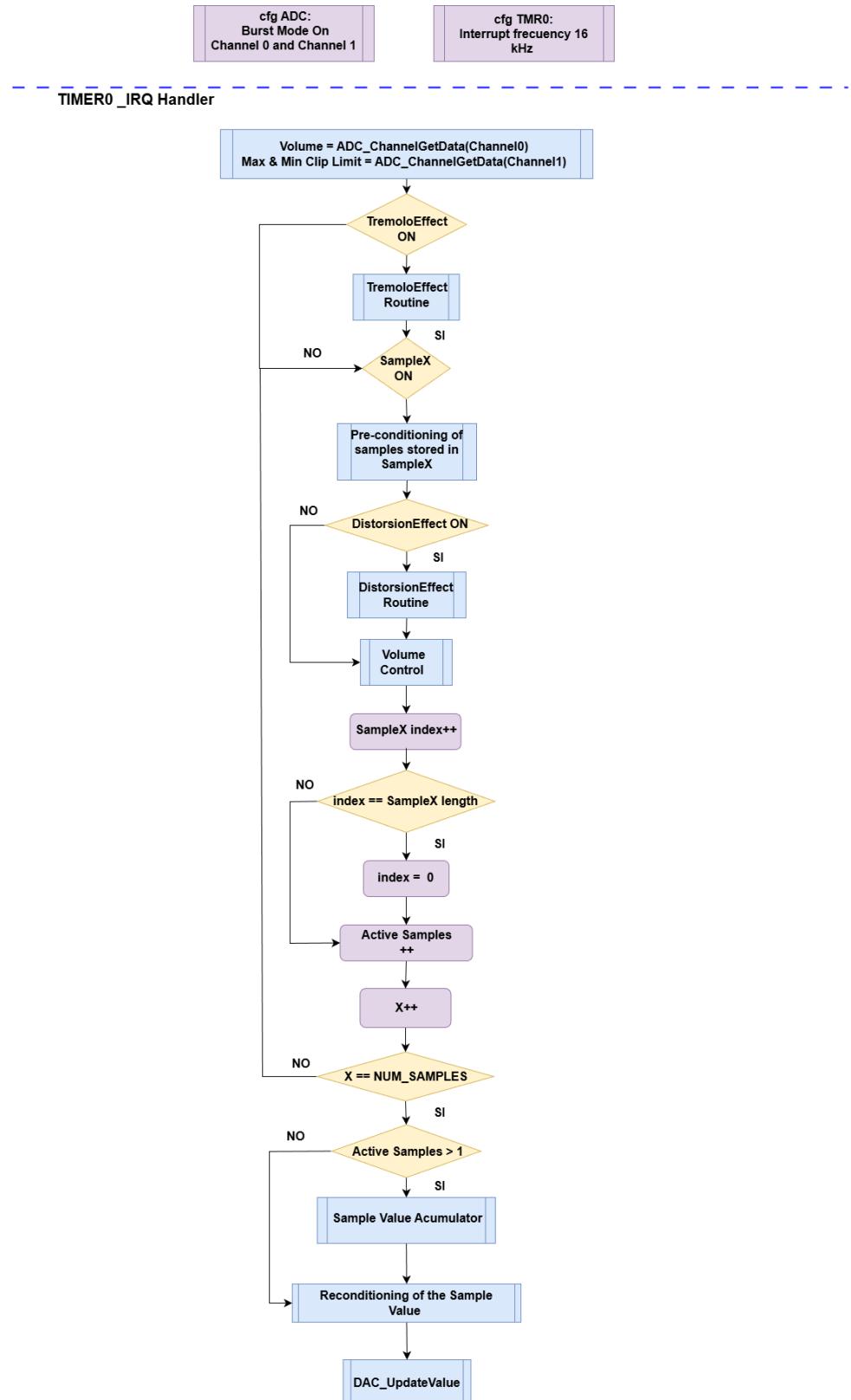


Figura 10: Fragmento de código de la configuración del periférico ADC.

Decidimos unir ambos bloques porque representan dos formas distintas de percibir la misma señal: la visualización (osciloscopio) y la señal acústica (woofer). Por lo tanto, la lógica interna del código es compartida.

En el **handler del TIMER0** se ejecuta todo el procesamiento crítico de audio: reproducción de samples, mezcla, efectos, control de volumen y envío final al DAC.

Los **samples se almacenan en memoria FLASH** debido a su mayor capacidad (512 kB), frente a los 64 kB disponibles en RAM. Por lo que decidimos sacrificar la velocidad que nos ofrece el uso de la RAM para obtener mayor capacidad a la hora de almacenar muestras. En promedio un sample de 1s, que contiene unas 1000 muestras de 8 bits cada una ocupa unos 9,77 kB. Por lo que para almacenar varios de ellos o aumentar su duración era la mejor opción. Un sample de 1 s, muestreado a 16 kHz con 8 bits por muestra, ocupa:

$$\text{Tamaño} = 10000 \text{ muestras/s} \times 1 \text{ byte/muestra} \approx 9,77 \text{ kB}$$

Para obtener los samples, primero se recorta la pista con la herramienta de grabación y edición de audio de código abierto **Audacity** a 16 kHz. Luego, se exporta en formato WAV y se convierte a código C mediante **WavToCode**, que genera arreglos **unsigned char** de 8 bits listos para compilar.

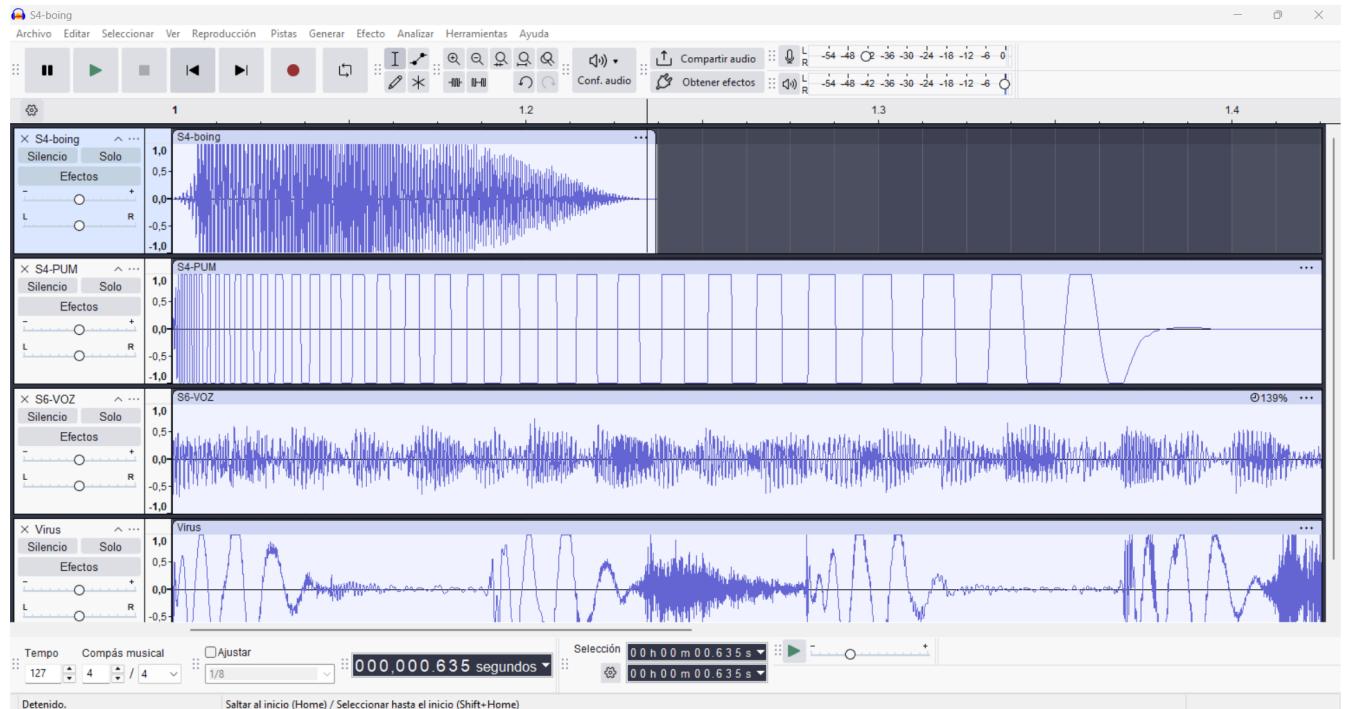


Figura 11: Captura de pantalla del software Audacity.

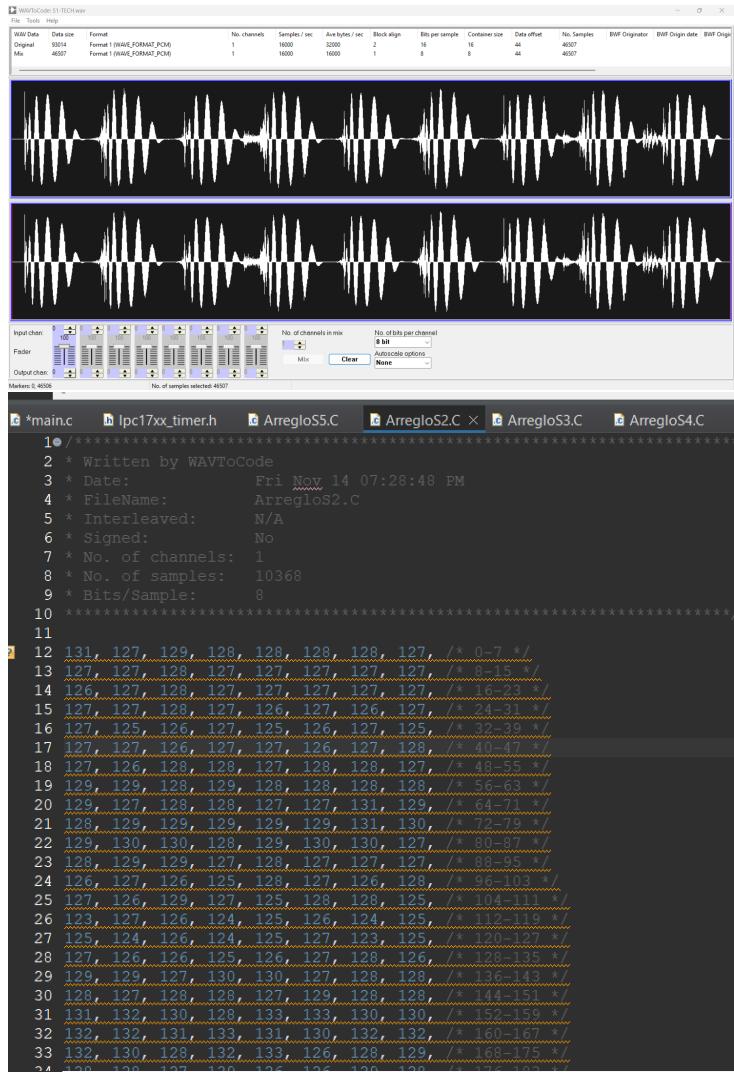


Figura 12: Captura de pantalla del software WaveToCode y arreglo generado

Para almacenar estos arreglos en **FLASH** se creó el header `samples.h..`. Allí se declaran los arreglos como `extern const` y las variables correspondientes al número de espacios de memoria que ocupan, esto hace óptimo el agregado o la modificación de samples a nuestro programa. Cada sample posee su propio archivo `sampleX_data.c` que contiene únicamente los datos en memoria. Esto mantiene limpio el archivo `main.c` y facilita agregar o quitar samples.

```

samples.h ×
1 #ifndef SAMPLES_H_
2 #define SAMPLES_H_
3
4 #include <stdint.h>
5
6 extern const uint8_t sampleS5[];
7 extern const uint8_t sampleS1[];
8 extern const uint8_t sampleS2[];
9 extern const uint8_t sampleS3[];
10 extern const uint8_t sampleS4[];
11 extern const uint8_t sampleS6[];
12
13
14 extern const uint32_t SAMPLE_S5_LEN;
15 extern const uint32_t SAMPLE_S1_LEN;
16 extern const uint32_t SAMPLE_S2_LEN;
17 extern const uint32_t SAMPLE_S3_LEN;
18 extern const uint32_t SAMPLE_S4_LEN;
19 extern const uint32_t SAMPLE_S6_LEN;
20
21
22
23 #endif
24

sampleS1_data.c ×
1 #include <samples.h>
2 #include <stdint.h>
3 #define NUM_ELEMENTS 21344
4
5 const uint8_t sampleS1[NUM_ELEMENTS] = {
6     #include "../ArregloS1.C"
7 };
8 const uint32_t SAMPLE_S1_LEN = NUM_ELEMENTS;
9

```

Figura 13: Fragmento de código de samples.h y sampleS1data

Como mencionadmos en el `TIMER0IRQHandler()`, ejecutado cada:

$$T = \frac{1}{16000} = 62,5 \mu s$$

Tiempo calculado con la función MATCH (ver `cfgTMR()` en el código final).

Se reproduce cada sample activo, se mezclan, se les aplica volumen y otros efectos digitales. Para organizar esta lógica se utiliza una estructura **SamplePlayer**, que encapsula:

- Puntero al arreglo de datos (`*data`).
- Longitud (`length`).
- Índice actual (`index`)
- Estado (`active = 1/0`).

```

23
24 typedef struct {
25     const uint8_t *data;           // puntero al arreglo en Flash
26     uint32_t length;          // cantidad de muestra
27     volatile uint32_t index;      // indice actual
28     volatile uint8_t active;      // 1 = esta reproduciendo
29 } SamplePlayer;
30
31 SamplePlayer players[NUM_SAMPLES];
32

269 void initSamples(void) {
270     players[0].data = sampleS1;
271     players[0].length = SAMPLE_S1_LEN;
272     players[0].index = 0;
273     players[0].active = 0;
274
275     players[1].data = sampleS5;
276     players[1].length = SAMPLE_S5_LEN;
277     players[1].index = 0;
278     players[1].active = 0;
279
280     players[2].data = sampleS2;
281     players[2].length = SAMPLE_S2_LEN;
282     players[2].index = 0;
283     players[2].active = 0;
284
285     players[3].data = sampleS3;
286     players[3].length = SAMPLE_S3_LEN;
287     players[3].index = 0;
288     players[3].active = 0;
289
290     players[4].data = sampleS4;
291     players[4].length = SAMPLE_S4_LEN;
292     players[4].index = 0;
293     players[4].active = 0;
294
295     players[5].data = sampleS6;
296     players[5].length = SAMPLE_S6_LEN;
297     players[5].index = 0;
298     players[5].active = 0;
299 }
300

```

Figura 14: Fragmento de código de struct **SamplePlayer** y **initSamples()**

Para el procesamiento de cada muestra se convierte la señal desde formato **unsigned 8 bits** a una representación centrada en cero, se le asigna al dato original una variable **int32 muestra** para evitar el overflow:

$$x[n] = \text{muestra\_8bit} - 128$$

Luego todos los samples activos se suman en la variable **mix**. Si hay más de un sample activo:

$$mix = \frac{mix}{2}$$

para evitar saturación. Finalmente, la señal se vuelve a desplazar a unsigned y se convierte a 10 bits para adaptarla a nuestro DAC:

$$y[n] = (mix + 128) \ll 2$$

```

153     //samples activos
154     uint8_t activos = 0;
155
156     //lógica principal aplica efectos a los samples que se suman
157     for (int i = 0; i < NUM_SAMPLES; i++) {
158         if (players[i].active) []
159             int32_t muestra = (int32_t)players[i].data[players[i].index] - 128;
160             int32_t salida = (muestra * vol_A) >> 12;
161             mix += salida;
162
163             // avanzar índice
164             if (++players[i].index >= players[i].length)
165                 players[i].index = 0;
166                 activos++;
167     }
168 }
169
170 // Si hay 2 samples sonando, dividimos por 2 para evitar saturación.
171 // Si hay 1 sample, 'mix' queda intacto (volumen completo).
172 if (activos > 1) {
173     mix = mix >> 1; // Divide por 2 (equivale a mix /= 2)
174 }
175
176 // Convertir a rango SIN SIGNO (0-255) para el DAC
177 mix += 128;
178
179 // Saturación (seguridad)
180 // Aseguramos que el valor esté en el rango de 8 bits
181 if (mix > 255) mix = 255;
182 if (mix < 0) mix = 0;
183
184 // Convertir 8 bits (0-255) a 10 bits (0-1023) para el DAC
185 uint16_t val10bit = (uint16_t)(mix << 2);
186 DAC_UpdateValue(val10bit);
187
188 TIM_ClearIntPending(LPC_TIM0, TIM_MR1_INT);
189

```

Figura 15: Fragmento de código del cálculo de mezcla y salida al DAC.

## Control digital de volumen

El volumen se obtiene mediante el ADC en el rango 0...4095. El valor se normaliza aplicándolo como ganancia, es decir se multiplica a la muestra por un valor que va desde 0 a 1:

$$y[n] = \frac{VolA}{4096} x[n]$$

Lo cual se implementa mediante:

$$y[n] = (x[n] \times VolA) \gg 12$$

```

147 void TIMER0_IRQHandler(void) {
148     int32_t mix = 0;
149     //variable que recibe los datos de CH0
150     vol_A = ADC_ChannelGetData(LPC_ADC, 0);
151     //variable que recibe CH1
152     uint16_t lecturaClip = ADC_ChannelGetData(LPC_ADC, 1);
153     maxClip = (lecturaClip)>> 4; //convierito a 8bits
154     //samples activos
155     uint8_t activos = 0;
156
157     //logica principal aplica efectos a los samples que se suman
158     for (int i = 0; i < NUM_SAMPLES; i++) {
159         if (players[i].active) {
160             int32_t muestra = (int32_t)players[i].data[players[i].index] - 128;
161             int32_t salida = (muestra * vol_A) >> 12;
162             mix += salida;
163

```

Figura 16: Fragmento del código del control de volumen.

### Efecto de Trémolo (Modulación de Amplitud)

El efecto de Trémolo consiste en una variación rítmica y cíclica del volumen de la señal de audio. Desde un punto de vista técnico, esto se logra mediante una Modulación de Amplitud (AM), donde la señal de audio original es multiplicada por una señal de control de baja frecuencia.

- **Funcionamiento del Algoritmo:** El sistema utiliza un Oscilador de Baja Frecuencia (LFO) generado por software; este LFO evoluciona lentamente generando una forma de onda triangular interna.
- **Generación del LFO:** En cada interrupción del temporizador, el sistema ajusta una variable de ganancia (`lfo_gain`) que oscila entre 0 (silencio) y 256 (volumen máximo). La velocidad de esta oscilación está determinada por la constante `LFO_SPEED`.
- **Aplicación:** Cada muestra de audio procesada se multiplica por el valor actual del LFO y luego se normaliza. El resultado es que el volumen del sonido "sube y baja" automáticamente, creando una sensación de pulsación o vibración sin alterar el tono de la nota.

### Efecto de Distorsión

La distorsión implementada es del tipo "*Hard Clipping*" (Recorte Duro). Este efecto busca simular la saturación al exceder sus límites de voltaje, lo que *recorta* los picos de la onda sonora, transformando ondas suaves (senoidales) en ondas más cuadradas y ricas en armónicos. Esto es percibido auditivamente como un sonido más agresivo o roto".

El comportamiento de este efecto es dinámico y controlable por el usuario mediante un potenciómetro conectado al canal 1 del ADC. La variable `maxClip` define el umbral de saturación, es decir, el límite máximo de amplitud permitido para la señal.

Las Etapas del procesamiento son:

- **Adquisición:** El sistema lee el valor del ADC (0 a 4095) y realiza un desplazamiento de bits a la derecha ( $\sim 4$ ) para escalar el valor a un rango de 8 bits (0 a 255). Este valor resultante es `maxClip`.
- **Pre-amplificación:** Antes de recortar, la señal de audio original se multiplica por un factor de ganancia (`CLIP_DRIVE`), forzando a la onda a crecer.
- **Comparación y Recorte:** El algoritmo compara la señal amplificada con el valor de `maxClip`:
  - Si la señal supera el valor de `maxClip`, se fuerza a valer exactamente `maxClip` (se recorta el "techo").
  - Si la señal es inferior al negativo de `maxClip`, se fuerza a valer `-maxClip` (se recorta el "piso").
  - Si la señal está dentro del rango, se mantiene intacta.
- **Interacción del Usuario:** Cuando el usuario gira el potenciómetro para disminuir el valor de `maxClip`, el "*techo*" y el "*piso*" de la señal se estrechan. Esto provoca que una mayor parte de la onda sea recortada, incrementando la intensidad de la distorsión auditiva. Por el contrario, un `maxClip` alto permite que la señal pase con menos alteraciones.

### 4.3. Firmware del SSE de Comunicación de Datos

#### UART IRQ Handler

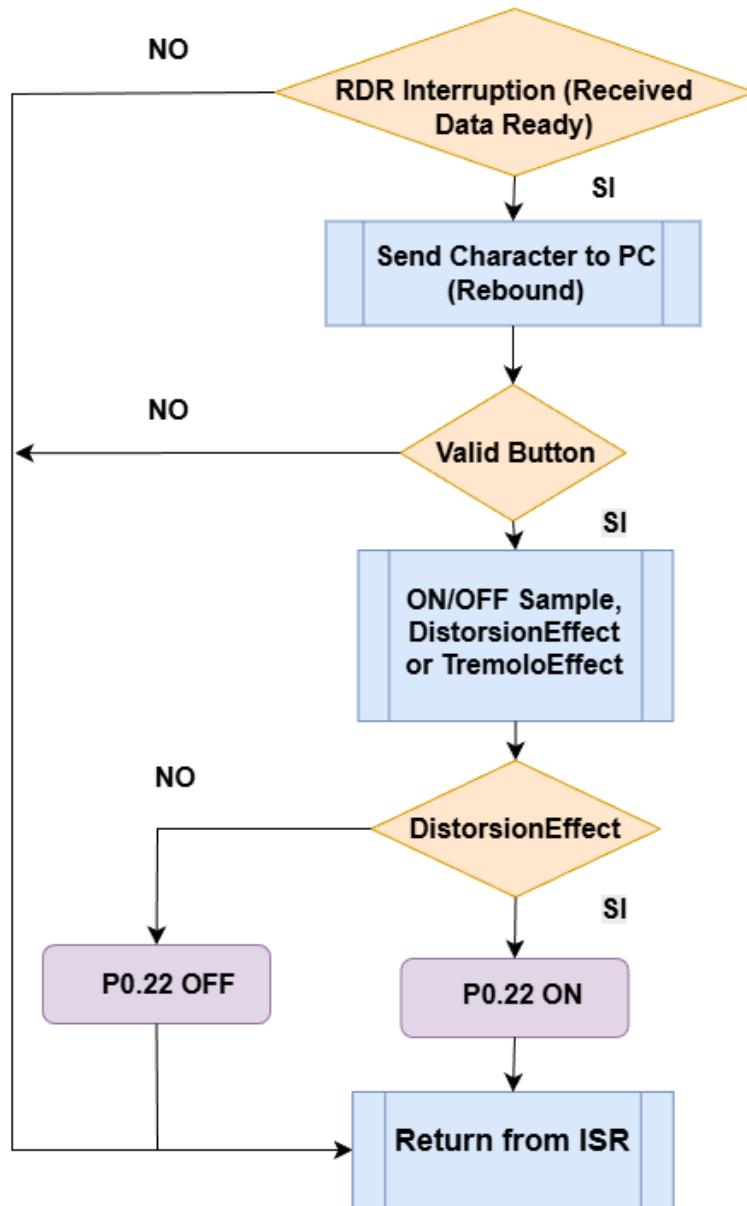


Figura 17: Diagrama de flujo.

El módulo UART cumple la función de ser la interfaz de usuario primaria. Su tarea específica es recibir comandos en tiempo real desde un operador para gobernar la reproducción de audio.

No se utiliza para transferir los datos de audio en sí, sino para recibir comandos de disparo (triggers). El sistema está diseñado para que, al recibir un carácter específico (por ejemplo, ‘1’), el firmware active o desactive la reproducción del sample correspondiente. Además, permite la modificación de estados globales del sistema, como la activación del

“Modo Distorsión” (tecla ‘d’) y el “Modo Tremolo” (tecla ‘t’), que alteran el comportamiento de reproducción de los samples.

## Implementación y Configuración Técnica

Para esta implementación, se utilizó el canal UART0 del microcontrolador LPC1769, configurando los pines del PORT0 para esta función:

- P0.2 como TXD0 (transmisión)
- P0.3 como RXD0 (recepción)

La configuración del periférico (definida en `cfgUART`) establece los parámetros de comunicación:

- **Velocidad de transmisión:** 115200 baudios. Es una velocidad estándar ampliamente soportada y con buen rendimiento para la interacción en tiempo real.
- **Formato de datos:** 8 bits de datos, sin paridad, 1 bit de parada (configuración por defecto).
- **Manejo de datos:** La recepción se realiza mediante interrupciones, habilitando la línea `UART_INTCFG_RBR` (Receive Data Ready).

Este mecanismo basado en interrupciones (rutina `UART0_IRQHandler`) es fundamental para la eficiencia del sistema. Permite que el microcontrolador ejecute otras tareas (como la mezcla de audio en la interrupción del `TIMER0`) sin necesidad de sondear constantemente el puerto (polling), respondiendo a los comandos del usuario sólo cuando llegan.

## Interacción del Usuario

La interacción del usuario con la consola se realiza mediante el Monitor Serie del IDE en la computadora personal, conectada al microcontrolador.

El flujo de interacción es el siguiente:

1. El usuario abre una conexión en el puerto COM correspondiente en la PC, configurada a 115200 baudios.
2. El usuario presiona una tecla en el teclado (por ejemplo: ‘1’, ‘2’, ‘3’, ‘4’, ‘d’, ‘t’).
3. El terminal envía el código ASCII de la tecla a través de la línea TX del PC, conectada a la línea RX (P0.3) del LPC1769.
4. El periférico UART0 recibe el byte y genera una interrupción (RBR).

5. La rutina `UART0_IRQHandler` se ejecuta, lee el byte con `UART_ReceiveByte`, y lo reenvía mediante `UART_SendByte`, generando un “eco” en la consola que confirma la recepción.
6. Un bloque `switch-case` dentro del handler analiza el byte recibido y modifica las variables globales `players[n].active`, `distortionMode` o `tremoloMode`.
7. Este cambio de estado es detectado por la interrupción del `TIMER0` en el siguiente ciclo de ejecución, iniciando, deteniendo o modificando la reproducción del audio según corresponda.

## 5. Pruebas de Sistema

Nro. Caso	Descripción	Paso	Resultado Esperado	Resultado Obtenido
1	<b>Comunicación de datos</b>	<ol style="list-style-type: none"> <li>1. Conectar la placa al PC mediante conversor USB-UART.</li> <li>2. Abrir terminal serie a 115200 baudios.</li> </ol>	Debe aparecer el mensaje de inicio en la terminal. El sistema queda a la espera de comandos.	<b>PASA</b>
2	<b>Señalización Acústica y Visualización de datos</b>	<ol style="list-style-type: none"> <li>1. Enviar el carácter '1' por la terminal.</li> <li>2. Verificar audio.</li> <li>3. Enviar nuevamente '1'.</li> </ol>	<ol style="list-style-type: none"> <li>1. Comienza a reproducirse el <i>Sample 1</i> en bucle.</li> <li>3. La reproducción del <i>Sample 1</i> se detiene inmediatamente (silencio).</li> </ol>	<b>PASA</b>
Continúa en la siguiente página...				

Tabla 1 – continúa de la página anterior

Nro. Caso	Descripción	Paso	Resultado Esperado	Resultado Obtenido
3	<b>Mezcla de dos canales</b>	<ol style="list-style-type: none"> <li>1. Activar <i>Sample 1</i> (tecla '1').</li> <li>2. Activar <i>Sample 2</i> (tecla '2') simultáneamente.</li> <li>3. Enviar '1' nuevamente.</li> </ol>	<p>1 y 2. Se deben escuchar/ver ambas señales sumadas. La amplitud pico a pico no debe duplicarse ni saturar violentamente (debido a la lógica mix ^ 1 en el código).</p> <p>3. Se apaga Sample 1 y queda sonando solo el Sample 2.</p>	<b>PASA</b>
4	<b>Control de Volumen General</b>	<ol style="list-style-type: none"> <li>1. Reproducir cualquier sample.</li> <li>2. Variar el potenciómetro conectado al pin <b>P0.23 (ADC0.0)</b> desde 0V a 3.3V.</li> </ol>	La amplitud de la señal de salida en el osciloscopio debe variar proporcionalmente al movimiento del potenciómetro (de silencio a volumen máximo).	<b>PASA</b>
5	<b>Control de Clipping y Efecto Distorsión</b>	<ol style="list-style-type: none"> <li>1. Reproducir un sample limpio.</li> <li>2. Activar el modo distorsión enviando 'd'.</li> <li>3. Variar el potenciómetro en <b>P0.24 (ADC0.1)</b> mientras suena un tono.</li> </ol>	Al girar el potenciómetro, la señal debe pasar de ser una onda completa a una onda recortada (cuadrada) progresivamente, indicando cambio en la variable max-Clip. El sonido debe percibirse más agresivo/roto.	<b>PASA</b>

Continúa en la siguiente página...

Tabla 1 – continúa de la página anterior

Nro. Caso	Descripción	Paso	Resultado Esperado	Resultado Obtenido
6	<b>Efecto de Trémolo</b>	1. Reproducir un sample sostenido. 2. Enviar tecla 't'. 3. Observar la envolvente de la señal.	La amplitud de la señal debe oscilar (subir y bajar) rítmicamente de forma automática.	<b>PASA</b>

## 6. Conclusiones

El trabajo fue realizado con éxito, implementando los distintos **módulos periféricos** aprendidos en la materia —**ADC**, **UART**, entre otros— dentro de una aplicación concreta, lo cual permitió comprender en mayor profundidad el funcionamiento real de cada uno.

La tarea de **almacena** **datos** resultó clave para afianzar el entendimiento sobre cómo se estructura el **mapa de memoria** de la **LPC1769**, así como para identificar las diferencias fundamentales entre el uso de **FLASH** y **RAM** en términos de capacidad, velocidad y accesibilidad.

Sin dudas, las mejoras posibles son numerosas. Entre las propuestas más relevantes se encuentran:

- Implementación de **DMA** para permitir la transferencia eficiente de datos desde la memoria **FLASH** hacia un **buffer** en **RAM**, habilitando la manipulación de muestras en tiempo real y reduciendo significativamente la carga de la **CPU**.
- Incorporación de una **entrada de audio** (jack) para procesar y aplicar efectos a **señales en tiempo real**, ampliando las capacidades del sistema.
- Desarrollo de un **modo de grabación**, permitiendo capturar audio durante algunos segundos (por ejemplo, 3 s), almacenarlo temporalmente y transferirlo a una **PC** para su posterior uso o edición.

## 7. Bibliografía y Referencias

## 8. Anexo

### 8.1. Circuito Esquemático del SEP

### 8.2. Firmware del SEP

```
1/* -----
2 *          HEADER
3 * -----
4 * */
5 #include "samples.h"
6 #include "LPC17xx.h"
7 #include "lpc17xx_gpio.h"
8 #include "lpc17xx_pinsel.h"
9 #include "lpc17xx_timer.h"
10 #include "lpc17xx_adc.h"
11 #include "lpc17xx_dac.h"
12 #include "lpc17xx_uart.h"
13 #include <string.h>
14/* -----
15 *          DEFINES
16 * -----
17 */
18 #define FREQ_MUESTREO 16000 // Frecuencia de muestreo DAC
19 #define PCLK_TIMER0 25000000 // Frecuencia periférica T0 (ajustar según tu config)
20 #define PORT_0 (uint8_t)0
21 #define PIN_1 (uint32_t)(1<<1)
22 #define NUM_SAMPLES 6
23 #define OUTPUT (uint8_t)1
24 #define INPUT (uint8_t)0
25 #define PIN_22 (1U << 22)
26 #define CLIP_DRIVE 4 // Ganancia de entrada
27 #define LFO_SPEED 4 // Velocidad del LFO (más rápido = más rápido)
28/* -----
29 *          VARIABLES
30 * -----
31 */
32typedef struct {
33    const uint8_t *data;      // Puntaje al apuntar en Flash
34    uint32_t length;         // cantidad de muestras
35    volatile uint32_t index; // indice actual
36    volatile uint8_t active; // * La data reproduciendo
37} SamplePlayer;
38
39 SamplePlayer players[NUM_SAMPLES];
40
41 volatile uint32_t idx = 0;
42 volatile uint16_t vol_A = 2048; // Valor inicial medio (ganancia 0.5 aprox)
43 volatile uint16_t maxClip = 255;
44 volatile uint8_t distortion = 0;
45 volatile uint8_t tremoloOn = 0;
46 static int16_t lfo_gain = 256; // Ganancia del LFO (256 = 100% volumen, 0 = 0%)
47 static int8_t lfo_dir = -1; // dirección del LFO (subiendo o bajando)
48 static uint16_t lfo_prescaler = 0; // Prescaler para controlar la velocidad
49/* -----
50 *          FUNCTIONS
51 * -----
52 */
53 void cfgDAC(void);
54 void cfgTMR(void);
55 void cfgGPIO(void);
56 void cfgADC(void);
57 void cfgUART(void);
58 void initSamples(void);
59 void sendWelcomeMessage(void);
60 void tremoloEffect(void);
61
62/* -----
63 *          MAIN
64 * -----
65 */
66int main(void) {
67    cfgGPIO();
68    cfgDAC();
69    cfgADC();
70    cfgTMR();
71    cfgUART();
72    initSamples();
73    sendWelcomeMessage();
74    while (1) {
75
76    }
77    return 0;
78}
79
80/* -----
81 *          GPIO
82 * -----
83 */
84
85void cfgGPIO(void) {
86    /* PBC (pin 26) */
87}
```

```

87     PINSEL_CFG_Type cfgPinDac;
88     cfgPinDac.portNum = 0;
89     cfgPinDac.pinNum = 26;
90     cfgPinDac.funcNum = 2;
91     cfgPinDac.pinMode = PINSEL_TRISTATE;
92     cfgPinDac.openDrain = PINSEL_OD_NORMAL;
93     PINSEL_ConfigPin(&cfgPinDac);
94
95     // ADC0.0 (P0.23)
96     PINSEL_CFG_Type cfgADC0;
97     cfgADC0.portNum = 0;
98     cfgADC0.pinNum = 23;
99     cfgADC0.funcNum = 1;
100    cfgADC0.pinMode = PINSEL_TRISTATE;
101    cfgADC0.openDrain = PINSEL_OD_NORMAL;
102    PINSEL_ConfigPin(&cfgADC0);
103
104    PINSEL_CFG_Type cfgPinLED, cfgPinRXD0, cfgPinTXD0;
105
106    // --- LED en P0.22 ---
107    cfgPinLED.portNum = PINSEL_PORT_0;
108    cfgPinLED.pinNum = PINSEL_PIN_22;
109    cfgPinLED.funcNum = PINSEL_FUNC_0;
110    cfgPinLED.pinMode = PINSEL_PULLUP;
111    cfgPinLED.openDrain = PINSEL_OD_NORMAL;
112    PINSEL_ConfigPin(&cfgPinLED);
113
114    // --- RXD0 (P0.3) ---
115    cfgPinRXD0.portNum = PINSEL_PORT_0;
116    cfgPinRXD0.pinNum = PINSEL_PIN_3;
117    cfgPinRXD0.funcNum = PINSEL_FUNC_1;
118    cfgPinRXD0.pinMode = PINSEL_PULLUP;
119    cfgPinRXD0.openDrain = PINSEL_OD_NORMAL;
120    PINSEL_ConfigPin(&cfgPinRXD0);
121
122    // --- TXD0 (P0.2) ---
123    cfgPinTXD0.portNum = PINSEL_PORT_0;
124    cfgPinTXD0.pinNum = PINSEL_PIN_2;
125    cfgPinTXD0.funcNum = PINSEL_FUNC_1;
126    cfgPinTXD0.pinMode = PINSEL_PULLUP;
127    cfgPinTXD0.openDrain = PINSEL_OD_NORMAL;
128    cfgPinTXD0.openDrain = PINSEL_OD_NORMAL;
129    PINSEL_ConfigPin(&cfgPinTXD0);
130
131    GPIO_SetDir(PORT_0, PIN_22, OUTPUT);
132    LPC_GPIO0->PIOSET |= (PIN_22);
133 }
134 */
135 *----- DAC -----
136 *----- *
137 */
138 void cfgDAC(void) {
139     DAC_Init();
140 }
141 */
142 *----- ADC -----
143 *----- *
144 */
145 void cfgADC(void) {
146     ADC_Init(200000);           // Frecuencia de muestreo del ADC = 200 kHz
147     ADC_ChannelCmd(0, ENABLE); // Canal 0 (P0.23)
148     ADC_IntConfig(ADC_ADINTEN0, DISABLE);
149
150     ADC_ChannelCmd(1, ENABLE); // Canal 1 (P0.23)
151     ADC_IntConfig(ADC_ADINTEN1, DISABLE);
152
153     ADC_BurstCmd(ENABLE);
154 }
155 */
156 */
157 *----- TIMER -----
158 *----- *
159 */
160 void cfgTMR(void) {
161     TIM_TIMERCFG_Type cfgTimer;
162     TIM_MATCHCFG_Type cfgMatch;
163
164     cfgTimer.prescaleOption = TIM_USVAL;
165     cfgTimer.prescaleValue = 1; // 1 µs
166
167     cfgMatch.matchChannel = 1;
168     cfgMatch.matchValue = (1000000 / FREQ_MUESTREO); // 62.5 µs
169     cfgMatch.intOnMatch = ENABLE;

```

```

170     cfgMatch.stopOnMatch = DISABLE;
171     cfgMatch.resetOnMatch = ENABLE;
172     cfgMatch.extMatchOutputType = TIM NOTHING;
173
174     TIM_Init(LPC_TIM0, TIM_TIMER_MODE, &cfgTimer);
175     TIM_ConfigMatch(LPC_TIM0, &cfgMatch);
176     NVIC_EnableIRQ(TIMER0 IRQn);
177     TIM_Cmd(LPC_TIM0, ENABLE);
178 }
179
180 void TIMER0_IRQHandler(void) {
181
182     int32_t mix = 0;
183     uint8_t activos = 0;
184     vol_A = ADC_ChannelGetData(0);
185     uint16_t lecturaClip = ADC_ChannelGetData(1);
186     maxClip = (lecturaClip)>> 4;
187     tremoloEffect();
188     for (int i = 0; i < NUM_SAMPLES; i++) {
189         if (players[i].active) {
190             int32_t muestra = (int32_t)players[i].data[players[i].index] - 128;
191             if (distorsion) {
192                 int32_t boosted_muestra = muestra * CLIP_DRIVE;
193                 int32_t umbral = (maxClip < 10) ? 10 : maxClip;
194
195                 if (boosted_muestra > umbral) {
196                     muestra = umbral;
197                 } else if (boosted_muestra < -umbral) {
198                     muestra = -umbral;
199                 } else {
200                     muestra = boosted_muestra; // Si no clippea, usa el valor con "drive"
201                 }
202             }
203             int32_t salida = (muestra * vol_A) >> 12;
204             mix += salida;
205             // avanzar indice
206             players[i].index++;
207             if (players[i].index >= players[i].length)
208                 players[i].index = 0;
209             activos++;
210         }
211     }
212     // Si hay 2 samples sonando, dividimos por 2 para evitar saturación.
213     // Si hay 1 sample, 'mix' queda intacto (volumen completo).
214     if (activos > 1) {
215         mix = mix >> 1; // Divide por 2 (equivale a mix /= 2)
216     }
217     // Convertir a rango SIN SIGNO (0-255) para el DAC
218     mix += 128;
219
220     if (mix > 255) mix = 255; // Saturación (seguridad)
221     if (mix < 0) mix = 0; // Aseguramos que el valor esté en el rango de 8 bits
222
223     // Convertir 8 bits (0-255) a 10 bits (0-1023) para el DAC
224     uint16_t val10bit = (uint16_t)(mix << 2);
225     DAC_UpdateValue(val10bit);
226     TIM_ClearIntPending(LPC_TIM0, TIM_MR1_INT);
227 }
228 */
229 *-----*
230 *-----*
231 * */
232 void cfgUART(void)
233 {
234     LPC_SC->PCONP |= (1 << 3);
235
236     UART_CFG_Type cfgUART0;
237     UART_FIFO_CFG_Type cfgUART0FIFO;
238
239     UART_ConfigStructInit(&cfgUART0);
240     cfgUART0.Baud_rate = 115200;
241
242     UART_Init((LPC_UART_TypeDef *)LPC_UART0, &cfgUART0);
243     UART_FIFOConfigStructInit(&cfgUART0FIFO);
244     UART_FIFOConfig((LPC_UART_TypeDef *)LPC_UART0, &cfgUART0FIFO);
245     UART_IntConfig((LPC_UART_TypeDef *)LPC_UART0, UART_INTCFG_RBR, ENABLE); // Habilita interrupt
246     UART_TxCmd((LPC_UART_TypeDef *)LPC_UART0, ENABLE);
247
248     NVIC_EnableIRQ(UART0 IRQn);
249 }
250

```

```

251 void UART0_IRQHandler(void)
252 {
253     uint8_t receivedData;
254     uint32_t iir_value;
255
256     // Leemos el registro de identificación de interrupción para saber qué la causó
257     iir_value = UART_GetIntId((LPC_UART_TypeDef *)LPC_UART0);
258
259     // Verificamos si la interrupción fue por "Receive Data Ready" (RDA)
260     if ((iir_value & UART_IIR_INTID_RDA) || (iir_value & UART_IIR_INTID_CTI))
261     {
262         receivedData = UART_ReceiveByte((LPC_UART_TypeDef *)LPC_UART0);
263
264         // UART_SendByte((LPC_UART_TypeDef *)LPC_UART0, receivedData);
265
266         switch(receivedData)
267         {
268             case '1': players[0].active = (players[0].active+1)%2; // Activo/Desactivo Sample 1
269                 if(players[0].active) players[0].index = 0;
270                 break;
271             case '2': players[1].active = (players[1].active+1)%2; // Activo/Desactivo Sample 2
272                 if(players[1].active) players[1].index = 0;
273                 break;
274             case '3': players[2].active = (players[2].active+1)%2; // Activo/Desactivo Sample 2
275                 if(players[2].active) players[2].index = 0;
276                 break;
277             case '4': players[3].active = (players[3].active+1)%2; // Activo/Desactivo Sample 2
278                 if(players[3].active) players[3].index = 0;
279                 break;
280             case '5': players[4].active = (players[4].active+1)%2; // Activo/Desactivo Sample 2
281                 if(players[4].active) players[4].index = 0;
282                 break;
283             case '6': players[5].active = (players[5].active+1)%2; // Activo/Desactivo Sample 2
284                 if(players[5].active) players[5].index = 0;
285                 break;
286             case 'd':
287                 distortion = ! distortion;
288                 break;
289             case 't':
290                 tremoloON = ! tremoloON;
291                 break;
292         }
293     }
294
295     void sendWelcomeMessage(void) {
296         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
297                     (uint8_t *)"\r\n=====\\r\\n", 35, BLOCKING);
298
299         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
300                     (uint8_t *)" 8 BITS MIXER \\r\\n", 37, BLOCKING);
301
302         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
303                     (uint8_t *)"=====\\r\\n", 35, BLOCKING);
304
305         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
306                     (uint8_t *)"[1] SAMPLE_BAT\\r\\n", 18, BLOCKING);
307
308         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
309                     (uint8_t *)"[2] SAMPLE_PUM\\r\\n", 18, BLOCKING);
310
311         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
312                     (uint8_t *)"[3] SAMPLE_BOING\\r\\n", 20, BLOCKING);
313
314         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
315                     (uint8_t *)"[4] SAMPLE_INTERF\\r\\n", 20, BLOCKING);
316
317         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
318                     (uint8_t *)"[5] SAMPLE_VIRUS\\r\\n", 20, BLOCKING);
319         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
320                     (uint8_t *)"[6] SENOIDAL_PRUEBA\\r\\n", 20, BLOCKING);
321         UART_Send((LPC_UART_TypeDef *)LPC_UART0,
322                     (uint8_t *)"\r\\nSeleccione un sample con 1, 2 o 3...\\r\\n", 41, BLOCKING);
323
324     }
325
326     /*
327     * ----- SAMPLES
328     * -----
329     */
330
331     void initSamples(void) {
332         players[0].data = sampleS1;
333         players[0].length = SAMPLE_S1_LEN;
334         players[0].index = 0;
335         players[0].active = 0;
336
337         players[1].data = sampleS5;

```

```

337     players[1].data = sampleS5;
338     players[1].length = SAMPLE_S5_LEN;
339     players[1].index = 0;
340     players[1].active = 0;
341
342     players[2].data = sampleS2;
343     players[2].length = SAMPLE_S2_LEN;
344     players[2].index = 0;
345     players[2].active = 0;
346
347     players[3].data = sampleS3;
348     players[3].length = SAMPLE_S3_LEN;
349     players[3].index = 0;
350     players[3].active = 0;
351
352     players[4].data = sampleS4;
353     players[4].length = SAMPLE_S4_LEN;
354     players[4].index = 0;
355     players[4].active = 0;
356
357     players[5].data = sampleS6;
358     players[5].length = SAMPLE_S6_LEN;
359     players[5].index = 0;
360     players[5].active = 0;
361 }
362 */
363 *-----*
364 *-----*
365 *-----*
366 void tremoloEffect(void){
367     // --- ACTUALIZACIÓN DEL LFO ---
368     if (tremoloON) {
369         if (++lfo_prescaler >= LFO_SPEED) { // Solo actualiza el LFO cada 'LFO_SPEED' muestras
370             lfo_prescaler = 0;
371             lfo_gain += lfo_dir; // Mueve la ganancia (onda triangular)
372
373             if (lfo_gain >= 256) { // Si llega al máximo (256)
374                 lfo_dir = -1;
375                 lfo_gain = 256;
376             }
377             if (lfo_gain <= 0) { // Si llega al mínimo (0)
378                 lfo_dir = 1;
379                 lfo_gain = 0;
380             }
381         }
382     } else {
383         lfo_gain = 256; // Si el efecto está apagado, ganancia = 100% (sin efecto)
384         lfo_prescaler = 0; // Resetea el prescaler
385     }
386 }
387

```

## 8.3. Lista de Materiales

### Lista de Elementos

- **LPC1769**

- **Protoboard**  $\times 2$

- **Potenciómetro**  $\times 3$

- **LM386**

- **Capacitores:**

- $100 \mu\text{F} \times 1$

- $220 \mu\text{F} \times 1$

- $100 \text{nF} \times 4$

- **Resistencias:**

- $10 \Omega$

- $220 \Omega$

- $470 \Omega$

- **LEDs**  $\times 3$

## 8.4. Hojas de Datos

### Referencias

- [1] Texas Instruments, *LM386 Low Voltage Audio Power Amplifier — Datasheet*. Disponible en: <https://www.alldatasheet.com/html-pdf/558009/TI1/LM386/168/3/LM386.html>
- [2] NXP Semiconductors, *LPC176x/5x User Manual*. Disponible en: <https://www.nxp.com/products/LPC1769FBD100>