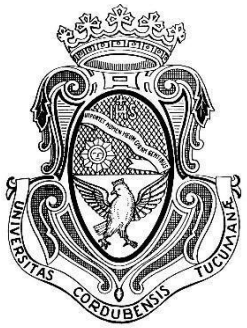


UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

CÁTEDRA DE ELECTRÓNICA DIGITAL III

TRABAJO PRÁCTICO INTEGRADOR

“AFINADOR DE GUITARRA GuitarTuna1769”

Grupo Nº 2

Alumnos:
Molina, Maria Wanda

Sabena, Maria Pilar

Verdú, Melisa Noel

Profesor:

Migliore, Emiliano

Comisión Nº 3.2

FECHA DE ENTREGA: 19/11

1. PROYECTO

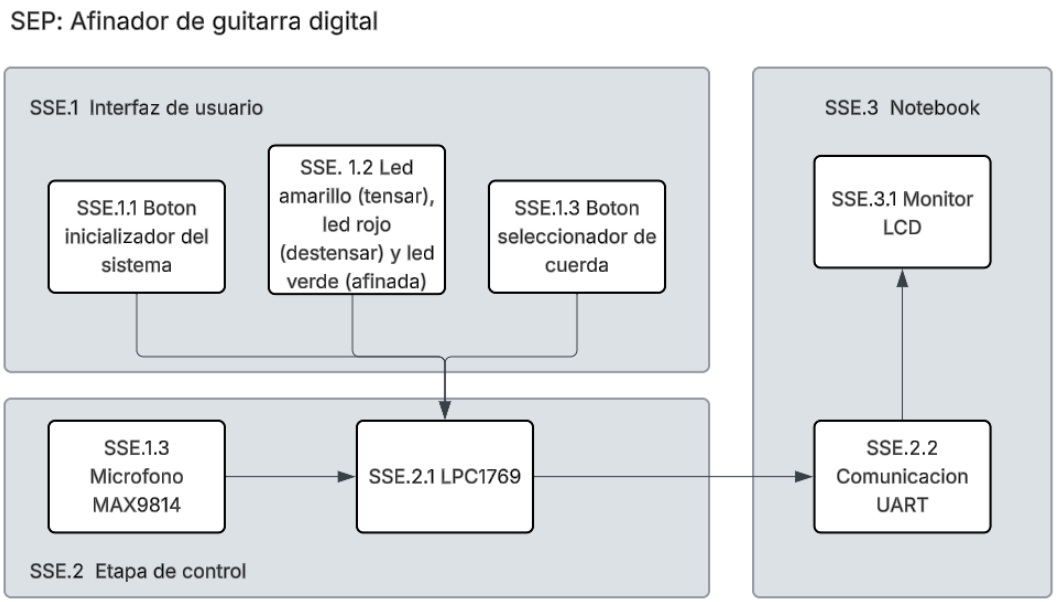
El presente proyecto tiene como objetivo diseñar y desarrollar un Sistema Electrónico Programable (SEP) basado en el microcontrolador LPC1769, capaz de detectar la frecuencia fundamental del sonido producido por una cuerda de guitarra y determinar si se encuentra afinada, o si requiere ser tensada o destensada.

El sistema permitirá analizar en tiempo real la señal acústica generada por las cuerdas de la guitarra, determinando su frecuencia fundamental para indicar al usuario si la cuerda se encuentra afinada o si debe ajustarse su tensión.

La señal acústica será captada mediante un módulo MAX9814, el cual integra un micrófono electret y un preamplificador incorporado. Esta señal se muestrea utilizando el ADC interno del microcontrolador, almacenando los valores obtenidos en la memoria SRAM para su posterior procesamiento.

El procesamiento digital incluirá la detección de cruces por cero en las muestras adquiridas, lo que permitirá calcular la frecuencia dominante del sonido y compararla con la frecuencia teórica correspondiente a la cuerda que se desea afinar.

Los resultados son enviados al usuario mediante comunicación UART, mostrando mensajes orientativos sobre el estado de afinación de la cuerda.

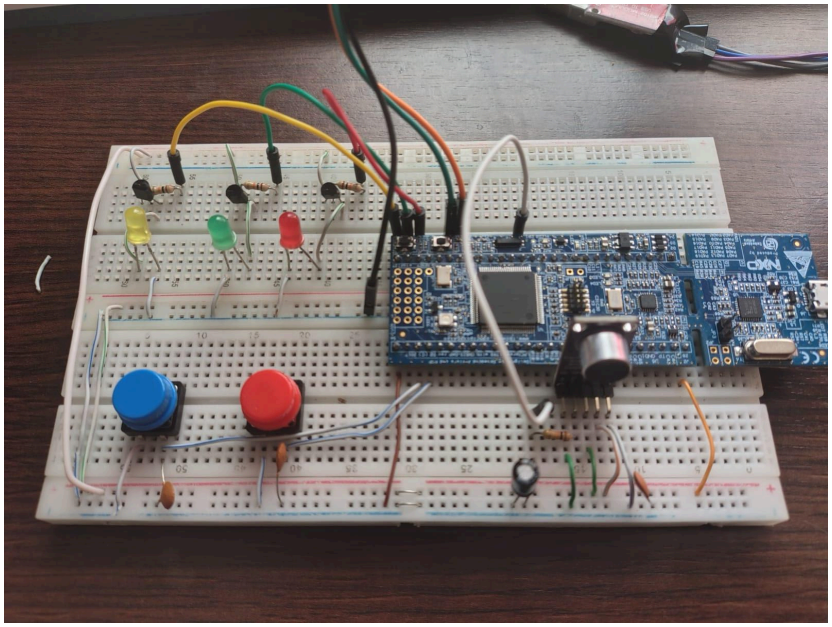


2. DESCRIPCIÓN DEL SEP

El Sistema Electrónico Programable (SEP) desarrollado se compone de un conjunto de subsistemas electrónicos (SSE) que interactúan entre sí para cumplir la función de detección, procesamiento y comunicación de los datos de afinación de cuerdas “al aire” de una guitarra criolla.

El sistema detecta la señal acústica emitida por la cuerda al ser pulsada, la convierte a una señal eléctrica mediante un micrófono, y la procesa digitalmente para obtener su frecuencia fundamental (identificación de la nota), la cuál es comparada con las teóricas.

En base a la comparación con la frecuencia teórica correspondiente a cada cuerda (E, A, D, G, B, E), el sistema determina si la cuerda está afinada o requiere ajuste.



El flujo general es el siguiente:

1. Inicio y Calibración (Interrupción Externa EINT0)

El sistema inicia en un estado de reposo o bucle de espera. La interacción comienza a través del botón conectado a EINT0 (P2.10).

Microfono MAX9814

El sistema utiliza un micrófono electret amplificado mediante el MAX9814, un preamplificador con AGC integrado. El micrófono se polariza a través del pin MICBIAS del MAX9814, y la señal se acopla mediante un capacitor al pin MICIN. El dispositivo amplifica la señal y entrega una salida centrada en 1.23 V, adecuada para ser digitalizada por el ADC de la LPC1769.

Para este proyecto se configuró el MAX9814 con la ganancia mínima (GAIN = VDD → 40 dB) para reducir ruido ambiente y evitar saturación del ADC, ya que la guitarra genera niveles acústicos suficientes para esta ganancia. El parámetro A/R se fijó en GND (ratio 1:500), lo que proporciona un ataque rápido y un release también rápido. Esta configuración es la más adecuada para instrumentos de cuerda: permite controlar los picos iniciales al atacar una cuerda y recupera la ganancia rápidamente durante el sostenido, que es la porción utilizada para calcular la frecuencia fundamental

1. Activación: Al presionar el botón, se genera una interrupción por flanco de bajada (`EINT0_IRQHandler`). Esto levanta la bandera de inicio (`start`) y activa el modo de calibración (`calibration_mode`).

2. Al arrancar, el sistema entra en modo de calibración.

3. Toma 256 muestras del micrófono (`NUM_SAMPLES_CALIBRATION`) mientras se asume que hay silencio.

4. Calcula el promedio (media) de estas muestras para determinar el "offset" de DC (el nivel de reposo del micrófono).

5. Calcula la desviación estándar (sigma) para medir el nivel de ruido ambiental. Este valor se usa como "umbral de ruido" (`noise_threshold`).

6. Si el ruido es bajo (`sigma <= SIGMA_THRESHOLD`), la calibración es exitosa y el programa pasa al modo de detección.

2.**Captura y Conversión de la Señal (ADC)**

El Timer0 está configurado para generar un evento de "match" cada 25 microsegundos (20 kHz). Cada evento del Timer0 dispara automáticamente una nueva conversión en el Canal 0 del ADC (pin P0.23), donde está conectado el micrófono. Cuando el ADC termina la conversión, genera una interrupción (`ADC_IRQHandler`).

La elección de muestrear con una frecuencia de 20kHz está dado por varias razones:

- Para capturar con precisión una señal, la frecuencia de muestreo debe ser al menos el doble de la frecuencia más alta presente $f_s \geq 2 \cdot f_{\max}$. Aunque las notas fundamentales de la guitarra solo llegan hasta 330 Hz, la calidad del sonido depende de sus armónicos, que pueden alcanzar hasta 6 kHz o más. Al elegir $f_s = 20\ \text{kHz}$, se establece la frecuencia de Nyquist en 10 kHz. Esto proporciona un margen de seguridad amplio, capturando todos los armónicos importantes sin riesgo de aliasing.

- El periodo de muestreo para 20kHz es de 50 us. Esta resolución es crucial para la detección de cruces por cero ya que permite tomar aproximadamente 60 puntos de datos por ciclo en la cuerda más aguda (330Hz). Una alta densidad de puntos de datos asegura una medición del tiempo transcurrido mucho más precisa y reduce el error de cuantificación.

- Usar frecuencias más altas aumenta la precisión marginalmente, pero generaría una gran cantidad de datos que sobrecargaron al microcontrolador (tanto al CPU como al DMA). Usan 20kHz es el punto óptimo que proporciona una calidad de señal excelente con una tasa de datos manejables para el procesamiento de filtros y cálculos de frecuencia en tiempo real.

3.**Procesamiento y Filtrado Digital (En la Interrupción)**

Dentro de la interrupción del ADC, el código lee el valor digital (`adc_value`). Se le resta el `calibration_offset` (calculado en el paso 1) para que la señal oscile alrededor del Amplifier Input BIAS, con un valor de 1,24V, según datasheet y práctica.

La señal de una guitarra es "sucia". Tiene armónicos y ruido. Si intentamos contar cruces por cero directamente, el afinador se volverá loco contando ruidos pequeños. El código implementa dos filtros en cascada usando **aritmética de punto fijo** (enteros) para ser ultra

rápido. Elimina cualquier componente DC residual o desviaciones lentas de voltaje. Solo deja pasar lo que vibra rápido (el sonido).

Filtro Pasa-Altos (HPF)

El filtro Pasa Alto se utiliza para eliminar el componente de corriente continua (**DC Offset**) de la señal. Este *offset* es el valor promedio (la calibración) que no contiene información de la vibración de la cuerda.

El filtro implementado es una versión discreta de un filtro pasa alto de primer orden, a menudo derivado de la diferencia entre la entrada y la salida de un filtro pasa bajo:

```
// High-pass filter (HPF) en forma discreta (implementado en enteros)
// hpf_output = alpha * (prev_output + x[n] - x[n-1])int32_t hpf_output = (ALPHA_COEFF * (prev_output + raw_sample_centered - prev_input)) / ALPHA_SCALER;
prev_input = raw_sample_centered; // actualizar x[n-1]
prev_output = hpf_output;         // actualizar estado del HPF
```

La fórmula implementada en el bloque de código es la siguiente:

$$y[n] = \alpha.\,(y[n-1] + x[n] - x[n-1])$$

- $y[n]$ es la salida actual del filtro (la variable `hpf_output`)
- $y[n-1]$ es la salida anterior del filtro (la variable `prev_output`).
- $x[n]$ es la muestra de entrada actual (la variable `raw_sample_centered`)
- $x[n-1]$ es la muestra de entrada anterior (la variable `prev_input`).
- α (alfa) es el coeficiente del filtro, que determina la frecuencia de corte

Suaviza la onda y elimina el "ruido blanco" de alta frecuencia o armónicos muy agudos que podrían provocar "falsos cruces por cero". Transforma la onda compleja de la guitarra en algo más parecido a una senoidal pura.

Los coeficientes utilizados en el código definen la frecuencia de corte del filtro:
`#define ALPHA_COEFF 990` $(\alpha = 0.99)$

Un valor de $\alpha.$ cercano a 1 significa que la frecuencia de corte es muy baja.

$$f_c \approx \frac{f_s}{2\pi} \cdot \frac{1-\alpha}{\alpha}$$

Teniendo en cuenta que la frecuencia de muestreo utilizada es de 20kHz:

$$f_c \approx \frac{20000\ \text{Hz}}{2\pi} \cdot \frac{1-0.99}{0.99} \approx 3200 \cdot \frac{0.01}{0.99} \approx 32\ \text{Hz}$$

Esta frecuencia es ideal para dejar filtrar ruido electrrico que se acopla a la señal recibida por el ADC.

Filtro Pasa-Bajo (LPF)

El filtro Pasa Bajo, aplicado después del HPF, tiene como objetivo **suavizar la señal** y eliminar el ruido de alta frecuencia que aún pueda estar presente después de la etapa de centrado.

```
// Low-pass / suavizado (LPF) exponencial aproximado con coeficientes enteros
// current_output = (1 - beta)*hpf_output + beta*prev_lpf_output
int32_t current_output = (ONE_MINUS_BETA_COEFF * hpf_output + BETA_COEFF *
prev_lpf_output) / BETA_SCALER;
prev_lpf_output = current_output; // actualizar estado del LPF
```

$$y[n] = (1 - \beta).x[n] + \beta.y[n-1]$$

- $y[n]$ es la salida actual del filtro (la variable `current_output`).
- $y[n-1]$es la salida anterior del filtro (la variable `prev_lpf_output`).
- $x[n]$ es la muestra de entrada actual (en tu caso, es la `hpf_output` del filtro anterior).
- β es la muestra de entrada anterior (la variable `prev_input`).

Los coeficientes utilizados en el código definen la frecuencia de corte del filtro:

`#define BETA_COEFF 700` ($\beta = 0.7$)
`#define ONE_MINUS_BETA_COEFF 300` ($1 - \beta = 0.3$)

El coeficiente β controla la "rapidez" con la que la salida sigue a la entrada. Un β más alto proporciona más suavizado (corte de frecuencia más bajo).

$$f_c \approx \frac{f_s}{2\pi} \cdot \frac{1-\beta}{\beta}$$

Sustituyendo los valores:

$$f_c \approx \frac{20000\ \text{Hz}}{2\pi} \cdot \frac{1-0.7}{0.7} \approx 3200 \cdot \frac{0.3}{0.7} \approx 1371\ \text{Hz}$$

Con esta frecuencia de corte se preservan las frecuencias fundamentales y los primeros armónicos de la cuerda. Además, atenúa significativamente el ruido de alta frecuencia, el antialiasing y los armónicos superiores que podrían provocar múltiples falsos cruces por cero.

4. Detección de Frecuencia (Cruce por Cero en Búfer)

Este paso también ocurre dentro de la función `estimate_frequency` sobre el lote de muestras ya filtradas.

1. **Comparación:** El código compara la salida final del LPF (`current_output`) con el `noise_threshold` (calculado en el paso 1).
2. **Detección de Flanco:** Se detecta la transición específica de un ciclo completo: cuando la señal cruza el cero en sentido ascendente (pasa de `current = -1` a `current = 1`).
3. **Conteo:** El algoritmo registra el índice de la **primera** (`first_cross`) y la **última** (`last_cross`) detección válida dentro del lote, y cuenta el **total de cruces** (`total_crosses`).

5. Cálculo de Período y Frecuencia (Promedio en Búfer)

Este método reemplaza el cálculo por SysTick, siendo más robusto al promediar sobre múltiples ciclos.

1. **Cálculo de Muestras:** Al final de estimate_frequency, se calcula el número total de muestras entre el primer y el último cruce (total_samples = last_cross - first_cross).

2. **Período Promedio (N):** Se calcula el período promedio *en muestras* dividiendo las muestras totales por el número de ciclos detectados: N = total_samples / (total_crosses - 1).

3. **Cálculo de Frecuencia:** La frecuencia del lote se calcula usando la tasa de muestreo (`ADC_RATE`):

$$frecuencia = \frac{TasaMuestreo(ADC_RATE)}{PeriodoPromedio(N)}$$

4. **Retorno:** La función `estimate_frequency` retorna este valor de frecuencia único, representativo de todo el lote.

6. Promediado, Selección y Comunicación (Main Loop + EINT1)

1. **Promediado Final:** La frecuencia devuelta por `estimate_frequency` (que ya es un promedio del lote) se almacena en un segundo búfer (`freq_buffer`) en el `main` loop. Se calcula un promedio (`avg_frequency`) de este búfer para obtener una lectura final **extremadamente estable**.

2. **Selección de Cuerda (EINT1):** El usuario utiliza el pulsador EINT1 (P2.11). Esta interrupción no compara nada; solo incrementa el índice `curr_string` para seleccionar la frecuencia objetivo (82Hz, 110Hz, 147Hz, etc.) del array `strings`.

3. **Comparación y Comunicación:** El `main` loop llama a `compare_frequency()`, pasándole la `avg_frequency`. Esta función:

    - Lee la frecuencia objetivo actual (ej. `strings[curr_string]`).
    - Compara la frecuencia medida (`avg_frequency`) con la objetivo.
    - Envía el diagnóstico final (ej: "CUERDA AFINADA ✓", "TENSAR (+5 Hz)") por **UART**.

Durante la validación del sistema, se observó una discrepancia sistemática y repetible entre las frecuencias teóricas de la escala musical (basadas en la afinación estándar A440) y los valores calculados por el algoritmo de detección de cruce por cero.

Este fenómeno se atribuye a la superposición de dos factores: la naturaleza armónica compleja de la cuerda de guitarra y la respuesta dinámica del módulo amplificador MAX9814. Este sensor integra un Control Automático de Ganancia (AGC) que, si bien estabiliza la amplitud de entrada al ADC, comprime el rango dinámico de la señal. Esto provoca que, durante el decaimiento de la nota, los armónicos superiores y el ruido de fondo sean amplificados, alterando sutilmente la forma de onda cerca de los puntos de cruce por cero respecto a una senoidal ideal.

Para compensar este comportamiento no lineal y garantizar una precisión funcional, se optó por un enfoque de calibración empírica. Se procedió a afinar el instrumento utilizando un afinador comercial de referencia y se realizaron series de mediciones (N=50) con el SEP. Los valores almacenados en el array de referencia (`strings[]`) corresponden al promedio estadístico de estas mediciones reales. De esta manera, el software se adapta al

comportamiento acústico real del instrumento procesado a través de la cadena de ganancia del MAX9814.

3. DESARROLLO HARDWARE

El hardware del SSEP se encuentra dividido en distintos Subsistemas Electrónicos (SSE) y un Subsistema Electrónico Programable (SSEP) central, encargado del control y coordinación del conjunto.

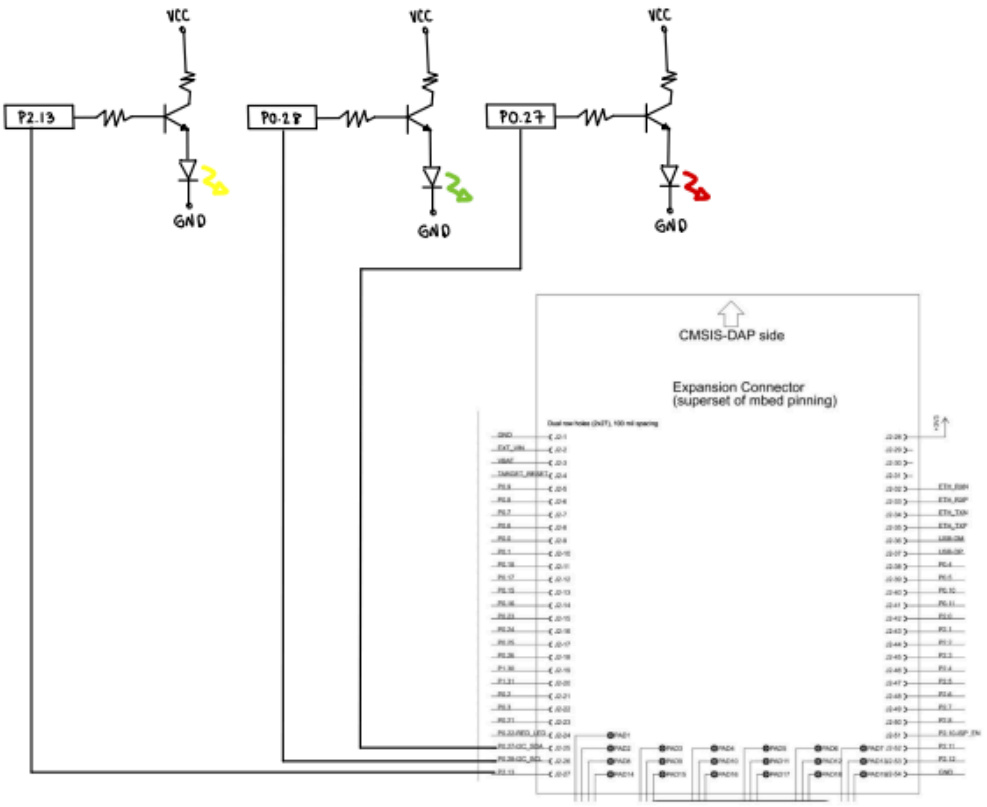
3.1 Funcionamiento bloque SSEP

El SSEP está basado en el LPC1769, un microcontrolador ARM Cortex-M3 de 32 bits que ofrece una frecuencia de operación de hasta 100 MHz, conversores ADC de 12 bits, comunicación UART, y soporte para operaciones matemáticas intensivas mediante la librería CMSIS-DSP.
Se alimenta a 3.3V y cuenta con un cristal externo de 12 MHz.
Los pines principales utilizados son, se encarga de:

- Inicializar todos los periféricos (SSE de hardware).
- Ejecutar la lógica de estado (esperar calibración, esperar DMA).
- Procesar los datos (ejecutar los SSEP de software como filtros y detección).
- Tomar decisiones (comparar la frecuencia).
- Controlar el flujo de datos entre todos los demás subsistemas.

3.2 Circuito de SSE de señalización óptica

Contamos con 3 LEDS los cuales acompañan a la señalización realizada por la visualización/Comunicación de datos (UART).



$$I_B = \frac{I_C}{\beta} \qquad I_C \approx I_{led} \qquad \therefore \quad I_B = \frac{18\,mA}{100} = 0{,}18\,mA$$

$$V_{IN} = 3{,}3\,V \qquad V_{BE} = 0{,}7\,V$$

$$\therefore \quad R_B = \frac{V_{IN} - V_{BE}}{I_B} = \frac{3{,}3V - 1{,}2V}{0{,}18mA} = 11666{,}66$$

$$R_{led} = \frac{V_{CC} - V_{led} - V_{ce}}{I_{led}} \qquad \boxed{I_C \approx I_{led} \approx 18\,mA}$$

$$\therefore \quad R_{led\,rojo} = \frac{3{,}3V - 2V - 0{,}7V}{18\,mA} = 33{,}33\,\Omega$$

$$R_{led\,verde} = \frac{3{,}3V - 2{,}2V - 0{,}7V}{18\,mA} = 22{,}22\,\Omega$$

$$R_{led\,Amarillo} = \frac{3{,}3V - 2{,}2V - 0{,}7V}{18\,mA} = 22{,}22\,\Omega$$

3.3 Circuito de SSE de señalización acústica

No se implementa en nuestro firmware, todo se delega a UART.

3.4 Circuito de SSE de visualización de datos

Como función tiene que recibir los bytes del SSE de Comunicación y mostrarlos en formato legible. Utilizamos un software de Terminal Serial en una PC para mostrar los datos.

3.5 Circuito de SSE de entrada de datos

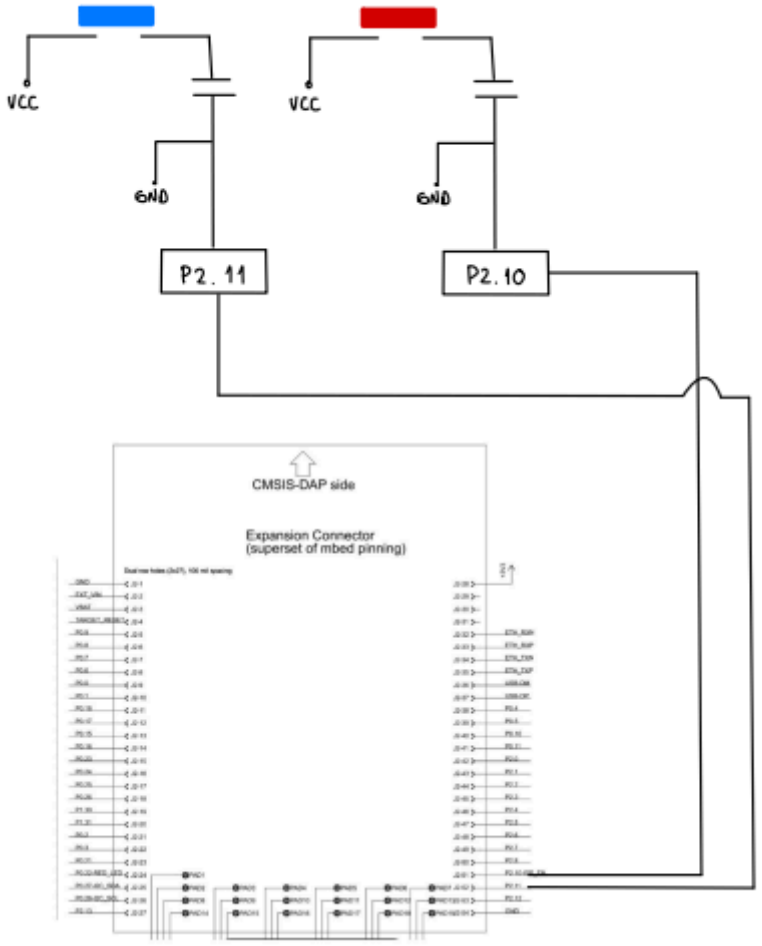
Utilizamos dos pulsadores (switches) los cuales proveen a la interfaz de usuario física iniciar el sistema y seleccionar la cuerda afinar, en un orden determinado que va desde la cuerda más aguda a la cuerda más grave.

Circuito:

1. **Pulsador 1 (Inicio/Calibrar):** Conectado entre el pin P2.10 (EINT0) y GND.
2. **Pulsador 2 (Selección):** Conectado entre el pin P2.11 (EINT1) y GND.

Ambos pulsadores contienen un capacitor de 100nF el cual evita el antirebote.

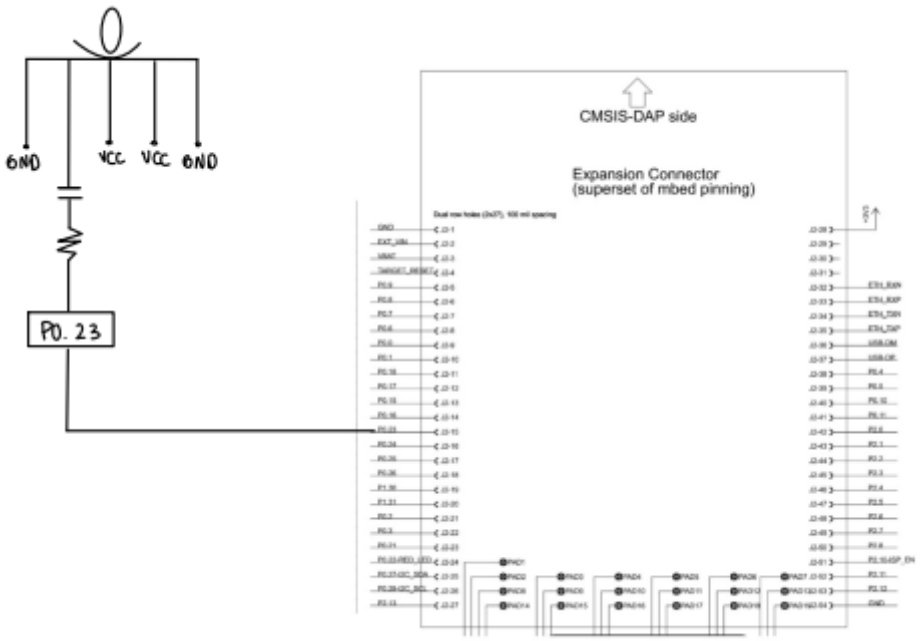
Configuración: No se requieren resistencias de pull-up externas, ya que el firmware configura las resistencias de pull-up internas del LPC1769 (`PINSEL_PINMODE_PULLUP`).



3.6 Circuito de SSE de adquisición de señales

Contamos con un micrófono **MAX9814** captura la onda sonora (señal acústica) y la convierte en una señal eléctrica analógica proporcional.

- VCC del módulo → 3.3V del LPC1769.
- GND del módulo → GND del LPC1769.
- AO (Analog Output) del módulo → Pin P0.23 (AD0.0) del LPC1769.
- GAIN → 3.3V del LPC1769 (40dB)
- A/R → GND del LPC1769 (1:500)



3.7 Circuito de SSE de comunicación de datos

Periférico **UART0** del LPC1769 y un **Conversor USB-TTL** externo.

Función: Traducir los datos paralelos del microcontrolador a un flujo de datos seriales (y viceversa) para la comunicación con el SSE de Visualización (la PC).

Circuito:

1. Pin **P0.2 (TXD0)** del LPC1769 -> Pin RX del conversor USB-TTL.
2. Pin **P0.3 (RXD0)** del LPC1769 -> Pin TX del conversor USB-TTL.
3. GND del LPC1769 -> GND del conversor.

   El conversor (ej. un módulo FTDI FT232R o CP2102) se conecta a la PC vía USB, creando un puerto COM virtual.

4. DESARROLLO DE FIRMWARE

4.1 Firmware del SSE de señalización óptica

Se implementan tres leds conectados a los pines P0.27, P0.28, y P2.13 que acompañan a la visualización de los datos enviados por UART.

4.2 Firmware del SSE de señalización acústica

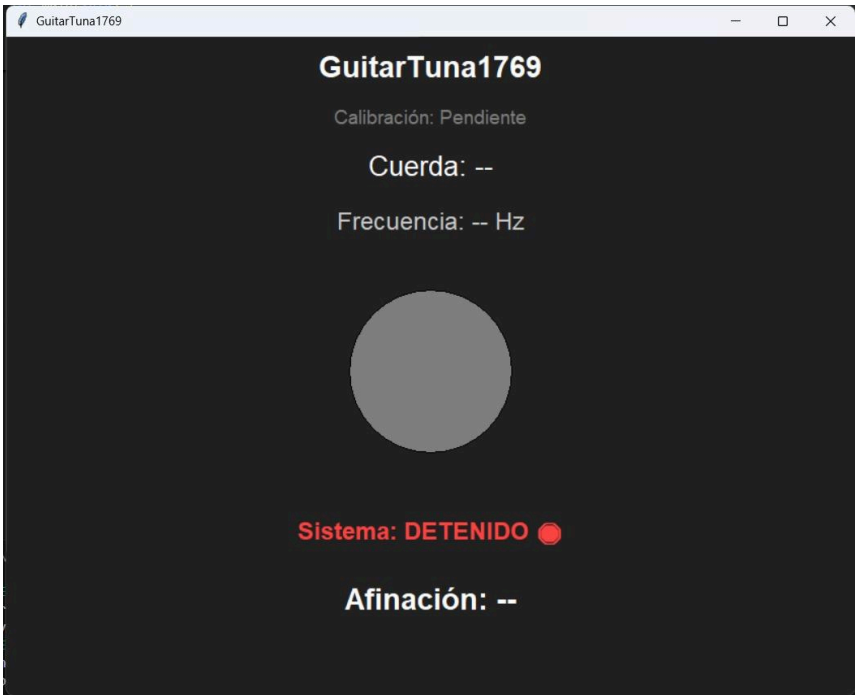
No se implementa en nuestro firmware, todo se delega a UART.

4.3 Firmware del SSE de visualización de datos

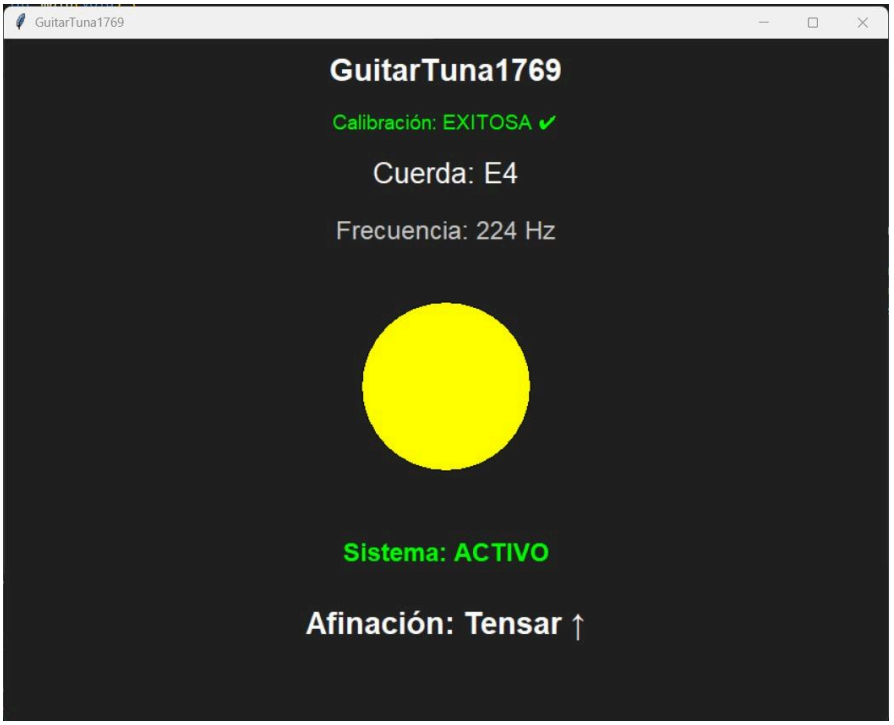
Mediante un script de Python, se levanta una GUI que permite visualizar los datos enviados mediante UART.

Sistema detenido

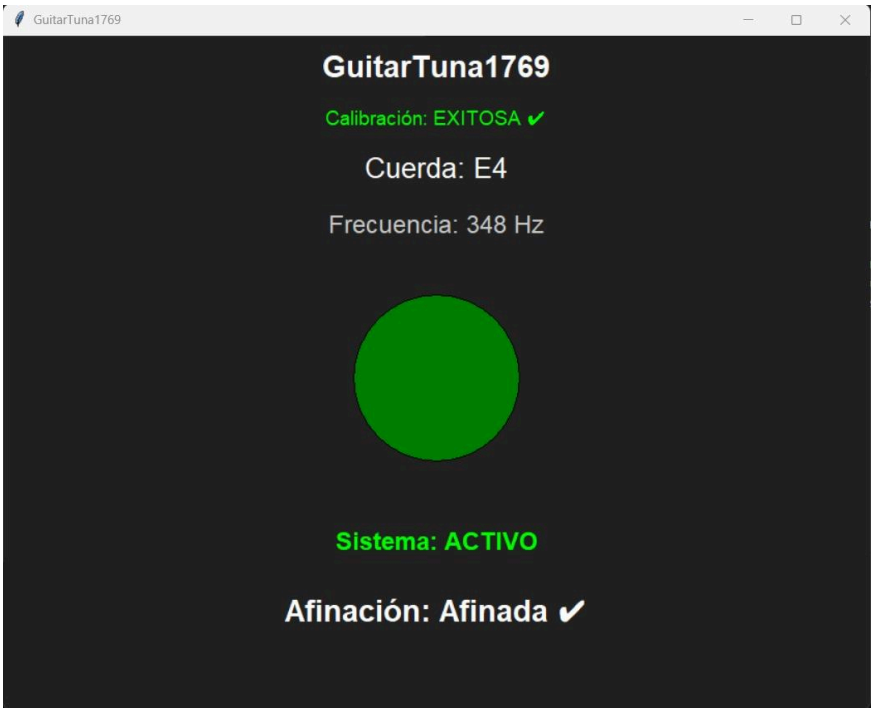
Apenas se conecta el sistema y se resetea o cuando se lo detiene con el botón rojo (EINT0).



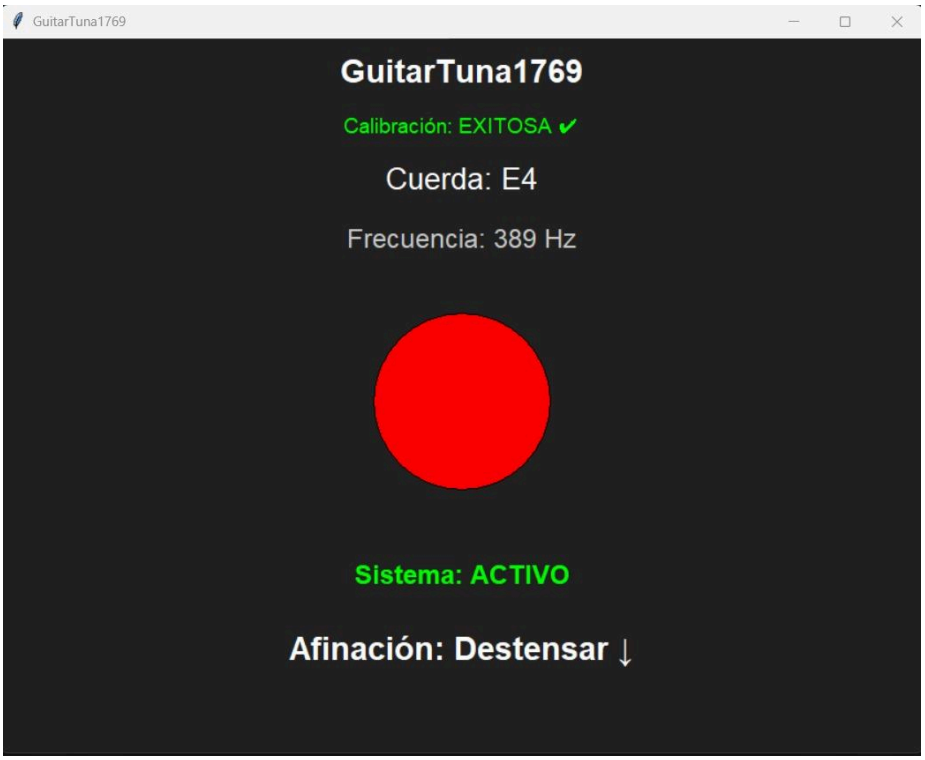
Sistema activo - Cuerda destensada



Sistema activo - Cuerda afinada

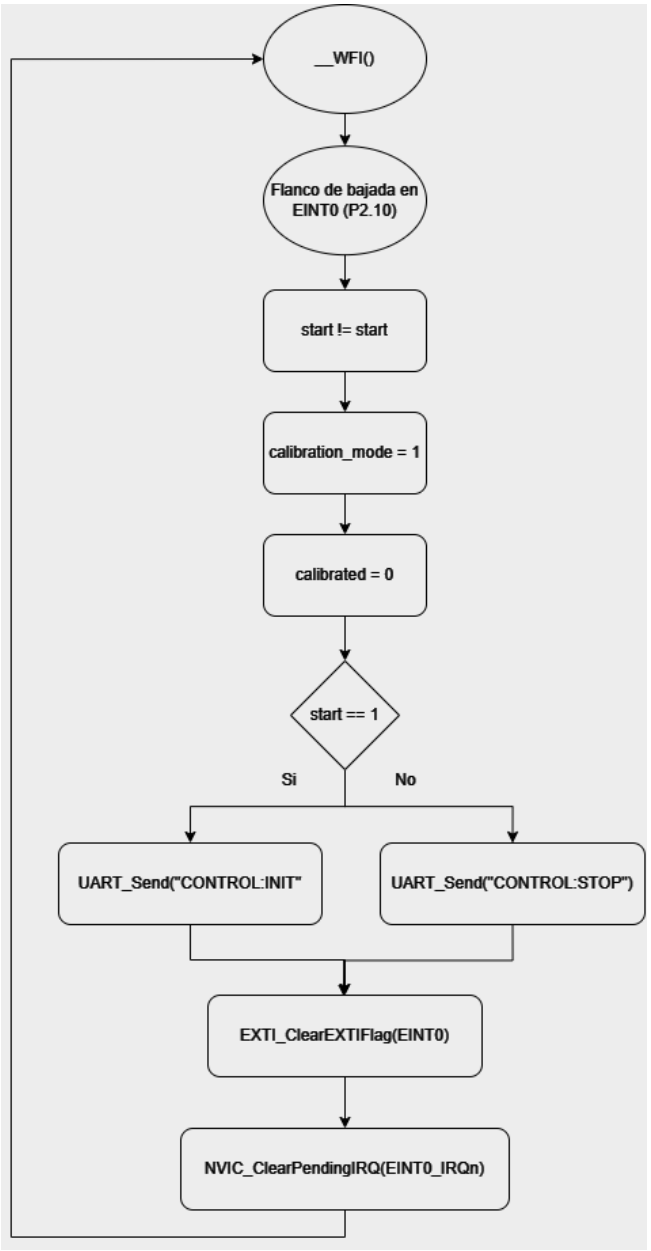


Sistema activo - Cuerda tensa



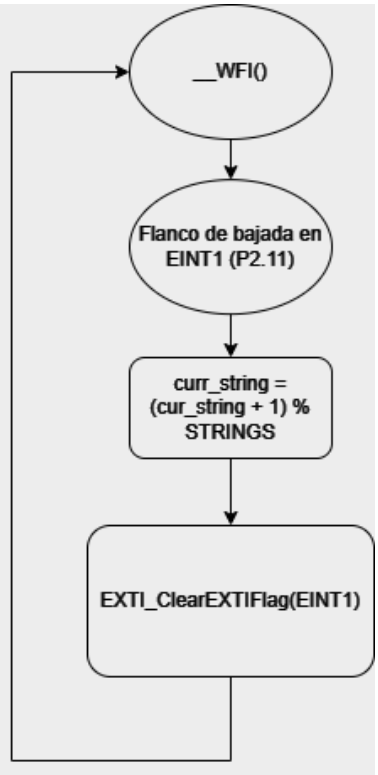
4.4 Firmware del SSE de entrada de datos

EINT0_IRQHandler()




```c
/**
 * @brief Handler de EINT0 para iniciar calibracion
 * Configura el sistema para iniciar la calibración del micrófono.
 */
void EINT0_IRQHandler(void){
    start ^= 1;             // Toggle start flag
    calibration_mode = 1;   // Entra en modo calibración
    calibrated = 0;        // Marca como descalibrado

    if(start){
        char msg_start[] = "CONTROL:INIT\r\n"; // Mensaje de inicio por UART
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)msg_start, strlen(msg_start), BLOCKING);
    }else{
        char msg_stop[] = "CONTROL:STOP\r\n"; // Mensaje de parada por UART
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)msg_stop, strlen(msg_stop), BLOCKING);
    }
    EXTI_ClearEXTIFlag(EXTI_EINT0);
    NVIC_ClearPendingIRQ(EINT0_IRQn);
}
```

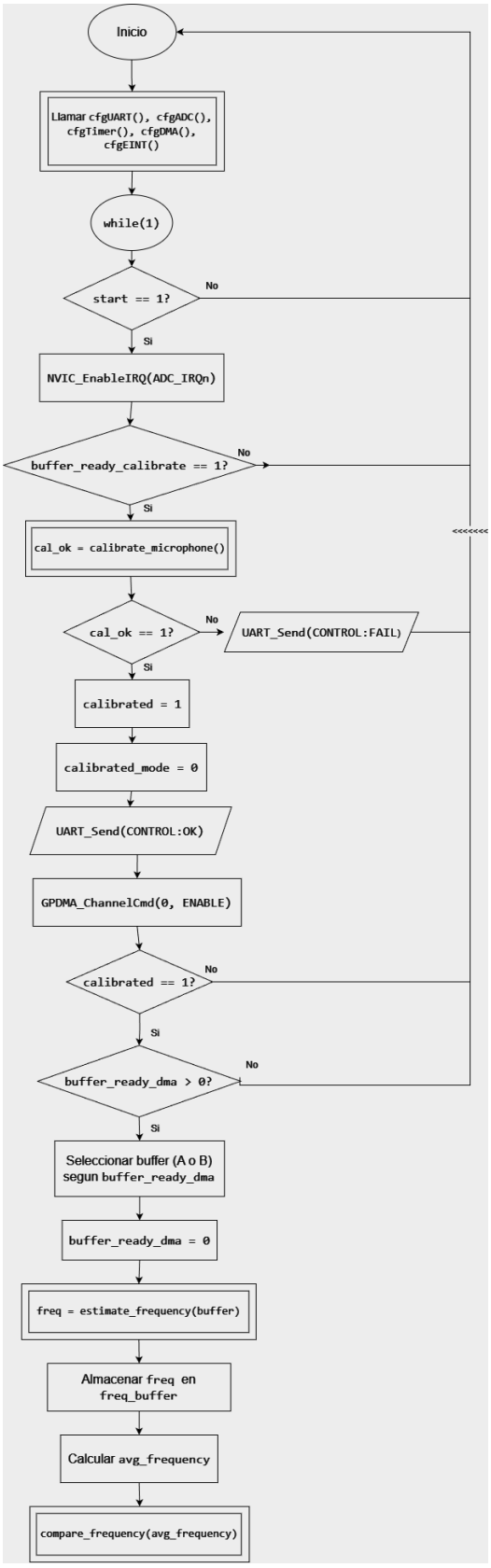
EINT1_IRQHandler()




```
/**
 * @brief Handler de EINT0 para iniciar calibracion
 * Cambia la cuerda actual a afinar.
 */
void EINT1_IRQHandler(void){
    curr_string = (curr_string + 1)%STRINGS;  // Cambia a la siguiente cuerda

    EXTI_ClearEXTIFlag(EXTI_EINT1);
    NVIC_ClearPendingIRQ(EINT1_IRQn);
}
```

4.5 Firmware del SSE de adquisición de señales

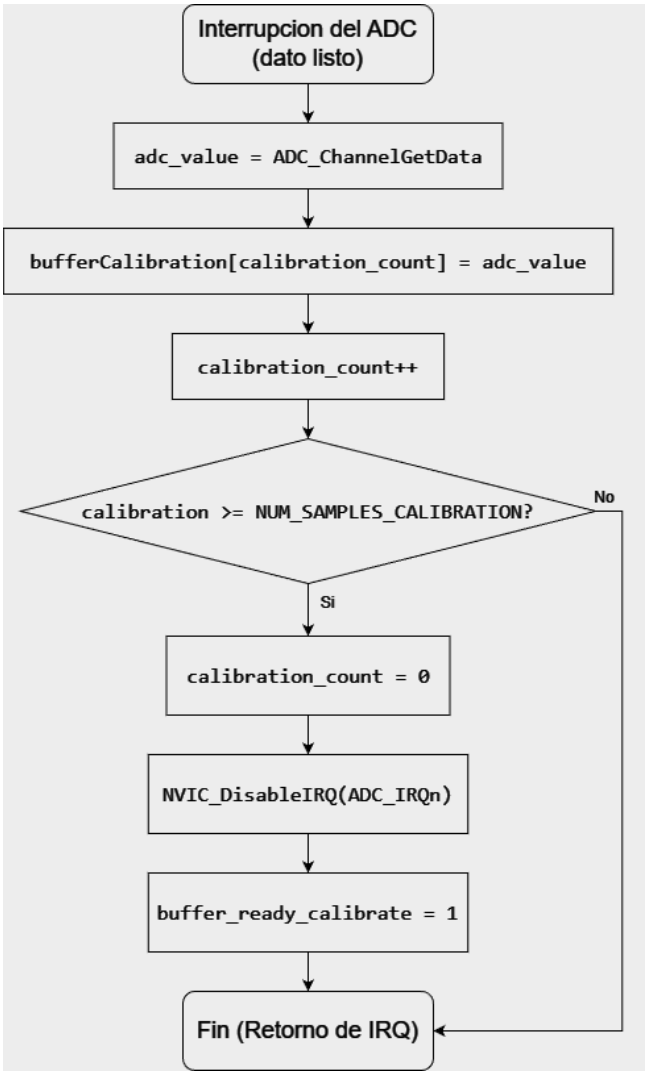
main() Lógica principal para adquisición de señales



```c
/**
* @brief Función principal
* Configura periféricos y entra en loop principal
*/
int main(void) {
    SystemInit();       // Configuración inicial del sistema
    cfgUART();          // Configura UART
    cfgADC();           // Configura ADC
    cfgTimer();         // Configura Timer
    cfgDMA();           // Configura DMA
    cfgEINT();          // Configura EINT
    cfgGPIO();           // Configura GPIO
    // Buffer circular para almacenar frecuencias recientes y promediar
    static uint32_t freq_buffer[FREQUENCY_BUFFER_SIZE];
    static uint8_t freq_idx = 0;                        // Índice del buffer circular
    static uint8_t freq_count = 0;                      // Cantidad de frecuencias almacenadas
    volatile uint32_t frequency = 0;                    // Frecuencia estimada

    while (1){
        if(start){
            // INICIO DEL SISTEMA -> DESCALIBRADO
            if(calibration_mode){
                //Entra en modo de calibracion
                NVIC_EnableIRQ(ADC_IRQn);               // Habilita interrupción ADC para calibración
                if (buffer_ready_calibrate) {           // Si el buffer de calibración está listo
                    buffer_ready_calibrate = 0;         // Resetear bandera
                    if (calibrateMicrophone()) {       // Procesar calibración
                        calibrated = 1;                 // Marcar como calibrado
                        calibration_mode = 0;           // Salir de modo calibración
                        char msg_calibrated[] = "CONTROL:OK\r\n"; // Mensaje de calibración por UART
                        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)msg_calibrated,
strlen(msg_calibrated), BLOCKING);
                        GPDMA_ChannelCmd(0, ENABLE);    // Habilitar canal DMA para muestreo continuo
                    } else {                // Si la calibración falla, permanecer en modo calibración
                          // Mensaje de fallo de calibración por UART
                        char msg_calibration_failed[] = "CONTROL:FAIL\r\n";
UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)msg_calibration_failed,
strlen(msg_calibration_failed), BLOCKING);
                        calibration_mode = 1;
                    }
                }
            }
            // Una vez calibrado el sistema, comienza a llenarse el buffer A para su posterior proceso
            if (calibrated){
                volatile uint32_t *buffer = NULL;      // Puntero al buffer listo
                if (buffer_ready_dma == 1){            // Buffer A listo
                        buffer = bufferADC_A;
                        buffer_ready_dma = 0;          // Resetear bandera
                    } else if (buffer_ready_dma == 2){ // Buffer B listo
                        buffer = bufferADC_B;
                        buffer_ready_dma = 0;
                    }
                if(buffer != NULL){
                    // Estima frecuencia con el buffer listo
                    frequency = estimateFrequency((uint32_t *)buffer);                }
                // Filtrar frecuencias plausibles antes de almacenar
                if(frequency > 80 && frequency < 400){
                    // Almacenar en buffer circular para posterior promediado
                    freq_buffer[freq_idx] = frequency;
```

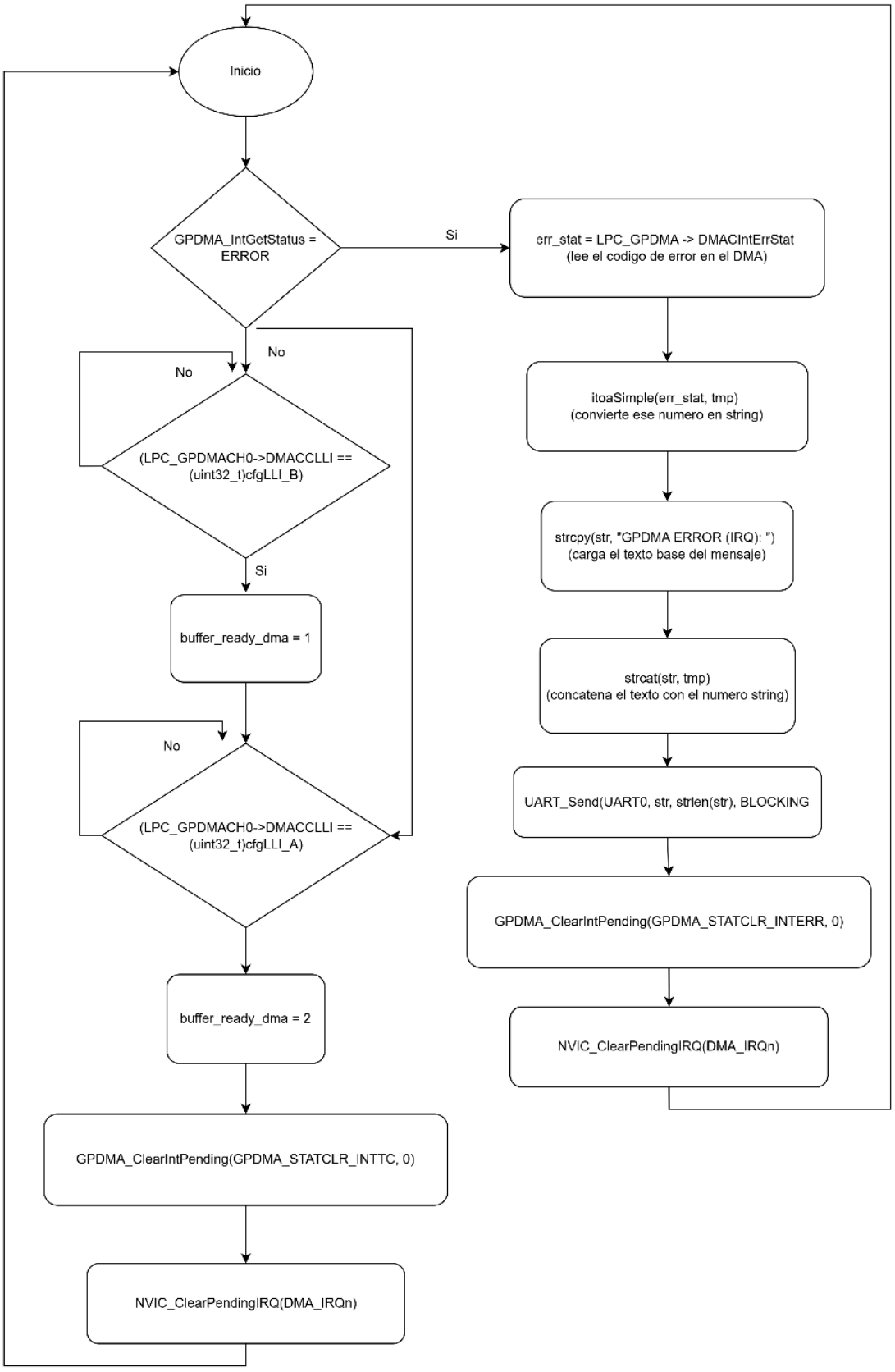
```
                    freq_idx = (freq_idx + 1) % FREQUENCY_BUFFER_SIZE;
                    if (freq_count < FREQUENCY_BUFFER_SIZE) {
                        freq_count++;
                    }
                }
                if (freq_count > 0) {
                // Calcular frecuencia promedio (reduce jitter)
                uint64_t sum_freq = 0;
                for (int i = 0; i < freq_count; i++) {
                    sum_freq += freq_buffer[i];
                }
                uint32_t avg_frequency = (uint32_t)(sum_freq / freq_count); // Frecuencia promedio
                compareFrequency(avg_frequency); // Compara con frecuencia objetivo
                }
            }
        }
    }
}
```

ADC_Handler() (usado solo para calibración)



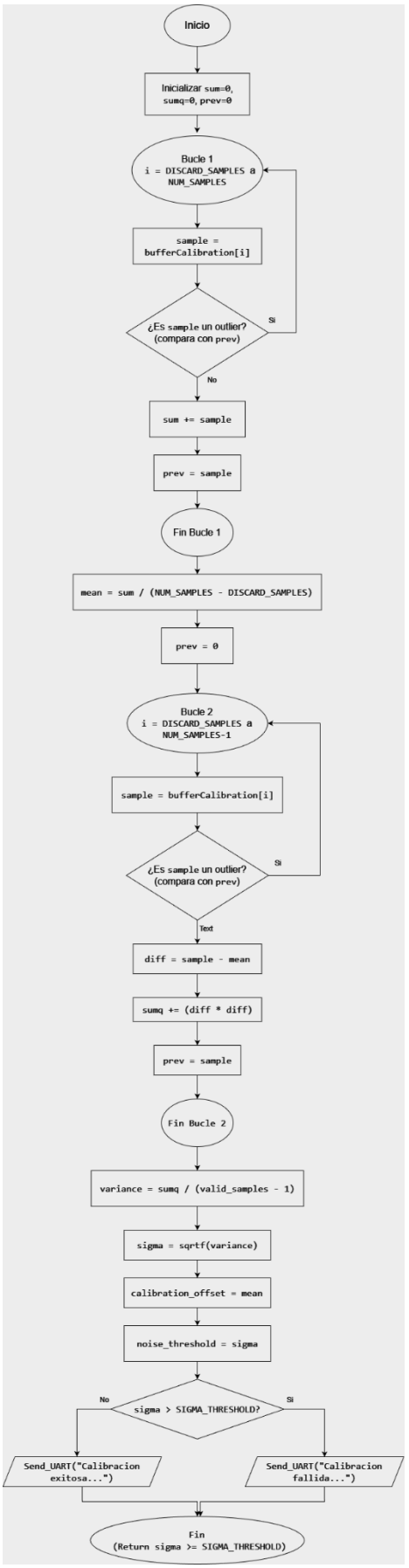

```c
/**
 * @brief Handler de interrupcion ADC
 * Maneja las interrupciones del ADC.
 * Durante la calibración, almacena muestras en el buffer de calibración.
 */
void ADC_IRQHandler(){
    if(ADC_ChannelGetStatus(LPC_ADC, 0, ADC_DATA_DONE)){
        uint32_t adc_value = ADC_ChannelGetData(LPC_ADC, 0);
        if(calibration_count < NUM_SAMPLES_CALIBRATION){
            bufferCalibration[calibration_count++] = adc_value;
            if(calibration_count >= NUM_SAMPLES_CALIBRATION){
                calibration_count = 0;
                NVIC_DisableIRQ(ADC_IRQn);
                buffer_ready_calibrate = 1; // Indica que la calibración está lista
            }
        }
    }
}
```

DMA_IRQHandler() (manejador de doble buffer)



```c
/**
 * @brief Handler de DMA
 * Maneja las interrupciones del GPDMA para el canal 0.
 * Actualiza las banderas de buffer listo según el descriptor LLI usado.
 */
void DMA_IRQHandler(void){

    // Maneja errores de DMA
    if (GPDMA_IntGetStatus(GPDMA_STAT_INTERR, 0)) {
        char str[50];
        strcpy(str, "GPDMA ERROR (IRQ): err_stat=");
        uint32_t err_stat = LPC_GPDMA->DMACIntErrStat;
        char tmp[12];
        itoaSimple(err_stat, tmp);
        strcat(str, tmp);
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)str, strlen(str), BLOCKING);

        GPDMA_ClearIntPending(GPDMA_STATCLR_INTERR, 0);
        NVIC_ClearPendingIRQ(DMA_IRQn);
        return;
    }
    // Maneja finalización de transferencia
    if (GPDMA_IntGetStatus(GPDMA_STAT_INTTC, 0)) {
        if(LPC_GPDMACH0->DMACCLLI == (uint32_t)cfgLLI_B){ // Buffer A listo
            buffer_ready_dma = 1;
        }else if(LPC_GPDMACH0->DMACCLLI == (uint32_t)cfgLLI_A){ // Buffer B listo
            buffer_ready_dma = 2;
        }

        GPDMA_ClearIntPending(GPDMA_STATCLR_INTTC, 0);
    }

    NVIC_ClearPendingIRQ(DMA_IRQn);
}
```

int calibrateMicrophone()



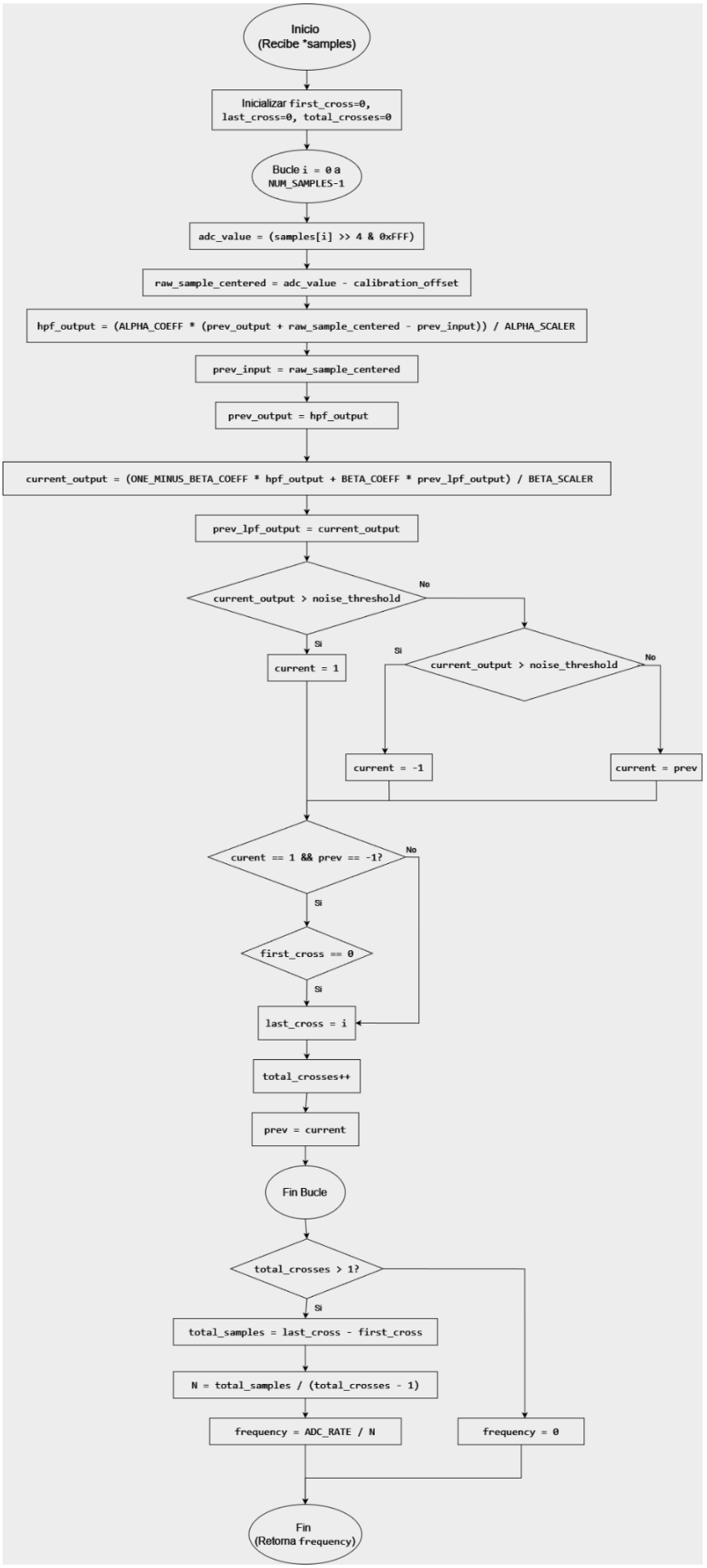
```
/* ------------------  CALIBRATION ------------------ */
/**
 * @brief Calibrar el offset del micrófono
 * Procesa las muestras almacenadas en bufferCalibration para calcular
 * el offset y el umbral de ruido (sigma). Descarta picos bruscos.
 */
int calibrateMicrophone(void) {
    uint32_t sum = 0;               // Suma de las muestras
    uint64_t sumsq = 0;             // Suma de los cuadrados de las muestras
    uint16_t sample;                // Muestra actual
    uint16_t prev = 0;              // Muestra previa para detección de outliers

    // Descartar primeras muestras
    for (int i = 0; i < DISCARD_SAMPLES; i++) {
        (void)bufferCalibration[i];
    }

    // Procesar muestras restantes
    for(int i = DISCARD_SAMPLES; i < NUM_SAMPLES_CALIBRATION; i++) {
        sample = bufferCalibration[i];
        if (i > DISCARD_SAMPLES && (sample > prev + OUTLIER_THRESHOLD || sample +
OUTLIER_THRESHOLD < prev)) {
            // ignora picos bruscos
            continue;
        }
        sum += sample; // Suma de las muestras
        prev = sample; // Actualiza la muestra previa
    }

    // Calculo de media y sigma
    int valid_samples = NUM_SAMPLES_CALIBRATION - DISCARD_SAMPLES;
    uint64_t mean = (uint64_t)sum / valid_samples;                  // Media de las muestras
    uint64_t variance = 0;                                          // Varianza de las muestras

    // Reiniciar para calcular varianza
    for(int i = DISCARD_SAMPLES; i < NUM_SAMPLES_CALIBRATION; i++) {
        sample = bufferCalibration[i];
        if (i > DISCARD_SAMPLES && (sample > prev + OUTLIER_THRESHOLD || sample +
OUTLIER_THRESHOLD < prev)) {
            // ignora picos bruscos
            continue;
        }
        int32_t diff = (int32_t)sample - (int32_t)mean; // Diferencia respecto a la media
        sumsq += ((uint64_t)diff * (uint64_t)diff); // Suma de los cuadrados de las diferencias
        prev = sample;                              // Actualiza la muestra previa
    }
    // Cálculo de varianza
    variance = sumsq / (valid_samples - 1);

    // Cálculo de desviación estándar (sigma)
    uint64_t sigma = 0;
    sigma = sqrtf((uint64_t)variance);
```

```
    calibration_offset = (uint16_t)mean; // Guarda el offset calculado
    noise_threshold = (uint16_t)sigma;  // Guarda el umbral de ruido

    char out[50];                        // Mensaje de salida por UART
    if(sigma > SIGMA_THRESHOLD) {
        strcpy(out, "Calibracion fallida: señal inestable\r\n");
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)out, strlen(out), BLOCKING);
    } else {
        strcpy(out, "Calibracion exitosa\r\n");
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)out, strlen(out), BLOCKING);
    }
    return sigma <= SIGMA_THRESHOLD; // Retorna éxito o fallo de calibración
}
```

estimateFrecuency(uint32_t *samples)

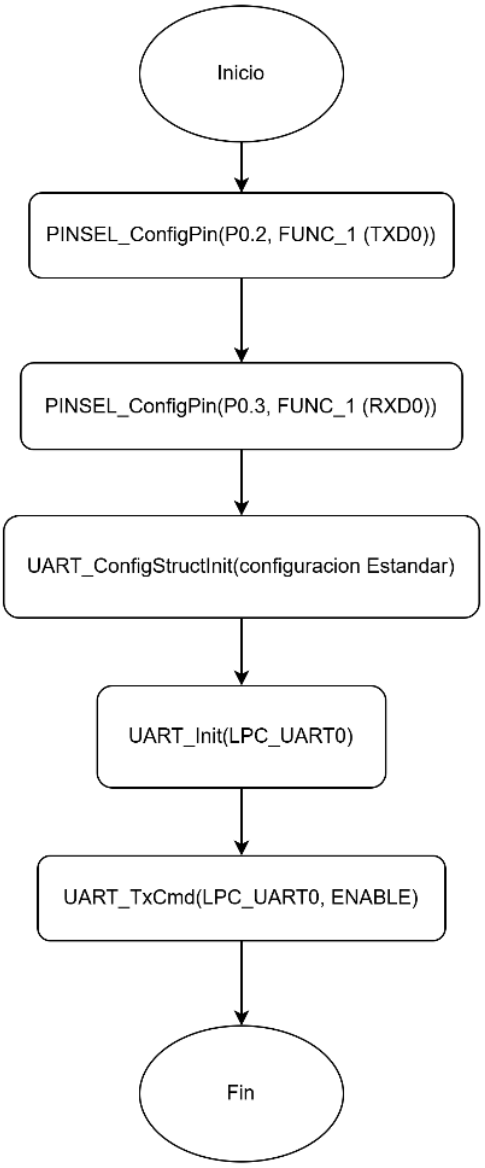
```
/* ------------------  ESTIMATE FREQUENCY  ------------------ */
/**
 * @brief Estima la frecuencia de la señal muestreada
 * @param samples Puntero al buffer de muestras ADC
 * @return Frecuencia estimada en Hz
 *
 * Procesa las muestras del buffer ADC para estimar la frecuencia
 * mediante la detección de cruces por cero en la señal filtrada.
 */
uint32_t estimateFrequency(uint32_t *samples){
    static int32_t prev_input = 0;          // Estado previo de entrada del HPF
    static int32_t prev_output = 0;         // Estado previo de salida del HPF

    static uint32_t frequency = 0;          // Frecuencia estimada

    static int current = 0;                 // Estado actual: 0 desconocido, 1 arriba, -1 abajo
    static int prev = 0;                    // Estado previo (para detectar transiciones)

    uint32_t first_cross = 0;               // Almacena el índice del primer cruce
    uint32_t last_cross = 0;                // Almacena el índice del último cruce
    uint32_t total_crosses = 0;             // Contador de cruces detectados
    uint32_t N = 0;                         // Periodo estimado en muestras

    static int32_t prev_lpf_output = 0;     // Estado previo del LPF (suavizado)

    for(int i=0; i<NUM_SAMPLES; i++){
        uint32_t adc_value = (samples[i]>>4)&0xFFF; // Extrae valor ADC de 12 bits
        // Centra la muestra respecto del offset calculado en la calibracion
        int32_t raw_sample_centered = (int32_t)adc_value - (int32_t)calibration_offset;

        // High-pass filter (HPF) - formato discreto
        // hpf_output = alpha * (prev_output + x[n] - x[n-1])
         int32_t hpf_output = (ALPHA_COEFF * (prev_output + raw_sample_centered - prev_input)) /
ALPHA_SCALER;
```

```
        prev_input = raw_sample_centered; // actualiza x[n-1]
        prev_output = hpf_output;         // actualiza el estado del HPF
        // Low-pass filter (LPF) - exponencial aproximado con coeficientes enteros
        // current_output = (1-beta)*hpf_output + beta*prev_lpf_output
               int32_t  current_output  =  (ONE_MINUS_BETA_COEFF  *  hpf_output  +  BETA_COEFF  *
prev_lpf_output) / BETA_SCALER;
        prev_lpf_output = current_output; // actualiza el estado del LPF

        // Decide si la señal esta por encima o por debajo del ruido
        if(current_output > noise_threshold){
            current = 1; // señal arriba
        }else if(current_output < -noise_threshold){
            current = -1; // señal abajo
        }else{
            // Dentro del rango de ruido -> mantener el estado previo para evitar rebotes
            current = prev;
        }
        // Detectar transicion de -1 a +1 (crece positivo) -> indica el periodo completo
        if(current == 1 && prev == -1){
            if(first_cross == 0){ // Guarda el índice del primer cruce
                first_cross = i; // primer cruce detectado
            }
            last_cross = i; // Actualiza el índice del último cruce
            total_crosses++; // Incrementa el contador de cruces
        }
        // Actualizar estado previo si hay un estado valido
        if(current != 0){
            prev = current; // Actualiza el estado previo
        }
    }
    if (total_crosses > 1) {
        // Distancia total en muestras / Número de periodos completos
        uint32_t total_samples = last_cross - first_cross;
        // N (periodo promedio) = (Total de muestras / Total de periodos completos)
        N = total_samples / (total_crosses - 1);
    } else {
        N = 0; // Menos de un ciclo completo detectado
    }
    if(N > 0){
        // Usar la tasa de muestreo adecuada (20000 o 40000)
        frequency = ADC_RATE / N; // Calcula la frecuencia estimada
    } else {
        frequency = 0; // Frecuencia no válida
    }

    return frequency; // Retorna la frecuencia estimada
}
```


Inicio
(Recibe *samples)
Inicializar first_cross=0,
last_cross=0, total_crosses=0
Bucle i = 0 a
NUM_SAMPLES-1
adc_value = (samples[i] >> 4 & 0xFFF)
raw_sample_centered = adc_value - calibration_offset
hpf_output = (ALPHA_COEFF * (prev_output + raw_sample_centered - prev_input)) / ALPHA_SCALER
prev_input = raw_sample_centered
prev_output = hpf_output
current_output = (ONE_MINUS_BETA_COEFF * hpf_output + BETA_COEFF * prev_lpf_output) / BETA_SCALER
prev_lpf_output = current_output
current_output > noise_threshold
No
Si
current = 1
Si
current_output > noise_threshold
No
current = -1
current = prev
curent == 1 && prev == -1?
No
Si
first_cross == 0
Si
last_cross = i
total_crosses++
prev = current
Fin Bucle
total_crosses > 1?
Si
total_samples = last_cross - first_cross
N = total_samples / (total_crosses - 1)
frequency = ADC_RATE / N
frequency = 0
Fin
(Retorna frequency)

4.6 Firmware del SSE de comunicación de datos

cfgUART()




```
/**
 * @brief Configuración de UART0 (TX P0.2, RX P0.3)
 * Configura UART0 para comunicación serial a 9600 baudios.
 */
void cfgUART() {
    PINSEL_CFG_Type cfgPinTXD0;
    PINSEL_CFG_Type cfgPinRXD0;

    cfgPinTXD0.Portnum = 0;
    cfgPinTXD0.Pinnum = 2;
    cfgPinTXD0.Funcnum = 1;
    cfgPinTXD0.Pinmode = PINSEL_PINMODE_NORMAL;
    cfgPinTXD0.OpenDrain = PINSEL_PINMODE_NORMAL;
```

```
    PINSEL_ConfigPin(&cfgPinTXD0);

    cfgPinRXD0.Portnum = 0;
    cfgPinRXD0.Pinnum = 3;
    cfgPinRXD0.Funcnum = 1;
    cfgPinRXD0.Pinmode = PINSEL_PINMODE_NORMAL;
    cfgPinRXD0.OpenDrain = PINSEL_PINMODE_NORMAL;
    PINSEL_ConfigPin(&cfgPinRXD0);

    UART_CFG_Type UARTConfig;
    UART_FIFO_CFG_Type FIFOConfig;

    UART_ConfigStructInit(&UARTConfig);
    UART_FIFOConfigStructInit(&FIFOConfig);

    UART_Init((LPC_UART_TypeDef *)LPC_UART0, &UARTConfig);
    UART_FIFOConfig((LPC_UART_TypeDef *)LPC_UART0, &FIFOConfig);
    UART_TxCmd((LPC_UART_TypeDef *)LPC_UART0, ENABLE);
}
```

5. PRUEBAS

Nro.	Descripción del Caso de Prueba	Paso a Seguir	Resultado Esperado	Resultado Obtenido	Observaciones
1	Prueba de Comunicación UART	1. Encender el sistema. 2. Conectar el conversor USB-TTL a la PC. 3. Abrir un terminal serial (ej. Termite) a 9600 bps.	El terminal se conecta correctamente. No se recibe ningún dato (el sistema espera el inicio).	PASO	Confirma que el microcontrolador arrancó y la UART está inicializada.
2	Prueba de Calibración Exitosa (EINT0)	1. Asegurar silencio en el micrófono. 2. Presionar el botón EINT0 (Inicio/Calibrar).	El sistema envía por UART el mensaje: "Calibracion exitosa\r\n".	PASO	Valida el ADC_IRQHandler, la lógica de calibrate_microphone() y la IRQ de EINT0.
3	Prueba de Calibración Fallida (Ruido)	1. Hacer ruido fuerte (soplar) en el micrófono. 2. Presionar el botón EINT0 (Inicio/Calibrar).	El sistema envía por UART el mensaje: "Calibración fallida: señal inestable\r\n".	PASO	Valida que el SIGMA_THRESHOLD está funcionando y rechaza calibraciones ruidosas.
4	Prueba de Selección de Cuerda (EINT1)	1. Realizar el Paso 2 (Calibrar). 2. Presionar el botón EINT1 3 veces. 3. Generar un tono de ~150 Hz.	El sistema debe mostrar: "frecuencia de comparacion: 147 Hz\r\n". (330->246->196->147).	PASO	Confirma el funcionamiento de la IRQ EINT1_IRQHandler y la lógica del array strings[curr_string].
5	Prueba de Lógica (Cuerda BAJA)	1. Calibrar. 2. Seleccionar la cuerda "La" (A2) (Presionar EINT1 4 veces -> 110 Hz). 3. Usar un generador de tonos para emitir 100 Hz.	El sistema debe reportar: "TENSAR (+10 Hz)\r\n" (o un valor cercano).	PASO	Valida el SSEP de Adquisición (DMA, DSP) y la lógica de "BAJA" en compare_frequency().
6	Prueba de Lógica (Cuerda ALTA)	1. Calibrar. 2. Seleccionar la cuerda "Sol" (G3) (Presionar EINT1 2 veces -> 196 Hz). 3. Usar un generador de tonos para emitir 205 Hz.	El sistema debe reportar: "DESTENSAR (-9 Hz)\r\n" (o un valor cercano).	PASO / FALLO	Valida la lógica de "ALTA" en compare_frequency().

6. CONCLUSIÓN

El TPI alcanzó exitosamente el objetivo de desarrollar un afinador de guitarra funcional en el LPC1769. El alcance se centró en la implementación robusta de la adquisición de datos por DMA con doble búfer, y la creación de una cadena DSP en punto fijo (HPF/LPF) para el filtrado en tiempo real. Un aprendizaje clave fue la necesidad de calibración empírica de las frecuencias, mitigando el *offset* sistemático generado por los armónicos de la guitarra y el AGC del MAX9814. Como indicación y mejora prioritaria, creemos que sería una mejor opción pasar la detección de frecuencia de cruce por cero a un algoritmo de FFT, lo que garantizaría una mayor estabilidad y una precisión espectralmente pura ante formas de onda.

7. BIBLIOGRAFÍA Y REFERENCIAS

- *NXP Semiconductors. (2010). UM10360: LPC176x/5x user manual (Rev. 3.1). NXP Semiconductors.*

- *NXP Semiconductors. (2009). LPC1769/68/67/66/65/64: 32-bit ARM Cortex-M3 microcontroller; up to 512 kB flash and 64 kB SRAM (Data Sheet). NXP Semiconductors.*

- *Maxim Integrated, MAX9814: Microphone Amplifier with AGC and Low-Noise Microphone Bias, Data Sheet, Rev. 0, Sunnyvale, CA, USA, 2007.*

- *Oppenheim, A. V., & Schafer, R. W. (1999).* ***Discrete-Time Signal Processing*** *(2nd ed.). Upper Saddle River, NJ: Prentice Hall.*

- *Proakis, J. G., & Manolakis, D. G. (2007).* ***Digital Signal Processing: Principles, Algorithms and Applications*** *(4th ed.). Upper Saddle River, NJ: Prentice Hall.*

8. ANEXO

Código

```
/**
 * @file main.c
 * @brief Afinador de guitarra para LPC1769 (AutoTune-1769)
 *
 * Programa principal que configura periféricos y orquesta el funcionamiento
 * del afinador:
 *  - ADC (canal 0, P0.23) muestreado a 20 kHz disparado por Timer0 (Match1).
 *  - GPDMA con descriptores LLI para transferencia continua de muestras a SRAM.
 *  - Timer0 para generar los eventos de muestreo.
 *  - UART0 para enviar estado/telemetría hacia una GUI.
 *  - GPIO y EXTI para control de LEDs e interacción por botones (inicio/cambio de cuerda).
 *
 * Características principales:
 *  - Calibración automática del micrófono (offset y umbral de ruido) con descarte
 *    de outliers y verificación de estabilidad.
 *  - Procesado en tiempo real: filtrado HPF + LPF, detección de cruces por cero y
 *    estimación de frecuencia por periodo medio de cruces.
 *  - Promediado de frecuencias recientes para reducir jitter y decisión de estado
 *    (OK / TENSAR / DESTENSAR) indicada por LEDs y enviada por UART.
 *
 * Requisitos: LPC1769, micrófono (ej. MAX9814), circuito de botones y LEDs.
 * Fecha: Noviembre 2024
 */

#include "LPC17xx.h"
#include "lpc17xx_adc.h"
#include "lpc17xx_uart.h"
#include "lpc17xx_timer.h"
#include "lpc17xx_gpdma.h"
#include "lpc17xx_pinsel.h"
#include "lpc17xx_exti.h"
#include "lpc17xx_gpio.h"
#include <string.h>
#include <math.h>

// Estructura para la configuración de la LLI del DMA
typedef struct {
    uint32_t SrcAddr;
    uint32_t DstAddr;
    uint32_t NextLLI;
    uint32_t Control;
} myLLI_t;

/* MACRO PARA EL MUESTREO CONTINUO DEL MICROFONO*/
#define ADC_RATE 20000                                  // Frecuencia de muestreo del ADC (20 kHz)
#define NUM_SAMPLES 2048                                // Número de muestras por buffer
#define LLI_SIZE (sizeof(myLLI_t))                      // Tamaño de una LLI
#define BUFFER_SIZE (NUM_SAMPLES * sizeof(uint32_t))    // Tamaño del buffer de muestras
```

```c
#define AHB_BASE_ADDR 0x2007C000UL                                  // Base real de la RAM AHB (32 KB)
#define AHB_BASE_ADDR2  (AHB_BASE_ADDR + LLI_SIZE + BUFFER_SIZE + 4) // Ajustar según NUM_SAMPLES

/* MACROS PARA LA CALIBRACION DEL MICROFONO */
#define NUM_SAMPLES_CALIBRATION 256                                 // Cantidad de muestras para calibración
#define SIGMA_THRESHOLD 76                                            // Umbral de sigma para considerar señal válida (~3
mV)
#define OUTLIER_THRESHOLD 50                                        // Umbral para descartar picos bruscos en la señal
#define DISCARD_SAMPLES 16

/* MACROS PARA LA ESTIMACION DE LA FRECUENCIA*/
#define ALPHA_SCALER 1000                                           // Escala para cálculos enteros
#define ALPHA_COEFF 990                                             // Coeficiente alpha = 0.99 (corte bajo ~100 Hz)
#define BETA_SCALER 1000                                            // Escala para cálculos enteros
#define BETA_COEFF 700                                              // Coeficiente Beta = 0.9 (Suavizado)
#define ONE_MINUS_BETA_COEFF 300                                    // Coeficiente 1 - Beta = 0.1
#define FREQUENCY_BUFFER_SIZE 50                                    // Tamaño del buffer circular de frecuencias

/* MACROS PARA LA COMPARACION DE FRECUENCIAS Y ESTADO DE LEDS */
#define STRINGS 6
#define FREQUENCY_THRESHOLD 10

/* MACROS DE LOS LEDS*/
#define LED_RED (1<<27)    // P0.27
#define LED_GREEN (1<<28)  // P0.28
#define LED_YELLOW (1<<13) // P1.13

/* Prototipos de funciones */
void cfgADC(void);
void cfgUART(void);
void cfgTimer(void);
void cfgDMA(void);
void cfgGPIO(void);
void cfgEINT(void);
void sendState(uint32_t frequency, uint8_t state, uint8_t string);
void itoaSimple(int, char*);
int calibrateMicrophone(void);
uint32_t estimateFrequency(uint32_t *samples);
void compareFrequency(uint32_t frequency);

myLLI_t *cfgLLI_A  = (myLLI_t *)AHB_BASE_ADDR;                      // Configuración LLI A
myLLI_t *cfgLLI_B  = (myLLI_t *)(AHB_BASE_ADDR + sizeof(myLLI_t));  // Configuración LLI B

/* Buffers */
volatile uint32_t bufferCalibration[NUM_SAMPLES_CALIBRATION]; // Buffer para calibración
volatile uint32_t *bufferADC_A  = (volatile uint32_t *)(AHB_BASE_ADDR + 2 * sizeof(myLLI_t)); // Buffer para datos ADC
volatile uint32_t *bufferADC_B    = (volatile uint32_t *)(AHB_BASE_ADDR2 + 2 * sizeof(myLLI_t) + NUM_SAMPLES *
sizeof(uint32_t)); // Buffer para datos ADC

/* Variables para la calibracion */
volatile uint16_t calibration_offset = 0;              // Offset calculado en la calibración
volatile uint16_t noise_threshold = SIGMA_THRESHOLD;   // Umbral de ruido para detección
```

```c
volatile uint16_t calibration_count = 0;                            // Contador de muestras para calibración
volatile uint8_t calibration_mode = 0;                      // Bandera para indicar modo calibración
volatile uint8_t calibrated = 0;                            // Bandera para indicar si ya se calibró

/* Flags de estado de buffers */
volatile uint8_t buffer_ready_dma = 0;          // Bandera para indicar que el buffer DMA está listo
volatile uint8_t buffer_ready_calibrate = 0;    // Bandera para indicar que el buffer de calibración está listo

/* Cuerdas */
static uint16_t strings[STRINGS] = {342, 280, 209, 165, 147, 202};  // Frecuencias objetivo de las cuerdas (Hz)
static uint8_t curr_string = 0;                                     // Índice de la cuerda actual

static uint8_t start = 0;       // Bandera para iniciar el sistema


/** ----------------  MAIN ------------------ */
/**
* @brief Función principal
* Configura periféricos y entra en loop principal
*/
int main(void) {
    SystemInit();       // Configuración inicial del sistema
    cfgUART();          // Configura UART
    cfgADC();           // Configura ADC
    cfgTimer();         // Configura Timer
    cfgDMA();           // Configura DMA
    cfgEINT();          // Configura EINT
    cfgGPIO();           // Configura GPIO

      static uint32_t freq_buffer[FREQUENCY_BUFFER_SIZE]; // Buffer circular para almacenar frecuencias recientes y
promediar
    static uint8_t freq_idx = 0;                        // Índice del buffer circular
    static uint8_t freq_count = 0;                      // Cantidad de frecuencias almacenadas
    volatile uint32_t frequency = 0;                    // Frecuencia estimada

    while (1){
        if(start){
            // INICIO DEL SISTEMA -> DESCALIBRADO
            if(calibration_mode){
                //Entra en modo de calibracion
                NVIC_EnableIRQ(ADC_IRQn);               // Habilita interrupción ADC para calibración
                if (buffer_ready_calibrate) {           // Si el buffer de calibración está listo
                    buffer_ready_calibrate = 0;         // Resetear bandera
                    if (calibrateMicrophone()) {       // Procesar calibración
                        calibrated = 1;                 // Marcar como calibrado
                        calibration_mode = 0;           // Salir de modo calibración
                        char msg_calibrated[] = "CONTROL:OK\r\n"; // Mensaje de calibración por UART
                             UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)msg_calibrated, strlen(msg_calibrated),
BLOCKING);
                        GPDMA_ChannelCmd(0, ENABLE);    // Habilitar canal DMA para muestreo continuo
                    } else {                            // Si la calibración falla, permanecer en modo calibración
                        char msg_calibration_failed[] = "CONTROL:FAIL\r\n"; // Mensaje de fallo de calibración por UART
```

```
                                        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)msg_calibration_failed,
strlen(msg_calibration_failed), BLOCKING);
                        calibration_mode = 1;
                    }
                }
            }
            // Una vez calibrado el sistema, comienza a llenarse el buffer A para su posterior proceso
            if (calibrated){
                volatile uint32_t *buffer = NULL;       // Puntero al buffer listo
                if (buffer_ready_dma == 1){             // Buffer A listo
                        buffer = bufferADC_A;
                        buffer_ready_dma = 0;           // Resetear bandera
                    } else if (buffer_ready_dma == 2){ // Buffer B listo
                        buffer = bufferADC_B;
                        buffer_ready_dma = 0;
                    }
                if(buffer != NULL){
                    frequency = estimateFrequency((uint32_t *)buffer); // Estima frecuencia con el buffer listo
                }
                // Filtrar frecuencias plausibles antes de almacenar
                if(frequency > 80 && frequency < 400){
                    // Almacenar en buffer circular para posterior promediado
                    freq_buffer[freq_idx] = frequency;
                    freq_idx = (freq_idx + 1) % FREQUENCY_BUFFER_SIZE;
                    if (freq_count < FREQUENCY_BUFFER_SIZE) {
                        freq_count++;
                    }
                }
                if (freq_count > 0) {
                // Calcular frecuencia promedio (reduce jitter)
                uint64_t sum_freq = 0;
                for (int i = 0; i < freq_count; i++) {
                    sum_freq += freq_buffer[i];
                }
                uint32_t avg_frequency = (uint32_t)(sum_freq / freq_count); // Frecuencia promedio
                compareFrequency(avg_frequency); // Compara con frecuencia objetivo
                }
            }
        }
    }
}
/** ---------------- CONFIGURACIONES ------------------ */
/**
 * @brief Configuracion del modulo GPIO para los LEDs
 * - Led Rojo: P0.27
 * - Led Verde: P0.28
 * - Led Amarillo: P2.13
 */
void cfgGPIO(void){
    PINSEL_CFG_Type cfgLed = {0};
    cfgLed.Portnum = 0;
    cfgLed.Pinnum = 27;
```

```
    cfgLed.Funcnum = 0;
    cfgLed.Pinmode = PINSEL_PINMODE_TRISTATE;
    cfgLed.OpenDrain = PINSEL_PINMODE_NORMAL;

    // Configuracion del Led Rojo
    PINSEL_ConfigPin(&cfgLed);
    GPIO_SetDir(cfgLed.Portnum, (1<<27), 1);

    // Configuracion del Led Verde
    cfgLed.Pinnum = 28;
    PINSEL_ConfigPin(&cfgLed);
    GPIO_SetDir(cfgLed.Portnum, (1<<28), 1);

    // Configuracion del Led Amarillo
    cfgLed.Portnum = 2;
    cfgLed.Pinnum = 13;
    PINSEL_ConfigPin(&cfgLed);
    GPIO_SetDir(cfgLed.Portnum, (1<<13), 1);

    // Apagar todos los LEDs al inicio
    GPIO_ClearValue(0, LED_RED);
    GPIO_ClearValue(2, LED_YELLOW);
    GPIO_ClearValue(0, LED_GREEN);
}

/**
 * @brief Configuracion del ADC canal 0 disparado por Match1 del Timer0
 * El ADC se configura para muestrear el canal 0 (P0.23) cada vez que
 * el Timer0 genera un evento de Match1 (cada 50 us). Los datos se transfieren
 * automáticamente a través del módulo GPDMA a buffers en SRAM. Muestreo a 20 kHz.
 */
void cfgADC(void){
    PINSEL_CFG_Type cfgCh0 = {0};
    cfgCh0.Portnum = 0;
    cfgCh0.Pinnum = 23;
    cfgCh0.Funcnum = 1;
    cfgCh0.Pinmode = PINSEL_PINMODE_TRISTATE;
    cfgCh0.OpenDrain = PINSEL_PINMODE_NORMAL;
    PINSEL_ConfigPin(&cfgCh0);

    ADC_Init(LPC_ADC, ADC_RATE);
    ADC_ChannelCmd(LPC_ADC, 0, ENABLE);
    ADC_BurstCmd(LPC_ADC, DISABLE);

    // Iniciar conversion ADC cuando ocurra Match1 en Timer0
    ADC_StartCmd(LPC_ADC, ADC_START_ON_MAT01);
    ADC_EdgeStartConfig(LPC_ADC, ADC_START_ON_RISING);

    // Habilitar interrupción para usar DMA
    LPC_ADC->ADINTEN = (1 << 8);
    // Prioridad 2: ADC (alta frecuencia durante calibración)
    NVIC_SetPriority(ADC_IRQn, 2);
```

```
    NVIC_DisableIRQ(ADC_IRQn);
}

/**
 * @brief Configuracion del Timer0 para generar Match1 periódico
 * El Timer0 se configura para generar un evento de Match1 cada 50 us,
 * lo que dispara una conversión ADC a 20 kHz.
 */
void cfgTimer(void) {
    TIM_DeInit(LPC_TIM0);  // Asegura que se limpia todo antes de reconfigurar

    TIM_TIMERCFG_Type cfgTimer;
    cfgTimer.PrescaleOption = TIM_PRESCALE_USVAL;
    cfgTimer.PrescaleValue = 1; //ponerlo en 1 para que cuente en us

    TIM_MATCHCFG_Type cfgMatcher;
    cfgMatcher.MatchChannel = 1;  // usar MAT1
    cfgMatcher.IntOnMatch = DISABLE;
    cfgMatcher.ResetOnMatch = ENABLE;
    cfgMatcher.StopOnMatch = DISABLE;
    cfgMatcher.ExtMatchOutputType = TIM_EXTMATCH_TOGGLE;
    cfgMatcher.MatchValue = 25;  // 100us --->ponerlo en 50 para que cuente 100us y tenga una frecuencia de 10kHz

    TIM_Init(LPC_TIM0, TIM_TIMER_MODE, &cfgTimer);
    TIM_ConfigMatch(LPC_TIM0, &cfgMatcher);
    TIM_Cmd(LPC_TIM0, ENABLE);
}

/**
 * @brief Configuracion de UART0 (TX P0.2, RX P0.3)
 * Configura UART0 para comunicación serial a 9600 baudios.
 */
void cfgUART() {
    PINSEL_CFG_Type cfgPinTXD0;
    PINSEL_CFG_Type cfgPinRXD0;

    cfgPinTXD0.Portnum = 0;
    cfgPinTXD0.Pinnum = 2;
    cfgPinTXD0.Funcnum = 1;
    cfgPinTXD0.Pinmode = PINSEL_PINMODE_NORMAL;
    cfgPinTXD0.OpenDrain = PINSEL_PINMODE_NORMAL;
    PINSEL_ConfigPin(&cfgPinTXD0);

    cfgPinRXD0.Portnum = 0;
    cfgPinRXD0.Pinnum = 3;
    cfgPinRXD0.Funcnum = 1;
    cfgPinRXD0.Pinmode = PINSEL_PINMODE_NORMAL;
    cfgPinRXD0.OpenDrain = PINSEL_PINMODE_NORMAL;
    PINSEL_ConfigPin(&cfgPinRXD0);

    UART_CFG_Type UARTConfig;
    UART_FIFO_CFG_Type FIFOConfig;
```

```
    UART_ConfigStructInit(&UARTConfig);
    UART_FIFOConfigStructInit(&FIFOConfig);

    UART_Init((LPC_UART_TypeDef *)LPC_UART0, &UARTConfig);
    UART_FIFOConfig((LPC_UART_TypeDef *)LPC_UART0, &FIFOConfig);
    UART_TxCmd((LPC_UART_TypeDef *)LPC_UART0, ENABLE);
}

/**
 * @brief Configuracion del modulo GPDMA
 * Configura el canal 0 del GPDMA para transferir datos del ADC
 * a buffers en SRAM usando descriptores LLI para muestreo continuo.
 */
void cfgDMA(void){
    GPDMA_Channel_CFG_Type cfgDMA;

    // Inicializa DMA (resetea lo necesario)
    GPDMA_Init();

    // Limpia flags globales antes de configurar
    LPC_GPDMA->DMACIntTCClear = 0xFF;
    LPC_GPDMA->DMACIntErrClr = 0xFF;

    cfgLLI_A->SrcAddr = (uint32_t)&LPC_ADC->ADGDR;
    cfgLLI_A->DstAddr = (uint32_t)bufferADC_A;
    cfgLLI_A->NextLLI = (uint32_t)cfgLLI_B; // apunta al propio descriptor
    cfgLLI_A->Control = (NUM_SAMPLES & 0xFFF)    // transfer size
                      | (2 << 18)               // src width = word (32 bits)
                      | (2 << 21)               // dst width = word
                      | (1 << 27)               // dst increment
                      | (1UL << 31);            // enable terminal-count interrupt

    cfgLLI_B->SrcAddr = (uint32_t)&LPC_ADC->ADGDR;
    cfgLLI_B->DstAddr = (uint32_t)bufferADC_B;
    cfgLLI_B->NextLLI = (uint32_t)cfgLLI_A; // apunta al propio descriptor
    cfgLLI_B->Control = (NUM_SAMPLES & 0xFFF)    // transfer size
                      | (2 << 18)               // src width = word (32 bits)
                      | (2 << 21)               // dst width = word
                      | (1 << 27)               // dst increment
                      | (1UL << 31);            // enable terminal-count interrupt

    cfgDMA.ChannelNum = 0;
    cfgDMA.TransferSize = NUM_SAMPLES;
    cfgDMA.TransferType = GPDMA_TRANSFERTYPE_P2M;
    cfgDMA.SrcMemAddr = 0;
    cfgDMA.DstMemAddr = (uint32_t)bufferADC_A;
    cfgDMA.SrcConn = GPDMA_CONN_ADC;
    cfgDMA.DstConn = 0;
    cfgDMA.DMALLI = (uint32_t)cfgLLI_A; // apunta al descriptor en AHB

    GPDMA_Setup(&cfgDMA);
```

```
    LPC_GPDMA->DMACIntTCClear = (1 << cfgDMA.ChannelNum);
    LPC_GPDMA->DMACIntErrClr = (1 << cfgDMA.ChannelNum);
    // Prioridad 1: DMA (la más alta para el flujo de datos)
    NVIC_SetPriority(DMA_IRQn, 1);
    NVIC_EnableIRQ(DMA_IRQn);
}

/**
 * @brief Configuracion de EINT0 para iniciar calibracion
 * Configura los pines y las interrupciones externas EINT0 y EINT1.
 * - EINT0 (P2.10) inicia eL sistema, desde el modo de calibracion.
 * - EINT1 (P2.11) cambia la cuerda actual a afinar.
 * Ambas interrupciones se disparan con flanco de bajada.
 */
void cfgEINT(void){
    PINSEL_CFG_Type pinEINT = {0};
    // Configura P2.10 como EINT0
    pinEINT.Portnum = 2;
    pinEINT.Pinnum = 10;
    pinEINT.Funcnum = 1;
    pinEINT.Pinmode = PINSEL_PINMODE_PULLUP;
    pinEINT.OpenDrain = PINSEL_PINMODE_NORMAL;
    PINSEL_ConfigPin(&pinEINT);

    EXTI_SetMode(EXTI_EINT0, EXTI_MODE_EDGE_SENSITIVE);
    EXTI_SetPolarity(EXTI_EINT0, EXTI_POLARITY_LOW_ACTIVE_OR_FALLING_EDGE); // Flanco de bajada
    EXTI_ClearEXTIFlag(EXTI_EINT0);

    // Configura P2.11 como EINT1
    pinEINT.Pinnum = 11;
    PINSEL_ConfigPin(&pinEINT);

    EXTI_SetMode(EXTI_EINT1, EXTI_MODE_EDGE_SENSITIVE);
    EXTI_SetPolarity(EXTI_EINT1, EXTI_POLARITY_LOW_ACTIVE_OR_FALLING_EDGE);
    EXTI_ClearEXTIFlag(EXTI_EINT1);

    // Prioridad 3: EINT0 y EINT1 (eventos de usuario)
    NVIC_SetPriority(EINT0_IRQn, 3);
    NVIC_SetPriority(EINT1_IRQn, 4);
    NVIC_EnableIRQ(EINT0_IRQn);
    NVIC_EnableIRQ(EINT1_IRQn);
}

/** ----------------  HANDLERS ------------------ */

/**
 * @brief Handler de DMA
 * Maneja las interrupciones del GPDMA para el canal 0.
 * Actualiza las banderas de buffer listo según el descriptor LLI usado.
 */
void DMA_IRQHandler(void){
```

```
    // Maneja errores de DMA
    if (GPDMA_IntGetStatus(GPDMA_STAT_INTERR, 0)) {
        char str[50];
        strcpy(str, "GPDMA ERROR (IRQ): err_stat=");
        uint32_t err_stat = LPC_GPDMA->DMACIntErrStat;
        char tmp[12];
        itoaSimple(err_stat, tmp);
        strcat(str, tmp);
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)str, strlen(str), BLOCKING);

        GPDMA_ClearIntPending(GPDMA_STATCLR_INTERR, 0);
        NVIC_ClearPendingIRQ(DMA_IRQn);
        return;
    }
    // Maneja finalización de transferencia
    if (GPDMA_IntGetStatus(GPDMA_STAT_INTTC, 0)) {
        if(LPC_GPDMACH0->DMACCLLI == (uint32_t)cfgLLI_B){ // Buffer A listo
            buffer_ready_dma = 1;
        }else if(LPC_GPDMACH0->DMACCLLI == (uint32_t)cfgLLI_A){ // Buffer B listo
            buffer_ready_dma = 2;
        }

        GPDMA_ClearIntPending(GPDMA_STATCLR_INTTC, 0);
    }

    NVIC_ClearPendingIRQ(DMA_IRQn);
}

/**
 * @brief Handler de EINT0 para iniciar calibracion
 * Configura el sistema para iniciar la calibración del micrófono.
 */
void EINT0_IRQHandler(void){
    start ^= 1;             // Toggle start flag
    calibration_mode = 1;   // Entra en modo calibración
    calibrated = 0;        // Marca como descalibrado

    if(start){
        char msg_start[] = "CONTROL:INIT\r\n"; // Mensaje de inicio por UART
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)msg_start, strlen(msg_start), BLOCKING);
    }else{
        char msg_stop[] = "CONTROL:STOP\r\n"; // Mensaje de parada por UART
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)msg_stop, strlen(msg_stop), BLOCKING);
    }
    EXTI_ClearEXTIFlag(EXTI_EINT0);
    NVIC_ClearPendingIRQ(EINT0_IRQn);
}

/**
 * @brief Handler de EINT0 para iniciar calibracion
 * Cambia la cuerda actual a afinar.
 */
```

```c
void EINT1_IRQHandler(void){
    curr_string = (curr_string + 1)%STRINGS;  // Cambia a la siguiente cuerda

    EXTI_ClearEXTIFlag(EXTI_EINT1);
    NVIC_ClearPendingIRQ(EINT1_IRQn);
}

/**
 * @brief Handler de interrupcion ADC
 * Maneja las interrupciones del ADC.
 * Durante la calibración, almacena muestras en el buffer de calibración.
 */
void ADC_IRQHandler(){
    if(ADC_ChannelGetStatus(LPC_ADC, 0, ADC_DATA_DONE)){
        uint32_t adc_value = ADC_ChannelGetData(LPC_ADC, 0);
        if(calibration_count < NUM_SAMPLES_CALIBRATION){
            bufferCalibration[calibration_count++] = adc_value;
            if(calibration_count >= NUM_SAMPLES_CALIBRATION){
                calibration_count = 0;
                NVIC_DisableIRQ(ADC_IRQn);
                buffer_ready_calibrate = 1; // Indica que la calibración está lista
            }
        }
    }
}

/* ------------------  CALIBRATION ------------------ */
/**
 * @brief Calibrar el offset del micrófono
 * Procesa las muestras almacenadas en bufferCalibration para calcular
 * el offset y el umbral de ruido (sigma). Descarta picos bruscos.
 */
int calibrateMicrophone(void) {
    uint32_t sum = 0;               // Suma de las muestras
    uint64_t sumsq = 0;             // Suma de los cuadrados de las muestras
    uint16_t sample;                // Muestra actual
    uint16_t prev = 0;              // Muestra previa para detección de outliers

    // Descartar primeras muestras
    for (int i = 0; i < DISCARD_SAMPLES; i++) {
        (void)bufferCalibration[i];
    }

    // Procesar muestras restantes
    for(int i = DISCARD_SAMPLES; i < NUM_SAMPLES_CALIBRATION; i++) {
        sample = bufferCalibration[i];
        if (i > DISCARD_SAMPLES && (sample > prev + OUTLIER_THRESHOLD || sample + OUTLIER_THRESHOLD < prev)) {
            // ignora picos bruscos
            continue;
        }
        sum += sample; // Suma de las muestras
        prev = sample; // Actualiza la muestra previa
```

```
    }

    // Calculo de media y sigma
    int valid_samples = NUM_SAMPLES_CALIBRATION - DISCARD_SAMPLES; // Muestras válidas consideradas
    uint64_t mean = (uint64_t)sum / valid_samples;                 // Media de las muestras
    uint64_t variance = 0;                                         // Varianza de las muestras

    // Reiniciar para calcular varianza
    for(int i = DISCARD_SAMPLES; i < NUM_SAMPLES_CALIBRATION; i++) {
        sample = bufferCalibration[i];
        if (i > DISCARD_SAMPLES && (sample > prev + OUTLIER_THRESHOLD || sample + OUTLIER_THRESHOLD < prev)) {
            // ignora picos bruscos
            continue;
        }
        int32_t diff = (int32_t)sample - (int32_t)mean; // Diferencia respecto a la media
        sumsq += ((uint64_t)diff * (uint64_t)diff);     // Suma de los cuadrados de las diferencias
        prev = sample;                                  // Actualiza la muestra previa
    }
    // Cálculo de varianza
    variance = sumsq / (valid_samples - 1);

    // Calculo de desviación estándar (sigma)
    uint64_t sigma = 0;
    sigma = sqrtf((uint64_t)variance);

    calibration_offset = (uint16_t)mean; // Guarda el offset calculado
    noise_threshold = (uint16_t)sigma;  // Guarda el umbral de ruido

    char out[50];                        // Mensaje de salida por UART
    if(sigma > SIGMA_THRESHOLD) {
        strcpy(out, "Calibracion fallida: señal inestable\r\n");
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)out, strlen(out), BLOCKING);
    } else {
        strcpy(out, "Calibracion exitosa\r\n");
        UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t *)out, strlen(out), BLOCKING);
    }
    return sigma <= SIGMA_THRESHOLD; // Retorna éxito o fallo de calibración
}

/* ------------------  ESTIMATE FREQUENCY  ------------------ */
/**
 * @brief Estima la frecuencia de la señal muestreada
 * @param samples Puntero al buffer de muestras ADC
 * @return Frecuencia estimada en Hz
 *
 * Procesa las muestras del buffer ADC para estimar la frecuencia
 * mediante la detección de cruces por cero en la señal filtrada.
 */
uint32_t estimateFrequency(uint32_t *samples){
    static int32_t prev_input = 0;          // Estado previo de entrada del HPF
    static int32_t prev_output = 0;         // Estado previo de salida del HPF
```

```
static uint32_t frequency = 0;          // Frecuencia estimada

static int current = 0;                 // Estado actual: 0 desconocido, 1 arriba, -1 abajo
static int prev = 0;                    // Estado previo (para detectar transiciones)

uint32_t first_cross = 0;               // Almacena el índice del primer cruce
uint32_t last_cross = 0;                // Almacena el índice del último cruce
uint32_t total_crosses = 0;             // Contador de cruces detectados
uint32_t N = 0;                         // Periodo estimado en muestras

static int32_t prev_lpf_output = 0;     // Estado previo del LPF (suavizado)

for(int i=0; i<NUM_SAMPLES; i++){

    uint32_t adc_value = (samples[i]>>4)&0xFFF; // Extrae valor ADC de 12 bits

    // Centra la muestra respecto del offset calculado en la calibracion
    int32_t raw_sample_centered = (int32_t)adc_value - (int32_t)calibration_offset;

    // High-pass filter (HPF) - formato discreto
    // hpf_output = alpha * (prev_output + x[n] - x[n-1])
    int32_t hpf_output = (ALPHA_COEFF * (prev_output + raw_sample_centered - prev_input)) / ALPHA_SCALER;
    prev_input = raw_sample_centered; // actualiza x[n-1]
    prev_output = hpf_output;         // actualiza el estado del HPF

    // Low-pass filter (LPF) - exponencial aproximado con coeficientes enteros
    // current_output = (1-beta)*hpf_output + beta*prev_lpf_output
    int32_t current_output = (ONE_MINUS_BETA_COEFF * hpf_output + BETA_COEFF * prev_lpf_output) / BETA_SCALER;
    prev_lpf_output = current_output; // actualiza el estado del LPF

    // Decide si la señal esta por encima o por debajo del ruido
    if(current_output > noise_threshold){
        current = 1; // señal arriba
    }else if(current_output < -noise_threshold){
        current = -1; // señal abajo
    }else{
        // Dentro del rango de ruido -> mantener el estado previo para evitar rebotes
        current = prev;
    }

    // Detectar transicion de -1 a +1 (crece positivo) -> indica el periodo completo
    if(current == 1 && prev == -1){
        if(first_cross == 0){ // Guarda el índice del primer cruce
            first_cross = i; // primer cruce detectado
        }
        last_cross = i; // Actualiza el índice del último cruce
        total_crosses++; // Incrementa el contador de cruces
    }
    // Actualizar estado previo si hay un estado valido
    if(current != 0){
        prev = current; // Actualiza el estado previo
    }
```

```
    }

    if (total_crosses > 1) {
        // Distancia total en muestras / Número de periodos completos
        uint32_t total_samples = last_cross - first_cross;
        // N (periodo promedio) = (Total de muestras / Total de periodos completos)
        N = total_samples / (total_crosses - 1);
    } else {
        N = 0; // Menos de un ciclo completo detectado
    }

    if(N > 0){
        // Usar la tasa de muestreo adecuada (20000 o 40000)
        frequency = ADC_RATE / N; // Calcula la frecuencia estimada
    } else {
        frequency = 0; // Frecuencia no válida
    }

    return frequency; // Retorna la frecuencia estimada
}

/* ------------------  COMPARE FREQUENCY  ------------------ */
/**
 * @brief Compara la frecuencia estimada con la frecuencia objetivo
 * y enciende los LEDs correspondientes según el estado.
  Además, envía el estado por UART.
  - LED Verde: Afinado (OK)
  - LED Amarillo: Tensar
  - LED Rojo: Destensar
 * @param f Frecuencia estimada en Hz
 */
void compareFrequency(uint32_t f){
    uint16_t frequency = (uint16_t)strings[curr_string];

    // Compara con frecuencia objetivo
    if ((f > frequency - FREQUENCY_THRESHOLD) && (f < frequency + FREQUENCY_THRESHOLD)){
        sendState(f, 0, curr_string); // Enviar estado OK

        // Encender LED verde, apagar los demás
        GPIO_SetValue(0, LED_GREEN);
        GPIO_ClearValue(0, LED_RED);
        GPIO_ClearValue(2, LED_YELLOW);
    }
    else if (f < frequency + FREQUENCY_THRESHOLD){
        sendState(f, 1, curr_string); // Enviar estado TENSAR

        // Encender LED amarillo, apagar los demás
        GPIO_SetValue(2, LED_YELLOW);
        GPIO_ClearValue(0, LED_RED);
        GPIO_ClearValue(0, LED_GREEN);
    }
    else {
```

```
        sendState(f, 2, curr_string); // Enviar estado DESTENSAR

        // Encender LED rojo, apagar los demás
        GPIO_SetValue(0, LED_RED);
        GPIO_ClearValue(0, LED_GREEN);
        GPIO_ClearValue(2, LED_YELLOW);
    }
}

/** -----------------  UTILITIES ------------------- */
/**
 * @brief Convierte un entero a string (itoa simple)
 * @param n Entero a convertir
 * @param s Cadena donde se almacena el resultado
 */
void itoaSimple(int n, char s[]) {
    int i, sign;
    if ((sign = n) < 0)
        n = -n;
    i = 0;
    do {
        s[i++] = n % 10 + '0';
    } while ((n /= 10) > 0);
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';

    int j = 0;
    char temp;
    i--;
    while (j < i) {
        temp = s[j];
        s[j] = s[i];
        s[i] = temp;
        j++;
        i--;
    }
}

/**
 * @brief Enviar string por UART0
 * @param frequency Frecuencia estimada
 * @param state Estado de la cuerda (0: OK, 1: TENSAR, 2: DESTENSAR)
 * @param string Índice de la cuerda actual
 * Envía un mensaje formateado para la GUI por UART0 con la frecuencia, estado y cuerda.
 * Ejemplo: "freq=220;state=OK;string=2\r\n"
 */
void sendState(uint32_t frequency, uint8_t state, uint8_t string){
    char str[100];
    char freq_str[10];
    char string_str[10];
    char state_str[10];
```

```
    itoaSimple(frequency, freq_str);
    itoaSimple(string, string_str);

    switch(state){
        case 0:
            strcpy(state_str, "OK");
            break;
        case 1:
            strcpy(state_str, "TENSAR");
            break;
        case 2:
            strcpy(state_str, "DESTENSAR");
            break;
        default:
            break;
    }
    strcpy(str, "freq=");
    strcat(str, freq_str);
    strcat(str, ";state=");
    strcat(str, state_str);
    strcat(str, ";string=");
    strcat(str, string_str);
    strcat(str, "\r\n");

    UART_Send((LPC_UART_TypeDef *)LPC_UART0, (uint8_t*)str, strlen(str), BLOCKING);
}
```

Repositorio de GitHub: https://github.com/wandamol1405/GuitarTuna-1769

Aquí encontrará el código en el archivo */src/main.c* y pruebas de cada periférico y algoritmo en */modules-tests/*. También se encuentra el código en Python de la GUI.