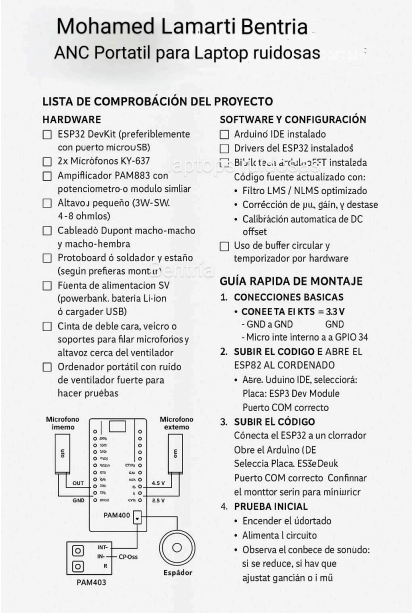

/ **Objetivo Inicial:** Reducir el ruido del ventilador de un portátil gaming.
* **Potencial de Adaptación:** Este diseño puede adaptarse para atenuar ruido en diversos entornos (vehículos, espacios de trabajo, hogares, etc.), ofreciendo una alternativa a soluciones pasivas de aislamiento acústico. Permite explorar la creación de "zonas de silencio" configurables mediante hardware y software.
* **Licencia:** CC BY-NC-SA 4.0 (Ver detalles: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es)

COMPONENTES PRINCIPALES (HARDWARE): presupuesto inicial: 62 €

* Controlador: ESP32 (AZ-Delivery Dev Kit V2 o similar)
* Micrófonos: 2x KY-037 (o similar, entrada analógica)
* Amplificador: 1x PAM8403
* Altavoz: 1x 8 Ohm, 0.5W (o similar)

ALGORITMO BASE:

* LMS (Least Mean Squares) con mejoras (Buffer Circular, Timer Hardware, Calibración DC Offset)

AUTOR:

* Mohamed Lamarti El Messous Ben Triaa

CONTACTO

mr.mohamedlamarti@gmail.com

FECHA (Última revisión):

* 2025-04-25

NOTA IMPORTANTE:

* Este es un prototipo funcional que demuestra el concepto. Requiere **pruebas experimentales y ajustes finos** de parámetros (tasa de aprendizaje `mu`, ganancia `output_gain`, `leakage`, etc.) para optimizar la cancelación en cada escenario específico.

```
// --- LIBRERÍAS ---
#include <driver/dac.h> // Para usar el DAC directamente
#include <driver/adc.h> // Para configurar y leer el ADC
#include "freertos/FreeRTOS.h" // Para delays precisos si no se usa timer
#include "freertos/task.h" // Para delays precisos si no se usa timer

// --- CONFIGURACIÓN DE DEBUG ---
#define DEBUG_ENABLED true // Cambiar a false para desactivar logs seguros

// --- CONFIGURACIÓN DE PINES ---
const adc1_channel_t MIC_REF_ADC_CHANNEL = ADC1_CHANNEL_6; // GPIO34 -> Micrófono de Referencia
const adc1_channel_t MIC_ERR_ADC_CHANNEL = ADC1_CHANNEL_7; // GPIO35 -> Micrófono de Error
const dac_channel_t DAC_OUTPUT_CHANNEL = DAC_CHANNEL_1; // GPIO25 -> Salida DAC 1

// --- PARÁMETROS DEL ALGORITMO Y SISTEMA ---
const int SAMPLE_RATE_HZ = 8000; // Frecuencia de muestreo (Hz) - ¡CRÍTICO PARA TIEMPO ISR!
const int FILTER_LENGTH = 128; // Longitud del filtro FIR adaptativo (taps) - ¡CRÍTICO PARA TIEMPO ISR!
float mu = 0.0005; // Tasa de aprendizaje (¡CRÍTICO! Empezar muy bajo)
const float leakage_factor = 0.0001; // Factor de leakage para estabilidad (opcional, ajustable)
const bool USE_NLMS = true; // ¿Usar LMS Normalizado? (true/false) - Aumenta carga computacional
const float nlms_epsilon = 1e-6; // Pequeño valor para evitar división por cero en NLMS
float output_gain = 0.6; // Ganancia de salida DAC (0.0 a <1.0) - ¡AJUSTAR!

// --- VARIABLES GLOBALES ---
float weights[FILTER_LENGTH] = {0}; // Coeficientes (pesos) del filtro adaptativo
float x_buffer[FILTER_LENGTH] = {0}; // Buffer CIRCULAR para muestras de ruido (x[n])
volatile int buffer_index = 0; // Índice para el buffer circular
float mic_ref_offset_dc = 2048.0; // Offset DC calibrado para Micrófono Referencia
float mic_err_offset_dc = 2048.0; // Offset DC calibrado para Micrófono Error
hw_timer_t *timer = NULL; // Puntero al timer hardware

// --- DECLARACIÓN DE FUNCIONES ---
```

```
float calibrateDCOffset(adc1_channel_t channel, int samples = 200);
float readMicrophone(adc1_channel_t channel, float offset_dc);
void updateNoiseBuffer(float new_sample);
float calculateFilterOutput();
void outputToDAC(float signal);
void updateLMSWeights(float error_signal);
void IRAM_ATTR processANC_ISR(); // ISR debe estar en IRAM
void printDebugInfo(float x, float y, float e); // Llamar desde loop() de forma segura

// --- FUNCIÓN SETUP ---
void setup() {
  Serial.begin(115200);
  Serial.println("Iniciando Prototipo ANC v3 (Timer, Circular Buffer, Calib)...");

  // 1. Configurar Canales ADC
  adc1_config_width(ADC_WIDTH_BIT_12);
  adc1_config_channel_atten(MIC_REF_ADC_CHANNEL, ADC_ATTEN_DB_11);
  adc1_config_channel_atten(MIC_ERR_ADC_CHANNEL, ADC_ATTEN_DB_11);

  // 2. Calibrar Offset DC de los Micrófonos
  Serial.println("Calibrando offsets DC de los micrófonos...");
  mic_ref_offset_dc = calibrateDCOffset(MIC_REF_ADC_CHANNEL);
  mic_err_offset_dc = calibrateDCOffset(MIC_ERR_ADC_CHANNEL);
  Serial.printf("Offset Ref: %.2f, Offset Err: %.2f\n", mic_ref_offset_dc, mic_err_offset_dc);

  // 3. Configurar Canal DAC
  dac_output_enable(DAC_OUTPUT_CHANNEL);
  dac_output_voltage(DAC_OUTPUT_CHANNEL, 128); // Salida media inicial

  Serial.println("Configuración ADC/DAC completa.");
  Serial.printf("Sample Rate: %d Hz, Filter Length: %d, Mu: %f\n", SAMPLE_RATE_HZ,
FILTER_LENGTH, mu);

  // 4. Configurar Timer Hardware
  timer = timerBegin(0, 80, true); // Timer 0, prescaler 80 -> 1MHz clock
  if (!timer) {
    Serial.println("Error al iniciar el Timer!"); while(1);
  }
  timerAttachInterrupt(timer, &processANC_ISR, true); // edge triggered
  uint64_t alarm_value = 1000000 / SAMPLE_RATE_HZ; // Periodo en microsegundos (125
us para 8kHz)
  timerAlarmWrite(timer, alarm_value, true); // auto-reload
  timerAlarmEnable(timer);

  Serial.printf("Timer configurado para %d Hz (periodo %llu us).\n", SAMPLE_RATE_HZ,
alarm_value);
  Serial.println("Sistema ANC iniciado. Esperando interrupciones...");
}
```

```
// --- FUNCIÓN LOOP (Vacía o para tareas no críticas) ---
void loop() {
  // Se puede añadir aquí llamada segura a printDebugInfo si se implementa con cola/flags
  vTaskDelay(pdMS_TO_TICKS(1000));
}

// --- FUNCIÓN PRINCIPAL ANC (ISR) ---
void IRAM_ATTR processANC_ISR() {
  // 1. Leer Micrófono Referencia -> x(n)
  float x_n = readMicrophone(MIC_REF_ADC_CHANNEL, mic_ref_offset_dc);

  // 2. Actualizar buffer circular
  updateNoiseBuffer(x_n); // O(1)

  // 3. Calcular salida filtro -> y(n) (Anti-Ruido)
  float y_n = calculateFilterOutput(); // O(N)

  // 4. Enviar Anti-Ruido al DAC
  outputToDAC(y_n); // O(1)

  // 5. Leer Micrófono Error -> e(n)
  // ¡IMPORTANTE! Existe latencia acústica entre outputToDAC y esta lectura.
  // LMS simple la ignora, FxLMS la modela.
  float e_n = readMicrophone(MIC_ERR_ADC_CHANNEL, mic_err_offset_dc);

  // 6. Actualizar pesos del filtro
  updateLMSWeights(e_n); // O(N) o O(N^2) si NLMS no optimizado
}


// --- FUNCIONES AUXILIARES ---

float calibrateDCOffset(adc1_channel_t channel, int samples) {
  long sum = 0;
  for (int i = 0; i < samples; i++) {
    sum += adc1_get_raw(channel);
    delayMicroseconds(100);
  }
  return (float)sum / samples;
}

// Nota: Considerar normalización simétrica: (adc_raw - offset_dc) / 2048.0;
float IRAM_ATTR readMicrophone(adc1_channel_t channel, float offset_dc) {
  int adc_raw = adc1_get_raw(channel);
  // Normalización robusta pero potencialmente distorsionante si offset no centrado:
  return (adc_raw - offset_dc) / (offset_dc > 2048.0 ? (4095.0 - offset_dc) : offset_dc);
}
```

```
void IRAM_ATTR updateNoiseBuffer(float new_sample) {
  x_buffer[buffer_index] = new_sample;
  buffer_index = (buffer_index + 1) % FILTER_LENGTH;
}

// Optimización posible: precalcular base_index fuera del loop
float IRAM_ATTR calculateFilterOutput() {
  float output = 0.0;
  int current_buffer_ptr = buffer_index;
  for (int i = 0; i < FILTER_LENGTH; i++) {
    int read_index = (current_buffer_ptr - 1 - i + FILTER_LENGTH) % FILTER_LENGTH;
    output += weights[i] * x_buffer[read_index];
  }
  return output;
}

void IRAM_ATTR outputToDAC(float signal) {
  // Considerar compresión suave (tanh) si se necesita evitar clipping fuerte
  int dac_value = 128 + (int)(output_gain * signal * 127.0);
  dac_value = (dac_value < 0) ? 0 : (dac_value > 255 ? 255 : dac_value);
  dac_output_voltage(DAC_OUTPUT_CHANNEL, dac_value);
}

void IRAM_ATTR updateLMSWeights(float error_signal) {
  float current_mu = mu;

  // --- Normalización NLMS (Opcional, O(N) cost) ---
  // Optimización O(1) posible si no se usa leakage (ver análisis previo)
  if (USE_NLMS) {
    float power = 0.0;
    int current_buffer_ptr = buffer_index;
    for (int i = 0; i < FILTER_LENGTH; i++) {
        int read_index = (current_buffer_ptr - 1 - i + FILTER_LENGTH) % FILTER_LENGTH;
        float x_ni = x_buffer[read_index];
        power += x_ni * x_ni;
    }
    current_mu = mu / (nlms_epsilon + power);
  }

  // --- Actualización de Pesos LMS / NLMS con Leakage ---
  int current_buffer_ptr_lms = buffer_index;
  for (int i = 0; i < FILTER_LENGTH; i++) {
    int read_index = (current_buffer_ptr_lms - 1 - i + FILTER_LENGTH) % FILTER_LENGTH;
    float x_ni = x_buffer[read_index];
    weights[i] = weights[i] * (1.0 - current_mu * leakage_factor) + current_mu * error_signal *
x_ni;
  }
```

```
}

// Implementar forma segura (cola FreeRTOS o variables volátiles con flags) si se necesita depuración en tiempo real
void printDebugInfo(float x, float y, float e) {
  if (!DEBUG_ENABLED) return;
  // ... (Implementación segura para llamar desde loop()) ...
  Serial.printf("Ref:%.2f, Anti:%.2f, Err:%.2f, W[0]:%.5f\n", x, y, e, weights[0]);
}
```

6. Calibración y Primeros Pasos
 * Compilar y Cargar: Usa tu IDE para compilar y cargar el código en el ESP32.
 * Monitor Serie: Abre el Monitor Serie (115200 baud). Deberías ver los mensajes de inicio y los valores de offset DC calibrados para cada micrófono. Asegúrate de que estos valores estén cerca del punto medio (aprox. 2048 para ADC de 12 bits). Si están muy lejos, revisa el cableado y la alimentación de los micrófonos.
7. Pruebas y Validación (¡Pasos Críticos!)
Estas pruebas son esenciales para saber si el sistema funciona mínimamente y si es viable. Necesitarás un osciloscopio.
 * Paso 1: Medir Tiempo de Ejecución de la ISR
   * Por qué: La ISR processANC_ISR DEBE ejecutarse en menos tiempo que el periodo de muestreo (1 / SAMPLE_RATE_HZ, que son 125µs para 8kHz). Si tarda más, el sistema fallará.
   * Cómo: Añade gpio_set_level(TU_PIN_DEBUG, 1); al inicio de processANC_ISR y gpio_set_level(TU_PIN_DEBUG, 0); justo al final. Mide el ancho del pulso en TU_PIN_DEBUG con el osciloscopio.
   * Qué hacer: Si el tiempo medido > 125µs, DEBES optimizar: reduce FILTER_LENGTH (p.ej., a 64), considera la optimización O(1) para NLMS si lo usas sin leakage, o reduce SAMPLE_RATE_HZ (lo que limita el ancho de banda de cancelación).
 * Paso 2: Prueba de Señal Básica (Tono Artificial)
   * Por qué: Verifica que toda la cadena (ADC -> Procesamiento -> DAC -> Amplificador) funciona y que el filtro puede generar una señal.
   * Cómo: Modifica temporalmente processANC_ISR para generar un tono simple en x_n (ej: x_n = 0.5 * sin(2.0 * PI * 200.0 * (float)sample_count / SAMPLE_RATE_HZ);) en lugar de leer el micrófono. Observa la salida del DAC (GPIO25) con el osciloscopio. Deberías ver el anti-tono generado.
 * Paso 3: Prueba de Estabilidad Inicial
   * Por qué: Verifica si el algoritmo converge (reduce el error) o diverge (se vuelve inestable).
   * Cómo: Vuelve al código original. Coloca los micrófonos y el altavoz en una configuración estable (ej: referencia cerca del ruido, error donde quieres silencio, altavoz emitiendo hacia el error). Empieza con mu muy bajo (0.0001), output_gain bajo (0.1-0.3), NLMS activado. Monitoriza la señal del micrófono de error (e_n). Idealmente, su amplitud debería disminuir lentamente. Si aumenta sin control o se vuelve loco, reduce mu o output_gain.
8. Ajuste Fino (Tuning)
Este es un proceso iterativo:

* mu (Tasa de Aprendizaje): Controla la velocidad de adaptación. Demasiado bajo = lento. Demasiado alto = inestable. NLMS lo hace menos sensible, pero sigue siendo clave. Aumenta gradualmente desde un valor bajo estable.
* output_gain (Ganancia de Salida): Ajusta la amplitud del anti-ruido. Debe ser suficiente para igualar el ruido original en el punto de error, pero no tanto que sature el DAC/amplificador o cause inestabilidad.
* FILTER_LENGTH (Longitud del Filtro): Filtros más largos capturan mejor las características del ruido (especialmente bajas frecuencias y reverberaciones) pero aumentan drásticamente la carga computacional y pueden requerir un mu más pequeño. Empieza con 64 o 128.
* NLMS/Leakage: Experimenta activando/desactivando para ver el impacto en la estabilidad y velocidad de convergencia con tu ruido específico.

9. Limitaciones Importantes y Próximos Pasos

* Hardware: Los componentes usados son básicos y limitarán el rendimiento máximo (resolución DAC, calidad de micros, potencia de altavoz).
* ¡Latencia Acústica y FxLMS!: El mayor problema del LMS simple es que no compensa el tiempo que tarda el sonido en viajar del altavoz al micrófono de error. Esto limita severamente la cancelación efectiva en el mundo real. Para mejorar significativamente, necesitas implementar Filtered-X LMS (FxLMS). Esto implica:
  * Estimar la "Ruta Secundaria": Medir o modelar la respuesta en frecuencia/impulso desde la salida del DAC hasta la entrada del ADC del micrófono de error.
  * Filtrar la Señal de Referencia: Usar la estimación de la ruta secundaria para filtrar la señal x_n antes de usarla en la actualización de los pesos LMS.
  * FxLMS es significativamente más complejo que LMS.
* Ancho de Banda: La cancelación será más efectiva a frecuencias bajas/medias. La tasa de muestreo y las características del hardware limitan las frecuencias altas.

10. Conclusión

Este proyecto es una excelente plataforma para aprender sobre los fundamentos del ANC y el DSP en tiempo real con hardware accesible. Sin embargo, sé consciente de sus limitaciones inherentes, especialmente la del algoritmo LMS simple frente a la latencia acústica. Considera este prototipo como un punto de partida para explorar el fascinante pero complejo mundo del control activo de ruido. ¡Mucha suerte con tus experimentos!