David Abad Pérez: 54916
Abel Bagué Madrigal: 54919
Jesús Marcos Torero: 54931

Sistemas Electrónicos Digitales
EE-403
Trabajo STM32

SISTEMAS ELECTRÓNICOS DIGITALES

TRABAJO DE MICROS:

CONTROL DE LUCES MEDIANTE AUDIO

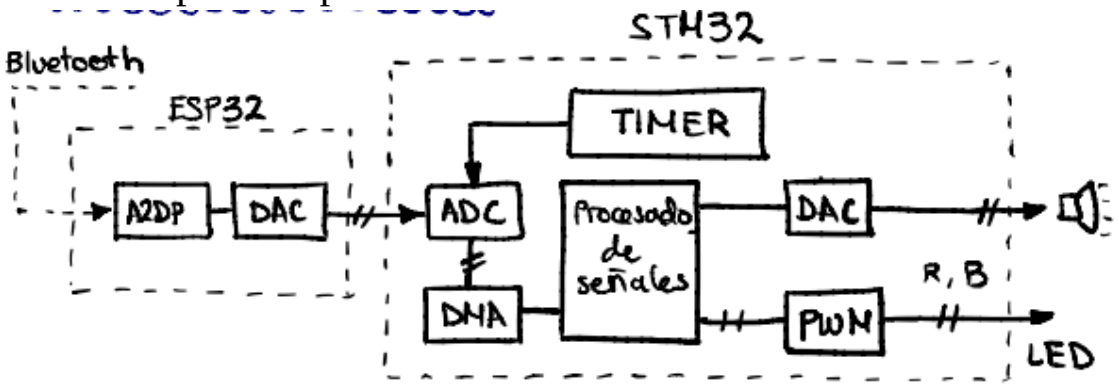
Índice

1 Introducción 3
1.1 Descripción superficial del funcionamiento 3
2 Estrategia y algoritmos 4
2.1 Configuración de la placa ESP32 4
2.2 DMA circular. 5
2.3 Interrupción HAL_ADC_ConvCpltCallback () 5
2.3.1 Tratamiento de la señal 5
2.3.2 Filtrado 6
2.3.3 Control PWM y conversión D/A 7
2.4 Funciones y parámetros de los filtros 7
2.4.1 Filtros L y R 8
2.4.2 Filtro paso bajo 8
2.4.3 Filtro paso alto 8
2.5 Debouncer 9
2.5.1 Mejora: Switch que cambia entre modos de audio 9
3 Distribución del trabajo 10
4 Código Repositorio 10

1 Introducción

El objetivo del trabajo es el diseño en STM32CubeIDE y la implementación en el microprocesador STM32F407 de un filtrado de audio y separación de frecuencias graves y agudas con la placa STM32 conectada a través de Bluetooh mediante la placa ESP32, de tal manera que la señal de audio controle mediante frecuencias graves a un LED rojo y mediante frecuencias agudas a un LED azul, mientras sigue manteniendo salida de audio para conectar altavoces o cascos.

1.1 Descripción superficial del funcionamiento



El programa cuenta con un total de ocho entidades, distribuidas en dos grupos. Un primer grupo compuesto por dos entidades destinadas a la gestión de la entrada (A2DP y DAC), realizada por la placa de Bluetooh ESP32 y, un segundo grupo formado por cuatro entidades que permiten el funcionamiento del programa (ADC, DMA, TIMER y procesado de señales) y, dos entidades que gestionan la salida (DAC y PWM).

- **Gestor de entradas:** El gestor de entradas funciona con la librería A2DP de la empresa “Espressif” para recibir la señal de audio a través del protocolo I2S y un conversor DAC de 8 bits para transformar la señal digital de la ESP32 a una señal analógica captada por la STM32, ya que la señal analógica ha sido la opción más viable y manejable para realizar este proyecto (la señal de salida por I2S está mezclando los bits del volumen con los bits de la señal de audio y tiene mucho ruido interno).
- **Funcionamiento del proyecto:** La señal analógica vuelve a ser transformada a digital para conectarse al DMA. Se trata de un DMA circular a modo de buffer de entrada que recibe la señal de audio a través de una interrupción general. Esta interrupción procesa la señal a través del filtro aplicado y se la pasa al gestor de salidas.
- **Filtrado de audio:** Se utilizan filtros de respuesta infinita (menos retardos) para construir las señales, y como se gobiernan las interrupciones de la DMA con un temporizador a 84kHz se puede saber la frecuencia de muestreo necesaria para construir estos filtros. La acción de filtrado se lleva a cabo en el callback “ConvCpltInterrupt( )” que llama la DMA cada vez que se ha guardado una medida en el buffer.

- **Gestor de salidas:** El gestor de salidas cuenta con un conversor DAC para transmitir la señal de audio a través de un altavoz y también dos temporizadores PWM para transmitir esta señal a los LEDs.

2 Estrategia y algoritmos

La estrategia seguida durante el desarrollo del proyecto ha sido la de dividir las tareas y clasificarlas por nivel de dificultad. De esta forma, conseguimos aumentar la complejidad del trabajo de forma progresiva, asegurándonos de que cada paso se realiza correctamente para que todo encaje a la hora de unir los diferentes bloques que componen el proyecto y, al mismo tiempo, que sea más sencillo aislar los errores.

Las funcionalidades a desarrollar han sido las siguientes:

2.1 Configuración de la placa ESP32

Lo que se ha tenido en cuenta a la hora de escoger esta placa es que tiene un coste y un consumo bajos, cuenta con un conversor DAC y nos permite recibir señales de audio en tiempo real a través de Bluetooth. La otra opción era usar la HC-06 (vista en clase), pero no permite una transmisión continua de datos complejos como se necesita en este caso.

Se ha configurado a través de la librería A2DP utilizando el estándar de interfaz de bus serie I2S. La placa se puede programar con el software de Arduino si se instalan los drivers correspondientes.

Código de la configuración del receptor bluetooth junto con el conversor DAC:

```
#include "BluetoothA2DPSink.h"

BluetoothA2DPSink a2dp_sink;

void setup() {
  static const i2s_config_t i2s_config = {
    .mode = (i2s_mode_t) (I2S_MODE_MASTER | I2S_MODE_TX | I2S_MODE_DAC_BUILT_IN),
    .sample_rate = 44100, // corrected by info from bluetooth
    .bits_per_sample = (i2s_bits_per_sample_t) 16, // the DAC module will only take the 8bits from MSB
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format = I2S_COMM_FORMAT_I2S_MSB,
    .intr_alloc_flags = 0, // default interrupt priority
    .dma_buf_count = 8,
    .dma_buf_len = 64,
    .use_apll = false
  };

  a2dp_sink.set_i2s_config(i2s_config);
  a2dp_sink.start("ESP32_Jesus");

}
```

A continuación, comenzamos a trabajar con la placa STM32F4 y el entorno de desarrollo STM32CubeIDE.

2.2 DMA circular.

El DMA circular nos permite transmitir pares de datos de forma continua a modo de buffer. Las interrupciones del DMA llevan asociadas un temporizador para disminuir la frecuencia a 84 kHz y de esta forma tener una frecuencia estable para construir los filtros digitales

Creamos una variable del mismo número de bits que la resolución del conversor ADC (8 bits) y de dos componentes (left and right):

```
volatile uint8_t buffer_in[2];
```

Comenzamos la transmisión de datos:

```
HAL_ADC_Start_DMA(&hadc1, buffer_in, 2);
```

2.3 Interrupción HAL_ADC_ConvCpltCallback ()

Dentro de esta función tiene lugar el tratamiento de la señal, la elección del filtro y la conversión D/A, además del control PWM del brillo de los LEDs.

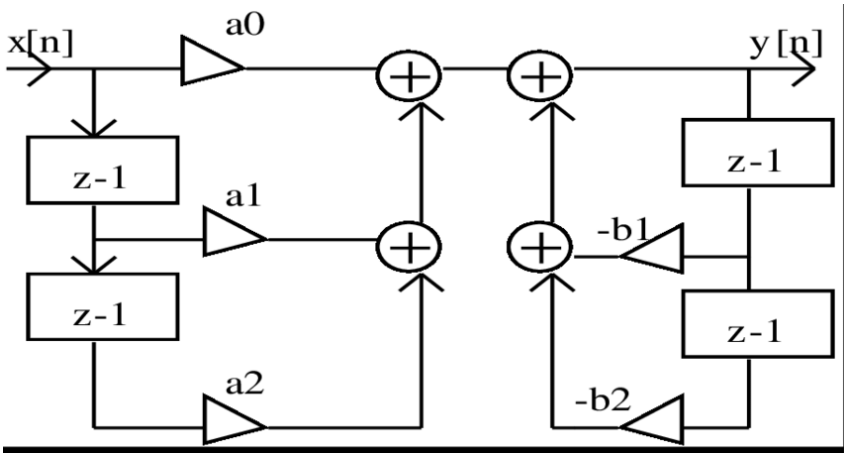
2.3.1 Tratamiento de la señal

Separamos las componentes de la señal estéreo y creamos una señal mono. Los LEDs recibirán la señal mono con el correspondiente filtro aplicado y elevada al cuadrado para quedarnos con los valores positivos, mientras que los altavoces recibirán una

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *AdcHandle){

    int muestra_l = buffer_in[0];//canal l
    int muestra_r = buffer_in[1];//canal r
    senal_mono = (buffer_in[0] >> 1) + (buffer_in[1] >> 1) - 141;

    //tratamiento de señal para las luces
    int temp_led_bajos = filtro_pb(senal_mono);
    int temp_led_agudos = filtro_pa(senal_mono);

    //elevar la señal al cuadrado
    temp_led_bajos = temp_led_bajos * temp_led_bajos;
    temp_led_agudos = temp_led_agudos * temp_led_agudos;
    //temp_led_bajos *= 4;
    //temp_led_agudos *= 4;

    led_bajos = temp_led_bajos;
    led_agudos = temp_led_agudos;
```

2.3.2 Filtrado

El filtrado se hace mediante filtros digitales de impulso infinito (IIR) porque su construcción permite menos retardos aunque sean más inestables que los de pulso finito (FIR). Los filtros responden al siguiente diagrama de bloques:



La programación con la STM32 se hace mediante la ecuación de diferencias, y guardando las muestras en memoria avanzando un paso cada vez que se ejecuta la función. Ejemplo:

```
//filtro paso bajo (led rojo)
int filtro_pb(int muestra){

    //formula del filtro
    float muestra_in = (float) muestra;
    float muestra_out = a0_pb * muestra_in + a1_pb * pbx_1 + a2_pb * pbx_2 - b1_pb * pby_1 - b2_pb * pby_2;
    //mover valores un puesto
    pbx_2 = pbx_1;
    pbx_1 = muestra_in;
    pby_2 = pby_1;
    pby_1 = muestra_out;
    //devolver valor
    return (int) muestra_out;
}

//filtro paso alto (led azul)
int filtro_pa(int muestra){

    //formula del filtro
    float muestra_in = (float) muestra;
    float muestra_out = a0_pa * muestra_in + a1_pa * pax_1 + a2_pa * pax_2 - b1_pa * pay_1 - b2_pa * pay_2;
    //mover valores un puesto
    pax_2 = pax_1;
    pax_1 = muestra_in;
    pay_2 = pay_1;
    pay_1 = muestra_out;
    //devolver valor
    return (int) muestra_out;
}
```

2.3.3 Control PWM y conversión D/A

Luego se agranda la señal a 3 bits y se le suma una cantidad para evitar que tome valores negativos. Después, se convierte a digital la señal y se configuran los ciclos de trabajo de los PWM que controlan la intensidad de brillo de los LEDs asociados a las frecuencias altas y bajas.

```
//agrandar la señal a 3 bits
muestra_l *= 4;
muestra_r *= 4;
//colocar un poco de continua para que no tome valores negativos
muestra_l += 1024;
muestra_r += 1024;
//DAC asincrono
HAL_DAC_SetValue(&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, muestra_l);
HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, muestra_r);

duty_cycle_1 = (100 * led_bajos) / 2048; // pondera la señal del 0 al 100
duty_cycle_2 = (100 * led_agudos) / 2048;
__HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, duty_cycle_1);
__HAL_TIM_SET_COMPARE(&htim4, TIM_CHANNEL_2, duty_cycle_2);
```

2.4 Funciones y parámetros de los filtros

La obtención de estos parámetros se ha llevado a cabo de forma empírica y con ayuda de diversos foros de internet, hasta dar con unos valores lo suficientemente adecuados.

```
//filtro paso bajo 20kHz
float a0 = 0.172893986607494;
float a1 = 0.345787973214988;
float a2 = 0.172893986607494;
float b1 = -0.5300387539709135;
float b2 = 0.22161470040088946;

//filtro paso bajo 250Hz
float a0_pb = 0.00008627874687691115;
float a1_pb = 0.0001725574937538223;
float a2_pb = 0.00008627874687691115;
float b1_pb = -1.9735555323380392;
float b2_pb = 0.9739006473255466;

//filtro paso alto 2kHz
float a0_pa = 0.09534175821687875;
float a1_pa = 0;
float a2_pa = -0.09534175821687875;
float b1_pa = -1.7891079133475518;
float b2_pa = 0.8093164835662425;
```

2.4.1 Filtros L y R

Se implementan filtros paso bajo a 20kHz para eliminar ruidos y cortar a una frecuencia conocida y así cumplir con el teorema del muestreo

```
//filtros para audio
int filtro_l(int muestra){

    //formula del filtro
    float muestra_in = (float) muestra;
    float muestra_out = a0 * muestra_in + a1 * lx_1 + a2 * lx_2 - b1 * ly_1 - b2 * ly_2;
    //mover valores un puesto
    lx_2 = lx_1;
    lx_1 = muestra_in;
    ly_2 = ly_1;
    ly_1 = muestra_out;
    //devolver valor
    return (int) muestra_out;
}

int filtro_r(int muestra){

    //formula del filtro
    float muestra_in = (float) muestra;
    float muestra_out = a0 * muestra_in + a1 * rx_1 + a2 * rx_2 - b1 * ry_1 - b2 * ry_2;
    //mover valores un puesto
    rx_2 = rx_1;
    rx_1 = muestra_in;
    ry_2 = ry_1;
    ry_1 = muestra_out;
    //devolver valor
    return (int) muestra_out;
}
```

2.4.2 Filtro paso bajo

Se trata de un filtro paso bajo de 250Hz para separar las frecuencias graves típicas de la señal de audio.

```
//filtro paso bajo (led rojo)
int filtro_pb(int muestra){

    //formula del filtro
    float muestra_in = (float) muestra;
    float muestra_out = a0_pb * muestra_in + a1_pb * pbx_1 + a2_pb * pbx_2 - b1_pb * pby_1 - b2_pb * pby_2;
    //mover valores un puesto
    pbx_2 = pbx_1;
    pbx_1 = muestra_in;
    pby_2 = pby_1;
    pby_1 = muestra_out;
    //devolver valor
    return (int) muestra_out;
}
```

2.4.3 Filtro paso alto

En realidad es un filtro paso banda porque de nuevo se necesita cortar el máximo de frecuencias siempre para cumplir con el teorema del muestreo. Se trata de un filtro paso banda cuya frecuencia de pico es de 2kHz.

```
//filtro paso alto (led azul)
int filtro_pa(int muestra){

    //formula del filtro
    float muestra_in = (float) muestra;
    float muestra_out = a0_pa * muestra_in + a1_pa * pax_1 + a2_pa * pax_2 - b1_pa * pay_1 - b2_pa * pay_2;
    //mover valores un puesto
    pax_2 = pax_1;
    pax_1 = muestra_in;
    pay_2 = pay_1;
    pay_1 = muestra_out;
    //devolver valor
    return (int) muestra_out;
}
```

2.5 Debouncer

Consiste en utilizar una interrupción externa para detectar la pulsación del botón. Una vez pulsado el botón, esperamos 50 ms (Preescaler = 16800 y Period = 500) para alcanzar el estado inactivo y comprobamos que de verdad se ha cumplido.

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
volatile int flag=0;
volatile int state=1;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_0 && state == 1){
        HAL_TIM_Base_Start_IT(&htim1);
        state = 0;
    }
}
```

Si el botón está inactivo 50 ms después de haber sido pulsado, procedemos a incrementar la variable que usamos para escoger el tipo de filtro que se aplica a la señal de audio.

```
/* USER CODE BEGIN 4 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim->Instance == TIM1){
        if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET){
            flag++;
            if (flag > 2){
                flag = 0;
            }
            state = 1;
            HAL_TIM_Base_Stop_IT(&htim1);
        }
    }
}
```

2.5.1 Mejora: Switch que cambia entre modos de audio

Esto ha sido una mejora implementada después de tener el prototipo del proyecto. Se trata de un switch que elige si la salida de audio será normal, filtrada paso bajo o paso alto, reutilizando los filtros de los leds.

```
    //filtro para el audio
    switch(flag){
        case 0:
            muestra_l = filtro_l(muestra_l) - 141;
            muestra_r = filtro_r(muestra_r) - 141;
            HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, 1);
            HAL_GPIO_WritePin(GPIOD, GPIO_PIN_13, 0);
            HAL_GPIO_WritePin(GPIOD, GPIO_PIN_14, 0);
            break;
        case 1:
            muestra_l = filtro_pb(senal_mono);
            muestra_r = muestra_l;
            HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, 0);
            HAL_GPIO_WritePin(GPIOD, GPIO_PIN_13, 1);
            HAL_GPIO_WritePin(GPIOD, GPIO_PIN_14, 0);
            break;
        case 2:
            muestra_l = filtro_pa(senal_mono);
            muestra_r = muestra_l;
            HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, 0);
            HAL_GPIO_WritePin(GPIOD, GPIO_PIN_13, 0);
            HAL_GPIO_WritePin(GPIOD, GPIO_PIN_14, 1);
            break;
    }
```

3 Distribución del trabajo

La distribución del trabajo se ha dividido por bloques principalmente, aunque todos han intervenido en todas las partes del trabajo con el objetivo de aumentar la lluvia de ideas de la entidad, su correcto funcionamiento y sus futuras ampliaciones.

David Abad Pérez se ha dedicado principalmente al diseño y a la programación de la interrupción HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *AdcHandle) que realiza el procesado de las señales y la interrupción para cambiar el filtro de la señal de audio.

Abel Bagué Madrigal se ha dedicado principalmente a la búsqueda de los parámetros para los filtros de audio aplicados y el diseño de las funciones de filtro paso bajo, alto, left y right, además del diseño e implementación de un método antirrebotes para el botón de cambio de filtro basado en el uso de interrupciones y temporizadores.

Jesús Marcos Torero se ha dedicado principalmente a la configuración de la placa Bluetooh ESP32 a través de la librería A2DP utilizando los protocolos I2S y a la comprobación del programa con el microchip ESP32 ya que era el único miembro del grupo que disponía de este chip.

Por otro lado, las mejoras realizadas por el equipo fueron la introducción de un botón para cambiar el filtro aplicado y los LEDs para comprobar visualmente el filtrado del audio y la separación de los sonidos graves y agudos

Por último, los problemas resueltos han sido la introducción de un DMA circular para poder transmitir los datos de 2 en 2 de forma continua, la introducción del filtro paso bajo a 20kHz para eliminar el ruido y por último poner un temporizador a las interrupciones de la DMA para bajar la f a 84kHz y poder hacer un filtro normalizado.

4 Código Repositorio

https://github.com/jesusmarcostorero/TRABAJO_SED_STM32.git
(ver README para conocer los distintos archivos que hay en el repositorio).