

Manual de ingeniería

APOLO I - V1.0

Apolo company


12 de diciembre del 2023

Confeccionado por:

- Guido Ignacio Glorioso Rego - guido.glorioso@frba.utn.edu.ar
- Nicolas Tobias Almaraz - nalamaraz@frba.utn.edu.ar

Índice

Índice.....	1
Estado del Arte.....	3
Principio de funcionamiento.....	3
Detalles de cada sección.....	4
Entrada y salida de audio del ESP32.....	4
Entrada y salida de audio del STM32.....	5
Interfaz usuario.....	6
Procesamiento de audio.....	9
Amplificación de audio.....	10
Esquemáticos.....	10
PCBs.....	14
Modelos 3D.....	15
Placa.....	15
Estructura.....	16
Porciones de código trascendentes.....	18
Códigos relevantes ST-Nucleo-F401RE.....	18
Códigos relevantes ESP32.....	30
Estructura de procesamiento de datos.....	33
Primer etapa.....	33
Segunda etapa.....	35
Tercer etapa.....	35
Manejo de ganancias.....	36
Consumos eléctricos.....	38
Cálculos generales.....	38
Dispositivos y periféricos empleados.....	38
Tipos de comunicación empleadas.....	38
Protocolo de control.....	39
Diagrama de flujo - Guardado de datos.....	39
Comandos para enviar:.....	42
Cronogramas de su proyecto.....	44
Costos del proyecto.....	45
Filtros sistema.....	46
Filtros de ecualización.....	46



Filtros de crossOver.....	49
Desafíos y problemáticas.....	49
Problemáticas.....	49
Soluciones.....	49
Posibles mejoras para una V2.0.....	50
Conclusiones.....	52
Bibliografía.....	53
Datasheet.....	53

Estado del Arte

En el contexto del mercado de amplificadores de audio con capacidad Bluetooth, se destaca un producto específico que ocupa una posición distintiva entre los productos de gama media. Este amplificador no solo ofrece una calidad de audio mejorada, característica típica de la gama media, sino que también se diferencia por sus altas capacidades de personalización.

En el panorama actual, los productos de gama media suelen equilibrar la calidad de audio y el precio, atrayendo a un público que busca un rendimiento aceptable sin incurrir en costos exorbitantes. Sin embargo, muchos de estos productos pueden carecer de opciones avanzadas de personalización.

El producto en cuestión rompe con esta tendencia al proporcionar a los usuarios una experiencia más rica y adaptada. La capacidad de personalización avanzada permite a los usuarios ajustar el rendimiento del amplificador según sus preferencias individuales, ofreciendo así una experiencia auditiva única.

Este enfoque único lo posiciona estratégicamente en el mercado, ya que satisface las necesidades de un público específico. Aquellos que buscan una combinación de calidad de audio mejorada y la posibilidad de personalizar su experiencia auditiva encontrarán en este producto una opción atractiva.

Principio de funcionamiento

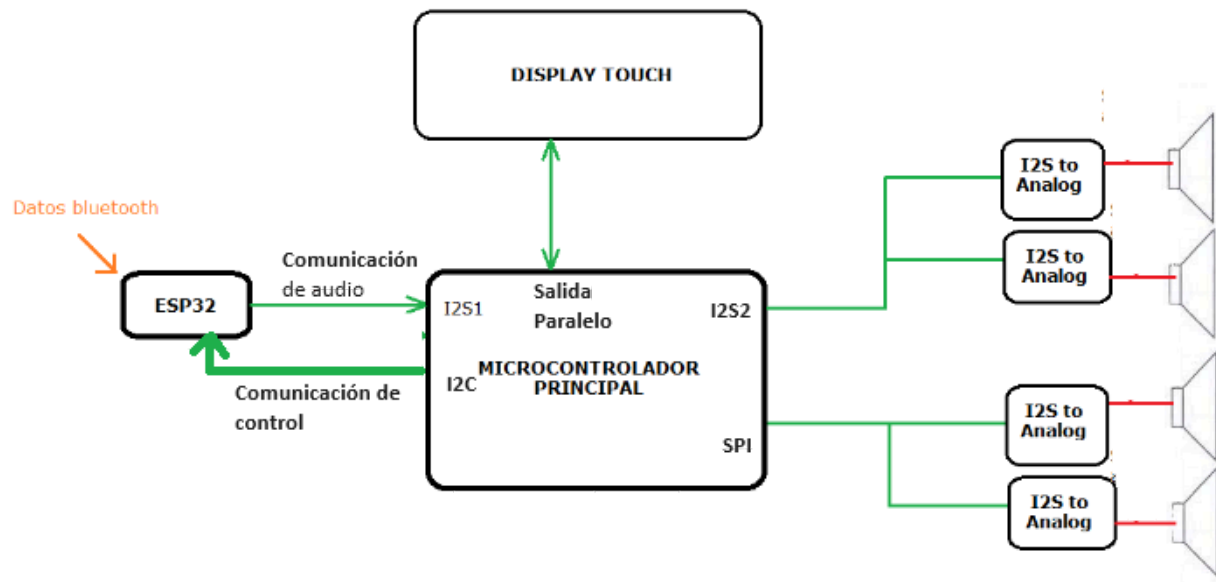
APOLO I consta de cuatro secciones bien definidas:

- Entrada de audio
- Interfaz usuario
- Procesamiento de audio
- Salida de audio y amplificación

APOLO I consta en un receptor bluetooth mediante el microcontrolador ESP32 que recibe los datos y transmite la señal de audio al microcontrolador Nucleo F401Re (microcontrolador principal) vía I2S. Dentro del microcontrolador principal se realiza todo el procesamiento de audio correspondiente y se transmite vía I2S y un I2S simulado a partir de SPI (ver apartado salida de audio y amplificación para más detalles) a los cuatro amplificadores de audio. Estos módulos convierten la señal con un DAC interno y amplifican la señal para poder ser reproducidos en parlantes de 4 a 8 ohm.

La interfaz usuario es procesada y ejecutada por el microcontrolador principal y vía una conexión paralelo se comunica con un módulo de pantalla TFT 3.5" con panel resistivo táctil.

Para guardar los datos de configuración y control del bluetooth se utiliza una conexión auxiliar entre el ESP32 y el microcontrolador principal por I2C.



NOTA: todos los microcontroladores fueron programados utilizando FreeRTOS.

Detalles de cada sección

Entrada y salida de audio del ESP32

APOLO I opera mediante la conexión Bluetooth 4.2 con un perfil A2DP. Su interfaz de entrada de audio se basa en un microcontrolador ESP32. Este componente juega un papel fundamental al recibir los datos a través de la conexión Bluetooth y transmitirlos al microcontrolador principal STM-Nucleo-401RE mediante el protocolo I2S. La elección de Bluetooth 4.2 garantiza una comunicación eficiente y confiable.

Los datos que se reciben están muestreados a una frecuencia de 44.1kHz y son datos estereo. La implementación del código para el ESP32 fue realizada bajo el SDK ESPRESSIF reutilizando código ya proporcionado por el mismo, permitiendo un desarrollo más veloz.

Dentro del ESP32 se realiza el control de ganancia automático (control de loudness), el cual es seteado por medio de la comunicación I2C desde el microcontrolador principal. Los paquetes de datos recibidos por el ESP32 son de una extensión de 820 datos PCM de 16 bits para cada canal.

Se cuenta con un sistema de debugging conectado por cable el ESP32 y a un PC por puerto serie. La velocidad es de 115200 baudios.

Entrada y salida de audio del STM32

En el microcontrolador principal la recepción y transmisión de datos de audio se implementa mediante el protocolo I2S. Un periférico de entrada y dos periféricos de salida. Cabe destacar que el protocolo I2S transmite un dato de canal izquierdo y otro de canal derecho en una misma trama.

Recordemos que la fuente de datos es el ESP32 y los consumidores de información serán los cuatro módulos amplificadores. Dos de ellos serán configurados en canal izquierdo y los otros dos en canal derecho. Para poder controlarlos independientemente se necesitan dos periféricos I2S.

Particularmente en este diseño se utiliza un esquema de double buffering (también llamado ping-pong buffering). Es decir, que trabajamos con DMA y simultáneamente ocurre el siguiente proceso:

DMA:

- Se encarga de transferir los datos del "I2S in" a un buffer A en memoria
- Se encarga de transferir los buffers B y C en memoria a los periféricos de salida "I2S out 1" e "I2S out 2"

Procesador:

- Lee los datos que están en el buffer D
- Procesa la información y escribe los buffers de salida E y F

Cuando ambos procesos terminan, hacen un switcheo de buffers:

DMA:

- Se encarga de transferir los datos del "I2S in" a un buffer D en memoria
- Se encarga de transferir los buffers E y F en memoria a los periféricos de salida "I2S out 1" e "I2S out 2"

Procesador:

- Lee los datos que están en el buffer A
- Procesa la información y escribe los buffers de salida B y C

Esto permite una optimización a la hora de trabajar con streams de audio en tiempo real.

Cabe destacar que el STM32 tiene un total de 2 periféricos I2S. Sin embargo, el proyecto requiere del uso de 3 periféricos. Para poder alcanzar la cantidad de 4 canales, fue necesario utilizar una emulación de I2S mediante SPI. Esto se debe a que el microcontrolador únicamente cuenta con dos salidas I2S nativas. Para dicha implementación se utilizó la nota de aplicación [“I2S protocol emulation on STM32L0 Series microcontrollers using a standard SPI peripheral ”](#) .

En esta básicamente dice que un puerto I2S de salida puede emularse con un puerto SPI y un timer. En donde se utiliza el pin MOSI como DATA y el pin de clock como clock de ambos periféricos. Por otro lado el word select se genera mediante el ingreso de la señal de clock como fuente de un timer configurado en el modo output compare toggle on match. En donde se switchea el word select cada vez que cuenta flancos de 16 clock.

Toda esta sección está implementada en los archivos “Driver_I2S.c” e “Driver_I2S.h”

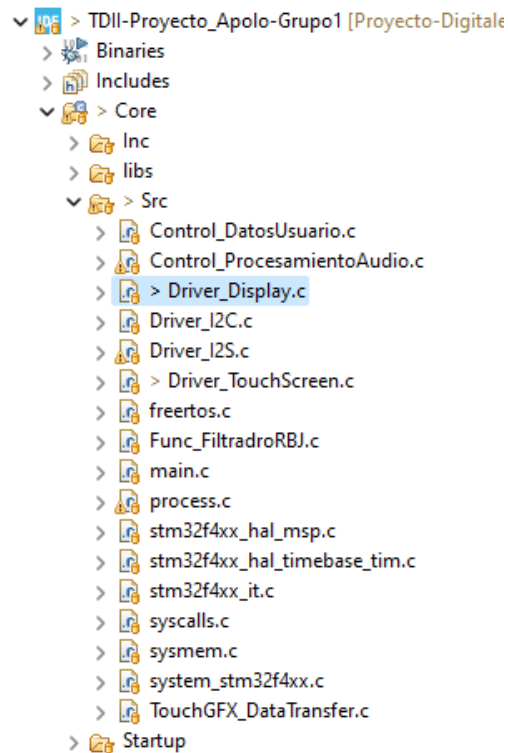
Interfaz usuario

Para el hardware de la interfaz usuario se cuenta con un display TFT de 3.5 pulgadas. Este utiliza un protocolo de comunicación paralelo con 8 pines para transmisión de datos y 4 pines para recibir comandos, siendo los pines de datos bidireccionales. El display está equipado con el controlador RM 68140 para gestionar la información visual. Se conecta directamente a los pines superiores del microcontrolador STM-Nucleo-401RE, lo que permite un reemplazo rápido y sencillo. En caso de mantenimiento o actualización, basta con desconectar uno y conectar otro. Este diseño garantiza una experiencia de usuario eficiente y facilita las tareas de mejora del sistema.

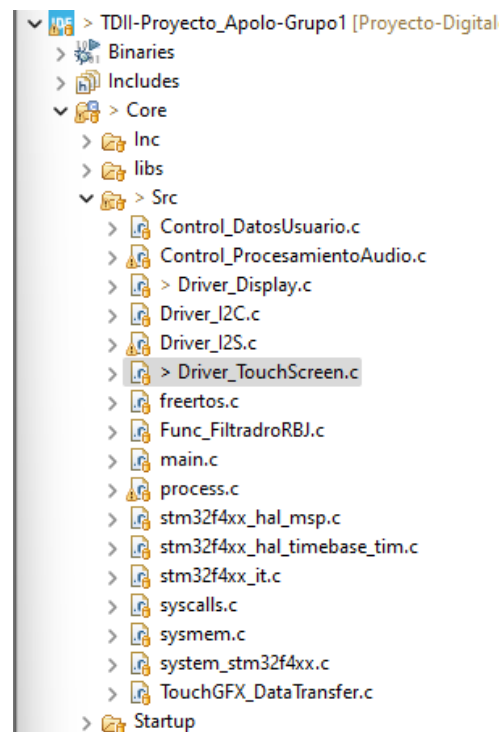
El display seleccionado, además de servir como medio de visualización, también cuenta con la capacidad de panel táctil resistivo. El tipo de interfaz que se utiliza es del tipo cuatro hilos, y para su desarrollo se utilizó la nota de aplicación de ATMEL [“AVR 341: Four and five-wire Touch Screen Controller ”](#). El manejo del panel táctil se realiza a través de 4 pines que se encuentran compartidos con el display, por lo que es necesario realizar un cambio de pinout a la hora de leer el valor presionado.

Para la capa de drivers se utilizaron partes de códigos de la librería [MCU FRIEND](#) que permiten interactuar con las capas de más alto nivel.

Los drivers del control del panel TFT están ubicados en el archivo Driver_Display.c y Driver_Display.h dentro de la carpeta core del proyecto.



Por otro lado, los drivers de manejo del panel táctil están ubicados en `Driver_TouchScreen.c` y `Driver_TouchScreen.h` dentro de la misma carpeta.



Las tareas ejecutadas en el microcontrolador con respecto al manejo de display tienen una prioridad menor que la encargada del procesamiento de audio. Esto garantiza que la funcionalidad principal del dispositivo carezca de retrasos afectando la experiencia del usuario. Sin embargo esto puede provocar algún tipo de retraso en el refresco del display.

Para el desarrollo de la interfaz se utilizó una extensión en el IDE de STM-Cube-IDE que permite diseñar e implementar interfaces gráficas en forma rápida. La extensión es TOUCH GFX y es proporcionada por el mismo fabricante STM.

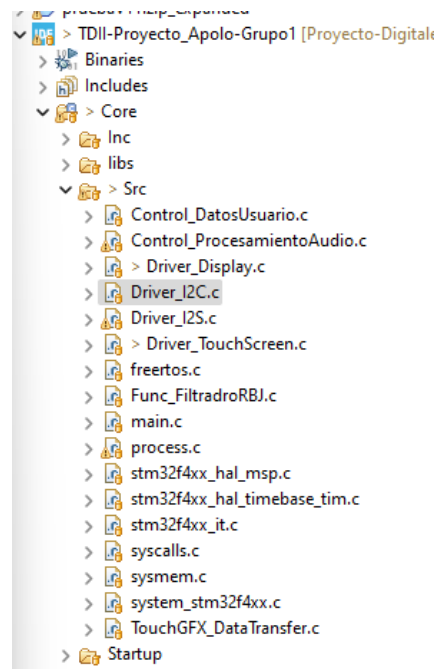
Para la programación de la interfaz se utiliza el lenguaje C ++.


Para guardar los datos de configuración usuario se implementó un método de guardado en flash del microcontrolador ESP32. La razón de esta medida es la baja disponibilidad de memoria en el microcontrolador principal.

Para realizar el guardado de datos se utiliza el protocolo I2C entre ambos microcontroladores, permitiendo enviar la información a guardar y poder obtenerla cuando se reinicie el dispositivo.

Además este canal de comunicación se utiliza para enviar comandos por bluetooth desde el microcontrolador principal al dispositivo conectado.

La ubicación de los archivos enfocados en esta transmisión están ubicados en Driver_I2C.c y Driver_I2C.h:





Resistencias de pullup externas son necesarias debido a que la internas de ambos dispositivos no eran suficientes para poder realizar con éxito la comunicación.

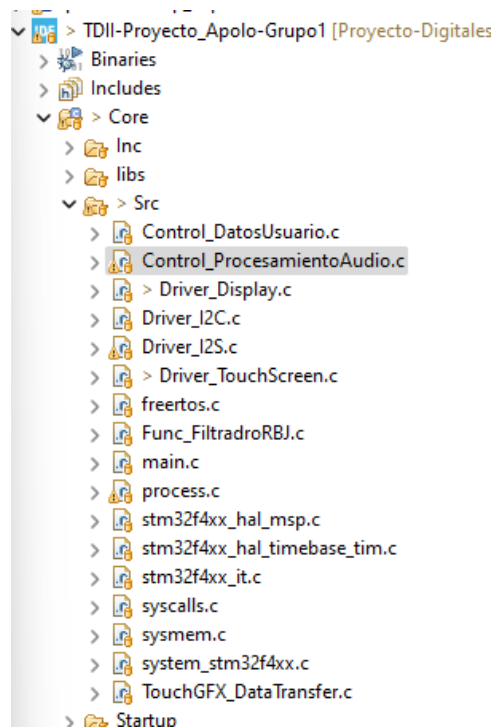
Procesamiento de audio

El procesamiento de audio en el proyecto se ejecuta internamente en el microcontrolador ST. Este proceso se centra en la manipulación de la señal de audio estéreo, abarcando operaciones fundamentales para mejorar la calidad sonora. Las etapas clave de procesamiento incluyen un filtrado general de la señal, filtrado independiente para cada canal de salida y ajuste de volumen.

Para llevar a cabo las operaciones de filtrado, el proyecto aprovecha la librería DSP (Procesamiento Digital de Señales) de CMSIS. Esta librería proporciona funciones y herramientas especializadas que facilitan la implementación eficiente de algoritmos de procesamiento de señales en el microcontrolador ST. Cabe destacar que se trabaja con datos de punto flotante ya que al contar con una unidad FPU el rendimiento es mejor en esta configuración de filtrado (se puede encontrar más información sobre esta librería [aquí](#)).

El sistema de procesamiento de audio está estrechamente vinculado a la información configurada por el usuario a través de la interfaz. La adaptabilidad de las operaciones de filtrado y ajuste de volumen se logra mediante la interacción directa con los parámetros definidos por el usuario en la interfaz. Este enfoque garantiza una experiencia de usuario personalizada y una respuesta adaptativa a las preferencias individuales.

Todo el procesamiento es realizado en la carpeta `Control_ProcesamientoAudio.c` y `Control_ProcesamientoAudio.h`:



Para los filtros que se utilizan, existen una serie de funciones en el archivo “FilterRBJ.c” que permiten obtener los coeficientes para filtros de distintas características. Esta forma de implementación permite en un futuro agregar distintos tipos de filtros sin necesidad de calcular coeficientes en forma externa y luego agregarlos. Basta con utilizar las funciones de diseño directamente en el código.

Estas funciones de cálculo de filtros fueron obtenidas de [RBJ audio EQ Cookbook](#).

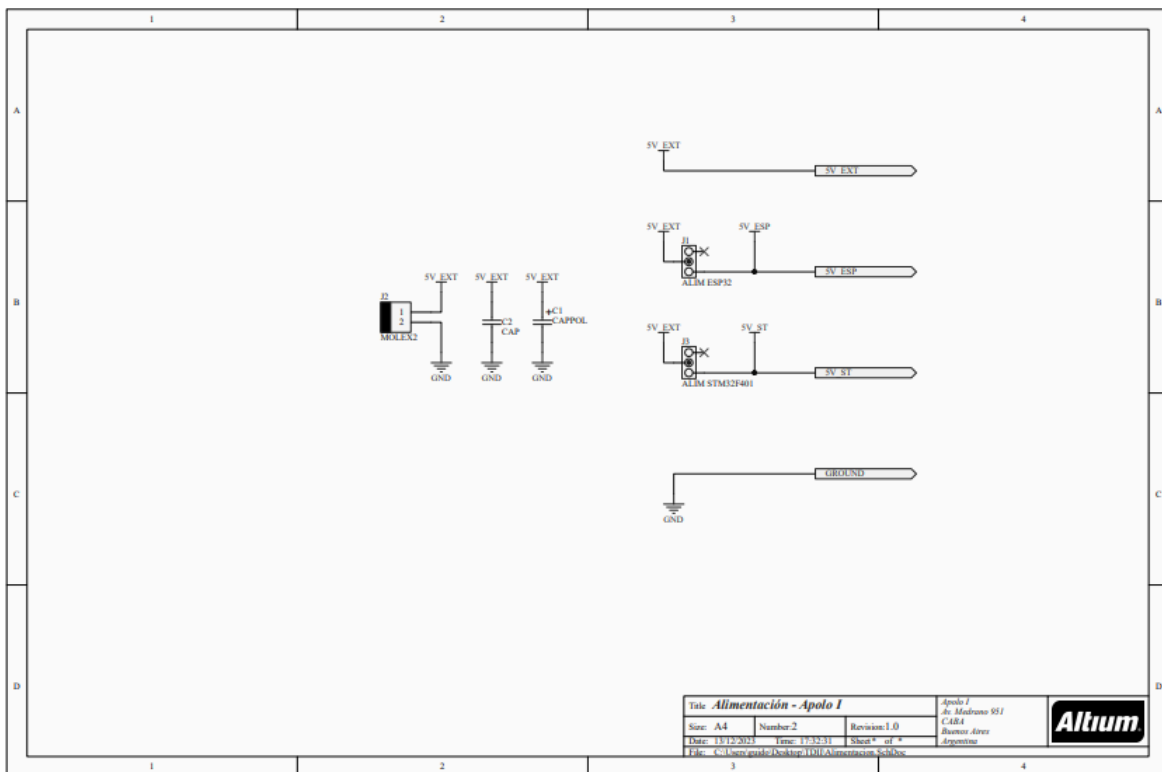
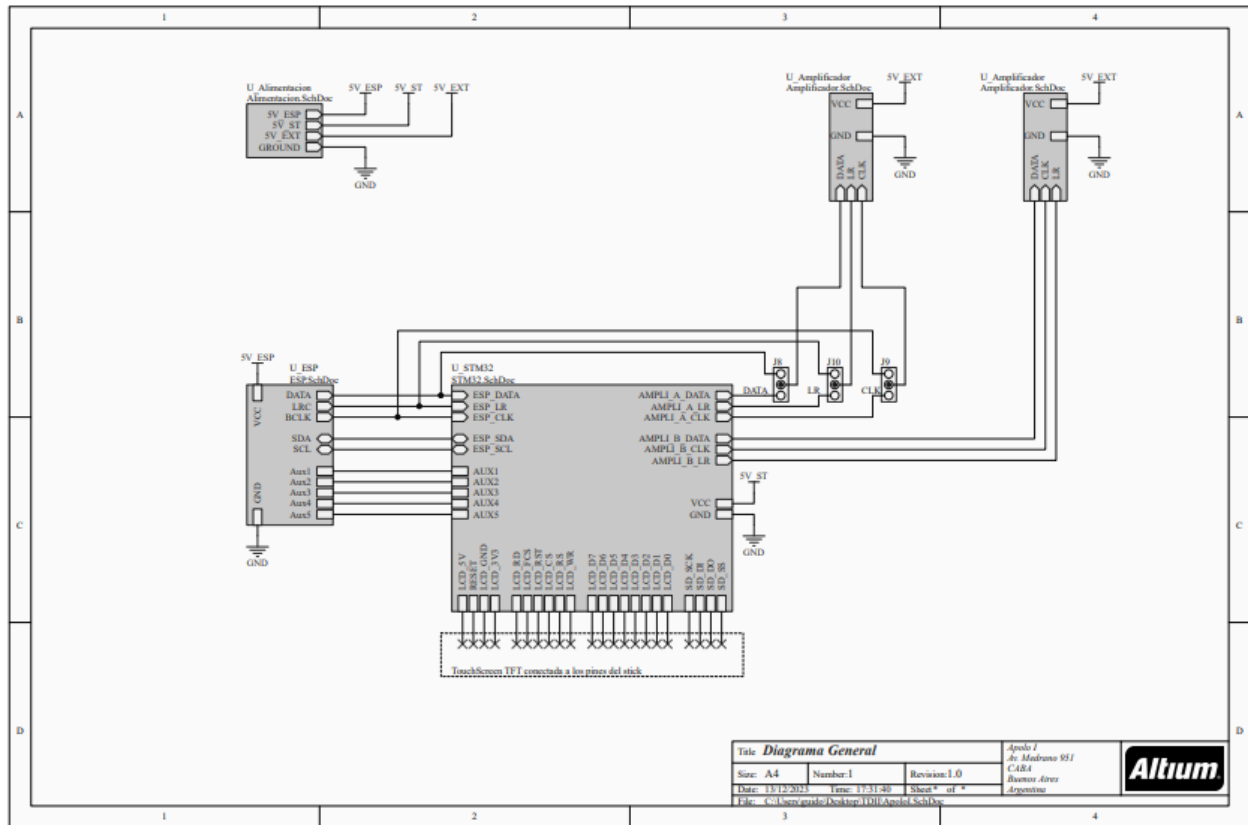
Amplificación de audio

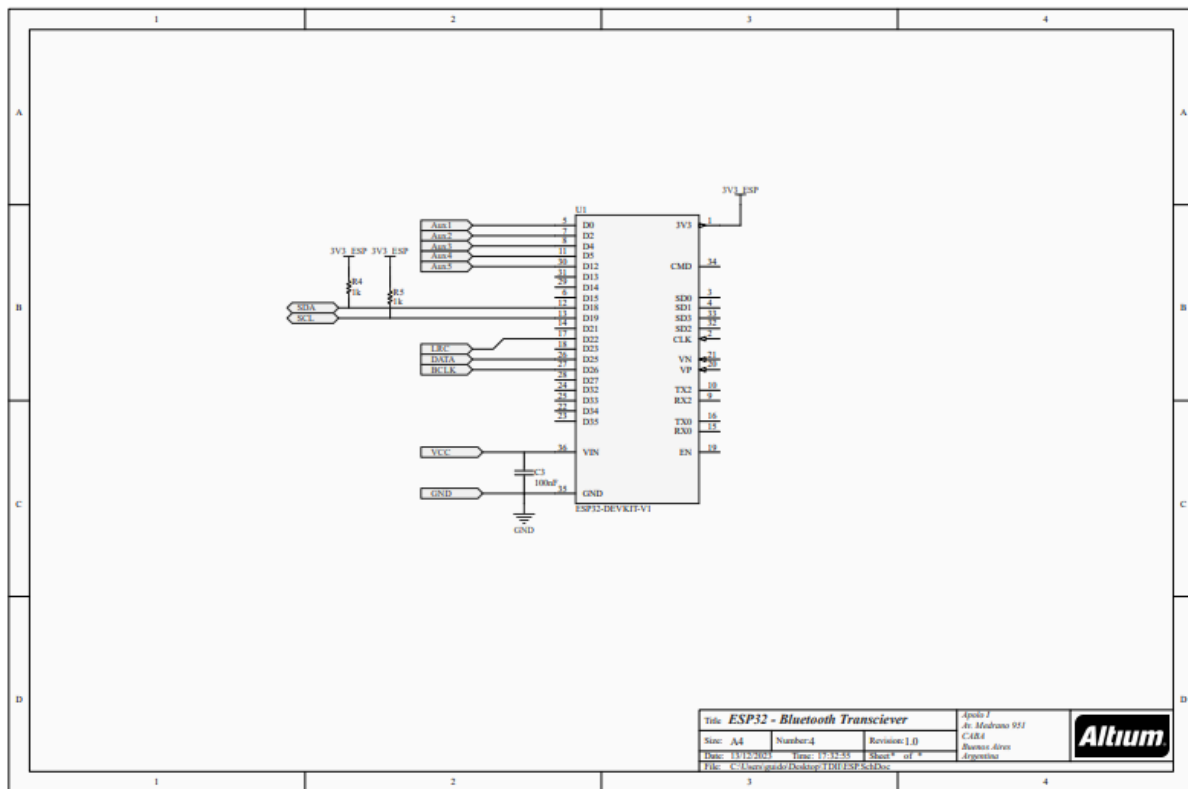
Para la salida de audio de APOLO I, se utilizan módulos basados en el integrado MAX98357A. Este módulo recibe en forma de I2S la señal de audio estéreo, y selecciona alguno de los dos canales mediante pines externos. El mismo integrado es capaz de hacer la conversión DAC y además amplificar la señal para poder ser reproducida en un parlante.

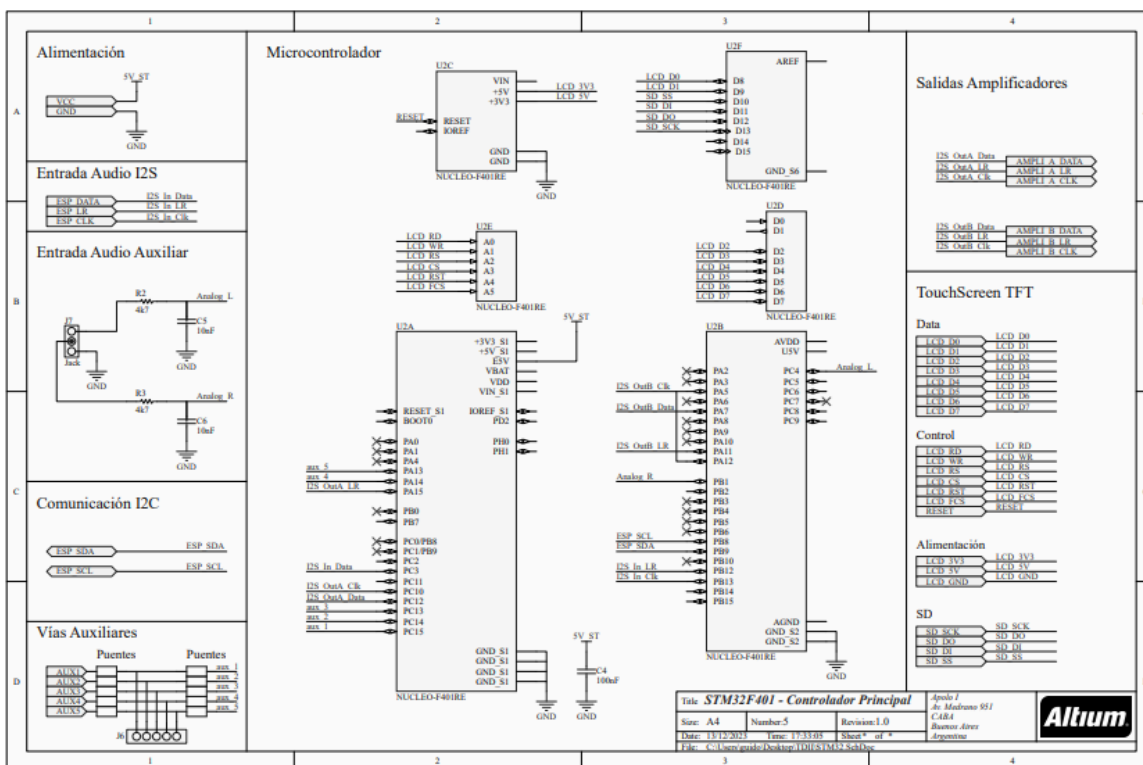
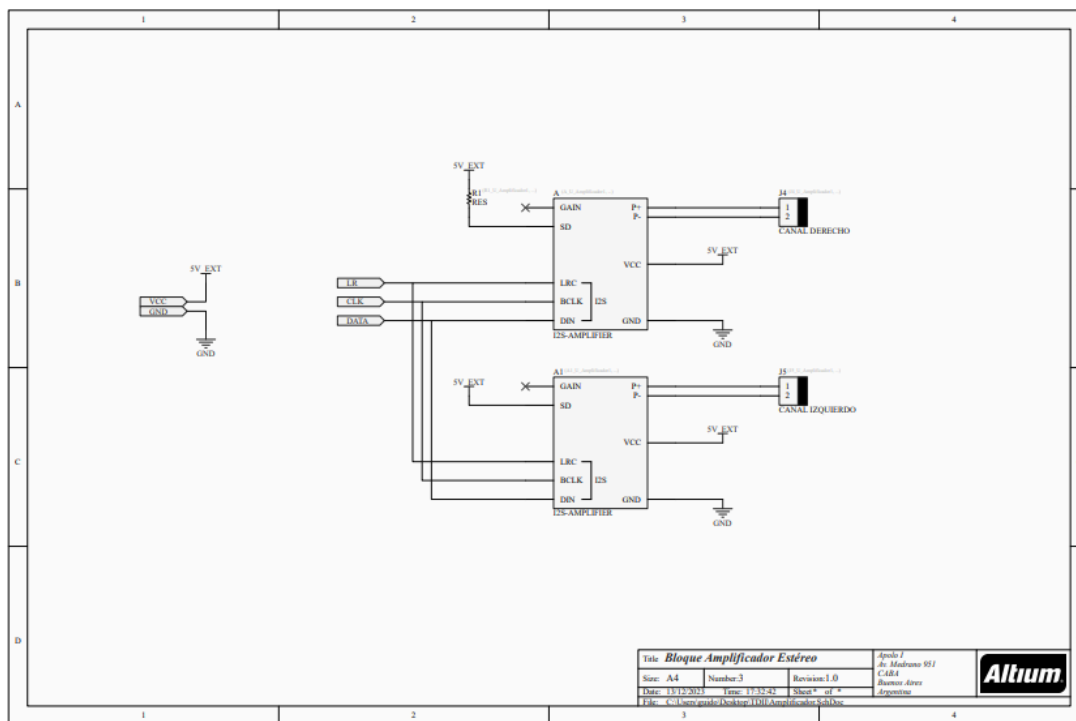
La salida es de tipo “bridge” permitiendo alcanzar un nivel de potencia máximo de 3.2W por canal con un parlante de 4 ohms.

Esquemáticos

Las placas del proyecto fueron desarrolladas con ALTIUM Designer.

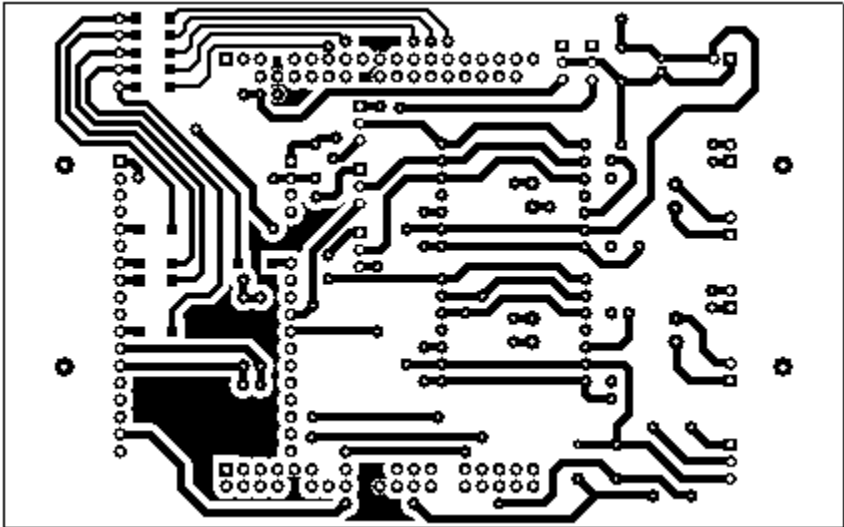






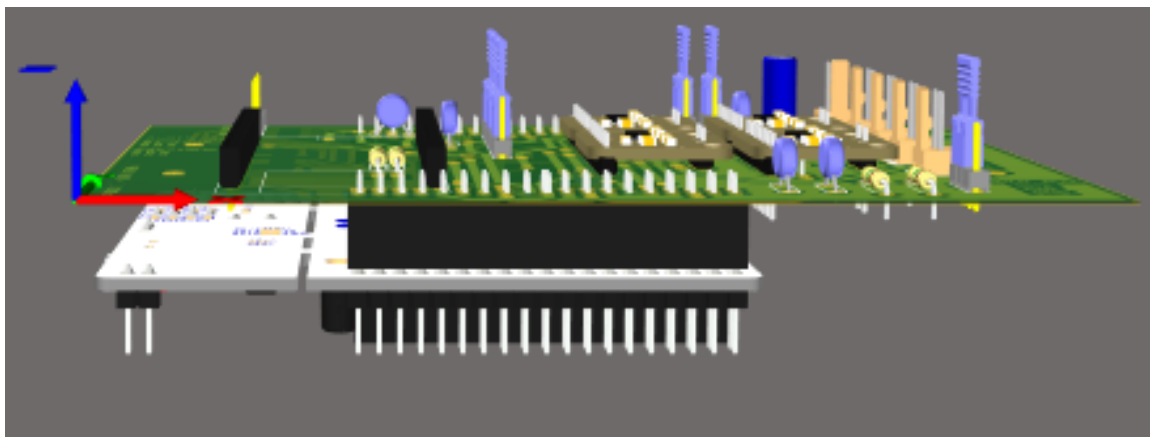
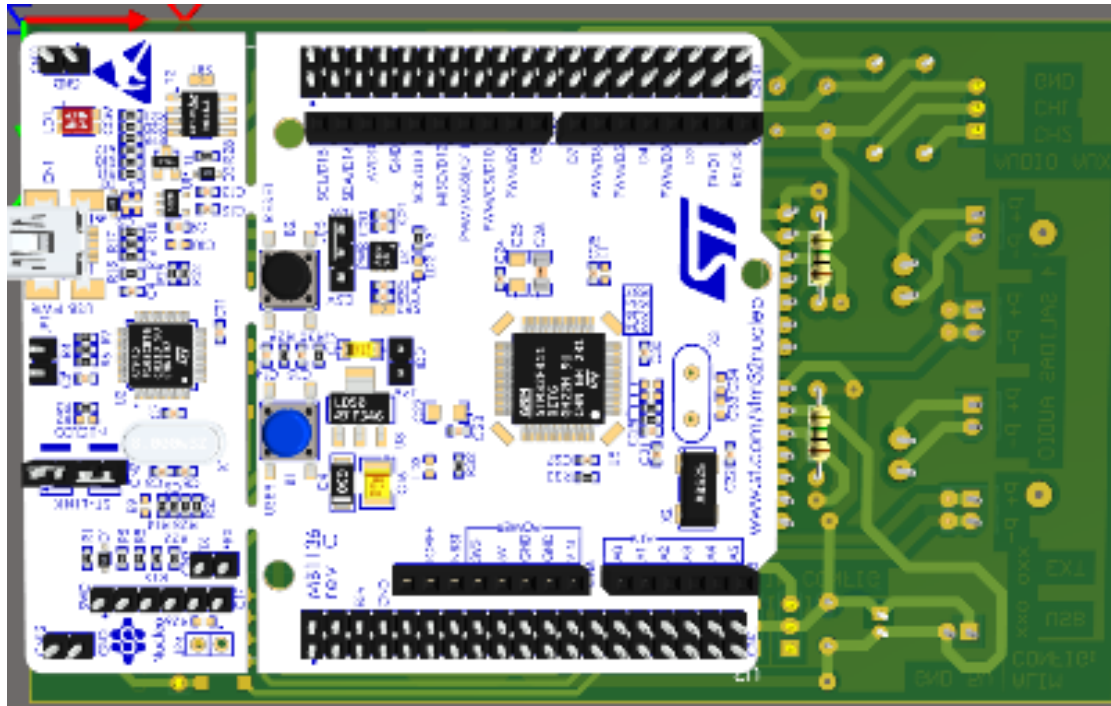
*Para más detalle se adjuntan los esquemáticos en el proyecto.

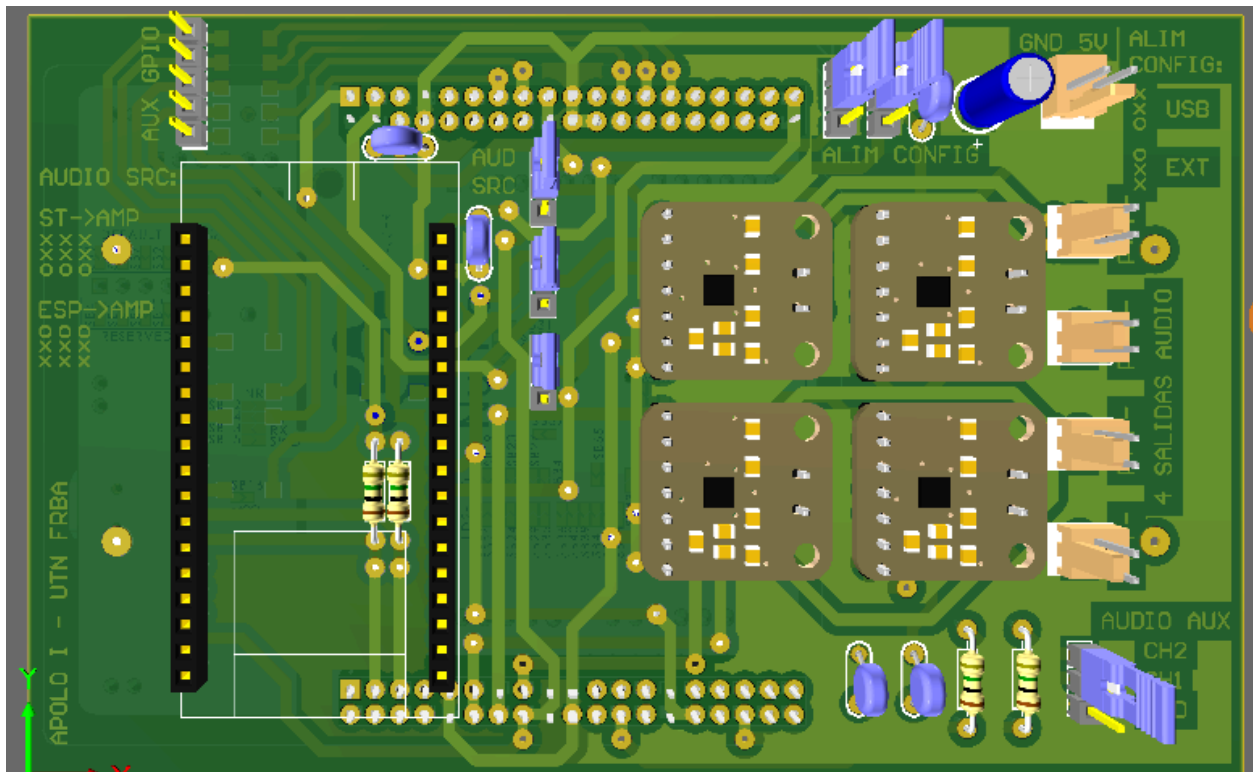
Page 10 of 10



Modelos 3D

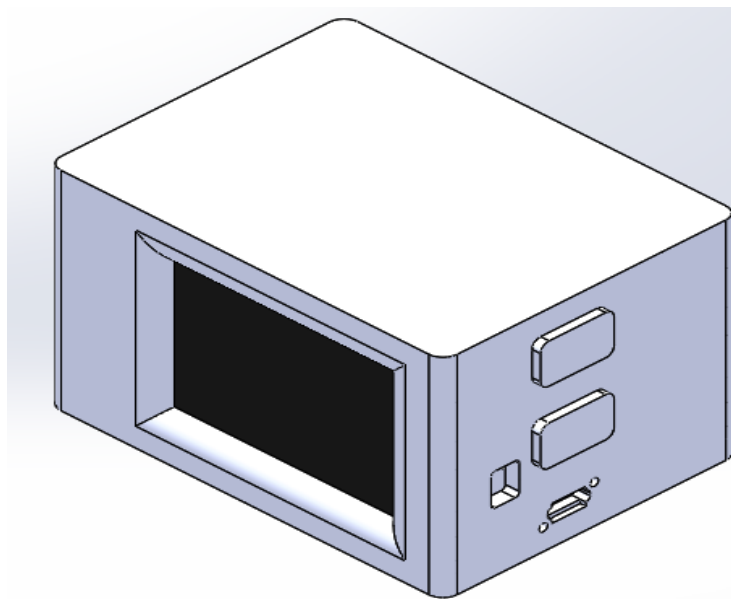
Placa



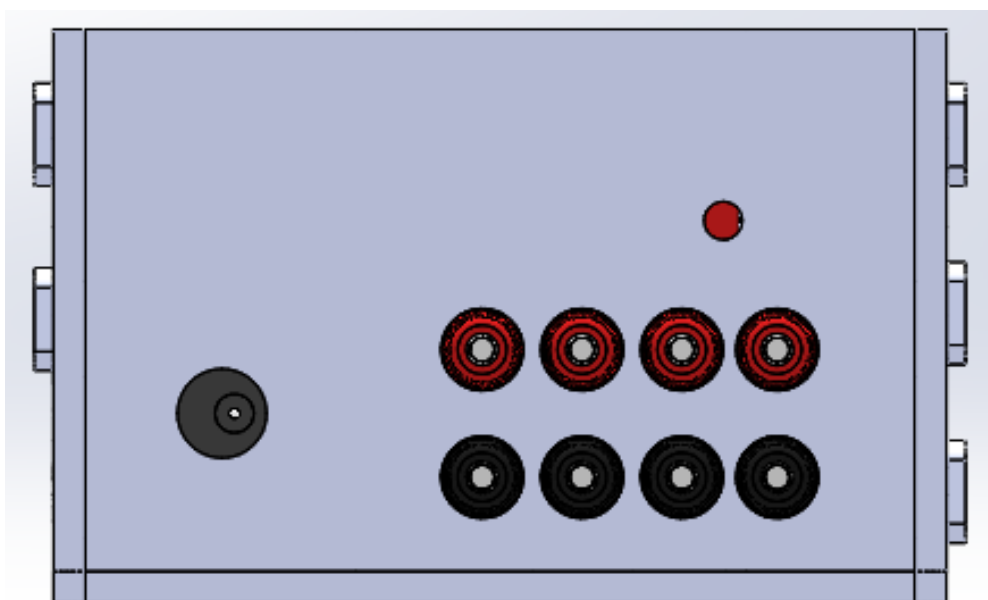


Estructura

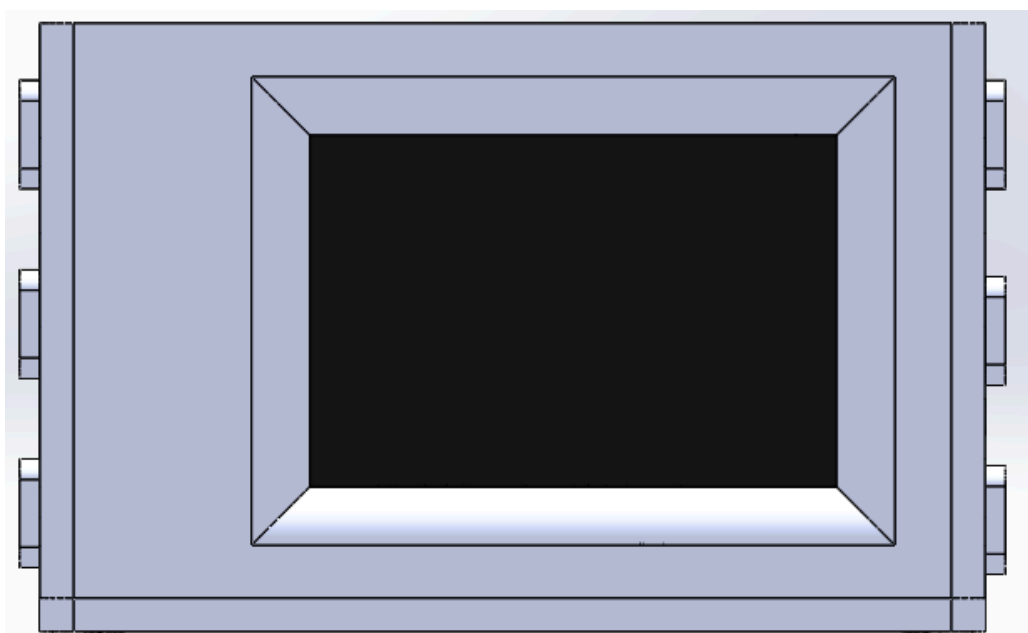
La estructura fue diseñada con SOLIDWORKS.



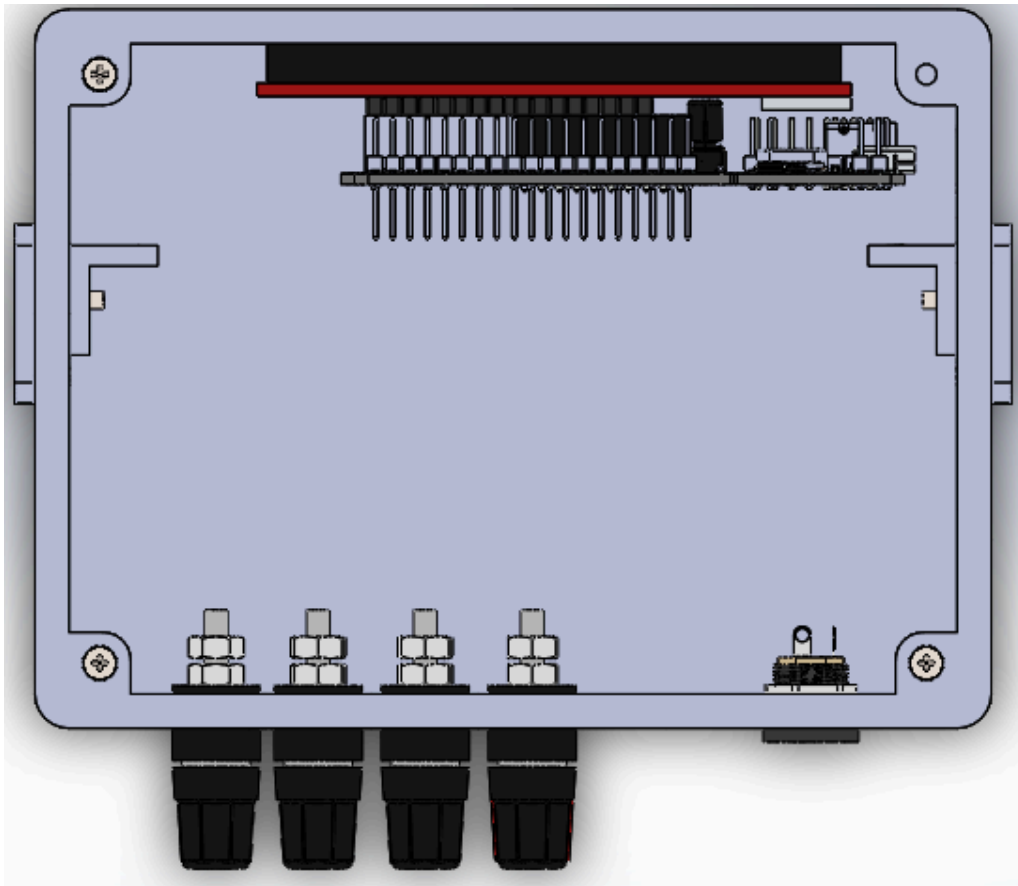
Vista superior derecha



vista trasera



vista frontal



vista inferior sin tapa

Porciones de código trascendentes

A continuación se detallan las partes de código relevantes. Para ello se especifica de donde es extraído el código y su función.

NOTA: Para más detalles revisar código con sus comentarios.

Códigos relevantes ST-Nucleo-F401RE

dirección: proyecto → core → inc → Driver_Display.h

Descripción: Drivers del display.

Configuración de pines y puertos: es importante saber que los pines del display se deben configurar durante tiempo de ejecución por lo que no es posible configurarlos desde el IOC.

```

#define RD_PORT GPIOA
#define RD_PIN 0//GPIO_PIN_0
#define WR_PORT GPIOA
#define WR_PIN 1//GPIO_PIN_1
#define CD_PORT GPIOA // RS PORT
#define CD_PIN 4//GPIO_PIN_4 // RS PIN
#define CS_PORT GPIOB
#define CS_PIN 0//GPIO_PIN_0
#define RESET_PORT GPIOC
#define RESET_PIN 1//GPIO_PIN_1

#define D0_PORT GPIOA
#define D0_PIN 9
#define D1_PORT GPIOC
#define D1_PIN 7
#define D2_PORT GPIOA
#define D2_PIN 10
#define D3_PORT GPIOB
#define D3_PIN 3
#define D4_PORT GPIOB
#define D4_PIN 5
#define D5_PORT GPIOB
#define D5_PIN 4
#define D6_PORT GPIOB
#define D6_PIN 10
#define D7_PORT GPIOA
#define D7_PIN 8

```

Dimensiones del display en pixeles

```

#define WIDTH ((uint16_t)320)
#define HEIGHT ((uint16_t)480)

```

Se optimizó el código mediante macros para poder obtener los tiempos de escritura en el display más reducidos posibles.

dirección: proyecto → core → src → Driver_Display.c

Inicialización del display, es requerido que se ejecute antes de enviar cualquier dato al display.

```

void tft_init(uint16_t ID)
{
    uint16_t *p16;           //so we can "write" to a const protected variable.
    const uint8_t *table8_ads = NULL;
    uint16_t table_size;
    _lcd_xor = 0;
    _lcd_ID = ID;

    _lcd_capable = AUTO_READINC | MIPI_DCS_REV1 | MV_AXIS;
    static const uint8_t RM68140_regValues_max[] = {           //
        0x3A, 1, 0x55,           //Pixel format .kbv my Mega Shield
    };
    table8_ads = RM68140_regValues_max, table_size = sizeof(RM68140_regValues_max);
    p16 = (uint16_t *) & height;
    *p16 = 480;
    p16 = (uint16_t *) & width;
    *p16 = 320;

    _lcd_rev = ((_lcd_capable & REV_SCREEN) != 0);
    if (table8_ads != NULL) {
        static const uint8_t reset_off[] = {
            0x01, 0,           //Soft Reset
            TFTLCD_DELAY8, 150, // .kbv will power up with ONLY reset, sleep out, display on
            0x28, 0,           //Display Off
            0x3A, 1, 0x55,           //Pixel read=565, write=565.
        };
        static const uint8_t wake_on[] = {
            0x11, 0,           //Sleep Out
            TFTLCD_DELAY8, 150,
            0x29, 0,           //Display On
        };
        init_table(&reset_off, sizeof(reset_off));
        init_table(table8_ads, table_size); //can change PIXFMT
        init_table(&wake_on, sizeof(wake_on));
    }
    setRotation(0);           //PORTRAIT
    invertDisplay(false);
}

```

Función de escritura de píxeles

```

void drawRGBBitmap(uint16_t x, uint16_t y, uint16_t *bitmap, uint16_t w, uint16_t h){

    setAddrWindow(x, y, x + w - 1, y + h - 1);
    CS_ACTIVE;
    WriteCmd(_MW);

    uint8_t hi, lo;
    uint16_t cant;
    const uint32_t total_data = h * w;
    for(cant = 0; cant < total_data ;cant++) {
        hi = bitmap[cant] >> 8;
        lo = bitmap[cant] & 0xFF;
        write8(hi);
        write8(lo);
    }
    CS_IDLE;
}

```

Esta función es necesaria ser incluida en la extensión TOUCHGFX para poder controlar el display

dirección: proyecto → core → inc → Driver_TouchScreen.h

Descripción: Drivers del panel táctil.

Definición de pines:

```
/*
 * A1(X-) -> PA1 : valor en X
 * A2(Y-) -> PA4 : valor en Y
 * D6(Y+) -> PB10 : valor en Y
 * D7(X+) -> PA8 : valor en X
 */

#define Xn_PORT GPIOA //A1_PORT
#define Xn_PIN GPIO_PIN_1 //A1_PIN
#define Yn_PORT GPIOA //A2_PORT
#define Yn_PIN GPIO_PIN_4 //A2_PIN

#define Yp_PORT GPIOB //D6_PORT
#define Yp_PIN GPIO_PIN_10 // D6_PIN
#define Xp_PORT GPIOA // D7_PORT
#define Xp_PIN GPIO_PIN_8 //D7_PIN
```

Es importante saber que estos pines son dinámicos, es decir cambiar su funcionamiento en tiempo de ejecución.

Configuración de parámetros para algoritmo de detección de toques:

```
#define NUMBER_OF_SAMPLES 15
#define OFFSET_REBOTE 5
```

Tipo de dato obtenido al presionar la pantalla:

```
//Tipo de dato para resultados
typedef struct TS_value{
    uint16_t x;
    uint16_t y;
}TS_value;
```

dirección: proyecto → core → Src → Driver_TouchScreen.c

Coeficientes de calibración del display:

```

float A_C=0.00170853978;
float B_C=-0.558122993;
float C_C=532.70874;
float D_C=-0.39182511;
float E_C=-0.00379674835;
float F_C=360.130005;

```

Estos coeficientes se obtienen mediante un programa externo.

Algoritmo de detección de toques:

```

/*
 * Maquina de estados que actualiza y calcula valores de coordenadas del display
 * Se debe actualizar como minimo cada lms
 */
void mde_TS(){
    static uint8_t estado = STANDBY;
    TS_value coords_TS;
    switch(estado){
        case STANDBY:
            if(check_standby())
                estado = MEDICION_VALORES;
            break;
        case MEDICION_VALORES:
            updateTouchScreen_x();
            updateTouchScreen_y();
            calculate_cords();
            coords_TS.x = CALIB_X();
            coords_TS.y = CALIB_Y();

            switch( is_valid_value(coords_TS) ){

                default:
                case NO_TOUCH:
                    estado = STANDBY;
                    break;
                case VALIDO:
                    flag_dato = 1;
                    _valueCoords = coords_TS; // @suppress("No break at end of case")
                case BUG_PIXEL:
                case REBOTE:
                    estado = MEDICION_VALORES;
                    break;
            }
            break;
    }
}

```

```

/*
 * Funcion que configura pines y lee el ADC
 */
* X+: GND (salida)
* X-: VCC (salida)
* Y+: HI_Z (entrada)
* Y-: ADC (entrada)
*/

```

```

void updateTouchScreen_x(){

    Set_InputPIN(Yp_PORT,Yp_PIN);
    Set_AnalogPIN(Yn_PORT,Yn_PIN);

    Set_OutputPIN(Xp_PORT,Xp_PIN);
    Set_OutputPIN(Xn_PORT,Xn_PIN);

    SET_PIN(Xn_PORT,Xn_PIN,HIGH);
    SET_PIN(Xp_PORT,Xp_PIN,LOW);

    // configuro ADC en modo polling.

    Set_channel_ADC(ADC_CHANNEL_4); // configuro que canal quiero convetir

    for(uint32_t i= 0;i<NUMBER_OF_SAMPLES;i++){
        HAL_ADC_Start(&hadcl);
        HAL_ADC_PollForConversion(&hadcl, 10); //timeout = 10ms
        _vect_x[i]= HAL_ADC_GetValue(&hadcl);
    }

    HAL_ADC_Stop(&hadcl);

    // deajo los pines listos para ser usados por el display
    Set_OutputPIN(Yp_PORT,Yp_PIN);
    Set_OutputPIN(Yn_PORT,Yn_PIN);
}

```

```

/*
 * Funcion que configura pines y lee el ADC
 */
* X+: HI_Z (entrada)
* X-: ADC (entrada)
* Y+: GND (salida)
* Y-: VCC (salida)
*/

```

```

void updateTouchScreen_y(){

    Set_InputPIN(Xp_PORT,Xp_PIN);
    Set_AnalogPIN(Xn_PORT,Xn_PIN);

    Set_OutputPIN(Yp_PORT,Yp_PIN);
    Set_OutputPIN(Yn_PORT,Yn_PIN);

    SET_PIN(Yn_PORT,Yn_PIN,HIGH);
    SET_PIN(Yp_PORT,Yp_PIN,LOW);

    Set_channel_ADC(ADC_CHANNEL_1); // configuro que canal quiero convetir

    for(uint32_t i= 0;i<NUMBER_OF_SAMPLES;i++){
        HAL_ADC_Start(&hadcl);
        HAL_ADC_PollForConversion(&hadcl, 10); //timeout = 10ms
        _vect_y[i]= HAL_ADC_GetValue(&hadcl);
    }

    HAL_ADC_Stop(&hadcl);

    // deajo los pines listos para ser usados por el display
    Set_OutputPIN(Xp_PORT,Xp_PIN);
    Set_OutputPIN(Xn_PORT,Xn_PIN);
}

```

```

/*Funcion que verifica que tipo de valor se obtuvo
 *Devuelve cero si el valor medido es valido
 *Devuelve codigo de Error en caso de no ser valido
 */
*/
uint8_t is_valid_value(struct TS_value pantalla){

    static struct TS_value previo;
    if(pantalla.x < 5 || pantalla.y<5)
        return NO_TOUCH;
    if(pantalla.x > 223 && pantalla.x < 228 && pantalla.y > 95 && pantalla.y < 97 )
        return BUG_PIXEL;
    if(pantalla.x > (previo.x + OFFSET_REBOTE) || pantalla.x < (previo.x - OFFSET_REBOTE)
        || pantalla.y > (previo.y + OFFSET_REBOTE) || pantalla.y < (previo.y - OFFSET_REBOTE)){
        previo = pantalla;
        return REBOTE;
    }
    previo = pantalla;
    return VALIDO;
}

```

Funcionamiento del algoritmo:

- Se detecta si el display fue presionado.
- Se configuran los pines y se obtienen las mediciones de X e Y.

- Se calculan las posiciones relativas del display en función con los datos calibrados.
- Se verifica el dato obtenido con el anterior. Si la distancia entre ambos es mayor al OFFSET REBOTE se descarta.

NOTA: Los otros errores son fallas del display.

La función void mde_TS() debe ser llamada periódicamente. Luego con la siguiente función:

```
uint8_t get_TS_cords(TS_value* value_str)

    if(flag_dato){
        flag_dato=0;
        *value_str = _valueCoords;
        return 1;
    }
    return 0;
}
```

se obtiene el valor de las coordenadas actuales.

dirección: proyecto → core → Src → TouchGFX_DataTransfer.c

Descripción: Drivers TouchGFX, Estas funciones permiten vincular el driver de la pantalla con las librería de touch GFX

```
extern void DisplayDriver_TransferCompleteCallback();

static uint8_t isTransmittingData = 0;

uint32_t touchgfxDisplayDriverTransmitActive(void)
{
    return isTransmittingData;
}

void touchgfxDisplayDriverTransmitBlock(uint8_t* pixels,
    uint16_t x, uint16_t y, uint16_t w, uint16_t h)
{
    isTransmittingData = 1;
    drawRGBBitmap(x,y,pixels, w, h);
    isTransmittingData = 0;
    DisplayDriver_TransferCompleteCallback();
}
```

dirección: proyecto → core → Inc → Control_DatosUsuario.h

Descripción: manejo de datos de configuración. En este archivo se definen los datos de guardado, así como las funciones que se utilizan para interactuar con estos datos.

Estructuras y datos que se utilizan en la configuración:

```
typedef struct{
    int8_t _20hz;
    int8_t _200hz;
    int8_t _500hz;
    int8_t _1000hz;
    int8_t _4000hz;
    int8_t _8000hz;
    int8_t _16000hz;
}Equalizer;
```

Estructura para guardar datos de ecualización. Los datos se escriben en dB para cada frecuencia

```
typedef struct{
    uint8_t channel;
    uint8_t state;
    uint8_t channel_volume;
    uint8_t type_equalizer;
    uint8_t channel_audio; // indica si saco canal izquierdo o derecho
}AudioOutput;
```

Estructura para guardar datos de configuración de salida. Se tiene una configuración por canal.

```
typedef struct {
    uint8_t system_type;
    uint8_t equalization_type;
    uint8_t max_volume;
    uint8_t loudness_state;
}GeneralConfig;
```

Estructura para almacenar los datos generales del sistema.

```
typedef struct {
    uint8_t Isinitialized;
    uint8_t main_volume;
    Equalizer general_equalizer;
    Equalizer general_equalizer_pers;
    AudioInput audio_input;
    AudioOutput audio_output[4];
    AudioOutput audio_output_pers[4];
    GeneralConfig general_config;
    uint8_t channelInUse; //workarround para problema en croosover edit
}UserData;
```

Estructura general de datos para APOLO I

dirección: proyecto → core → Src → Control_DatosUsuario.c

Función que inicializa todas las variables del sistema en valores predefinidos. Verifica si ya hay datos precargados (datos obtenidos de configuraciones previas).

```
void init_userdata()
```

Funciones para adquirir o devolver configuración:

```
void set_user_data(UserData new_data){  
    user_data = new_data;  
}  
void get_user_data(UserData *read_data){  
    *read_data = user_data;  
}
```

dirección: proyecto → core → Inc → Driver_I2C.h

Descripción: Establece comunicación I2C con el ESP32 y envía comandos y datos que sean necesarios.

OPCODES de I2C:

```
-  
3 #define BUFFER_LEN 50  
4 #define SLAVE_ADDR (0x18 << 1)  
5  
6 #define LOAD_CONFIG_CMD 0x11  
7 #define GET_CONFIG_CMD 0x12  
8 #define STATE_COMMAND_CMD 0x13  
9 #define SAVE_FLASH_CMD 0x14  
0 #define NEXT_SONG_CMD 0x15  
1 #define PREVIOUS_SONG_CMD 0x16  
2 #define STOP_SONG_CMD 0x17  
3 #define LOAD_CONFIG_CMD 0x18  
4 #define GET_MUSIC_STATE_CMD 0x19  
5 #define GET_BT_STATE_CMD 0x20  
6  
7 #define COMMAND_FAILED 0xFF  
8 #define COMMAND_OK 0x33  
9 #define MAX_CMD_SEND 5
```

dirección: proyecto → core → Src → Driver_I2C.c

Función para guardar configuración de usuario en el ESP32:

```

uint8_t save_config_esp(){
    uint8_t nbr_of_try = 0;
    do{
        Buffer_tx[0] = LOAD_CONFIG_CMD;
        HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDR, Buffer_tx , 1 ,HAL_MAX_DELAY); // Envio comando de guardar

        get_user_data(&local_data.datos);
        //memcpy(&local_data.datos,&user_data,STRUCT_LENGTH); // Copio los datos del usuario en la estructura

        HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDR, local_data.datosRaw , STRUCT_LENGTH ,HAL_MAX_DELAY); // Envio los datos del usuario

        vTaskDelay(300); // Cambiar por un vTaskDelay

        Buffer_tx[0] = STATE_COMMAND_CMD;
        HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDR, Buffer_tx , 1 ,HAL_MAX_DELAY); // Envio comando de verificar estado previo

        vTaskDelay(300);

        HAL_I2C_Master_Receive(&hi2c1,SLAVE_ADDR,Buffer_rx,1,HAL_MAX_DELAY);

        nbr_of_try++;
    }while((Buffer_rx[0] != COMMAND_OK)&&(nbr_of_try<MAX_CMD_SEND));

    if(nbr_of_try >= MAX_CMD_SEND)
        return ERROR;
    return SUCCESS;
}

```

Funcion para obtener información de usuario del ESP32:

```

uint8_t get_config_esp(){
    uint8_t nbr_of_try = 0;
    do{
        Buffer_tx[0] = GET_CONFIG_CMD;
        HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDR, Buffer_tx , 1 ,HAL_MAX_DELAY); // Envio comando de guardar

        vTaskDelay(300); // Cambiar por un vTaskDelay

        HAL_I2C_Master_Receive(&hi2c1,SLAVE_ADDR,local_data.datosRaw,STRUCT_LENGTH,HAL_MAX_DELAY);

        Buffer_tx[0] = STATE_COMMAND_CMD;

        HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDR, Buffer_tx , 1 ,HAL_MAX_DELAY); // Envio comando de verificar estado previo

        vTaskDelay(300);

        HAL_I2C_Master_Receive(&hi2c1,SLAVE_ADDR,Buffer_rx,1,HAL_MAX_DELAY);

        nbr_of_try++;
    }while((Buffer_rx[0] != COMMAND_OK)&&(nbr_of_try<MAX_CMD_SEND));

    if(nbr_of_try >= MAX_CMD_SEND)
        return ERROR;
    set_user_data(local_data.datos);
    //memcpy(&user_data,&local_data.datos,STRUCT_LENGTH); // Copio los datos del usuario en la estructura

    return SUCCESS;
}

```

Función para enviar comandos al ESP32:

```

uint8_t send_cmd_esp(uint8_t CMD){
    uint8_t nbr_of_try = 0;
    do{
        Buffer_tx[0] = CMD;
        HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDR, Buffer_tx , 1 ,HAL_MAX_DELAY); // Envio comando de guardar

        Buffer_tx[0] = STATE_COMAND_CMD;
        HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDR, Buffer_tx , 1 ,HAL_MAX_DELAY); // Envio comando de verificar estado previo

        vTaskDelay(300);

        HAL_I2C_Master_Receive(&hi2c1,SLAVE_ADDR,Buffer_rx,1,HAL_MAX_DELAY);

        nbr_of_try++;
    }while((Buffer_rx[0] != COMAND_OK)&&(nbr_of_try<MAX_CMD_SEND));

    if(nbr_of_try >= MAX_CMD_SEND)
        return ERROR;
    return SUCCESS;
}

```

Comando para conocer el estado de la conexión Bluetooth.

```

4@uint8_t get_bt_estado_esp(){
5    uint8_t nbr_of_try = 0;
6    uint8_t temp=0;
7    xSemaphoreTake(semI2CResource,portMAX_DELAY);
8    do{
9        Buffer_tx[0] = GET_BT_STATE_CMD;
10       HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDR, Buffer_tx , 1 ,HAL_MAX_DELAY); // Envio comando de estado BT
11
12       vTaskDelay(300);
13
14       HAL_I2C_Master_Receive(&hi2c1,SLAVE_ADDR,Buffer_rx,1,HAL_MAX_DELAY);
15
16       temp = Buffer_rx[0]; // Recibo el porcentaje de la cancion que se encuentra reproducido
17
18       Buffer_tx[0] = STATE_COMAND_CMD;
19
20       HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDR, Buffer_tx , 1 ,HAL_MAX_DELAY); // Envio comando de verificar estado previo
21
22       vTaskDelay(300);
23
24       HAL_I2C_Master_Receive(&hi2c1,SLAVE_ADDR,Buffer_rx,1,HAL_MAX_DELAY);
25
26       nbr_of_try++;
27   }while((Buffer_rx[0] != COMAND_OK)&&(nbr_of_try<MAX_CMD_SEND));
28   xSemaphoreGive(semI2CResource);
29   if(nbr_of_try >= MAX_CMD_SEND)
30       return COMAND_FAILED;
31
32   return temp;
33 }

```

dirección: proyecto → core → Src → Func_FiltradoRBJ.c

Descripción: funciones para calcular coeficientes de filtros.

```
// Funciones diseño filtros RBJ

float32_t convertBwToOctaves(double fc_inf, double fc_sup);
float32_t get_alpha(float32_t w0, float32_t Q);
float32_t get_A(float32_t dbGain);
float32_t get_w0(float32_t f0);
float32_t get_alpha_fromBW(float32_t w0, float32_t BW);
void low_shelf_pass(float32_t f0, float32_t dbGain, float32_t Q, float32_t* b, float32_t* a);
void high_shelf_pass(float32_t f0, float32_t dbGain, float32_t Q, float32_t* b, float32_t* a);
void peaking_filter(float32_t f0, float32_t dbGain, float32_t BW, float32_t* b, float32_t* a);
void band_pass(float32_t f0, float32_t dbGain, float32_t BW, float32_t* b, float32_t* a);
void low_pass(float32_t f0, float32_t Q, float32_t* b, float32_t* a);
void high_pass(float32_t f0, float32_t Q, float32_t* b, float32_t* a);
void designFilter(float32_t gainsdB[7], float32_t Q_BW[7], float32_t *sos, type_data_filter type);
void designCrossover(float32_t *buff, uint8_t Type);
```

Dirección: proyecto → core → Src → process.c

Descripción: programa principal.

Creación de tareas:

```
xTaskCreate(Touchscreen_process, "UPD TS", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, NULL);
xTaskCreate(SaveData_process, "SaveData", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 4, NULL);
xTaskCreate(volume_process, "volSYS", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, NULL);
xTaskCreate(upd_progressVar_process, "progVar", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, NULL);
xTaskCreate(Check_bt_process, "btcheck", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, NULL);
xTaskCreate(TouchGFXSYNC_process, "GFX SYNC", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 2, NULL);
xTaskCreate(TouchGFX_Task, "UPD GFX", 3000, NULL, tskIDLE_PRIORITY + 2, NULL);
xTaskCreate(FilterData_process, "ProcessData", 2000, NULL, tskIDLE_PRIORITY + 3, NULL); // se le debe dar la mayor prioridad
xTaskCreate(CalculateCoefs_process, "CalcCoefsData", 500, NULL, tskIDLE_PRIORITY + 3, NULL); // Process que calcula los coeficientes aun m
// La Tarea de I2S se inicializa por medio de la tarea "btcheck" en el momento que se detecte una conexión bluetooth su prioridad es idle + 3
```

Process de tareas:

```
void TouchGFXSYNC_process(void *arguments){
    extern void touchgfxSignalVSync(void);

    while(1){
        touchgfxSignalVSync();
        vTaskDelay(pdMS_TO_TICKS(1));
    }
}

void Touchscreen_process(void *arguments){

    while(1){
        mde_TS();
        vTaskDelay(pdMS_TO_TICKS(1));
    }
}
```

```

void SaveData_process(void *arguments){
    get_config_esp();
    init_userdata();
    while(1){

        xSemaphoreTake(semSaveData,portMAX_DELAY);

        if(save_config_esp() == SUCCESS)
            send_cmd_esp(SAVE_FLASH_CMD);

    }
}

void FilterData_process(void *arguments){

    filter_init_system(); // inicializo los filtros

    while(1){

        // Verifico si hay datos nuevos para filtrar
        xSemaphoreTake(semProcessData,portMAX_DELAY);
        process_filter();
        process_set_gains();
        xSemaphoreGive(semProcessData);
        vTaskDelay(pdMS_TO_TICKS(20));

    }
}

```

```

26 void volume_process(void *arguments){
27     while(1){
28         xSemaphoreTake(semCalcVolume,portMAX_DELAY);
29         process_set_gains();
30
31         //vTaskDelay(pdMS_TO_TICKS(50));
32     }
33 }
34
35
36
37 void upd_progressVar_process(void *arguments){
38     uint8_t valor_progress=0;
39     while(1){
40         valor_progress = get_music_estate_esp();
41
42         xQueueSend(queue_progres_var,&valor_progress,portMAX_DELAY);
43         vTaskDelay(pdMS_TO_TICKS(1000));
44     }
45 }
46

```

```

17 void Check_bt_process(void *arguments) {
18     uint8_t previo=0;
19     uint8_t estado_actual=0;
20     while(1){
21         estado_actual = get_bt_estate_esp();
22         if(previo != estado_actual && previo != COMAND_FAILED){
23             previo = estado_actual;
24             if(estado_actual == BT_CONNECTED){
25                 initI2SDriver();
26             }
27             if(estado_actual == BT_DISCONNECTED){
28                 deinitI2SDriver();
29             }
30         }
31         vTaskDelay(pdMS_TO_TICKS(1500));
32     }
33 }
34 }
--

void task_I2S_recieve() {

    //initI2SDriver();

    while(1) {

        // Revizo si se termino de recibir y enviar los datos
        xSemaphoreTake(semTxAB_I2S,portMAX_DELAY);

        //xSemaphoreTake(semTxCD_I2S,portMAX_DELAY);

        xSemaphoreTake(semRx_I2S,portMAX_DELAY);

        //Proceso los datos

        //Libero el semaforo asi la tarea de procesamiento arranca
        xSemaphoreGive(semDataReady);

        //pruebaLoopback(); // comentar si no se desea loopback
    }
}

```


Códigos relevantes ESP32

Dirección: proyecto → main → main.c

Descripción: inicialización de tarea comunicación I2C.

```
void app_main(void)
{
    //Init I2C
    comi2c_start_up();
}
```

```
void comi2c_start_up() {
    esp_err_t err=i2c_slave_init();
    i2c_set_pin(I2C_SLAVE_NUM, I2C_SLAVE_SDA_IO, I2C_SLAVE_SCL_IO, GPIO_PULLUP_DISABLE, GPIO_PULLUP_DISABLE, I2C_MODE_SLAVE);

    if(err == ESP_OK){
        #if DEBUG
            printf("Se inicio bien I2C\n");
        #endif
    }

    err = init_nvs();
    if(err == ESP_OK){
        #if DEBUG
            printf("Se inicio bien NVS\n");
        #endif
    }

    read_nvs(&local_user_data.user_data);

    //local_user_data.user_data.Isinitialized = 0; // descomentar estas lineas para reiniciar la config
    //write_nvs(&local_user_data.user_data);

    //print_data(local_user_data.user_data);

    xTaskCreate(i2c_task, "i2c_task", 1024 * 2, (void *)0, 11, NULL);
}
```

Dirección: proyecto → main → comI2C.c

Descripción: manejo y desarrollo de comunicación de control.

Funciones de control y manejo de la unidad NVS (Non-Volatile Storage)

```
esp_err_t init_nvs(void);
esp_err_t read_nvs(UserData *value);
esp_err_t write_nvs(UserData *value);
```

Tarea de recepción de datos I2C y procesamiento:

```

void i2c_task (void* parameters){
    int8_t size = 0;

    while(1){
        size = i2c_slave_read_buffer(I2C_SLAVE_NUM, buff_rx, 1, portMAX_DELAY);

        if(size == ESP_FAIL){
            #if DEBUG
                printf("Fallo el Read\n");
            #endif
        }
        else{
            if(size == 0){
                #if DEBUG
                    //printf("No se recibio mensaje\n");
                #endif
            }
            else{
                #if DEBUG
                    printf("Se recibio el cmd: %i \n",buff_rx[0]);
                #endif
                command_state = analyses_message(buff_rx[0]);
            }
        }

        vTaskDelay(pdMS_TO_TICKS(10));
    }
}

```

Procesamiento de tramas recibidas:

```

uint8_t analyses_message(uint8_t command){
    int8_t err = 0;

    switch(command){
        case LOAD_CONFIG_CMD: // Se pasa la configuracion a guardar.
            vTaskDelay(pdMS_TO_TICKS(300)); // Espero 300 ms para recibir todo.

            err = i2c_slave_read_buffer(I2C_SLAVE_NUM, buff_rx,STRUCT_LENGTH, pdMS_TO_TICKS(100));

            #if DEBUG
            printf("Se recibieron %i datos\n",err);
            #endif

            if(err == 0)
                return ERROR;
            if(err !=STRUCT_LENGTH)
                return INSUFICIENT_DATA;

            memcpy(local_user_data.datosRaw, buff_rx, STRUCT_LENGTH);

            #if DEBUG
            print_data(local_user_data.user_data);
            #endif
            break;
        case GET_CONFIG_CMD: // Se pide la configuracion local
            i2c_reset_tx_fifo(I2C_SLAVE_NUM);
            err = i2c_slave_write_buffer(I2C_SLAVE_NUM, local_user_data.datosRaw,STRUCT_LENGTH,pdMS_TO_TICKS(100));

            if(err == ESP_FAIL)
                return ERROR;

            break;
        case STATE_COMAND_CMD: // Se pide el estado del comando anterior (si fue exitoso o no)

            if(command_state == ESP_OK)
                buff_tx[0] = COMAND_OK;
            else
                buff_tx[0] = COMAND_FAILED;

            i2c_reset_tx_fifo(I2C_SLAVE_NUM);
            err = i2c_slave_write_buffer(I2C_SLAVE_NUM, buff_tx,1,pdMS_TO_TICKS(100));

            vTaskDelay(pdMS_TO_TICKS(50)); // Espero 50 ms para reciba todo.

            #if DEBUG
            printf("Se envia un %i \n",buff_tx[0]);
            #endif

            if(err == ESP_FAIL)
                return ERROR;

            break;
        case SAVE_FLASH_CMD: // Guardar en flash datos
            err = write_nvs(&local_user_data.user_data);
            if(err != ESP_OK)
                return ERROR;

            break;
        case NEXT_SONG_CMD: // Siguiente cancion
            #if DEBUG
            printf("Comando Next Song\n");
            #endif

            esp_avrc_ct_send_passthrough_cmd(0, ESP_AVRC_PT_CMD_FORWARD, ESP_AVRC_PT_CMD_STATE_PRESSED);
            esp_avrc_ct_send_passthrough_cmd(0, ESP_AVRC_PT_CMD_FORWARD, ESP_AVRC_PT_CMD_STATE_RELEASED);

            break;
        case PAUSE_CMD: // Pausar musica
    
```

```

    case PREVIOUS_SONG_CMD: // Anterior cancion
        #if DEBUG
            printf("Comando Previous Song\n");
        #endif
        esp_avrc_ct_send_passthrough_cmd(0, ESP_AVRC_PT_CMD_BACKWARD, ESP_AVRC_PT_CMD_STATE_PRESSED);
        esp_avrc_ct_send_passthrough_cmd(0, ESP_AVRC_PT_CMD_BACKWARD, ESP_AVRC_PT_CMD_STATE_RELEASED);

        break;
    case STOP_SONG_CMD: // Pausar cancion
        #if DEBUG
            printf("Comando STOP Song\n");
        #endif
        static uint8_t fStop = false;
        if(fStop) {
            esp_avrc_ct_send_passthrough_cmd(0, ESP_AVRC_PT_CMD_PLAY, ESP_AVRC_PT_CMD_STATE_PRESSED);
            esp_avrc_ct_send_passthrough_cmd(0, ESP_AVRC_PT_CMD_PLAY, ESP_AVRC_PT_CMD_STATE_RELEASED);
        } else {
            esp_avrc_ct_send_passthrough_cmd(0, ESP_AVRC_PT_CMD_PAUSE, ESP_AVRC_PT_CMD_STATE_PRESSED);
            esp_avrc_ct_send_passthrough_cmd(0, ESP_AVRC_PT_CMD_PAUSE, ESP_AVRC_PT_CMD_STATE_RELEASED);
        }
        fStop = !fStop;
        break;

    default:
        #if DEBUG
            printf("Comando Invalido\n");
        #endif
        return INVALID_COMMAND;
        break;

    return ESP_OK;
}

```

NOTA: para código de bluetooth leer documentación de espressif link [bibliografía](#)

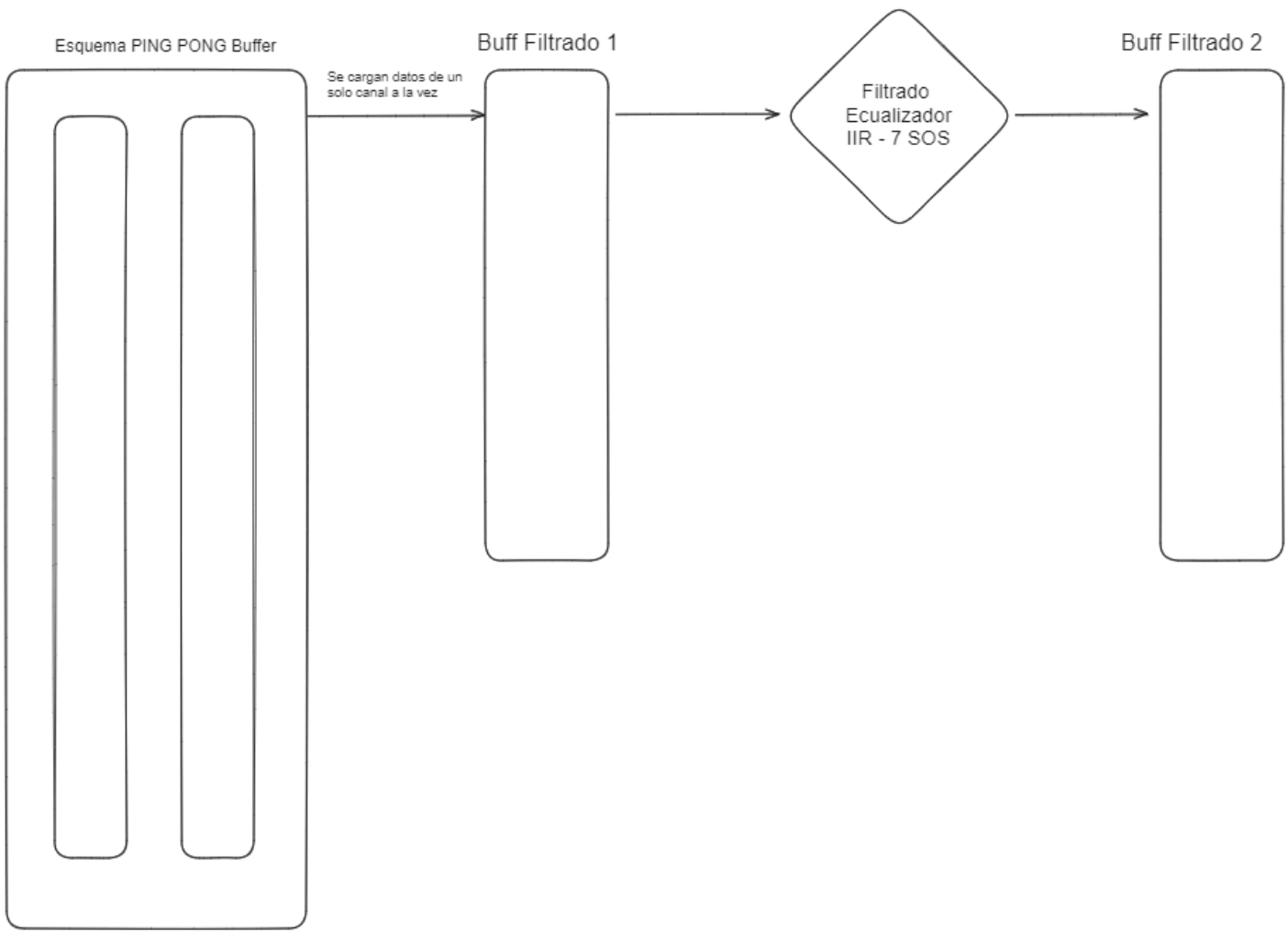
Estructura de procesamiento de datos

Primer etapa

En esta etapa se ecualiza la señal ingresada según las configuraciones determinadas. La ecualización se hace por canal, y solo se tiene un único canal filtrado al mismo tiempo.

Una vez filtrado el canal se utilizan los datos para lo que sea necesario y se descartan los datos para dar lugar al filtrado del otro canal.

En el esquema PING PONG del DMA se reciben datos y se procesan en dos buffers para poder realizar ambas tareas en simultáneo.



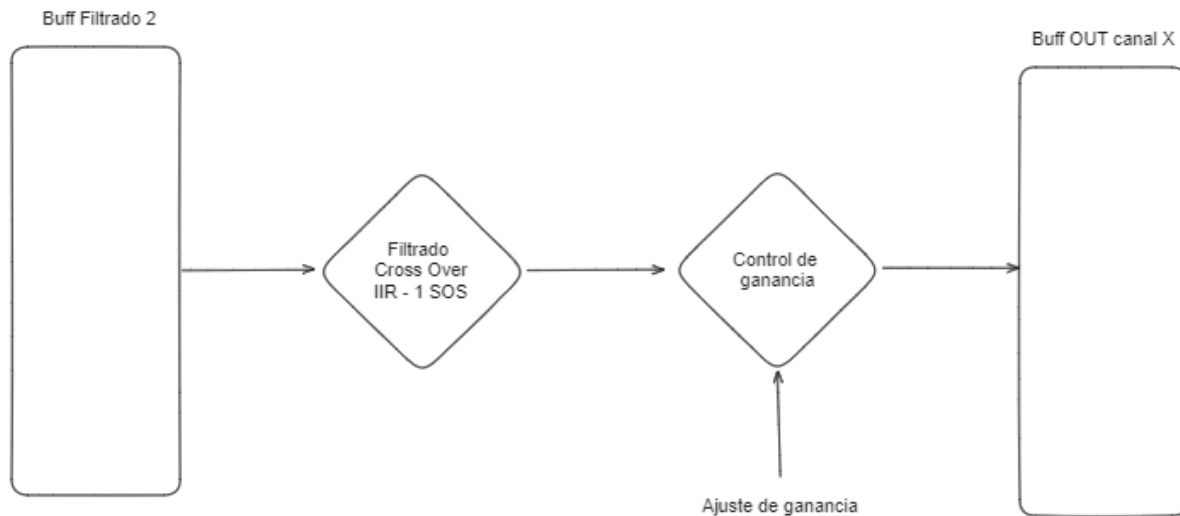
Esquema primer etapa

Segunda etapa

En la segunda etapa del procesamiento se analiza para cada canal de salida cuál es el crossover correspondiente, se aplica el filtrado y luego se aplica el control de ganancia correspondiente. Finalmente se almacena en un buffer de salida que luego es ingresado en DMA para ser enviado.

Es importante que para que se transmita la información a los amplificadores es necesario que se hayan procesado ambas señales.

El buff filtrado 2 corresponderá al canal que tenga asignada dicha salida.



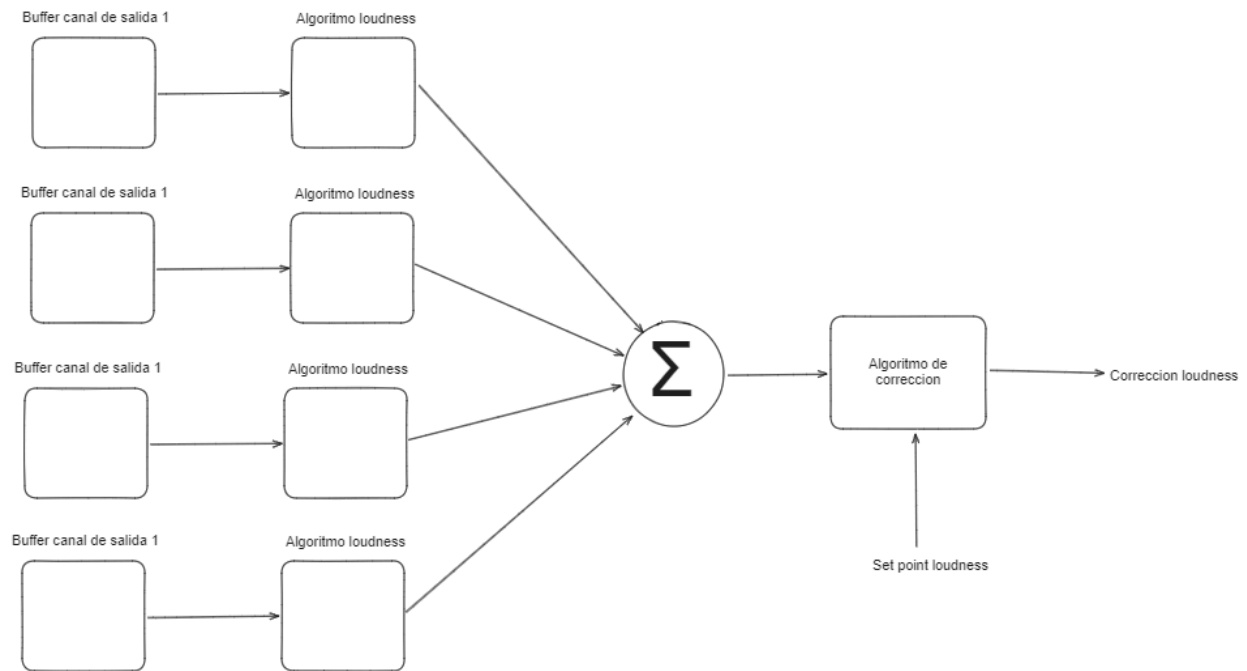
Tercer etapa

Es una etapa que permite medir la potencia acústica de una señal y en base a ello establecer un nivel constante de loudness. El procesamiento realizado consiste en filtrar, y calcular la media cuadrática de los distintos canales de salida. Luego un algoritmo de corrección ajusta la ganancia de los canales para conseguir un volumen deseado.

NOTA: esta etapa no se encuentra actualmente dentro del microcontrolador principal debido a falta de recurso. Sin embargo si implemento dentro del ESP32 manejando únicamente la señal de entrada estéreo.

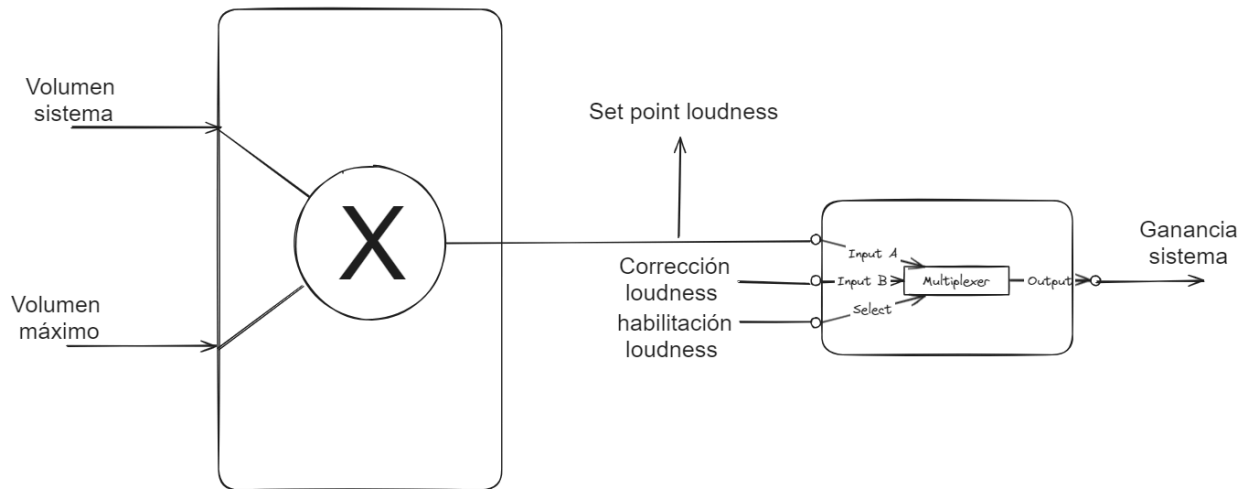
La ventaja de procesar y ajustar la ganancia dentro del ESP32 es que no se carga al microcontrolador principal, como desventaja no se puede tomar la señal que realmente es entregada al usuario.

Esquema original



Manejo de ganancias

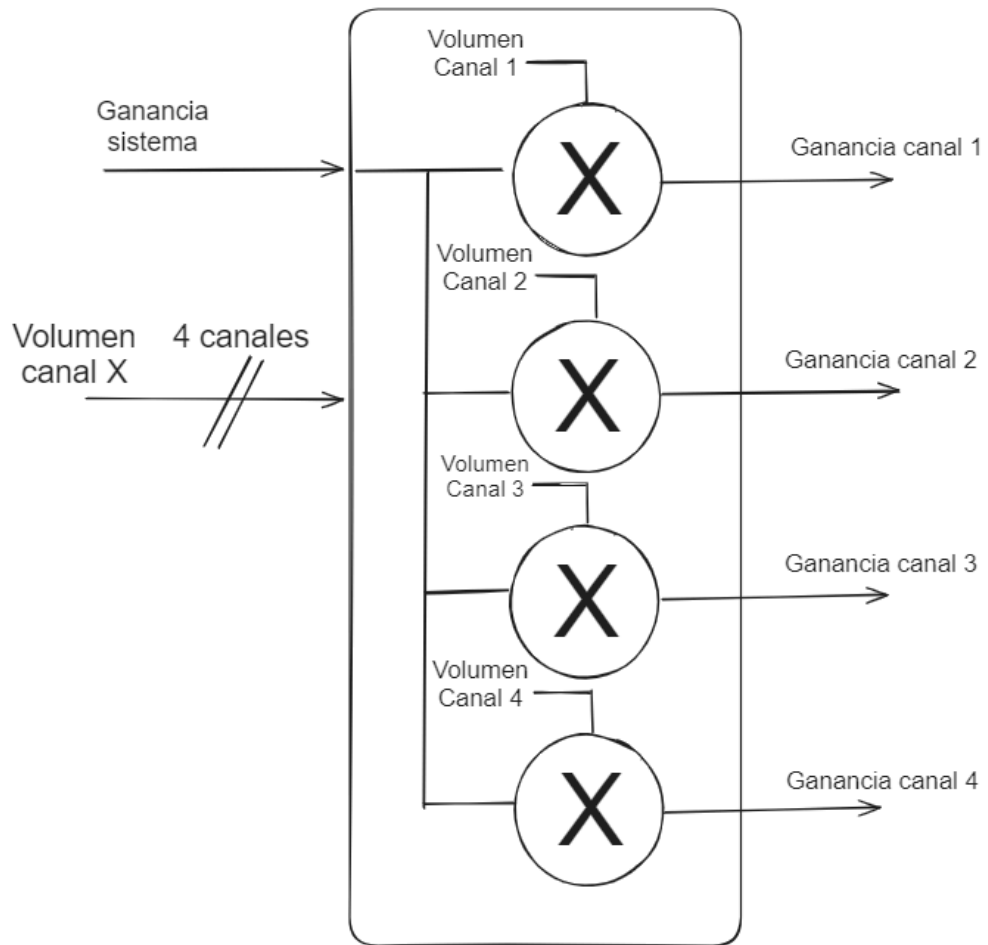
Para el manejo de la ganancia del sistema se tiene el siguiente diagrama, en donde "VOLUMEN SISTEMA" es el volumen determinado en el menú principal y el volumen máximo es el parámetro que se configura en el menú general.



Para el caso de la corrección de loudness la ganancia es implementada dentro del ESP32 mientras que el resto de ganancias es aplicado dentro del ST.

El setpoint del nivel deseado será establecido por el volumen máximo y el volumen del sistema, y el mismo es enviado al ESP32 por medio de I2C seteando así el nivel deseado y corrigiendo este según la medición que haga.

Para establecer la ganancia de cada canal se combinan la ganancia del sistema general con el volumen configurado a cada canal. De ambos se obtiene para cada canal una ganancia que es la que finalmente se aplica a la señal que se envía a los amplificadores.



Consumos eléctricos

Voltaje nominal: 5VDC \pm 0.4v

Corriente nominal: 0,5 a 3 A

Consumo nominal: 4W @ dos canales en uso volumen 50%.

Consumo de máximo: 15W

Consumo en standby: 0.5W

Temperatura En funcionamiento: 50 a 95 °F (10 a 35 °C)

Almacenado: -4 a 114 °F (-20 a 50 °C)

Cálculos generales

No se realizaron cálculos relevantes.

Dispositivos y periféricos empleados.

Para APOLO I se utilizaron una amplia cantidad de dispositivos y periféricos:

- Módulos MAX98357A: se utilizan como DAC y amplificador de audio.
- ESP32: se utiliza como dispositivo de almacenamiento de datos y receptor de datos bluetooth.
- ST Nucleo F401Re: microcontrolador principal, empleado para procesamiento de datos e interfaz usuario.
- ADC: se utiliza para obtener la medición de una pulsación en la pantalla táctil.

Tipos de comunicación empleadas

Las comunicaciones empleadas en APOLO I son las siguientes:

- I2S: utilizado para transmitir y recibir datos desde el ESP32 al Nucleo-401Re, y desde el Nucleo-401Re a los módulos MAX98357.
- SPI: se utiliza para emular una comunicación I2S.

- UART: se utiliza para debuggear ambos dispositivos en caso de reparación. 115200 baudios (8 Bits de datos, Sin bit de paridad, y 1 bit de stop)
- I2C: comunicación de control entre ambos microcontroladores y de guardado de datos.
- Bluetooth: se utiliza para recibir las señales de audio que se quieran reproducir.

Protocolo de control

Para establecer la comunicación entre el ESP32 y el microcontrolador principal se establecen una serie de comandos y protocolos para conseguir una comunicación segura y estable.

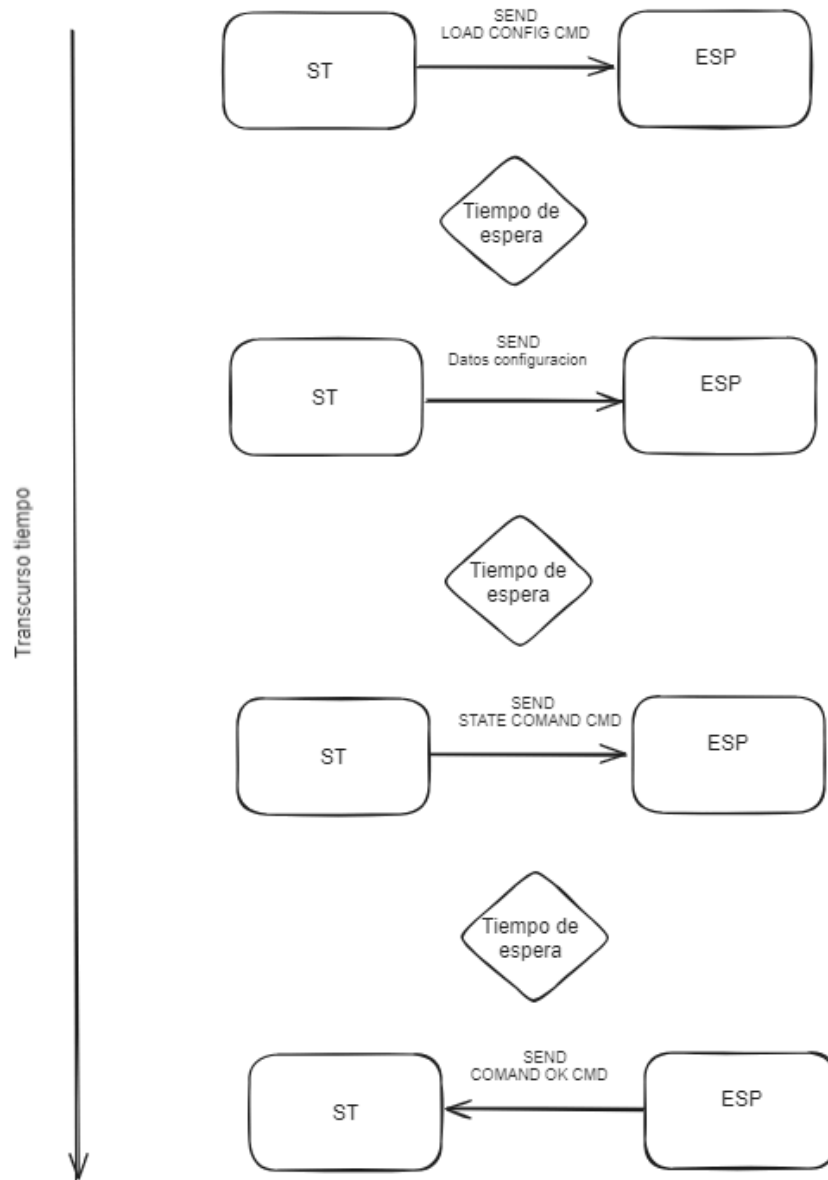
SLAVE ADDRESS: 0x18

Diagrama de flujo - Guardado de datos

Protocolo para enviar datos del usuario al ESP32 y que éste los guarde en FLASH.

COMANDO	CÓDIGO
LOAD_CONFIG_CMD	0x11
STATE_COMAND_CMD	0x13
COMAND_OK	0x33

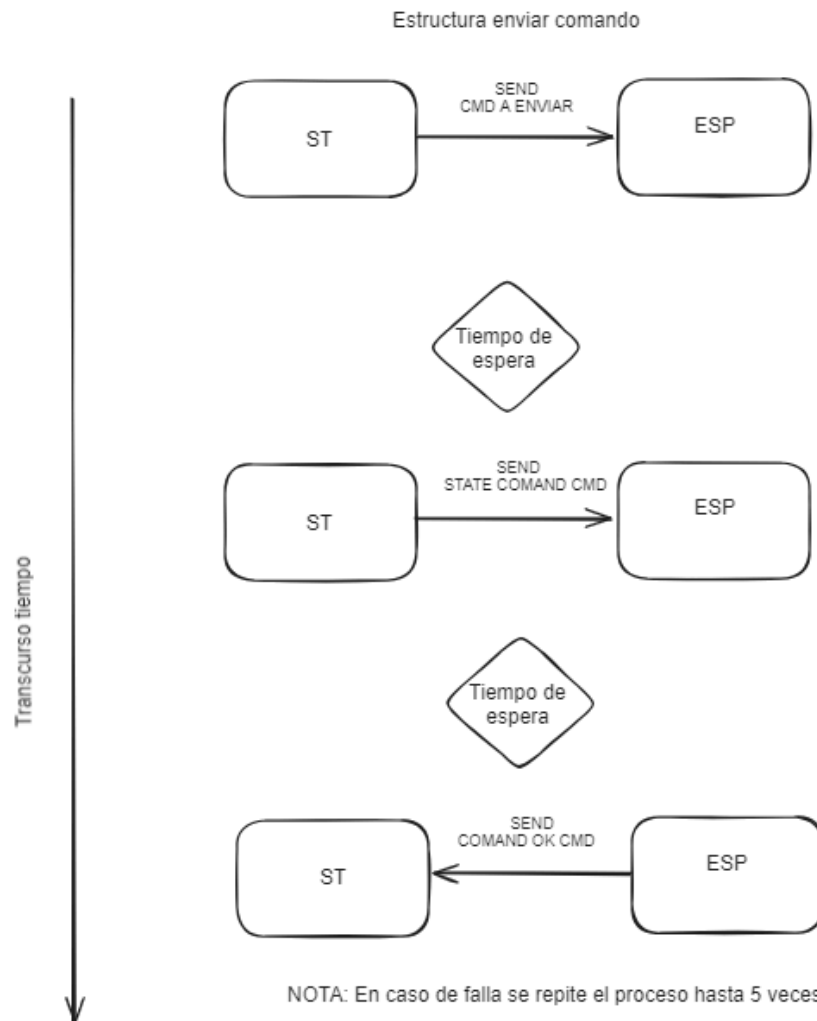
Estructura Guardado de datos



NOTA: En caso de falla se repite el proceso hasta 5 veces

Diagrama de flujo - enviar comando

COMANDO	CÓDIGO
CMD	0xXX
STATE_COMAND_CMD	0x13
COMAND_OK	0x33



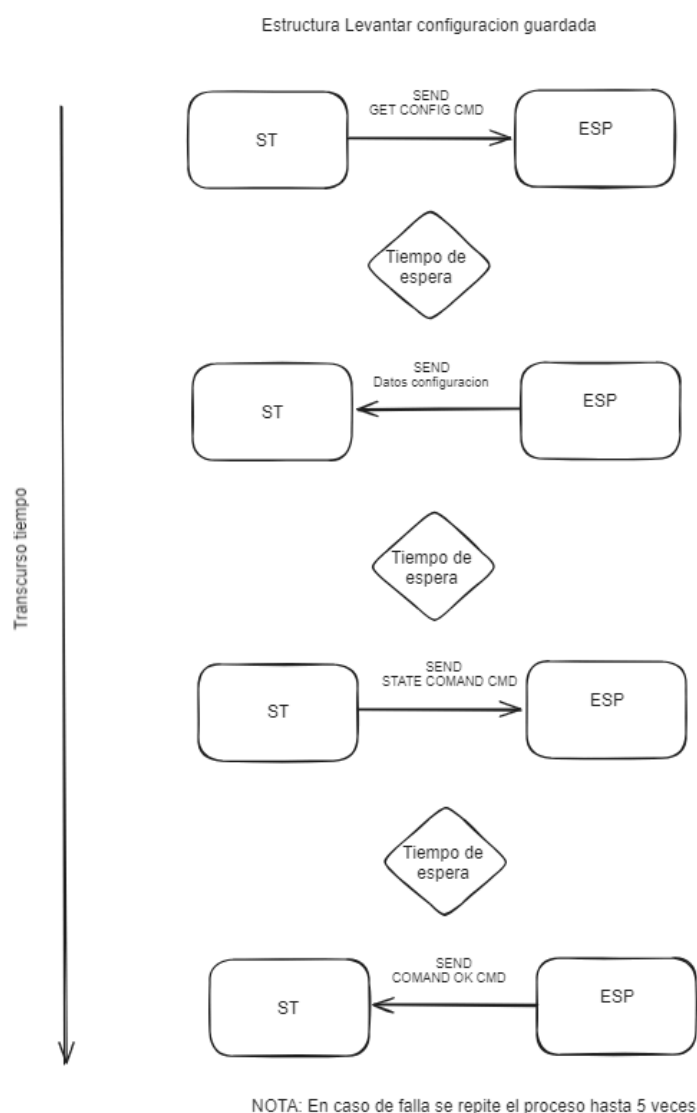
Comandos para enviar:

COMANDO	Descripción	CÓDIGO
SAVE_FLASH_CMD	Indica al ESP32 que debe guardar los datos en flash	0x14
NEXT_SONG_CMD	Indica al ESP32 que debe cambiar a la siguiente canción	0x15
PREVIOUS_SONG_CMD	Indica al ESP32 que debe cambiar a la anterior canción	0x16
STOP_SONG_CMD	Indica al ESP32 que debe parar o arrancar la canción	0x17
LOUD_CONFIG_CMD	Indica al ESP32 que nivel de loudness debe setear	0x18
GET_MUSIC_STATE_CMD	Pregunta al ESP cuál es el tiempo que transcurrió de la canción	0x19
GET_BT_STATE_CMD	Pregunta al ESP cuál es el estado del bluetooth, es decir si esta o no conectado.	0x20

Diagrama de flujo - pedir datos guardados

Protocolo para pedir los datos guardados en el ESP32 (si los hay) y recibirlos en el microcontrolador principal.

COMANDO	CÓDIGO
GET_CONFIG_CMD	0x12
STATE_COMAND_CMD	0x13
COMAND_OK	0x33



Cronogramas de su proyecto

Nicolas	Guido	Ambos	Tercero
---------	-------	-------	---------

Mes 1				Mes 2				Mes 3				Mes 4				Mes 5			
Sem 1	Sem 2	Sem 3	Sem 4	Sem 5	Sem 6	Sem 7	Sem 8	Sem 9	Sem 10	Sem 11	Sem 12	Sem 13	Sem 14	Sem 15	Sem 16	Sem 17	Sem 18	Sem 19	Sem 20
investigación inicial			compras recursos 1		Desarrollo y pruebas bluetooth						diseño comunicación I2C - control	desarrollo guardado datos		diseño estructura de procesamiento					
		anteproyecto				Diseño comunicación audio entre microcontrolador		Diseño de placa				pruebas placa				Integración general			
					Desarrollo drivers y pruebas pantalla			Desarrollo interfaz usuario		pruebas generales interfaz usuario				correcciones estructura (lijado, armado, etc)		desarrollo procesamiento audio			
							Diseño estructura mecánica					impresión 3D							

Costos del proyecto

item	Descripción	Costo unitario (USD)	cantidad	costo total (USD)
Componentes varios	Conector - componentes pasivos - Placa PCB - Cable	\$20,00	1	\$20,00
Módulos MAX98357	Pack 4 modulos + envio	\$20,00	1	\$20,00
ESP32	Módulo Bluetooth	\$12,00	1	\$12,00
STM-Nucleo-F401RE	Microcontrolador principal	\$25,00	1	\$25,00
Carcaza	Estructura impresa en 3D	\$15,00	1	\$15,00
Tornilleria	tornillos, tuercas y arandelas	\$3,00	1	\$3,00
Pantalla	Pantalla TFT resistiva táctil 3.5"	\$10,00	1	\$10,00
Hora hombre	Hora hombre	\$8,00	500	\$4.000,00
Logísticas	Traslados y envíos	\$30,00	2	\$60,00
Herramientas	Herramientas varias	\$30,00	1	\$30,00
Instrumentos laboratorio	Instrumentos requeridos para llevar a cabo el trabajo (analizador lógico, osciloscopio, fuentes, GAF, etc)	\$500,00	1	\$500,00

item	importe
Total sin impuestos (USD):	\$4.695,00
Total en pesos:	\$4.695.000,00
Total impuestos:	\$985.950,00
Total con impuestos:	\$5.680.950,00
Cotizacion dolar:	\$1.000,00

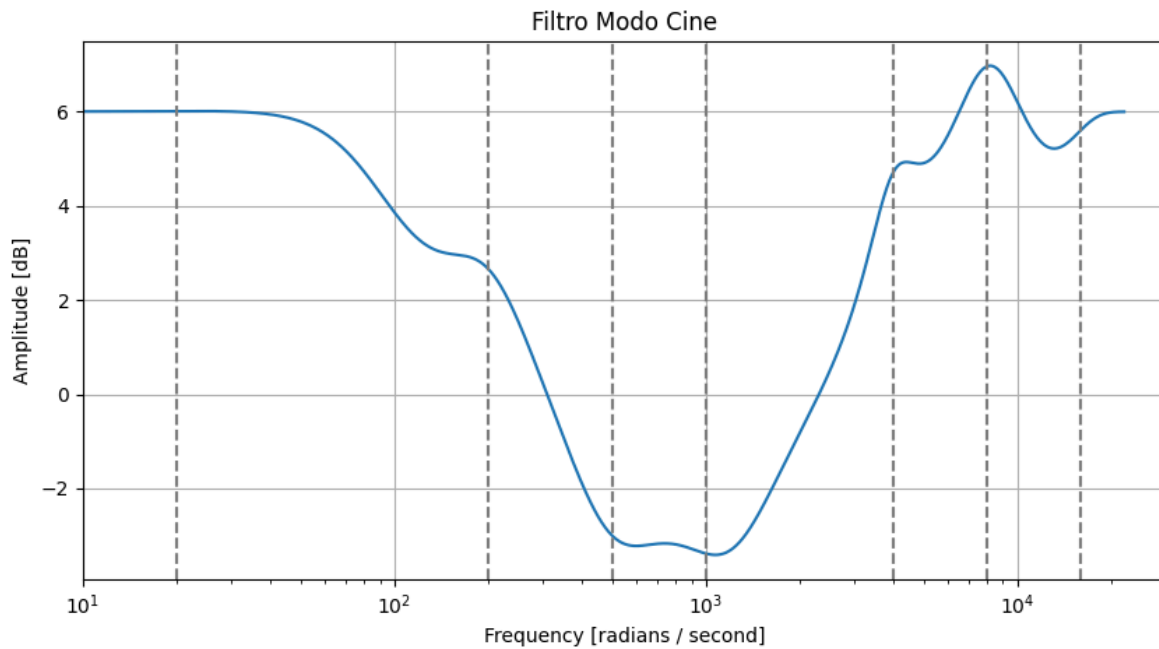
Filtros sistema

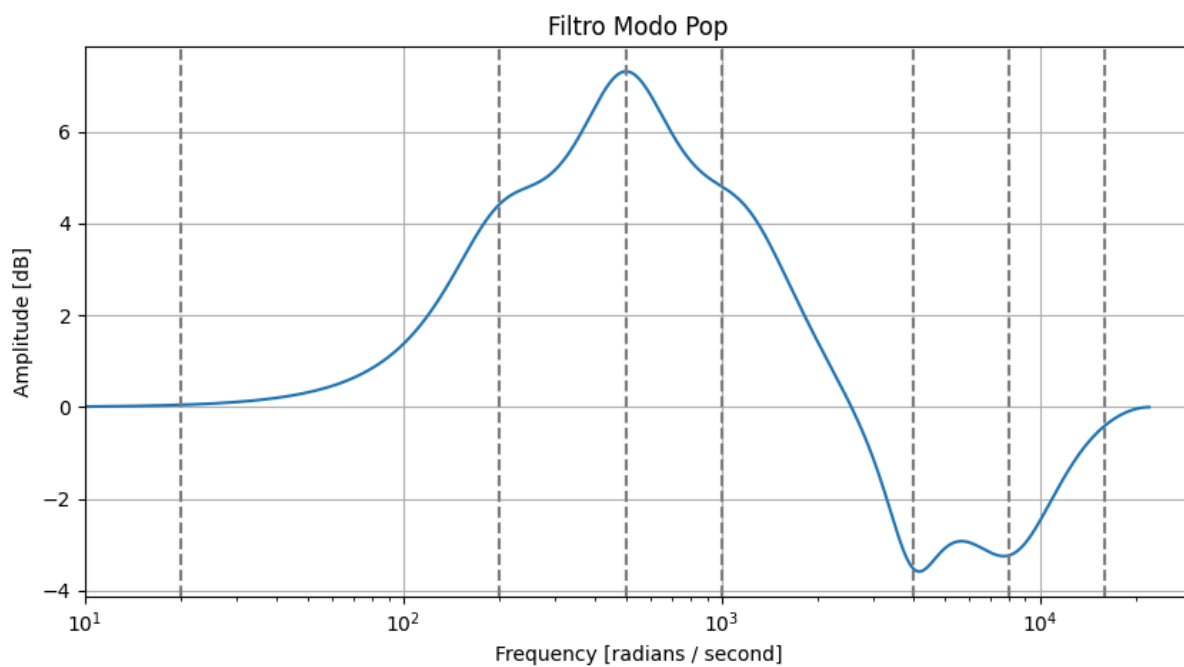
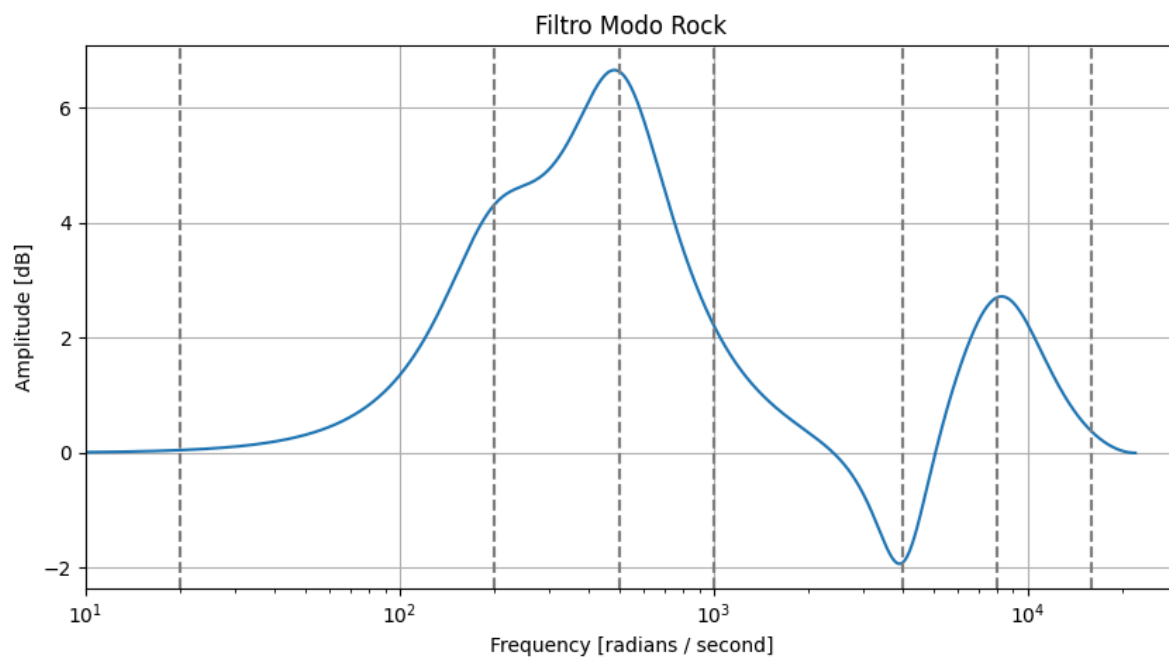
Para APOLO I se deben implementar una serie de filtros tanto de ecualización como de crossover. Para el diseño de estos filtros se utilizó la guía de “EQ cookbook RBJ” en donde se proveen funciones de diseño para poder obtener los coeficientes de los filtros en forma económica computacionalmente.

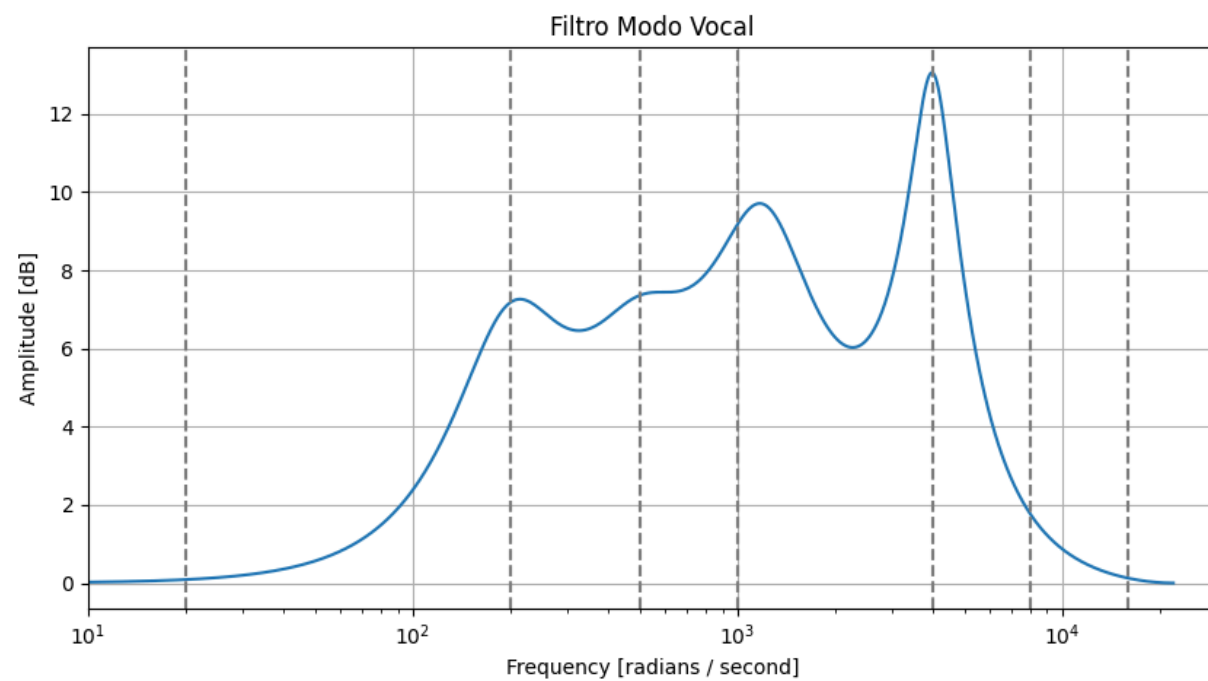
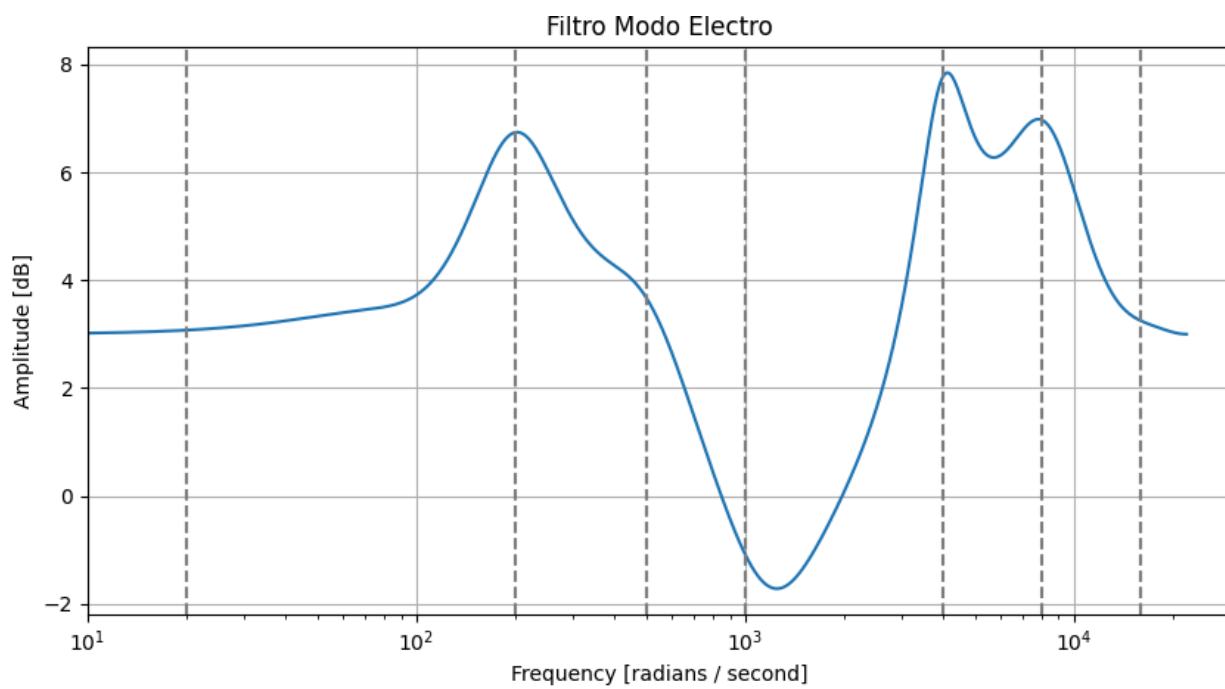
Esta implementación permite la capacidad de ajustar los filtros en forma simple y agregar nuevas configuraciones sin excesivos cambios en el código.

A continuación se muestra la respuesta en frecuencia de todos los filtros utilizados en esta revisión.

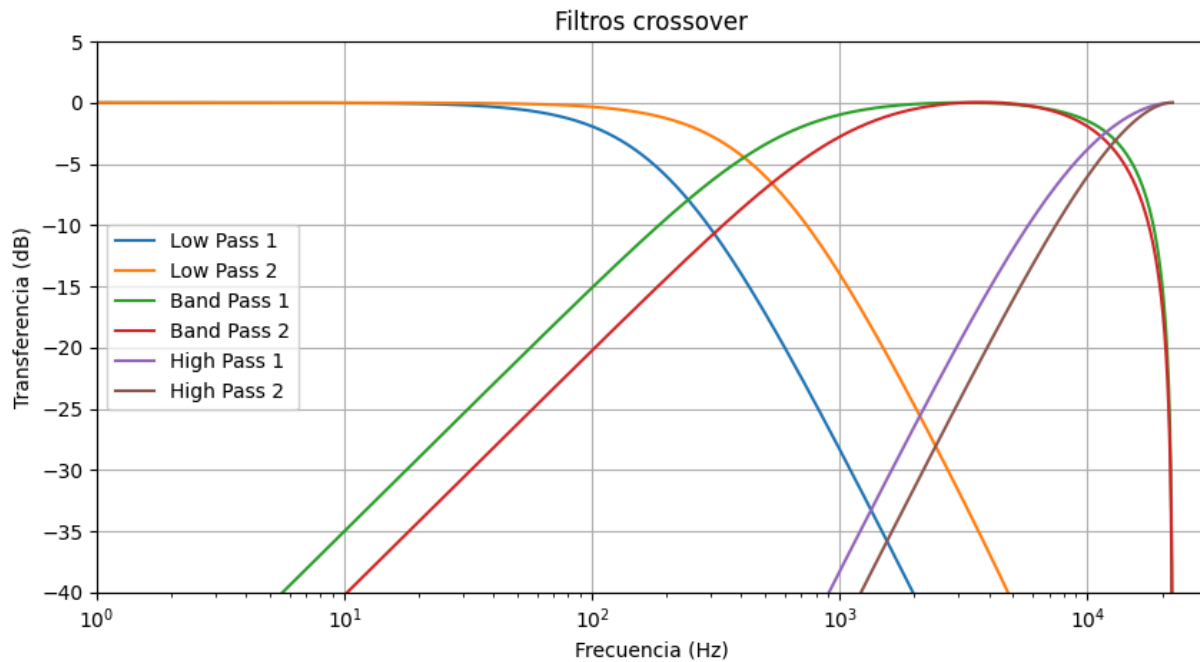
Filtros de ecualización







Filtros de crossOver




Desafíos y problemáticas

Problemáticas

- Falta de puertos I2S
- Ausencia de pantallas SPI
- Calibración de panel Táctil y toques "fantasma"
- Problema en recepción de datos
- Falta de recursos interfaz gráfica
- Falta de recursos procesamiento digital

Soluciones

Falta de puertos I2S: Una gran limitación del proyecto fue la ausencia de un tercer I2S. Para poder paliar esta ausencia se utilizó una emulación de puerto I2S mediante SPI y un timer. Debido a la falta de un clock de audio la salida por SPI no tuvo buenos resultados.



Ausencia de pantallas SPI: Debido a la ausencia en el mercado de pantallas SPI de un tamaño considerable a un costo acorde al proyecto se tuvo que optar por una pantalla con comunicación paralelo. Como consecuencia de esta elección se tuvo que hacer un mayor proceso de selección de drivers y el refresco de la pantalla no es tan fluido como se hubiese deseado.

Calibración de panel Táctil y toques “fantasma”: Por defecto los paneles táctiles de este tipo de dispositivos suelen ser de baja calidad o con algunas deficiencias. Para poder obtener resultados aceptables se utilizó la nota de aplicación de ATMEL para paneles de cuatro hilos y se realizó un algoritmo que permite filtrar aquellos toques que no eran realizados por el usuario.

Además se realizó una calibración específica para este display. Es importante que al cambiar el mismo es necesario realizar una re calibración para poder tener una experiencia adecuada.

Problema en recepción y transmisión de datos I2S: Poder elegir una frecuencia de transmisión de 44.1kHz fue un desafío dado que no era tan sencilla la configuración de los clocks para poder conseguir dicho resultado. Debido a que el clock no es un múltiplo exacto de la frecuencia de muestreo resulta imposible conseguir una sincronización perfecta, lo que resulta en ciertas distorsiones debido a la desincronización de la señal.


Falta de recursos interfaz gráfica: Si bien TOUCHGFX permite hacer un manejo eficiente de los recursos, el microcontrolador usado tiene una disponibilidad reducida de memoria flash. Esto impide poder usar una cantidad de widgets elevada en la interfaz provocando que sea poco estética en ciertos casos. Para solucionar la falta de memoria se llevó al mínimo la cantidad de imágenes que requería la interfaz.

Falta de recursos procesamiento digital: Inicialmente se planteó agregar una opción de ajuste de loudness automático a la salida de todos los canales, sin embargo los recursos del microcontrolador no fueron suficientes para poder implementarlo. Se optó por realizar el control de loudness directamente en el ESP32 con la señal de entrada que recibe el nucleoF401RE

Posibles mejoras para una V2.0

Problemática:

La interacción con la pantalla actual, que utiliza tecnología resistiva, presenta desafíos significativos en términos de fluidez y eficiencia. La naturaleza de esta pantalla dificulta una experiencia del usuario ágil, y la implementación de DMA (Acceso Directo a Memoria) se complica debido a las peculiaridades del protocolo de comunicación. Además, la limitación



inherente de la pantalla resistiva impide la variación del brillo, lo que podría afectar la visibilidad y la adaptabilidad del dispositivo en entornos cambiantes.

Posible Mejora:

Con el objetivo de superar estos obstáculos, se propone migrar a una tecnología de pantalla más avanzada, como OLED o LCD con panel táctil capacitivo. Este cambio no solo mejoraría la fluidez de la interacción, sino que también permitiría una mayor flexibilidad en la implementación de DMA mediante una comunicación simplificada a través de SPI (Interfaz Periférica en Serie). La utilización de SPI facilita la transmisión eficiente de datos mediante DMA, evitando transiciones lentas y optimizando el rendimiento general.

Esta mejora no sólo resolvería los problemas actuales de interacción y comunicación, sino que también brindaría la oportunidad de explorar funcionalidades adicionales, como la variación de brillo, mejorando aún más la versatilidad y la calidad de la interfaz de usuario. Este enfoque estratégico busca no solo abordar las limitaciones actuales sino también sentar las bases para futuras mejoras en la experiencia del usuario.


Problemática:

La elección actual del microcontrolador principal revela limitaciones significativas que han afectado el desarrollo y desempeño del proyecto. La memoria FLASH de capacidad limitada ha condicionado numerosas implementaciones, restringiendo la complejidad del software y afectando la capacidad de almacenamiento y gestión de datos cruciales para el funcionamiento del sistema. Además, la escasez de puertos I2S ha impactado directamente en la conectividad y el procesamiento de audio, limitando la integración con dispositivos externos y la manipulación eficiente de señales de alta calidad. Asimismo, la insuficiencia de la memoria RAM ha generado desafíos adicionales, limitando la capacidad del sistema para gestionar datos temporales y operaciones simultáneas.

Posible Mejora:

Para superar estas limitaciones, se propone la adopción de un microcontrolador más adecuado para el proyecto. Este nuevo microcontrolador debería contar con una memoria FLASH de mayor capacidad para abordar las restricciones actuales de almacenamiento de programas y datos. Además, una mayor cantidad de puertos I2S mejorarán la conectividad y el procesamiento de audio, facilitando la integración y manipulación de señales.

Asimismo, la elección de un microcontrolador con una memoria RAM más amplia resolverá los desafíos relacionados con la gestión eficiente de datos temporales y operaciones concurrentes,



proporcionando una base sólida para desarrollos más avanzados y aplicaciones más exigentes en términos de recursos.

La revisión y selección cuidadosa de un microcontrolador más adecuado abordará de manera integral las limitaciones actuales, permitiendo un desarrollo más robusto y versátil del proyecto a corto y largo plazo.

Posible Mejora:

Una mejora sustancial para el proyecto implica la transición hacia un microcontrolador que incorpora Bluetooth de manera integrada. La elección de un microcontrolador con funcionalidades Bluetooth incorporadas no sólo permitirá una notable reducción del espacio en la placa, sino que también mejorará significativamente los tiempos de transmisión entre dispositivos.

La integración nativa del Bluetooth eliminará la necesidad de componentes externos y simplificará el diseño, optimizando el espacio disponible en la placa. Esta mejora no solo contribuirá a la eficiencia física del proyecto, sino que también proporcionará beneficios tangibles en términos de velocidad y estabilidad en las comunicaciones inalámbricas.

Conclusiones

APOLO I es una buena primera versión para este producto. Los resultados fueron acordes a lo esperado y se cumplieron las expectativas básicas del proyecto.

Para una segunda implementación se mencionaron en el apartado de [Posibles mejoras para una V2.0](#) correcciones para mejorar los resultados obtenidos.

Se pudo lograr alcanzar los objetivos principales que fueron:

- Establecer comunicación bluetooth y controlarla.
- Procesamiento en tiempo real de datos de audio incluyendo transmisión y recepción entre microcontroladores.
- Diseño adecuado de estructura y placa.
- Interfaz usuario cómoda y gráfica.
- Posibilidad de personalización para el usuario.
- Correcta integración entre distintos procesos.

Como falencias se pueden destacar:

- Hardware de interfaz usuario deficiente.

- Elección del microcontrolador no adecuada para la aplicación.
- Estructura lógica poco eficiente en desarrollo (necesidad de utilizar dos microcontroladores).
- Falta de integración de opciones debido a deficiencias en el hardware.

En conclusión los resultados obtenidos fueron adecuados para una primera versión pero todavía se requieren mejoras para poder ser un producto viable.

Bibliografía

- [Application note: "I2S protocol emulation on STM32L0 Series microcontrollers using a standard SPI peripheral" - STelectronics](#)
- [Audio EQ Cookbook - Robert Bristow-Johnson](#)
- [Application note: "A basic Guide to I2C" - Texas Instrument](#)
- [Application note: "AVR341: Four and five-wire Touch Screen Controller" - ATME](#)
- [Espressif - Bluetooth® API](#)
- [Espressif - Inter-Integrated Circuit \(I2C\)](#)
- [Espressif - Non-Volatile Storage](#)
- [Application note: "Increased Data Transfer Rate Using Ping-Pong Buffering" - Texas instrument](#)
- [FreeRTOS documentation](#)
- [TouchGFX documentation](#)
- [CMSIS - DSP documentation](#)

Datasheet

- [Raydium - RM68140 \(display controller\)](#)
- [Analog Devices - MAX98357 \(I2S PCM Class D Amplifier\)](#)
- [ST Electronics - Nucleo F401Re \(documentation\)](#)
- [Espressif - ESP32](#)