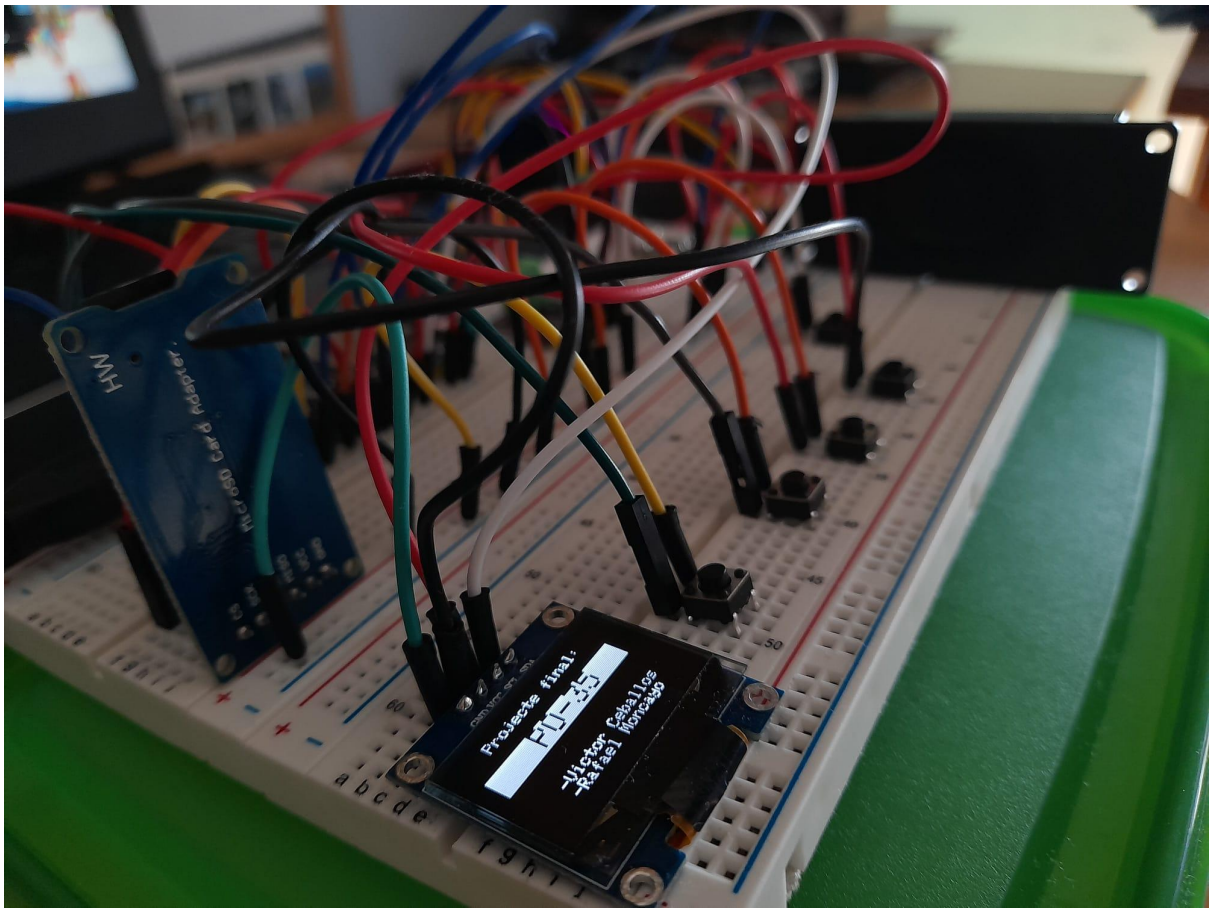
Proyecto final Procesadores digitales:

PO-35



-Victor Ceballos Fouces
-Rafael E. Moncayo Palate

Victor Ceballos

Rafael E. Moncayo

Introducción:

La intención de este proyecto, desde buen inicio, siempre ha sido poder trabajar con las opciones que nos ofrece la ESP32 destinada al audio y a su procesado. Y, como personas que estamos metidas dentro del mundo del hardware de audio, se nos encendió la bombilla rápidamente cuando nos dimos cuenta de que podríamos hacer un intento de réplica de un sintetizador real llamado PO-35 que está destinado a la sintetización de la voz.*
Los PO (Pocket Operator) son una familia de sintetizadores muy pequeños de la empresa Teenage Engineering. Estos aparte de poder sintetizar sonido vienen con capacidad de grabar, reproducir y secuenciar el audio introducido. Nosotros decimos que nuestro proyecto es un "intento" de lo que es el PO-35 porque es bastante obvio que no podríamos llegar a hacer un componente de este calibre, pero al menos intentarlo con las opciones mínimas de grabadora que ofrece el hardware junto con un menú en el display.

Material:

El montaje requiere de un material bastante extenso pero que no ha sido complicado conseguir porque más de la mayoría lo teníamos gracias a prácticas anteriores.
Así que, enumerando todo, para la realización de este proyecto necesitamos:

Una protoboard grande
Muchos cables Macho x Macho
Muchos cables Hembra x Macho
Un cable USB mini
5x botones
1x Display
1x SD XC mini
1x Lector de SD mini
1x microfono (MEM o Analog)
1x amplificador
1x speaker
1x ESP32

Victor Ceballos

Rafael E. Moncayo

Código:

Main:

RECORD:

Función void() donde iniciamos la grabación y la guardamos en un archivo .wav dentro de la SD. Los parámetros de entrada son: I2S Sampler *input (señal entrada I2S del micrófono) y const char *fname (nombre del fichero).

```cpp
void record(I2SSampler *input, const char *fname)
```

Primero inicializamos el objeto input con start()
instalando los drivers I2S y configurando los pines. Seguidamente abrimos un archivo dentro de la SD y creamos un WAV FILEWriter con el que empezar a escribir sobre el, pasandole el nombre del mismo y el sample rate del input del micrófono (m_i2s_config).

```cpp
int16_t *samples = (int16_t *)malloc(sizeof(int16_t) * 1024);
  ESP_LOGI(TAG, "Start recording");
  input->start();

  FILE *fp = fopen(fname, "wb");

  WAVFileWriter *writer = new WAVFileWriter(fp, input->sample_rate());
```

Después establecemos el bucle para solo hacer la lectura/escritura mientras se mantenga pulsado el botón. Vamos leyendo la señal I2S en el contenedor samples a la vez que lo escribimos con el writer y contando también el tamaño total del archivo escrito en la SD. También se escribe por pantalla el número total de samples escritos y el tiempo
que se ha tardado.

```cpp
while (gpio_get_level(GPIO_BUTTON) == 1)
  {
    int samples_read = input->read(samples, 1024);
    int64_t start = esp_timer_get_time();
    writer->write(samples, samples_read);
    int64_t end = esp_timer_get_time();
    ESP_LOGI(TAG, "Wrote %d samples in %lld microseconds",
samples_read, end - start);
  }
```

Finalmente para acabar la función paramos el input del I2S Sampler, paramos la escritura del writer, cerramos el archivo de la SD y liberamos memoria borrando objetos y limpiando el contenedor de samples.

```cpp
input->stop();

  writer->finish();
  fclose(fp);
  delete writer;
  free(samples);
  ESP_LOGI(TAG, "Finished recording");
}
```

PLAY:

Función void() donde leemos el .wav escrito anteriormente en la SD y lo reproducimos por el speaker. Los parámetros de entrada són: Output *output (señal salida I2S del speaker) y const char *fname (nombre del fichero).

```cpp
void play(Output *output, const char *fname)
```

Para empezar abrimos el archivo dentro de la tarjeta y creamos un objeto tipo WAV FileReader que apunte a este para empezar a leerlo. Para esto solo necesitamos pasarle el nombre del archivo. Seguimos inicializando output
con start() instalando los drivers I2S y configurando los pines, además de pasarle el sample_rate asociado al reader.

```cpp
int16_t *samples = (int16_t *)malloc(sizeof(int16_t) * 1024);

  FILE *fp = fopen(fname, "rb");

  WAVFileReader *reader = new WAVFileReader(fp);
  ESP_LOGI(TAG, "Start playing");
  output->start(reader->sample_rate());
  ESP_LOGI(TAG, "Opened wav file");
```

Con el .wav abierto y todo preparado para empezar, declaramos el bucle que leerá el archivo hasta que no haya más muestras. Dentro del while, con **samples_read** almacenamos las muestras totales usando el reader (función **read**()). Si no encontramos samples paramos el bucle e indicamos que no hay audio. Después
indicamos el número total de muestras leídas y finalmente con la función write del output preparamos los datos a la vez que los vamos escribiendo en el periférico I2S, en nuestro caso el speaker, reproduciendo así el audio del archivo .wav hasta que no haya más muestras.

```cpp
while (true)
  {
    int samples_read = reader->read(samples, 1024);
    if (samples_read == 0)
    {
      break; Serial.print("No hay audio");
    }
    ESP_LOGI(TAG, "Read %d samples", samples_read);
    output->write(samples, samples_read);
    ESP_LOGI(TAG, "Played samples");
  }
```

Para acabar simplemente paramos el output, cerramos el archivo, borramos el reader y liberamos memoria de samples.

```cpp
output->stop();
  fclose(fp);
  delete reader;
  free(samples);
  ESP_LOGI(TAG, "Finished playing");
}
```

DISPLAY MENÚ:

Esta es una función que se repetirá todo el rato porque lo que nos interesa es que se mantenga el menú en el Display de manera continua y por eso será la única acción que estará en el **void loop()**. No hace falta meterle nada por parámetros porque todo lo que usa el menú (altavoz, micro, display, botones,…) estará declarado fuera o dentro de este. Las primeras líneas de código nos permiten definir las variables que nos representarán los botones.Estos los guardaremos como valores enteros sacados del digitalread del pin en el que estén conectados.

```cpp
  int down = digitalRead(25);
  int up = digitalRead(33);
  int enter = digitalRead(19);
  int back = digitalRead(5);
```

Como podemos ver usaremos 4 botones: down, up, enter, back, los cuales nos van a ayudar a movernos a través del menú.
Los **if**s que hay a continuación hacen lo comentado previamente: ayuda a seleccionar y a entrar dentro de las 3 diferentes opciones.Estos trabajan con los valores **selected** y

entered que nos ayudan a plantear las diferentes posibilidades dentro del menú. **Selected** nos ayuda a ver cómo se seleccionan los valores en el display y porque cambian de color, y el **enter** nos ayuda a entrar en la opcion de menú que nos interesa.

```cpp
for (int i = 0; i < 3; i++) {
      if (i == selected) {
        display.setTextColor(SSD1306_BLACK, SSD1306_WHITE);
        display.println(options[i]);
      } else if (i != selected) {
        display.setTextColor(SSD1306_WHITE);
        display.println(options[i]);
      }
    }
```

Cuando estamos en la opción **enter==-1** estaremos en el menú inicial en el que proporcionamos este bucle for en el que cambiamos el color del fondo y de la letra con el **display.setTextColor()** cada vez que **selected** coincida con cada opción definida de la siguiente manera:

```cpp
const char *options[3] = {
    " 1.- Grabar",
    " 2.- Reproducir",
    " 3.- Creditos"
  };
```

Cuando se entra dentro de cualquiera de las posibilidades se hace un **display.clearDisplay()** para poder limpiar la pantalla y tener una pequeña presentación de la opción de menú que hemos escogido.
En el documento de **config.h** ya habremos definido un botón extra que servirá para ejecutar la función de interés: si es grabar se usará para grabar, y si es reproducir para reproducir, valga la redundancia.

```cpp
// multifunction button
#define GPIO_BUTTON GPIO_NUM_23
```

Si la elección del usuario es Grabar, se creará un nuevo micrófono gracias a la librería de **I2SSampler** (en la que se crearà el Sampler I2S) y el **I2SMEMSampler** (MEMs:Micro Electro Mechanical System) y por parámetros al constructor se le introducirán los puertos I2S, los pines definidos en el "**config.cpp**" y la configuración del micrófono.
El código también nos permite, a falta de un micrófono MEM, poder crear un nuevo input con entrada analógica.
En el caso de que se presione el botón, por tanto si el **GPIO_BUTON** está a nivel alto, se usará la función **record** para poder grabar.

```cpp
    /*ESP_LOGI(TAG, "Creating microphone");*/
    #ifdef USE_I2S_MIC_INPUT
      I2SSampler *input = new I2SMEMSSampler(I2S_NUM_0, i2s_mic_pins,
i2s_mic_Config);
    #else
      I2SSampler *input = new ADCSampler(ADC_UNIT_1, ADC1_CHANNEL_7,
i2s_adc_config);
    #endif
    while (gpio_get_level(GPIO_BUTTON) == 1){
        /*wait_for_button_push();*/
          record(input, "/sdcard/test.wav");
    }
```

Algo parecido sucederá con la alternativa **Reproducir**. Se creará el altavoz y si el botón **GPIO_BUTON** está a nivel alto se usará la función **play** para poder reproducir el último sonido capturado por el micrófono que estará guardado en la SD.

```cpp
#ifdef USE_I2S_SPEAKER_OUTPUT
      Output *output = new I2SOutput(I2S_NUM_0, i2s_speaker_pins);
    #else
      Output *output = new DACOutput(I2S_NUM_0);
    #endif

    while (gpio_get_level(GPIO_BUTTON) == 1)
      {
        //spectrum_analyzer();

        play(output, "/sdcard/test.wav");

      }
```

Como opción final tenemos los créditos donde simplemente saldrá el nombre de los integrantes del grupo que ha hecho este proyecto y un pequeño comentario por parte de los mismos.

SETUP():

En primera instancia se hace un Serial.begin(115200) y seguidamente se crea el objeto de la SD, el código también da la opción de que se pueda usar la memoria interna de la Esp 32 (SPIFFS). La SD se crea con los siguientes parámetros: el nombre de la dirección de la SD, y los pines SPI para comunicarnos con el lector de tarjetas.

```cpp
 Serial.begin(115200);

  ESP_LOGI(TAG, "Starting up");

#ifdef USE_SPIFFS
  ESP_LOGI(TAG, "Mounting SPIFFS on /sdcard");
  SPIFFS.begin(true, "/sdcard");
#else
  ESP_LOGI(TAG, "Mounting SDCard on /sdcard");
  new SDCard("/sdcard", PIN_NUM_MISO, PIN_NUM_MOSI, PIN_NUM_CLK,
PIN_NUM_CS);
#endif
```

Seguidamente se definen los 4 botones del menú y pasamos a limpiar el display y a que salga una presentación del proyecto de 4 segundos.

```cpp

  pinMode(25, INPUT_PULLUP);
  pinMode(33, INPUT_PULLUP);
  pinMode(19, INPUT_PULLUP);
  pinMode(5, INPUT_PULLUP);

  if (!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
    Serial.println(F("SSD1306 allocation failed"));
    for (;;); // Don't proceed, loop forever
  }
  display.clearDisplay();
  // Draw a single pixel in white
  display.drawPixel(10, 10, SSD1306_WHITE);
  display.display();
  delay(2000); // Pause for 2 seconds

  //sampling_period_us = round(1000000 * (1.0 / SAMPLING_FREQUENCY));
Para FFT

  display.clearDisplay();
  display.setTextColor(SSD1306_WHITE);
```

```
  display.setCursor(0, 0);
  display.setTextSize(1);
  display.println("   Projecte final:\n");
  display.setTextSize(2);
  display.setTextColor(SSD1306_BLACK, SSD1306_WHITE);
  display.println("   PO-35  \n");
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.println("   -Victor Ceballos");
  display.println("   -Rafael Moncayo");
  display.display();
  delay(4000);
```

Librerías:

Ahora comentaremos por encima las librerías que usamos en el código principal y un poco más en profundidad las que nos parecen más importantes para la correcta comprensión del funcionamiento del código principal .

En todos los .h hay un `#pragma once` que hace que el fichero solo se incluya en el programa una vez en una sola compilación.

WAV:

El **WAVFile.h** nos declara, sobre todo, lo que es el header del fichero wav.

```cpp
_wav_header()
  {
    riff_header[0] = 'R';
    riff_header[1] = 'I';
    riff_header[2] = 'F';
    riff_header[3] = 'F';
    wave_header[0] = 'W';
    wave_header[1] = 'A';
    wave_header[2] = 'V';
    wave_header[3] = 'E';
    fmt_header[0] = 'f';
    fmt_header[1] = 'm';
    fmt_header[2] = 't';
    fmt_header[3] = ' ';
    data_header[0] = 'd';
    data_header[1] = 'a';
    data_header[2] = 't';
    data_header[3] = 'a';
  }
```

La classe **WAVFileWriter.cpp**, tal y como dice el nombre, es nuestro soporte fundamental a la hora de grabar, ya que serà lo que guarde los datos directamente del micrófono a un fichero WAV para meter dentro de la SD.
Esta clase, como es de ver venir, se vale mucho de la clase **FILE** para poder escribir el fichero encima de la cabecera ya existente. Es por eso que vemos tanto **fwrite.**

```cpp
WAVFileWriter::WAVFileWriter(FILE *fp, int sample_rate)
{
  m_fp = fp;
  m_header.sample_rate = sample_rate;
  // write out the header - we'll fill in some of the blanks later
  fwrite(&m_header, sizeof(wav_header_t), 1, m_fp);
  m_file_size = sizeof(wav_header_t);
}

void WAVFileWriter::write(int16_t *samples, int count)
{
  // write the samples and keep track of the file size so far
  fwrite(samples, sizeof(int16_t), count, m_fp);
  m_file_size += sizeof(int16_t) * count;
}
```

Y pasará algo muy parecido con el WAVFileReader. Que es la parte de la clase que nos será útil para poder leer el fichero WAV existente de dentro de la SD.

```cpp
int WAVFileReader::read(int16_t *samples, int count)
{

    size_t read = fread(samples, sizeof(int16_t), count, m_fp);
    return read;
    //lee datos de un stream sado dentro del puntero de array.
}
```

SD():

Aquí definimos el constructor de la clase SDCard para nuestra tarjeta, reemplazando la librería SD original. Primero se declara el mount_point donde montaremos la SD y una variable ret que se usará para detectar errores durante el proceso.

```cpp
SDCard::SDCard(const char *mount_point, gpio_num_t miso, gpio_num_t mosi, gpio_num_t clk, gpio_num_t cs)
{
  m_mount_point = mount_point;
  esp_err_t ret;
```

A continuación con mount_config retocamos algunas opciones para montar el sistema de ficheros. format_if_mount_failed a true nos permitirá formatear la SD en caso de que falle en el montaje, max_files limita el máximo a 5 y .allocation_unit_size define el tamaño del ubicación en nuestra unidad.

```cpp
esp_vfs_fat_sdmmc_mount_config_t mount_config = {
      .format_if_mount_failed = true,
      .max_files = 5,
      .allocation_unit_size = 16 * 1024};
```

Empezamos con la inicialización de la tarjeta: con sdmmc_host_t host = **SDSPI_HOST_DEFAULT**() establecemos la configuración del host para el driver SPI en la SD y conseguimos no tener el dispositivo como "slave" dentro del sistema sino como master. Esto nos permite tanto enviar como recibir datos. Después tenemos que cambiar la velocidad por defecto del host que define **SDSPI_HOST_DEFAULT**() porque es demasiado alta para nuestra breakout board donde tenemos la microSD instalada. Con **host.max_freq_khz** = 4000 declaramos el valor recomendado por la librería SDFat con esp32/esp8266. A continuación con sdspi_slot_config_t slot_config = **SDSPI_SLOT_CONFIG_DEFAULT**(); configuramos los pines para el host SPI y usando **slot._config.gpio_** los asignamos a los que pasamos como valores de entrada de la función (miso,mosi,clk,cs).

```cpp
 ESP_LOGI(TAG, "Initializing SD card");

  sdmmc_host_t host = SDSPI_HOST_DEFAULT();
  host.max_freq_khz = 4000;
  sdspi_slot_config_t slot_config = SDSPI_SLOT_CONFIG_DEFAULT();
  slot_config.gpio_miso = miso;
  slot_config.gpio_mosi = mosi;
  slot_config.gpio_sck = clk;
  slot_config.gpio_cs = cs;
```

Con la función esp_vfs_fat_sdmmc_mount() montamos el filesystem en la SD de la siguiente manera: pasamos el path donde se registrará la partición (e.g. "/sdcard") con m_mount_point, inicializamos el driver SPI con la configuración en host, inicializamos los pines de la SD con la configuración en slot_config, pasamos los parámetros extra de montaje con mount_config y finalmente guardamos la información de la estructura de la SD montada en m_card.

```cpp
ret = esp_vfs_fat_sdmmc_mount(m_mount_point.c_str(), &host,
&slot_config, &mount_config, &m_card);
```

Todo este proceso lo llamamos dentro de la variable esp_err_t ret para después en un sencillo if indicar al usuario, en el caso que falle la función, si el error se ha producido en el montaje del sistema de ficheros o directamente no se ha podido inicializar la tarjeta.

```cpp
if (ret != ESP_OK)
  {
    if (ret == ESP_FAIL)
    {
      ESP_LOGE(TAG, "Failed to mount filesystem. "
                    "If you want the card to be formatted, set the
format_if_mount_failed");
    }
    else
    {
      ESP_LOGE(TAG, "Failed to initialize the card (%s). "
                    "Make sure SD card lines have pull-up resistors in
place.",
               esp_err_to_name(ret));
    }
    return;
  }
```

Para acabar y si todo funciona correctamente, escribimos por pantalla el mount_point donde se encuentra la SD y su información almacenada previamente en la variable m_card.

```cpp
ESP_LOGI(TAG, "SDCard mounted at: %s", m_mount_point.c_str());

  // Card has been initialized, print its properties
  sdmmc_card_print_info(stdout, m_card);
}
```

I2S Sampler:

I2SSampler actua cómo clase base tanto para I2SMEMSSampler como para ADCSampler.

```cpp
class I2SSampler
{
protected:
    i2s_port_t m_i2sPort = I2S_NUM_0;
    i2s_config_t m_i2s_config;
    virtual void configureI2S() = 0;
    virtual void unConfigureI2S(){};
    virtual void processI2SData(void *samples, size_t count){
        // nothing to do for the default case
    };

public:
    I2SSampler(i2s_port_t i2sPort, const i2s_config_t &i2sConfig);
    void start();
    virtual int read(int16_t *samples, int count) = 0;
    void stop();
    int sample_rate()
    {
        return m_i2s_config.sample_rate;
    }
};
```

Aquí simplemente definimos la función start() y stop(). En la primera, pasándole como valores de entrada nuestro puerto y configuración, instalamos e inicializamos el driver I2S a la vez que llamamos a configureI2S() para establecer la configuración I2S que contiene cada subclase (I2SMEMS/ADC).

```cpp
void I2SSampler::start()
{
    //install and start i2s driver
    i2s_driver_install(m_i2sPort, &m_i2s_config, 0, NULL);
    // set up the I2S configuration from the subclass
    configureI2S();
}
```

Al contrario, con stop() hacemos un clear de la configuración y i2s_driver_uninstall (m_i2sPort) nos desinstala los drivers I2S del puerto especificado.

```cpp
void I2SSampler::stop()
{
    // clear any I2S configuration
    unConfigureI2S();
    // stop the i2S driver
    i2s_driver_uninstall(m_i2sPort);
}
```

I2S MEMS Sampler:

I2S MEMS Sampler es una clase derivada de I2S Sampler preparada para establecer una conexión I2S con micrófonos MEMS digitales. En el constructor definimos las variables del puerto, los pines y la configuración necesaria para la creación del objeto.

```cpp
I2SMEMSSampler::I2SMEMSSampler(
    i2s_port_t i2s_port,
    i2s_pin_config_t &i2s_pins,
    i2s_config_t i2s_config,
    bool fixSPH0645) : I2SSampler(i2s_port, i2s_config)
{
    m_i2sPins = i2s_pins;
    m_fixSPH0645 = fixSPH0645;
}
```

Después, con la función configureI2S() establecemos solamente el número de pines de la configuración I2S estándar que contienen las variables m_i2sPort(I2S_NUM_0 o I2S_NUM_1) y m_i2sPins (I2S estructura pin).

```cpp
void I2SMEMSSampler::configureI2S()
{
    if (m_fixSPH0645)
    {
        // FIXES for SPH0645
        REG_SET_BIT(I2S_TIMING_REG(m_i2sPort), BIT(9));
        REG_SET_BIT(I2S_CONF_REG(m_i2sPort), I2S_RX_MSB_SHIFT);
    }

    i2s_set_pin(m_i2sPort, &m_i2sPins);
```

```
}
```

Read es una función tipo int que lee la señal I2S del micrófono, almacenando el número de muestras totales leídas en la variable samples_read y haciendo el retorno de la misma al final de la función.

```cpp
int I2SMEMSSampler::read(int16_t *samples, int count)
{
    // read from i2s
    int32_t *raw_samples = (int32_t *)malloc(sizeof(int32_t) * count);
    size_t bytes_read = 0;
    i2s_read(m_i2sPort, raw_samples, sizeof(int32_t) * count,
&bytes_read, portMAX_DELAY);
    int samples_read = bytes_read / sizeof(int32_t);
    for (int i = 0; i < samples_read; i++)
    {
        samples[i] = (raw_samples[i] & 0xFFFFFFF0) >> 11;
    }
    free(raw_samples);
    return samples_read;
}
```

Output:

Output actúa como clase base tanto para I2S Output como para DAC Output.

```cpp
class Output
{
protected:
  i2s_port_t m_i2s_port = I2S_NUM_0;

public:
  Output(i2s_port_t i2s_port);
  virtual void start(int sample_rate) = 0;
  void stop();
  // override this in derived classes to turn the sample into
  // something the output device expects - for the default case
  // this is simply a pass through
  virtual int16_t process_sample(int16_t sample) { return sample; }
  void write(int16_t *samples, int count);
};
```

```
Dentro del métodos de la clase aquí definimos el stop() y write(int16_t *samples, int count).

En cuanto al primero, solamente para y desinstala el driver I2S conectado a la señal de salida.
```cpp
void Output::stop()
{
  // stop the i2S driver
  i2s_stop(m_i2s_port);
  i2s_driver_uninstall(m_i2s_port);
}
```

La función tipo void() write es la encargada de preparar las muestras que se pasan por entrada para que sean enviadas al periférico I2S a la vez que también las va escribiendo en nuestro dispositivo de salida. El contenedor donde se almacenan es la variable int16_t *frames.
```cpp
void Output::write(int16_t *samples, int count)
{

  int16_t *frames = (int16_t *)malloc(2 * sizeof(int16_t) *
NUM_FRAMES_TO_SEND);
  int sample_index = 0;
```
Después definimos el bucle donde se prepara toda la información antes de ser enviada y que continuará ejecutándose hasta que se hayan acabado todas las muestras gracias a la variable count (valor de entrada).
```cpp
while (sample_index < count)
  {
    int samples_to_send = 0;
    for (int i = 0; i < NUM_FRAMES_TO_SEND && sample_index < count;
i++)
    {
      int sample = process_sample(samples[sample_index]);
      frames[i * 2] = sample;
      frames[i * 2 + 1] = sample;
      samples_to_send++;
      sample_index++;
    }
```
```

Finalmente escribimos la información en nuestro periférico I2S utilizando i2s_write(), que nos permite escribir data en un transmisor I2S con DMA (Direct Memory access) a la vez que se va expandiendo el número de bits por muestra.

```cpp
size_t bytes_written = 0;
    i2s_write(m_i2s_port, frames, samples_to_send * sizeof(int16_t) *
2, &bytes_written, portMAX_DELAY);
    if (bytes_written != samples_to_send * sizeof(int16_t) * 2)
    {
      ESP_LOGE(TAG, "Did not write all bytes");
    }
  }
  free(frames);
}
```

I2S Output:

I2S Output es una clase derivada de Output con las definiciones necesarias de los métodos start y constructor para la correcta inicialización de la conexión I2S con el periférico, además de los pines.

```cpp
#include "Output.h"

class I2SOutput : public Output
{
private:
    i2s_pin_config_t m_i2s_pins;

public:
    I2SOutput(i2s_port_t i2s_port, i2s_pin_config_t &i2s_pins);
    void start(int sample_rate);
};
```

En el constructor simplemente añadimos la variable de entrada de los pines i2s.

```cpp
I2SOutput::I2SOutput(i2s_port_t i2s_port, i2s_pin_config_t &i2s_pins) :
Output(i2s_port), m_i2s_pins(i2s_pins)
{
}
```

Con el método start(int sample_rate) englobamos todos los procesos necesarios para poder conectar nuestro dispositivo i2s. Primero encontramos i2s_config que contiene todos los valores de las variables necesarias como el modo (master), la frecuencia de muestreo (pasada por entrada en sample_rate), el canal, dma buffer, etc.

```cpp
i2s_config_t i2s_config = {
        .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_TX),
        .sample_rate = sample_rate,
        .bits_per_sample = I2S_BITS_PER_SAMPLE_16BIT,
        .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
        .communication_format =
(i2s_comm_format_t)(I2S_COMM_FORMAT_I2S),
        .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
        .dma_buf_count = 2,
        .dma_buf_len = 1024,
        .use_apll = false,
        .tx_desc_auto_clear = true,
        .fixed_mclk = 0};
```

Después con **i2s _driver_ install** instalamos el driver I2S pasando el puerto y la configuración mostrada anteriormente.

```cpp
 i2s_driver_install(m_i2s_port, &i2s_config, 0, NULL);
```

Con i2s_set_pin definimos los pines I2S almacenados en m_i2s_pins para nuestro puerto.

```cpp
i2s_set_pin(m_i2s_port, &m_i2s_pins);
```

Finalmente las últimas dos funciones són: **i2s_zero_dma_buffer** que limpia el buffer DMA de nuestro puerto y i2s_start que inicializa el driver I2S instalado. Esta última no hace falta realmente ya que después de **i2s _driver_ install** se inicializa automáticamente, aun asi si que la necesitamos después de llamar a **i2s_stop**.

```cpp
 i2s_zero_dma_buffer(m_i2s_port);


    i2s_start(m_i2s_port);
}
```

Conclusiones:

Con este proyecto en funcionamiento delante de nosotros, podemos decir con seguridad que nos ha ayudado mucho a consolidar los conocimientos adquiridos en esta asignatura, a perder el miedo delante de las librerías grandes y sobre todo a analizar código para una fácil implementación de este. La resolución de problemas que hemos tenido durante el proceso de tanto el montaje como del código nos ha sido útil para poder entender más en profundidad los buses que usamos: I2C, I2S, SPI y nos ha mostrado lo que se puede hacer con los SDMMC.

Aún con esas, nos deja un sabor de boca agridulce porque no hemos podido acabar con el broche de oro que nos habíamos propuesto al inicio del proyecto. Para empezar, el proyecto era el que hemos acabado y puesto en marcha, pero queríamos ir más allá implementando la librería arduinoFFT para poder mostrar el espectro frecuencial por la pantalla del display. El uso del código no tenía una alta dificultad pero lo que nos fallaba era el micrófono y la señal que nos llegaba de este y salía del altavoz. El micrófono era un micrófono MEM (MicroElectroMecánico) que es un tipo de micrófono que se basa de, principalmente, cápsulas electret yque tienen integrado convertidores ADC, por tanto este nos proporciona señal digital y no podíamos usar esa señal porque era directamente digital y para poder hacer la FFT necesitamos señal analógica.
Y por último: el reverse. Entendiendo el funcionamiento de las librerías necesarias para escribir y leer la señal dentro de un wav, creíamos que podríamos ser capaces de girar los samples del WAV, respetando los 16 bits de cabecera, y poder reproducir un wav de fin a inicio. En ningún momento tuvimos en cuenta que el tipo de compresión de audio también era algo que afectaba a los samples del sonido registrado y por tanto lo que nosotros creíamos que era una tarea muy simple acabó siendo una funcionalidad del proyecto que estaba por encima de nuestras posibilidades. Siendo así la cosa, nos volveremos a poner con estas funcionalidades en un futuro porque le vemos mucho potencial de mejora a este proyecto.

Video definitivo:

Proyecto final Ceballos Moncayo