Práctica 3: Filtrado.

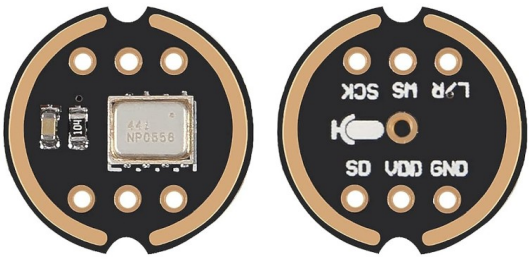
Adquisición y Procesamiento de Señal

Objetivo

El objetivo de esta práctica es la implementación en el microcontrolador de un filtro en el dominio del tiempo (convolución) para procesar en tiempo real señales de audio capturadas desde un micrófono. La señal de audio procesada será enviada al PC para su análisis.

Sensor

Utilizaremos el micrófono omnidireccional INMP441, un **sensor digital** que utiliza el interfaz **I2S** para transferir muestras de sonido de **24 bits** con una frecuencia de muestreo de hasta **48 kHz** al microcontrolador.



La hoja técnica del sensor puede consultarse en:

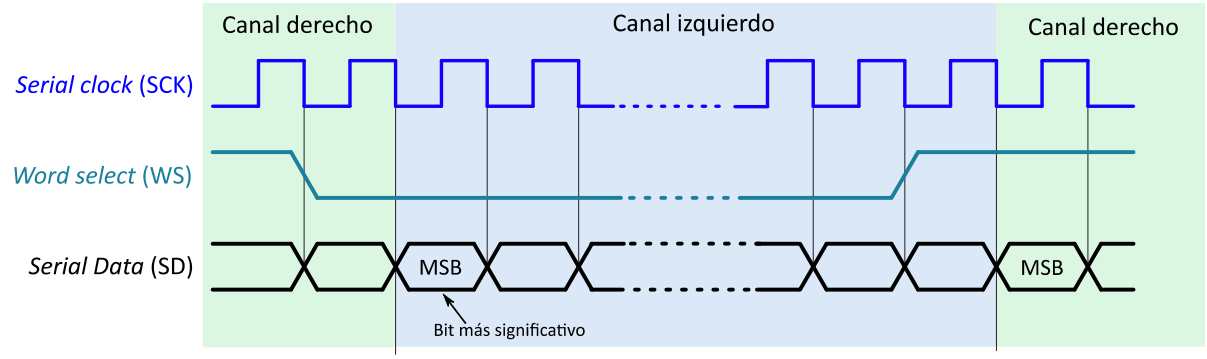
https://invensense.tdk.com/wp-content/uploads/2015/02/INMP441.pdf

I2S

El protocolo I2S (***I**nter-**I**ntegrated circuit **S**ound*) es un protocolo serie que se utiliza para transferir muestras digitales de sonido de dos canales (estéreo) y codificadas en PCM entre chips. Utiliza tres líneas de datos (**SCK**, **SD** y **WS**) para transferir las muestras desde un micrófono (ADC) o hacia un altavoz (DAC).

Cuando se utiliza con un micrófono la líneas tienen las siguientes funciones:

- **SCK** (*Serial CLock*): escrita por el microcontrolador, leída por el micrófono. Lleva la señal de reloj. Marca el ritmo al que el micrófono debe escribir los bits en la línea de datos (es un **protocolo síncrono**). Cada vez que esta señal pasa de nivel alto a nivel bajo el micrófono escribe un bit en SD. Similar a SCL en I2C.
- **SD** (*Serial Data*): escrita por el micrófono, leída por el microcontrolador. Lleva los bits de las muestras capturadas. Similar a SDA en I2C. Se envía primero el bit más significativo de cada muestra (formato *big-endian*).
- **WS** (*Word Select*): escrita por el microcontrolador, leída por el micrófono. Indica al micrófono que debe comenzar a enviar la siguiente **muestra** en el ciclo de SCK inmediatamente posterior a que WS cambie. Si cambia de nivel alto a nivel bajo debe comenzar a enviar la muestra del canal izquierdo, y si cambia de nivel bajo a nivel alto la muestra del canal derecho. La frecuencia de la señal WS es igual a la frecuencia de muestreo.



En el caso del INMP441, cada muestra de sonido tiene 24 bits. Es decir, el ADC en el INMP441 tiene 2^{24} = 16777216 niveles. La salida del ADC son **números enteros (con signo) codificados en complemento a 2** representando la onda de sonido. Sin embargo, aunque las muestras sean de 24 bits, el INMP441 necesita 32 ciclos de reloj para enviar cada muestra (ver hoja técnica: sección I²S Data Interface y figuras 8 a 10), por lo que será necesario configurar el periférico I2S en el ESP32 en modo **32 bits** (en la inicialización del objeto `I2S` del MicroPython), y descartar durante el procesado los 8 bits menos significativos de cada muestra.

DMA

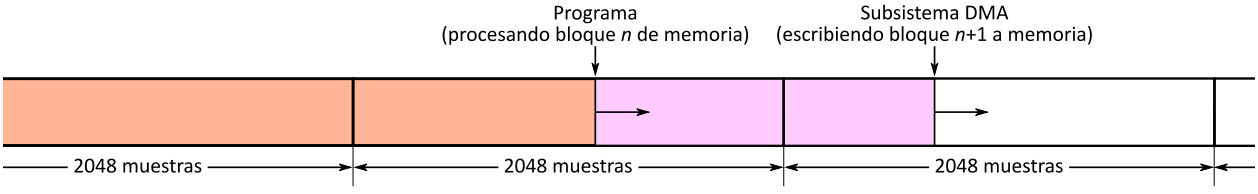
Al contrario que en la práctica anterior, en la que leíamos cada muestra desde el programa mediante entrada/salida programada o mediante interrupciones, en este caso usaremos **DMA (*Direct Memory Access*).**

Con DMA el microcontrolador es capaz de **leer las muestras directamente a memoria desde un bus** (I2C, SPI, I2S, etc.) **sin intervención de la CPU.** Esto nos permite procesar las muestras de sonido sin gastar ciclos de reloj de CPU en la entrada/salida: nuestro programa simplemente configurará la lectura mediante DMA durante la inicialización del objeto `I2S` y, en el bucle principal, leerá directamente desde un *buffer* en memoria RAM las muestras que haya obtenido el microcontrolador de forma autónoma.

DMA es el método de entrada/salida más apropiado cuando se necesita transferir información hacia o desde un periférico a **alta velocidad**. El microcontrolador debe disponer de soporte hardware (con un **controlador DMA**), algo prácticamente asegurado en microcontroladores modernos de 32 bits.

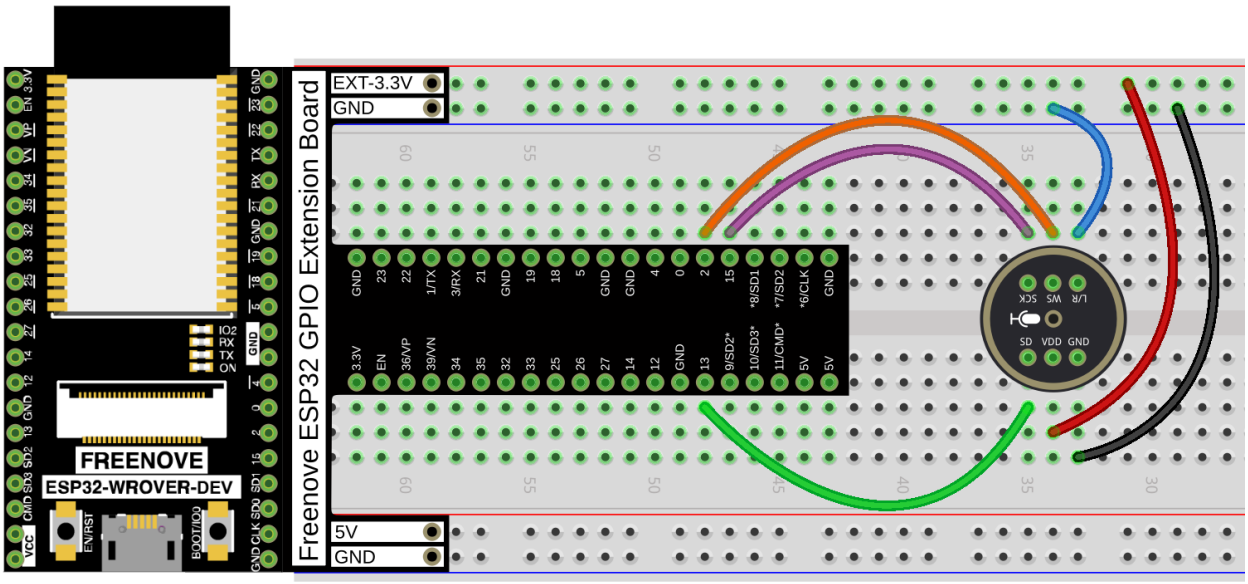
Para procesar las muestras nuestro programa esperará a que haya **2048** muestras disponibles. Para ello se debe pasar a la función `I2S.readinto` un buffer del tamaño adecuado. Si al llamar a `readinto` el controlador DMA aún no ha escrito suficientes muestras desde el bus a memoria para llenar el buffer, el programa permanecerá a la espera de más muestras (lo que se conoce como **E/S bloqueante**).

Para que el programa funcione correctamente **el procesado debe ser más rápido que la captura.** Si el programa termina de procesar el bloque *n* de muestras **antes** de que el controlador DMA haya escrito a memoria el bloque *n* + 1, el programa simplemente permanecerá en espera en la llamada a `readinto` hasta que se termine de capturar a memoria el siguiente bloque completo. Si, por el contrario, el procesado del bloque es demasiado lento, el buffer interno se llenará y el controlador DMA escribirá sobre muestras aún no procesadas por nuestro programa, lo que hará que se pierdan y el sonido se oirá entrecortado.



Montaje

Conecta el INMP441 en la placa del microcontrolador de la siguiente forma:



Los pines de alimentación del micrófono (VCC y GND) van a las líneas de 3.3 V y tierra, respectivamente. Conecta los pines propios del protocolo I2S: SCK, WS y SD a los pines 15, 2 y 13, respectivamente.

Dado que el micrófono no es estéreo, solo enviará bits en el intervalo correspondiente a uno de los canales (izquierdo o derecho). El pin **L/R** sirve para seleccionar si envía en uno u otro. Con L/R conectado a GND, como en la figura anterior, el micrófono envía los datos en el intervalo del canal izquierdo; podría combinarse con otro micrófono con el canal L/R conectado a VCC para capturar sonido estéreo.

Código

El código (disponible en moodle) consta de un módulo que se ejecutará **en el PC** (`capture`) y tres módulos que se ejecutarán **en el microcontrolador** (`net`, `profiler` y `mic`) y deben ser copiados a su flash interna.

El módulo `capture` se ejecutará en el Python local (en una **instancia distinta de Thonny** o en Spyder), y se conectará al microcontrolador a través de la red Wi-Fi, solicitará que inicie la captura con los parámetros adecuados (frecuencia de muestreo, duración de la captura en segundos y respuesta al impulso del filtro a aplicar a la señal) y creará un archivo de sonido WAV con los datos que reciba. La IP del microcontrolador y los parámetros que pasa a `mic.py` son parte del código; es necesario editar el fichero `capture.py` con los valores desados (especialmente la dirección IP, que no tiene un valor por defecto).

En cuanto a los módulos que se ejecutarán en el microcontrolador:

- El módulo `net` tiene el código para conectarse a la red Wi-Fi (en la inicialización del objeto `Net`) y quedar en espera de una solicitud con los parámetros para la captura y el filtrado (función `get_params`). También se encarga de enviar las muestras capturadas por la red (función `send`). No es necesario modificarlo.
- El módulo `profiler` se encarga de imprimir el tiempo que le lleva al microcontrolador procesar cada bloque (`print_partial`), en microsegundos (tanto el total del bloque como el tiempo de cada tarea: recepción, procesado y envío), así como el tiempo medio dedicado a todos los bloques (`print_average`). Tampoco es necesario modificarlo.

- El módulo `mic` contiene el programa principal: se conecta a la red creando un objeto `net` (hay que configurar el SSID y contraseña de la red Wi-Fi) y espera en el bucle principal a una nueva solicitud de captura. Es necesario implementar la inicialización del objeto I2S y el bucle de lectura, procesado y envío de muestras como se pide en las siguientes tareas.

Herramientas

Para la realización de la práctica es necesario utilizar:

https://onlinetonegenerator.com/multiple-tone-generator.html, página web que permite generar tonos de diferentes frecuencias desde el navegador.

https://www.audacityteam.org, editor de audio para analizar el espectro (FFT) de los ficheros WAV capturados.

Tarea 1: Inicialización de I2S y lectura de muestras.

El primer paso, tras verificar que `mic.py` recibe correctamente las solicitudes de `capture.py` a través de la red, es inicializar el objeto `I2S` pasándole los pines, el modo (RX), el número de bits por muestra, el formato (mono), la tasa de muestreo (recibida de `get_params`) y el **tamaño del buffer interno** en el que guardará las muestras mediante DMA (**en bytes**, no en número de muestras). Las posteriores llamadas a `I2S.readinto` copiarán las muestras desde este buffer interno al que pasemos como argumento a `readinto`. El tamaño del buffer interno debe ser **mayor** que el de nuestro buffer (por ejemplo, 2 veces mayor) para que el controlador pueda escribir muestras sin afectar a las que vamos a copiar a nuestro buffer.

Consulta la documentación de I2S en https://docs.micropython.org/en/latest/library/machine.I2S.html.

Tras la inicialización del objeto `I2S` se deben leer las muestras en bloques de 2048 muestras. Para ello crea un nuevo bucle (interno al bucle principal) en el que, en cada iteración, se leerá un bloque a un `bytearray` del tamaño adecuado (argumento pasado a `readinto`). Como el tamaño de este buffer nunca cambia, créalo fuera del bucle principal. Este bucle de lectura debe terminar cuando se hayan leído las muestras solicitadas, calculadas a partir de la duración en segundos y la frecuencia de muestreo.

Imprime los 16 primeros bytes del buffer (4 muestras) en cada iteración del bucle y verifica que se reciben datos correctos. Ten en cuenta que, aunque las muestras llegan a través de I2S en formato *big-endian*, `readinto` reordena los bytes a *little-endian* para facilitar el procesado nativo en ESP32, por lo que cada muestra en el buffer comenzará con el byte menos significativo.

Tras terminar el bucle de lectura se cerrará la conexión (`net.close`, ya en `mic.py`) y comenzará una nueva iteración del bucle principal, quedando a la escucha de una nueva solicitud desde `capture.py`.

Preguntas

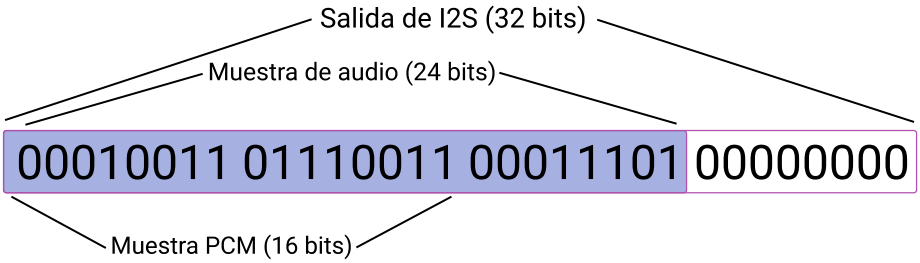
1. Utiliza la función `time.ticks_us` para guardar el instante de simulación (en microsegundos) al principio del bucle de lectura (`t0`) y tras la llamada a `readinto` (`t1`). Pasa estos valores en cada iteración a `profiler.print_partial(t0, t1)` para imprimir el tiempo de procesado de cada bloque y llama a `profiler.print_average` (sin argumentos) **tras el bucle** de procesado para imprimir el tiempo medio de todos los bloques.

¿Cada cuánto se procesa en término medio un bloque de muestras cuando se solicita una frecuencia de muestreo de 8000 Hz? ¿Cuál es el tiempo teórico que debería tardar?

2. Repite lo anterior a 16000 Hz y a 48000 Hz.

Tarea 2: Escalado y envío.

Envía las muestras por la red al PC para crear el archivo WAV. El programa `capture.py` espera recibir del ESP32 muestras de **16 bits en complemento a 2** (calidad CD), por lo que es necesario **escalar** las muestras *crudas* de 32 bits recibidas en el buffer a un valor apropiado. Esto se puede lograr desplazando 16 bits a la derecha o, equivalentemente, dividiendo por 2^{16}, pero puede ser necesario dividir por un valor menor para que la señal de sonido resultante tenga un volumen más alto.



En MicroPython se puede convertir el `bytearray` con las muestras crudas a una lista de enteros con `struct.unpack`, como en la práctica anterior, pero como necesitamos operar con él es preferible utilizar la función `util.from_int32_buffer`, que lo convierte directamente a un array de NumPy. El array resultante tendrá tipo de datos *float*, porque `ulab.numpy` no soporta enteros de 32 bits como tipos de datos en arrays.

Escala las muestras (dividiendo, pues no se pueden realizar operaciones de desplazamiento de bits sobre variables de tipo flotante) y envíalas como un array de enteros de 16 bits. Para ello guarda el resultado del escalado en un nuevo array de NumPy con `dtype=np.int16` y envíalo con `net.send`.

Preguntas

- Captura a 8000 Hz 2 tonos, uno con frecuencia 700 Hz y otro con frecuencia 5500 Hz. Escucha el fichero generado por `capture`. ¿Se oyen los dos tonos?
- Abre el fichero resultante en **Audacity** y muestra el espectro (*Analyze* → *Plot Spectrum…*). Configura el eje x en modo lineal en el desplegable *Axis* → *Linear frequency*. ¿Qué se observa? ¿Por qué ocurre esto?
- Repite lo anterior capturando la señal a 16000 Hz. ¿Qué ocurre ahora? ¿Por qué es diferente al caso anterior?
- Captura a 8000 Hz un tono de 700 Hz y otro de 4500 Hz. ¿Qué se observa en el espectro? ¿A qué se debe?

Tarea 3: Filtrado.

Filtra la señal con la respuesta al impulso recibida desde `capture.py`. Para ello utiliza la función `convolve` de NumPy (https://micropython-ulab.readthedocs.io/en/latest/numpy-functions.html#convolve) tras el escalado y antes del envío.

Para que el bloque filtrado tenga el mismo tamaño que el de entrada (2048) descarta las últimas $L - 1$ muestras, donde L es la longitud de la respuesta al impulso del filtro (`ulab.numpy.convolve` solo soporta el modo ***'full'***, que devuelve ambas colas de la convolución).

Para crear el filtro en `capture.py` usa la función `scipy.signal.firwin`:

https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.firwin.html

Elige los valores adecuados de `numtaps` (número de taps), `cutoff` (frecuencia de corte), `fs` (frecuencia de muestreo) y `pass_zero` (filtro paso alto o bajo). Modifica el array `FILTER` en `capture.py` con los coeficientes del filtro devueltos por `firwin`. Asegúrate de mantener el `dtype='float32'`, que es el tipo de datos esperado por `mic.py`.

En `mic.py`, el filtro es el tercer valor devuelto por la llamada a `get_params`.

Preguntas

1. Diseña un filtro de $L = 10$ *taps* que elimine tonos por encima de 1000 Hz (filtro **paso bajo**) cuando son muestreados a 16000 Hz. Captura a esta frecuencia una señal compuesta por 3 tonos, uno de 800 Hz, otro de 2000 Hz y otro de 5000 Hz y pásala por el filtro. Muestra el espectro de la señal resultante en Audacity y explica el resultado.
2. Añade medidas de tiempos: `t2` tras el filtrado y `t3` tras el envío, y pásalas junto a `t0` y `t1` también a `print_partial(t0, t1, t2, t3)`. Diseña filtros de tamaños cada vez mayores ($L = 15, 20, 25$, etc.). ¿Cómo cambian los tiempos? ¿Hasta qué tamaño de filtro le da tiempo a procesar los bloques sin perder muestras?
3. Muestra capturas del espectro de la señal filtrada con 10 taps, 20 taps y 30 taps. ¿Qué efecto tiene en la señal de salida aumentar el tamaño del filtro?
4. ¿Puedes construir un filtro que deje pasar un tono de 800 Hz y elimine completamente uno de 1200 Hz? ¿Por qué?

Tarea 4: Filtrado corregido.

Al realizar el filtrado por bloques el resultado de la convolución será distinto al que se obtendría si se filtrase la señal completa, debido a que la convolución necesita, además de la muestra actual, las L - 1 muestras anteriores para obtener cada muestra de salida. Si cada bloque se procesa de forma independiente, al principio de cada bloque no tenemos las muestras anteriores disponibles.

Mejora el filtrado de la tarea anterior teniendo en cuenta que cada bloque es parte de una señal completa. Para ello es necesario guardar las **últimas $L - 1$ muestras** del bloque actual y **concatenarlas al principio del siguiente** antes de la convolución. Ahora para obtener el mismo número de muestras de entrada que de salida (2048) debes descartar las primeras $L - 1$ y las últimas $L - 1$ muestras.

Preguntas

1. Genera un tono de 800 Hz y pásalo por el filtro paso bajo de 30 taps del apartado anterior, con la convolución sin corregir y corregida. ¿Escuchas alguna diferencia? Muestra una captura de ambas señales de salida **en tiempo** donde se vea el límite entre dos bloques y explica la diferencia.