14 dic 2021

Reporte Modulo MP45DT02 (Usermod)

Versión 1.0

DIONICIO MEZA SOLANO

Índice

Marco teórico

Modulación PDM

Es un tipo de codificación conocido como modulación por densidad de pulsos (Pulse Density Modulation) basado en 1 bit y sobremuestreo.

Es usado en la representación y conversión de señales analógicas al dominio digital (y a la inversa), la amplitud se representa por densidad o número de impulsos en función del tiempo. Así, por ejemplo, para representar una señal constante y nula, tendríamos una sucesión de ...1010101010... de forma que la densidad sería mínima. Por otro lado, para hacer lo mismo con una señal constante, pero de valor elevado y positivo, tendríamos gran densidad de “unos”: ...1111111011...

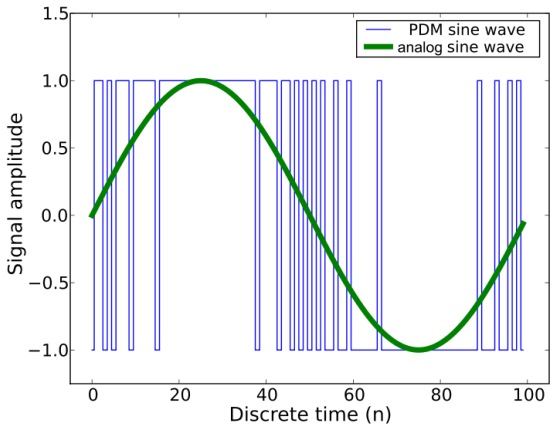

Ilustración 1. Señal PDM

Modulación PCM

La modulación por impulsos codificados (MIC o PCM por las siglas en inglés de Pulse Code Modulation) es un procedimiento de modulación utilizado para transformar una señal analógica en una secuencia de bits (señal digital), es una forma estándar de audio digital en computadoras, discos compactos, telefonía digital y otras aplicaciones similares. Es una representación más intuitiva: para cada instante de tiempo discretizado, se asigna un dato multi-bit. En la *ilustración 2* se puede ver un ejemplo de una señal PCM de 4 bits

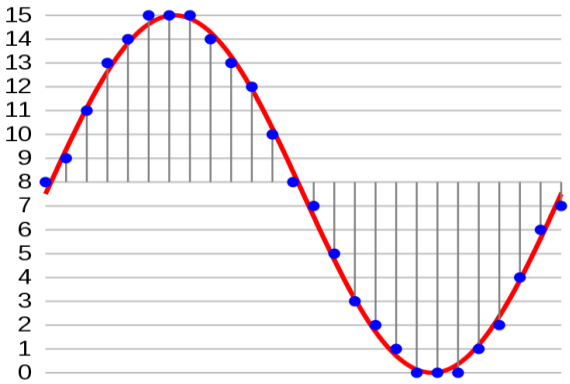

Ilustración 2. Señal PCM de 4 bits.

Conversión PDM a PCM

En este proceso se realiza la conversión de la salida de flujo de 1 bit de un micrófono PDM a una señal digital estándar en formato PCM. Este proceso consta de dos pasos principalmente, el primero es el filtrado pasa bajos que se encarga de remover el ruido de alta frecuencia característico del formato PDM, este filtro se debe implementar de forma digital utilizando un filtro de respuesta de impulso finito o por sus siglas en ingles FIR, aplicando un promedio móvil para el cual se deben calcular un conjunto de coeficientes de ponderación específicos.

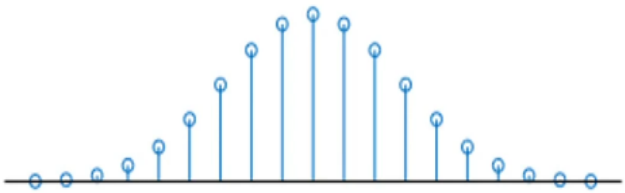

Ilustración 3. Respuesta en el tiempo del filtro FIR.

La primera muestra en formato PCM se obtiene de la suma del producto de cada coeficiente del filtro por el bit correspondiente de la muestra en PDM.

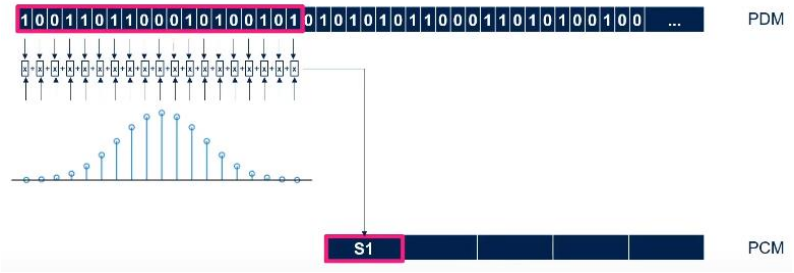

Ilustración 4. Obtención de la primera muestra en PCM.

La segunda muestra PCM se obtiene realizando el mismo procedimiento anterior, pero en esta ocasión se debe recorrer un bit en el flujo PDM.

Ilustración 5, Obtención de la segunda muestra PCM.

Este proceso se debe repetir contantemente para ir obteniendo las muestras PCM. El siguiente paso es la decimacion, que consiste en descartar una cierta cantidad de muestras para reducir la frecuencia de muestreo, en la *ilustración 6* se muestra un ejemplo con una decimacion de 3.

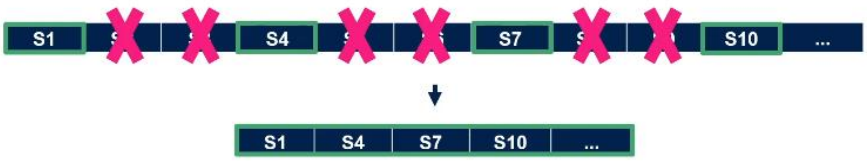

Ilustración 6. Decimacion de las muestras.

En este caso la frecuencia de muestreo para el formato PCM es igual a:

$$Frecuancia\ PCM = Frecuancia\ PDM / Factor\ de\ decimación.$$

Filtro pasa bajo

Este tipo de filtro deja pasar todas las frecuencias desde 0 a la frecuencia de corte y bloquea las frecuencias que están sobre esta. A Las frecuencias que se ubican entre 0 hasta la frecuencia de corte se le denomina banda pasante mientras las frecuencias que están encima de la frecuencia de corte se le denomina banda eliminada. Las zonas entre la banda pasante y la banda eliminada se le llaman banda de transición.

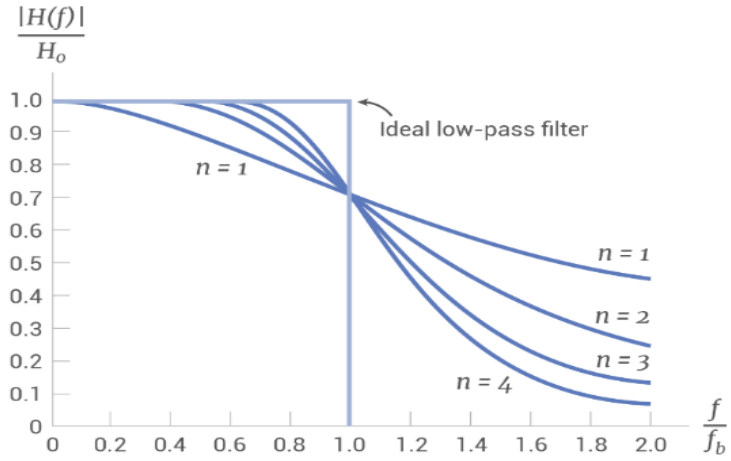

Ilustración 7. Filtro pasa bajo.

Implementación

Diseño del filtro FIR (promedio móvil)

El filtro FIR se diseñó con la ayuda del software MatLab, utilizando un filtro Blackman de 64 ventanas a una frecuencia de muestreo de 44Khz, los coeficientes arrojados se graficaron en la ilustración 8.

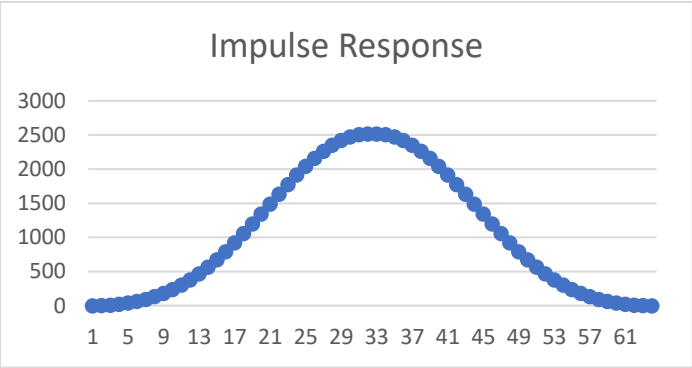

Ilustración 8. Respuesta al impulso del filtro calculado.

Algoritmo de conversión PDM a PCM en STM32CubeIDE

Se creó un nuevo proyecto STM32 y se configuró el microcontrolador STM32f407VG que está integrado en la tarjeta de desarrollo Ophyra. En la configuración del reloj se estableció un reloj externo de 8Mhz y se modificaron los prescalers para obtener una velocidad máxima de 192 Mhz en el reloj de la comunicación I2S como se muestra en la *ilustración 9*.

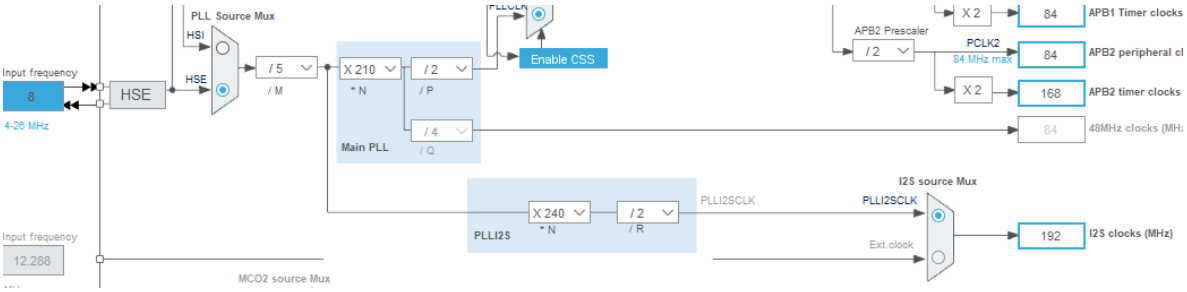

Ilustración 9. Configuración del reloj I2S.

Se activó la comunicación I2S2 en un modo Half-Duplex Master para que el microcontrolador sea quien envíe la señal de reloj hacia el micrófono, en los parameros generales se configuró: el modo de transmisión como Master Receive, la comunicación estándar como I2S Phillips, el tamaño del frame o muestra PDM se definió a 16 bits, por último para establecer una frecuencia de muestreo PDM se realizó lo siguiente, se tomó en cuenta las 64 ventanas del filtro FIR, el tamaño de la muestra PDM (16 bits) y la velocidad del audio de salida en este caso 44khz,

primero se calculó la cantidad de muestras necesarias para obtener la misma cantidad de bits que las ventanas del filtro:

$$\frac{64\ ventanas_{fir}}{16\ bits} = 4\ muestras_{PDM}$$

Lo que indica que para poder obtener una muestra en PCM se necesitan 4 muestras de 16 bits en PDM por lo tanto la velocidad del reloj PDM debe ser por lo menos 4 veces mayor, con ese dato se estableció la relación entre la frecuencia PDM y PCM:

$$frec_{PDM} = 4\ frec_{PCM}$$
$$frec_{PDM} = 4\ (44khz) = 176\ khz$$

Con lo anterior el puerto I2S2 quedo configurado como se muestra en la *ilustración 10*.

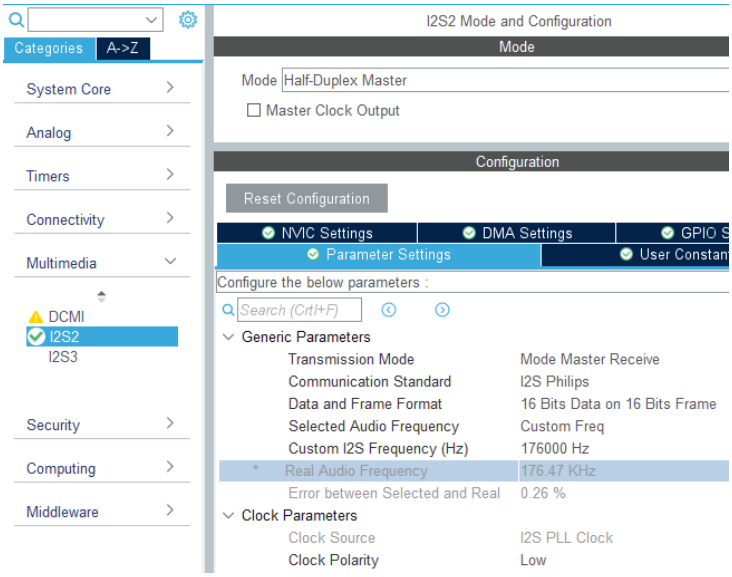

Ilustración 10. Configuración del periférico I2S.

Se configuró un DMA circular para almacenar las muestras obtenidas por el puerto I2S2.

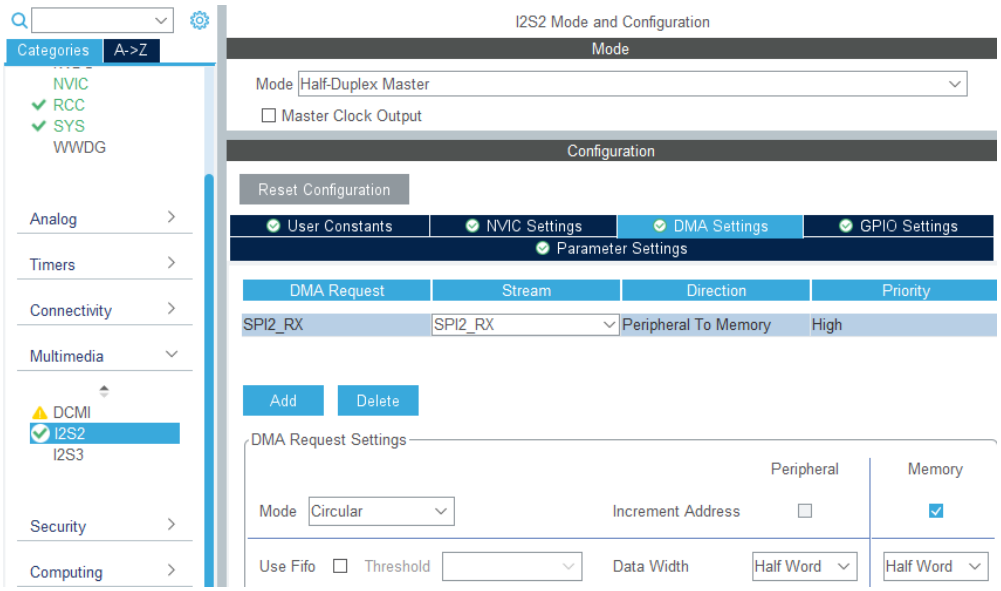

Ilustración 11. Configuración del DMA.

Para llevar un control general de la toma de muestras PDM y procesarlas a PCM de forma sincronizada se decidió utilizar las herramientas que ofrece el microcontrolador como son las interrupciones, en este caso se seleccionaron 2 interrupciones ligadas a la recepción de datos del puerto I2S:

- **HAL_I2S_RxHalfCpltCallback** – Indica que el buffer proporcionado para recibir las muestras PDM se ha llenado a la mitad.
- **HAL_I2S_RxCpltCallback** – Indica que el buffer proporcionado para recibir las muestras PDM se ha llenado por completo.

Estas interrupciones se utilizaron de tal forma que al disparase cualquiera de ella se genere una muestra PCM. Anteriormente se calculó que la frecuencia PDM es 4 veces mayor que la PCM, por lo tanto, el buffer para la recepción de datos PDM también debe ser 4 veces mayor.

$$1\ muestra_{PCM} = 4\ muestras_{PDM}$$

La primera interrupción se crea al llenarse la mitad del buffer esto debe generar una muestra PCM por lo tanto al llenarse por completo el buffer se habrán creado dos

muestras PCM, por lo tanto, el buffer para el DMA debe ser de 8 muestras de 16 bits, cuatro para la primera muestra y otras 4 para la segunda muestra.

```
uint16_t pdmRxBuf[8];
HAL_I2S_Receive_DMA(&hi2s2, &pdmRxBuf[0],4);
```

Después se creó el algoritmo para convertir el formato PDM a PCM, este algoritmo está formado por los siguientes pasos:

1. Se creo una macro llamada PDM_REPEAT_LOOP_16 la cual realiza un bucle que permitirá recorrer los 16 bits de cada muestra PDM.

```
#define PDM_REPEAT_LOOP_16(X) X X X X X X X X X X X X X X X X
```

2. En la línea 1 del *código 1* se declara una variable llamada *runningsum* en esta variable se irá almacenando la suma resultante de multiplicar el valor de cada ventana por cada bit de la muestra PDM.
3. En la línea 2 se declara una variable de tipo puntero que permite acceder a la dirección de cada uno de los 64 coeficientes del filtro pasa bajo.
4. En la línea 3 se crea un ciclo *for* que se repite 4 veces y dentro de ese ciclo se utiliza la macro PDM_REPEAT_LOOP_16 esto para poder multiplicar los 64 coeficientes por los 64 bits obtenidos de 4 muestras de 16 bits. Dentro de la macro se realiza una comparación de cada bit empezando por el menos significativo, si el bit es 1 se realiza la suma del coeficiente correspondiente, sino solo se recorre el bit y la posición del coeficiente.
5. En la línea 8 se avanza por cada una de las direcciones de los coeficientes y en línea 9 se avanza por cada uno de los bits de la muestra PDM.
6. Al finalizar el ciclo *for* se habrá obtenido la muestra PCM almacenada en la variable *runningsum.*

```
1.  uint16_t runningsum = 0;
2.  uint16_t *sinc_ptr = sincfilter;
3.  for (uint8_t i=0; i < 4 ; i++) {
4.      PDM_REPEAT_LOOP_16({
5.          if (pdmRxBuf[i] & 0x1) {
6.              runningsum += *sinc_ptr;
```

```
7.          }
8.          sinc_ptr++;
9.          pdmRxBuf[i] >>= 1;
10.     })
11. }
```

Código 1. Algoritmo para convertir PDM a PCM.

Para implementar este algoritmo dentro de las interrupciones se debe declarar la interrupción mediante:

- *void HAL_I2S_RxHalfCpltCallback (I2S_HandleTypeDef *hi2s)* para la mitad del buffer
- *void HAL_I2S_RxCpltCallback (I2S_HandleTypeDef *hi2s)* para el buffer lleno.

A continuación, se muestra el ejemplo para la interrupción a mitad del buffer, adicionalmente al algoritmo de conversión en la línea 16 del *código 2* la muestra PCM se almacena en un nuevo buffer que almacenara las muestras convertidas, en la línea 13 se crea una condición para que en el momento que se supere en máximo número de muestras del buffer se regrese al índice 0.

```
1.  void HAL_I2S_RxHalfCpltCallback (I2S_HandleTypeDef *hi2s) {
2.      uint16_t runningsum = 0;
3.      uint16_t *sinc_ptr = sincfilter;
4.      for (uint8_t i=0; i < 4 ; i++) {
5.          PDM_REPEAT_LOOP_16({
6.          if (pdmRxBuf[i] & 0x1) {
7.              runningsum += *sinc_ptr;
8.          }
9.          sinc_ptr++;
10.         pdmRxBuf[i] >>= 1;
11.         })
12.     }
13.     if (p >= maxBufTx){
14.         p = 0;
15.     }
16.     txBuf[p] = runningsum;
17.     p++;
18. }
```

Código 2. Ejemplo del uso del algoritmo en una interrupción.

Adicionalmente al filtro pasa bajo previamente diseñado se creó otro filtro para limpiar el audio generado, pues el primer filtro solo es para la conversión de PDM a PCM mediante un promedio móvil.

El diseño del segundo filtro se realizo con la ayuda del programa Tfilter y re implemento con el algoritmo del *código 3* el cual realiza la multiplicación de los coeficientes del filtro por cada muestra anterior y realiza el corrimiento de las muestras.

```
float filtro(int in){
 float y=0;

 x_n[0] = in;
 y = h[0] * x_n[0];
 for(int j=1;j<M;j++){
   y += h[j] * x_n[j];
 };

 for (int i=M;i>0;i--){
   x_n[i]=x_n[i-1];
 };
 return y;
}
```

Código 3. Algoritmo del filtro pasa bajos para el filtrado del audio.

El código anterior toma una estructura como la de la *ilustración 12.*

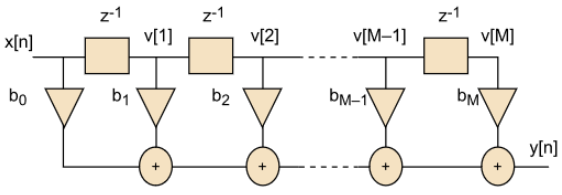

Ilustración 12. Estructura del filtro FIR pasabajos.

Con el diseño de este nuevo filtro se reemplaza la línea 16 del *código 2* por:

```
txBuf[p] = filtro(runningsum);
```

con esto último ya se tienen muestras de audio en formato PCM almacenadas en el buffer *txBuf* y listas para ser reproducidas por el DAC a una frecuencia de 44khz o almacenadas en algún archivo de audio.

Desarrollo del modulo MP45DT02 en C (usermod)

Con el algoritmo diseñado y con el programa de ejemplo realizado en STM32CubeIDE se procedió a realizar la implementación del módulo para el firmware de MicroPython.

Dentro de la carpeta modules se creó una nueva carpeta con el nombre *ophyra_mp45dt02* y dentro de esta se crearon dos archivos.

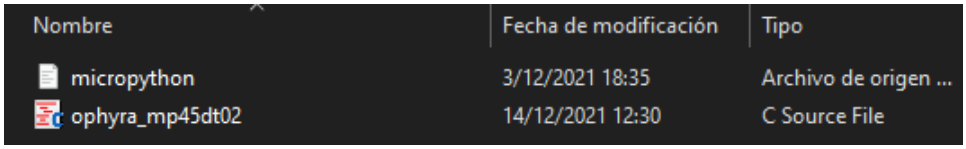

Ilustración 13. Contenido de la carpeta ophyra_mp45dt02.

El primer archivo llamado micropython.mk es el archivo Makefile para la compilación, este archivo debe contener el nombre del archivo que contiene la programación principal del código.

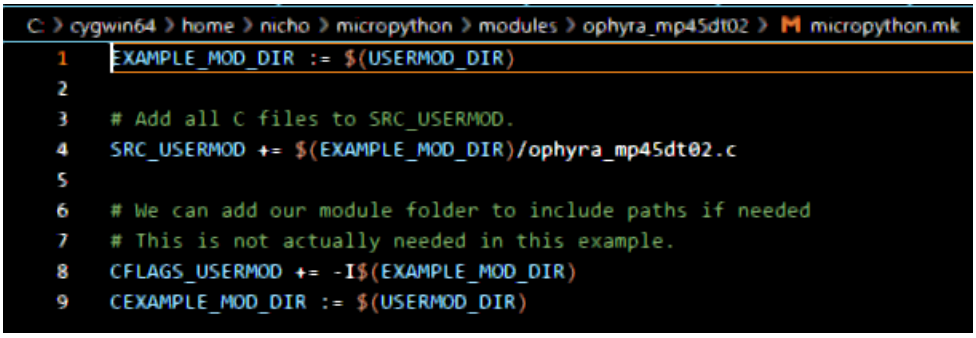

Ilustración 14. Contenido del archivo make.

El segundo archivo contiene el código fuente del módulo MP45DT02.

Estructura interna del archivo Ophyra_mp45dt02

Inicialmente en el encabezado del archivo se agregan los headers necesarios para tener disponibles las macros y modulos de MicroPython.

```
#include <stdbool.h>
#include "py/obj.h"
#include "py/runtime.h"
#include "py/mphal.h"
#include "py/misc.h"
#include "py/stream.h"
#include "py/objstr.h"
#include "modmachine.h"
#include "pin.h"
#include "dma.h"
#include "py/mphal.h"
```

Se creó una estructura llamada *non_blocking_descriptor* la cual contiene elementos que ayudan al control de del proceso en un modo sin bloqueo es decir que pueda trabajar en segundo plano, estos elementos son el buffer que almacenará los datos en formato PCM, el índice del buffer y una señal que permite poner en marcha la toma de muestras

```
typedef struct _non_blocking_descriptor_t {
    mp_buffer_info_t appbuf;
    uint32_t index;
    bool copy_in_progress;
} non_blocking_descriptor_t;
```

Se creó otra estructura relacionada directamente con el módulo, esta estructura contiene elementos como la base, una devolución de llamada que se realiza al llenarse el buffer proporcionado, la longitud del buffer, el descriptor del modo sin bloqueo, la configuración del puerto I2S y la configuración del DMA.

```
typedef struct _mp45dt02_obj_t {
    mp_obj_base_t base;
    mp_obj_t callback_for_non_blocking;
    uint16_t dma_buffer[LONG_BUF];
    non_blocking_descriptor_t non_blocking_descriptor;
    I2S_HandleTypeDef hi2s2;
```

```
    DMA_HandleTypeDef hdma_rx;
    const dma_descr_t *dma_descr_rx;
} mp45dt02_obj_t;
```

Después de esto se definieron los coeficientes de los dos filtros, sin embargo, como son muy extensos se agregaron en el anexo 1 al final de este documento.

La función make_new es el constructor de la clase, cuando se declara esta función se crea el objeto de la clase, dentro de esta función se utiliza la función mp45dt02_init_helper(self) la cual se encarga de inicializar los periféricos como el DMA y la comunicación I2S.

```
STATIC mp_obj_t mp45dt02_make_new(const mp_obj_type_t *type, size_t n_pos_args, size_t n_kw_args,
const mp_obj_t *args) {
    mp_arg_check_num(n_pos_args, n_kw_args, 0, 0, false);

    mp45dt02_obj_t *self;

    if (mp45dt02_obj == NULL) {
        self = m_new_obj(mp45dt02_obj_t);
        mp45dt02_obj = self;
        self->base.type = &mp45dt02_type;
    } else {
        self = mp45dt02_obj;
        mp45dt02_deinit(MP_OBJ_FROM_PTR(self));
    }

    mp45dt02_init_helper(self);
    return MP_OBJ_FROM_PTR(self);
}
```

La función mp45dt02_irq se encarga de definir la devolución de llamada que se realizara en el momento que el buffer proporcionado por el usuario este lleno.

```
STATIC mp_obj_t mp45dt02_irq(mp_obj_t self_in, mp_obj_t handler) {
    mp45dt02_obj_t *self = MP_OBJ_TO_PTR(self_in);

    if (handler != mp_const_none && !mp_obj_is_callable(handler)) {
        mp_raise_ValueError(MP_ERROR_TEXT("invalid callback"));
    }
```

```
    self->callback_for_non_blocking = handler;
    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_2(mp45dt02_irq_obj, mp45dt02_irq);
```

La función mp45dt02_stream_read se encarga de vincular el buffer proporcionado por el usuario al buffer que almacena las muestras PCM, iniciar el índice en 0 y activar la bandera para comenzar a convertir las muestras de PDM a PCM.
(Aunque el nombre está definido como read esta función no obtiene directamente las muestras)

```
STATIC mp_uint_t mp45dt02_stream_read(mp_obj_t self_in, void *buf_in, mp_uint_t size, int *errcode)
{
    mp45dt02_obj_t *self = MP_OBJ_TO_PTR(self_in);

    if (size == 0) {
        return 0;
    }

    self->non_blocking_descriptor.appbuf.buf = (void *)buf_in;
    self->non_blocking_descriptor.appbuf.len = size;
    self->non_blocking_descriptor.index = 0;
    self->non_blocking_descriptor.copy_in_progress = true;
    return size;
}
```

Al igual que en el ejemplo con el STM32CubeIDE el control de la toma de muestras y conversión está ligado a las interrupciones del puerto I2S, sin embargo, en este caso al llenarse el buffer se crea una interrupción para que el usuario pueda tener un mejor control es su programa.

```
void HAL_I2S_RxHalfCpltCallback(I2S_HandleTypeDef *hi2s2) {
    mp45dt02_obj_t *self;
    self = mp45dt02_obj;
    if (self->non_blocking_descriptor.copy_in_progress)  {
        uint16_t runningsum = 0;
        uint16_t *sinc_ptr = sincfilter;
        for (uint8_t i=0; i <= 3 ; i++) {
            PDM_REPEAT_LOOP_16({
```

```
            if (self->dma_buffer[i] & 0x1) {
                runningsum += *sinc_ptr;
            }
            sinc_ptr++;
            self->dma_buffer[i] >>= 1;
            })
        }
        if (self->non_blocking_descriptor.index*2 >= self->non_blocking_descriptor.appbuf.len){
            self->non_blocking_descriptor.copy_in_progress = false;
            mp_sched_schedule(self->callback_for_non_blocking, MP_OBJ_FROM_PTR(self));

        }
        ((uint16_t *)self->non_blocking_descriptor.appbuf.buf)[self->non_blocking_descriptor.index]
= (uint16_t)filtro(runningsum);
        self->non_blocking_descriptor.index++;
    }
}
```

Finalmente se crearon las funciones init y deinit las cuales inicializan o detienen a la clase.

```
STATIC mp_obj_t mp45dt02_init(size_t n_pos_args, const mp_obj_t *pos_args, mp_map_t *kw_args) {

    mp45dt02_obj_t *self = MP_OBJ_TO_PTR(pos_args[0]);
    mp45dt02_deinit(MP_OBJ_FROM_PTR(self));
    mp45dt02_init_helper(self);
    self->non_blocking_descriptor.copy_in_progress = true;
    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_KW(mp45dt02_init_obj, 1, mp45dt02_init);

STATIC mp_obj_t mp45dt02_deinit(mp_obj_t self_in) {

    mp45dt02_obj_t *self = MP_OBJ_TO_PTR(self_in);
    dma_deinit(self->dma_descr_rx);
    HAL_I2S_DeInit(&self->hi2s2);
    __HAL_RCC_SPI2_CLK_DISABLE();
    HAL_GPIO_DeInit(GPIOC, GPIO_PIN_3);
    HAL_GPIO_DeInit(GPIOB, GPIO_PIN_12|GPIO_PIN_13);
    __SPI2_FORCE_RESET();
    __SPI2_RELEASE_RESET();
    __SPI2_CLK_DISABLE();
    return mp_const_none;
}
```

```
STATIC MP_DEFINE_CONST_FUN_OBJ_1(mp45dt02_deinit_obj, mp45dt02_deinit);
```

En la siguiente tabla se define el string que corresponde a cada función, la cual se escribirá en MicroPython para invocarla.

```
STATIC const mp_rom_map_elem_t mp45dt02_locals_dict_table[] = {
    { MP_ROM_QSTR(MP_QSTR_init),            MP_ROM_PTR(&mp45dt02_init_obj) },
    { MP_ROM_QSTR(MP_QSTR_readinto),        MP_ROM_PTR(&mp_stream_readinto_obj) },
    { MP_ROM_QSTR(MP_QSTR_deinit),          MP_ROM_PTR(&mp45dt02_deinit_obj) },
    { MP_ROM_QSTR(MP_QSTR_irq),             MP_ROM_PTR(&mp45dt02_irq_obj) },

};
MP_DEFINE_CONST_DICT(mp45dt02_locals_dict, mp45dt02_locals_dict_table);
```

Después en la constante mp45dt02_type se vinculan los parameros que tendrá la clase, como el nombre, la función print, la función make_new etc.

```
const mp_obj_type_t mp45dt02_type = {
    { &mp_type_type },
    .name = MP_QSTR_ophyra_mp45dt02,
    .print = mp45dt02_print,
    .getiter = mp_identity_getiter,
    .iternext = mp_stream_unbuffered_iter,
    .protocol = &i2s_stream_p,
    .make_new = mp45dt02_make_new,
    .locals_dict = (mp_obj_dict_t *)&mp45dt02_locals_dict,
};
```

Una última matriz se define para asociar el string _name_ con el nombre del módulo y el nombre de la clase y se convierte a un diccionario.

```
STATIC const mp_rom_map_elem_t ophyra_mp45dt02_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_ophyra_mp45dt02) },
    { MP_ROM_QSTR(MP_QSTR_MP45DT02), MP_ROM_PTR(&mp45dt02_type) },
};
STATIC MP_DEFINE_CONST_DICT(mp_module_ophyra_mp45dt02_globals, ophyra_mp45dt02_globals_table);
```

Por último, se registra el módulo para ser visible desde MicroPython

```
MP_REGISTER_MODULE(MP_QSTR_ophyra_mp45dt02, ophyra_mp45dt02_user_cmodule,
MODULE_OPHYRA_MP45DT02_ENABLED);
```

En el archivo mpconfigport.h se debe incluir el tercer argumento como macro para poder habilitar el módulo.

Bibliografía

[1] P. Thomas Kite, «PDM Microphones,» de *Understanding PDM Digital Audio (PDF)*, 2012, p. 6.

[2] L. G. Morales, Introducción al Audio Digital, Alemania: Books on Demand (BoD), 2020.

[3] L. e. l. Wikipedia, «Pulse-code modulation,» [En línea]. Available: https://en.wikipedia.org/wiki/Pulse-code_modulation. [Último acceso: 5 12 2021].

[4] STMicroelectronics, «STM32 Microphone Audio Acquisition: Part 3, PDM to PCM Conversion,» 22 12 2020. [En línea]. Available: https://www.youtube.com/watch?v=5lH-tQw0tlU. [Último acceso: 13 10 2021].

[5] M. A. S. S. Erick Matías Balboa Morales, ANÁLISIS, DISEÑO Y CONSTRUCCIÓN, CONCEPCIÓN – CHILE: Universidad del Bío-Bío, 2016, p. 17.

Anexos

Anexo 1 Coeficiente del filtro FIR (promedio móvil)

0, 2, 9, 21, 39, 63, 94, 132, 179, 236, 302, 379, 467, 565, 674, 792,920, 1055, 1196, 1341, 1487, 1633, 1776, 1913, 2042, 2159, 2263, 2352, 2422, 2474, 2506, 2516, 2516,2506, 2474, 2422, 2352, 2263, 2159, 2042, 1913, 1776, 1633, 1487, 1341, 1196, 1055, 920, 792,674, 565, 467, 379, 302, 236, 179, 132, 94, 63, 39, 21, 9, 2, 0

Anexo 2 Coeficiente del filtro FIR (filtrar audio)

0.005383878543905767, 0.0008745285494715447, 0.0008482121881024766, 0.0007460661434193918, 0.0005591943982935655, 0.0002791528160262167, -0.00009918091362182439, -0.0005811661980721277, -0.0011671893566906496, -0.0018565212849891023, -0.002645092214276294, -0.0035243971082051908, -0.00448229643396019, -0.005504049776296296, -0.00657090631514472, -0.007660384600351277, -0.00874671862109321 4, -0.009800854443203912, -0.010792801755240751, -0.011690626248753846, -0.012460044110719715, -0.013066138954722997, -0.013478050976535164, -0.013661419705534182, -0.01358814972261887, -0.013230659567661602, -0.012569445996477512, -

0.01157485826919379, -0.01025972395735936, -0.008591627155855458, -0.006579739518908161 5, -0.004235785042586487, -0.0015738584167381312, 0.0013871360655679007, 0.004618933545599627 5, 0.008086638890839394, 0.011747688496100916, 0.015554266016657506, 0.019454870341107575, 0.023394135061082386, 0.02731338070322758, 0.031152907654284957, 0.03485296501818983, 0.03835421967664724, 0.04159966314102669, 0.04453367705215179, 0.04710590599599399, 0.049273780397548254, 0.05100223257635541, 0.052255648506699295, 0.05301808055521172, 0.05327320160592689, 0.05301808055521172, 0.052255648506699295, 0.05100223257635541, 0.049273780397548254, 0.04710590599599399, 0.04453367705215179, 0.04159966314102669, 0.03835421967664724, 0.03485296501818983, 0.031152907654284957, 0.02731338070322758, 0.023394135061082386, 0.019454870341107575, 0.015554266016657506, 0.011747688496100916, 0.008086638890839394, 0.004618933545599627 5, 0.0013871360655679007, -0.0015738584167381312, -0.004235785042586487, -0.006579739518908161 5, -0.008591627155855458, -0.01025972395735936, -0.01157485826919379, -0.012569445996477512, -0.013230659567661602, -0.01358814972261887, -0.013661419705534182, -0.013478050976535164, -0.013066138954722997, -0.012460044110719715, -0.011690626248753846, -0.010792801755240751, -0.009800854443203912, -0.008746718621093214, -0.007660384600351277, -0.00657090631514472, -0.005504049776296296, -0.004482296433960019, -0.0035243971082051908, -0.002645092214276294, -0.0018565212849891023, -0.0011671893566906496, -0.0005811661980721277, -0.0009991809136218243 9, 0.0002791528160262167, 0.0005591943982935655, 0.0007460661434193918, 0.0008482121881024766, 0.0008745285494715447, 0.005383878543905767