Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERÍAS INDUSTRIALES

Grado en Ingeniería en Electrónica Industrial y Automática

ESTUDIO DE PROCESADORES ARM CORTEX-M4 CON UNIDAD DE COMA FLOTANTE

Autor:

Carrasco Alonso, David

Tutor:

Plaza Pérez, Francisco José

Dpto. Tecnología Electrónica

Valladolid, Mayo 2020

RESUMEN

Este Trabajo Fin de Grado consiste en el estudio de un microprocesador ARM Cortex-M4 con unidad de coma flotante.

Se estudia qué beneficios se consiguen al invertir el coste requerido para obtener un procesador que tenga coma flotante, y qué dificultades se presentan al tratar de programar en punto fijo.

Para llevar esto a cabo se realiza una aplicación en una placa de STMicroelectronics que contiene un Cortex-M4, y se utilizan las herramientas software de desarrollo puestas a disposición de los usuarios por STMicroelectronics y ARM: CubeMX y µVision.

PALABRAS CLAVE procesador microcontrolador arm stmmicroelectronics flotante

ABSTRACT

This Final Degree Project consist of the study of an ARM Cortex-M4 microprocessor with floating point unit.

We will study the benefits of investing in a processor that has a floating point unit, and the difficulties that arise when trying to program using fixed point math.

In order to do this we will develop an application for one of STMicroelectronics' Discovery boards that contain a Cortex-M4, using the software tools made available to developers by STMicroelectronics and ARM: CubeMX and µVision.

KEYWORDS processor microcontroller arm stmicroelectronics floating

Contenido

Introducción

Objetivos

En este TFG se va a estudiar el procesador Cortex-M4F, diseñado por la compañía ARM Holdings, el cual tiene una unidad de coma flotante (FPU, Floating Point Unit). En particular, se pretende comprender qué ofrece al programador del microcontrolador, de forma práctica, el hecho de tener o no una FPU, para ayudar a determinar en qué casos puede convenir el uso de un procesador con coma flotante respecto a uno que no la tenga.

Se estudiará el desarrollo de aplicaciones en punto fijo debido a la carencia de FPU para ahorrar costes, y sus ventajes e inconvenientes respecto a coma flotante en función del tipo de proyecto.

También se desea ofrecer una guía introductoria de programación de placas tipo “Discovery” proporcionadas por STMicroelectronics utilizando las herramientas software que ARM pone a disposición de los programadores. Para ello se ha elegido la placa STM32f429-Discovery.

Justificación

Interesa estudiar qué implica trabajar con o sin FPU porque los procesadores con FPU son más caros que sus equivalentes que no la tienen. La FPU es una parte que se añade al procesador diseñada para realizar operaciones con números representados en coma flotante más rápidamente (menos ciclos de reloj). Permite ejecutar instrucciones específicas para sumar, restar, multiplicar, dividir, acumular y hallar raíz cuadrada de números representados en coma flotante.

Si no se dispone de FPU, realizar estas operaciones en coma flotante requiere el uso de otras instrucciones, las mismas que se usan para realizar operaciones con enteros. No poder utilizar instrucciones específicas para ejecutar operaciones aritméticas en coma flotante provoca que los tiempos de ejecución de estas operaciones crezca. Para evitar este problema, el programador tiene la opción de realizar estas operaciones en punto fijo, en lugar de en coma flotante.

Operar en punto fijo permite representar números fraccionarios utilizando números enteros, pero tiene el inconveniente de que conlleva un esfuerzo extra para el programador. En esta memoria se verá por qué es más difícil hacer operaciones de números fraccionarios en punto fijo que en coma flotante.

Se tratará de comprender qué supone para el desarrollador de una aplicación tener que trabajar un procesador sin FPU en lo que a tiempo de desarrollo se refiere, y comparar dicho coste de tiempo de un ingeniero respecto al coste extra de comprar procesadores con FPU, que son más caros pero pueden ahorrar tiempo al programador.

Desarrollo

Aplicación a realizar

Para observar los efectos de tener una FPU interesaría desarrollar una aplicación en la que se requiera realizar operaciones matemáticas continuamente de forma intensiva, y en la que se pueda modificar fácilmente la carga de trabajo de la CPU.

Para ello se ha hecho una aplicación que consiste en realizar un filtrado de una señal que la placa recibe a través de uno de sus convertidores digital a analógico, y que tras ser filtrada se emite a través de uno de uno de sus convertidores analógico a digital.

Realizar un filtrado es una buena forma de aprender a programar una de estas placas, ya que implica configurarla para recibir una señal, realizar operaciones con esta señal y luego emitir otra señal. También es una buena manera de comprobar el efecto de tener FPU, ya que podemos modificar fácilmente parámetros como la frecuencia de muestreo, o la complejidad y número de operaciones que se realizan en cada periodo de muestreo, lo cual nos permitirá explorar los límites del procesador cuando usamos la FPU y cuando no la usamos.

Además, aprovecharemos la oportunidad para aprender a programar aplicaciones que usen la pantalla táctil que posee nuestra placa. En nuestro caso vamos a poder introducir manualmente ciertos parámetros del filtrado a través de la pantalla. El usuario de la aplicación podrá elegir si quiere que el filtrado se realice en coma flotante o punto fijo, la frecuencia de corte y la frecuencia de muestreo.

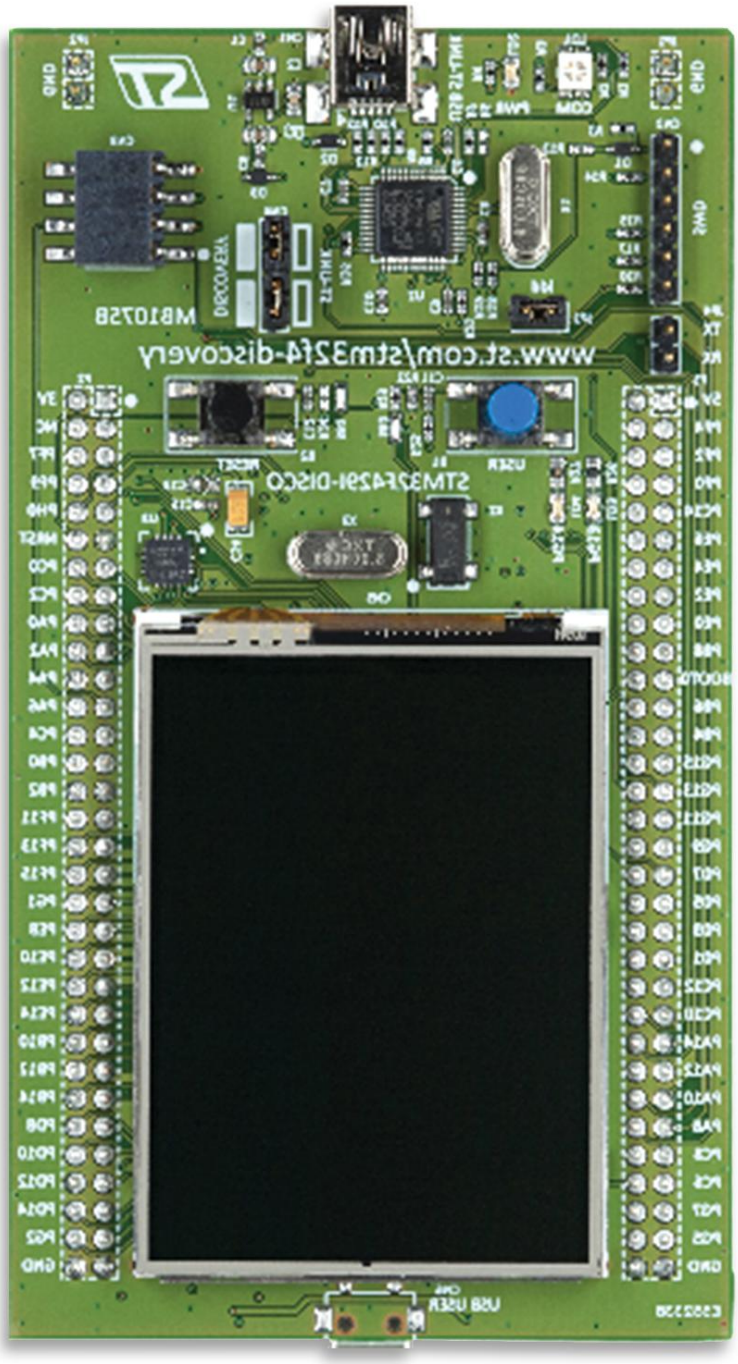

Ilustración 1: Placa ARM STM32F429

Herramientas software utilizadas

Para la realización del proyecto se utilizará µVision, la IDE ofrecida por ARM, la compañía que diseña el procesador:

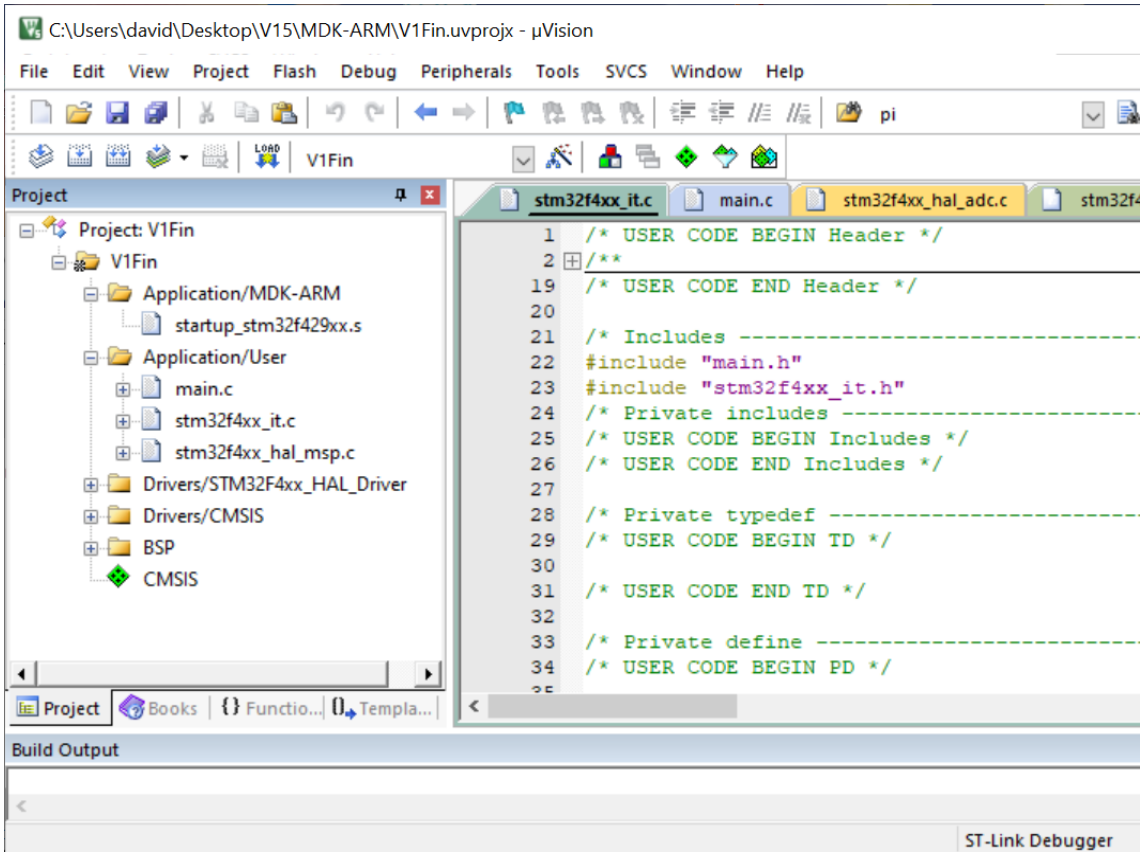

Ilustración 2: µVision

Al descargar MDK tendremos a nuestra disposición las herramientas necesarias para compilar y depurar nuestro software, así como compatibilidad con STM32CubeMX.

STM32CubeMX es un programa que escribe parte del código básico que de otra manera tendríamos que escribir nosotros (el programador del microcontrolador). Esto incluye todas las líneas necesarias para la configuración de temporizadores, relojes, convertidores analógico a digital, digital a analógico, memoria, drivers, etc.

Esto nos permite ahorrar mucho tiempo al programar porque no es necesario que el programador se preocupe de aprender todos los detalles específicos de cada microcontrolador o placa necesarios antes de programar en dicho microcontrolador o placa.

El link de descarga de STM32CubeMX puede consultarse en el Anexo.

Una vez descargado e instalado, ejecutamos el programa:

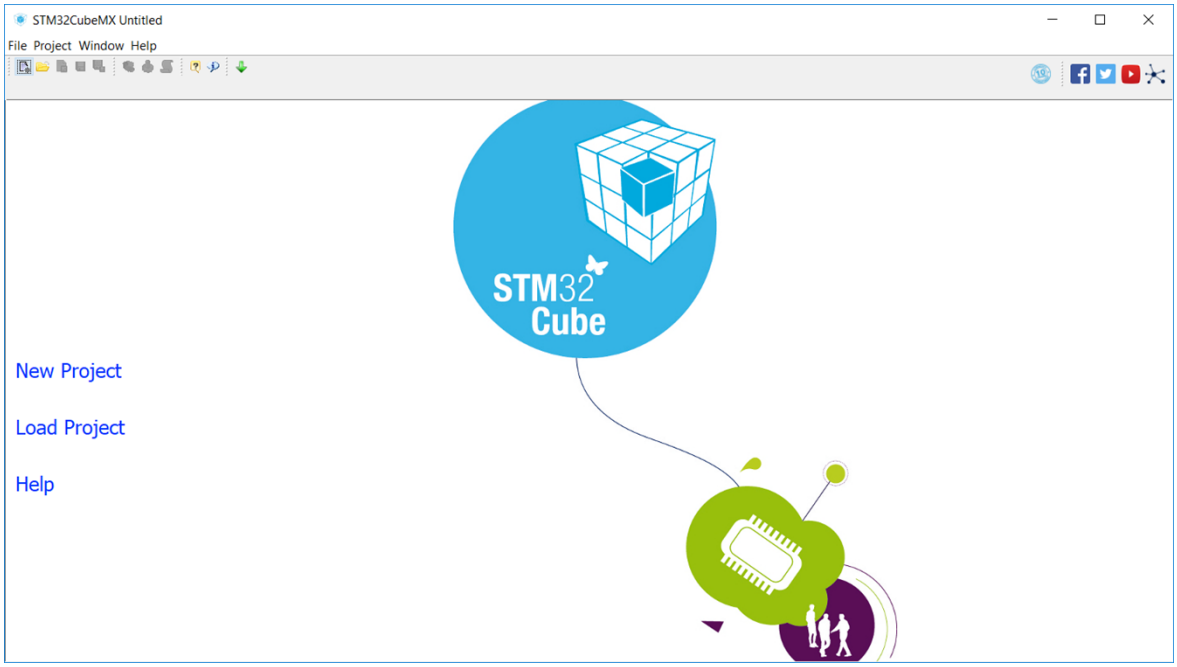

Ilustración 3: Lanzar STM32CubeMX

Al hacer click en “New Project” podemos seleccionar nuestra placa, en este caso STM32F429:

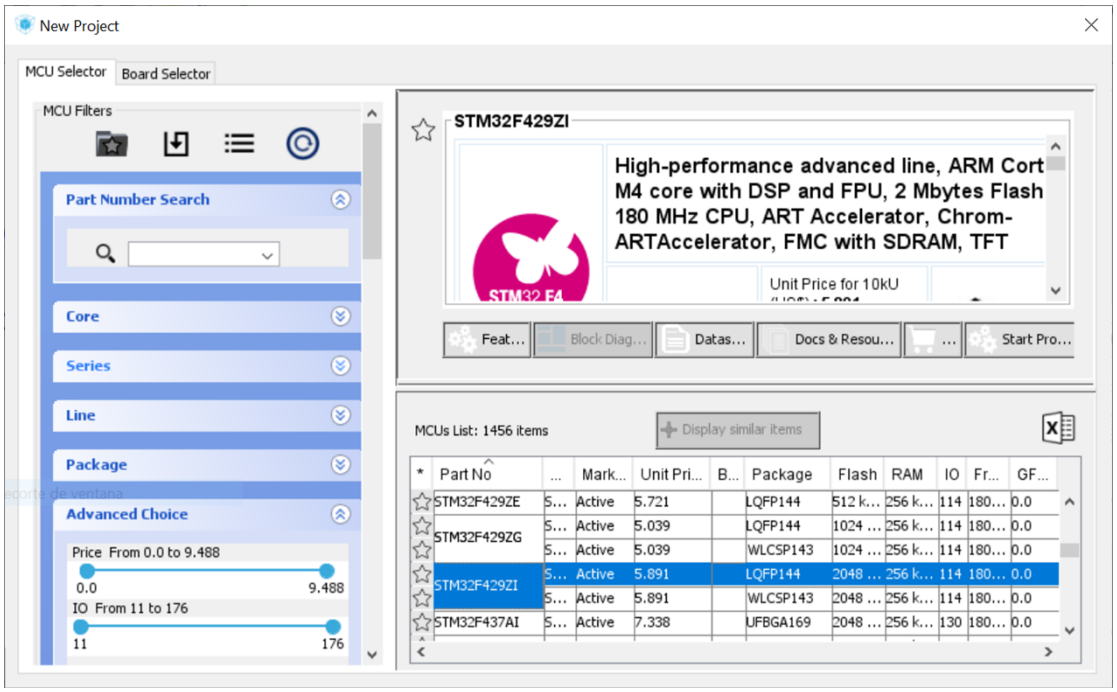

Ilustración 4: Seleccionar nuestra placa en STM32CubeMX

Una vez hemos seleccionado nuestra placa, hacemos click en “Start Project” y aparece la siguiente ventana:

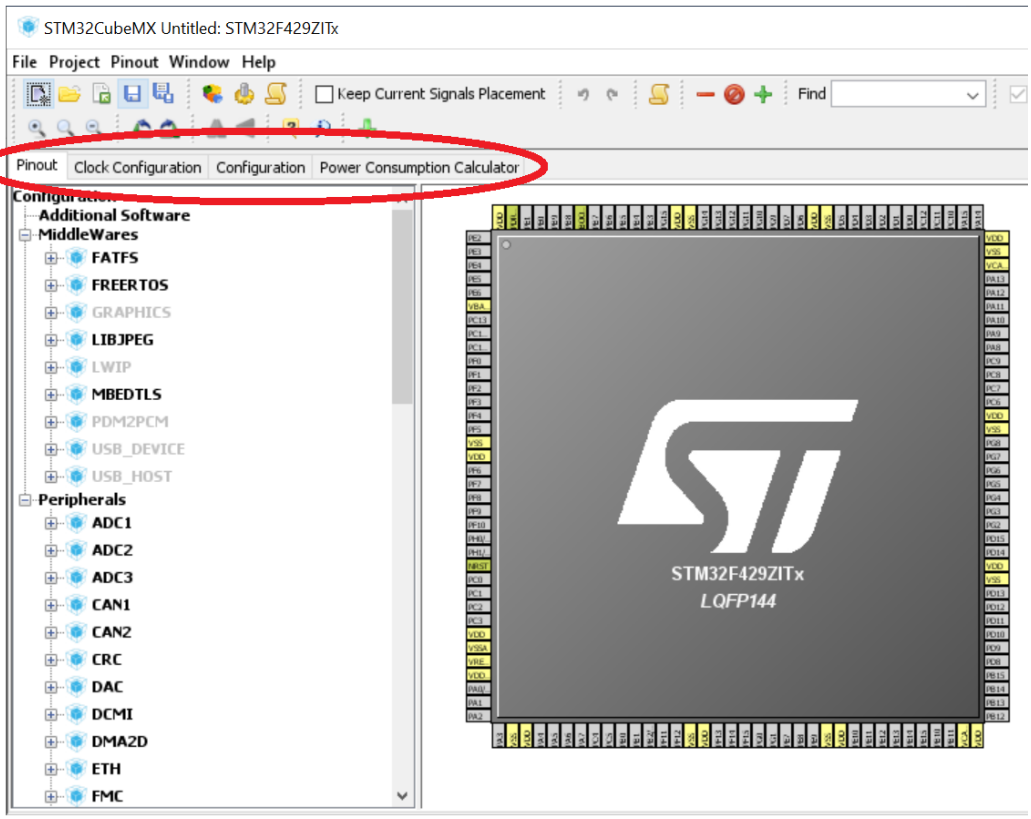

Ilustración 5: Pestañas de configuración

Aquí tenemos las pestañas Pinout, Clock Configuration, Configuration y Power Consumption Calculator.

En la ventana Pinout podemos activar los pines necesarios para nuestra aplicación. En nuestro caso, como vamos a realizar un filtrado, necesitamos un convertidor analógico digital (ADC) y un convertidor digital analógico (DAC).

También queremos que estos ADC y DAC operen a una frecuencia determinada, la frecuencia de muestreo. Para ello necesitaremos un temporizador. Tenemos varios temporizadores a nuestra disposición; elegimos Timer 6.

El ADC realizará un muestreo cada vez que reciba la señal del Timer 6.

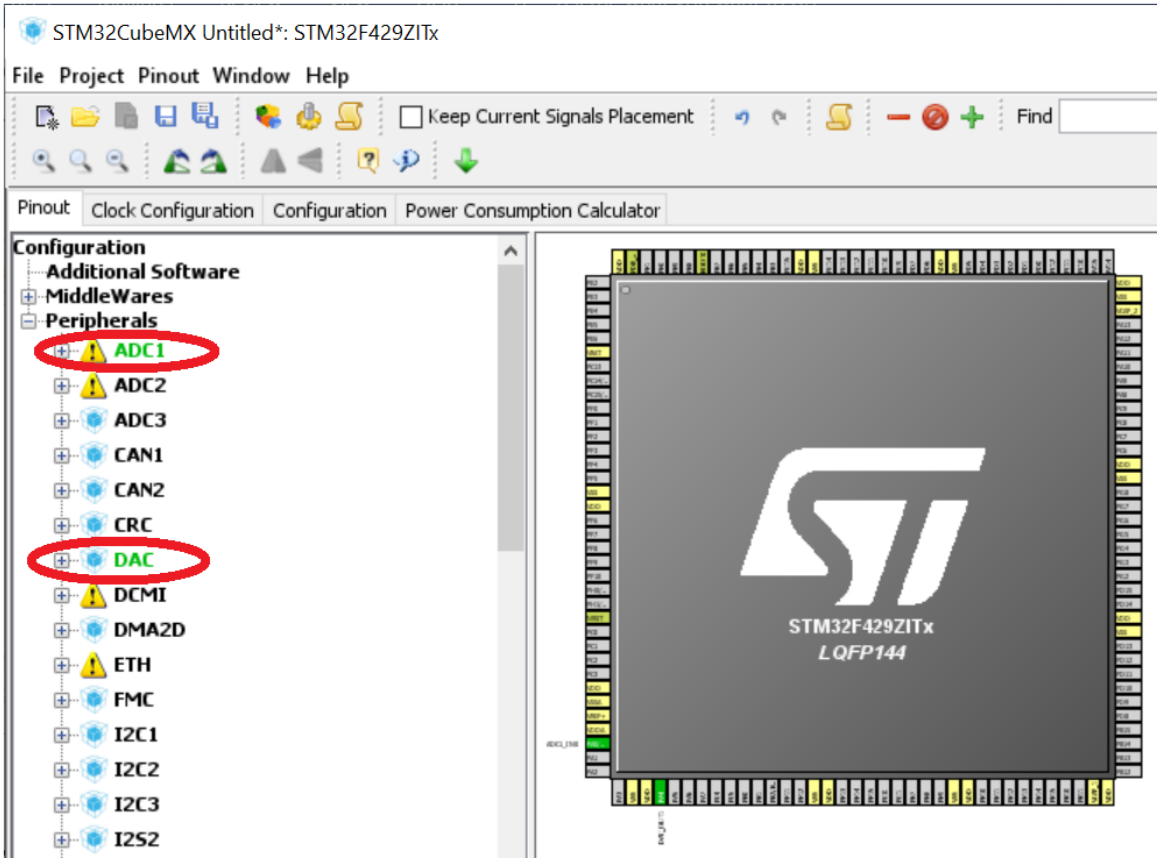

Ilustración 6: Activación de pines

En la pestaña “clock” vamos a seleccionar la máxima frecuencia; para esta placa es 180Mhz:

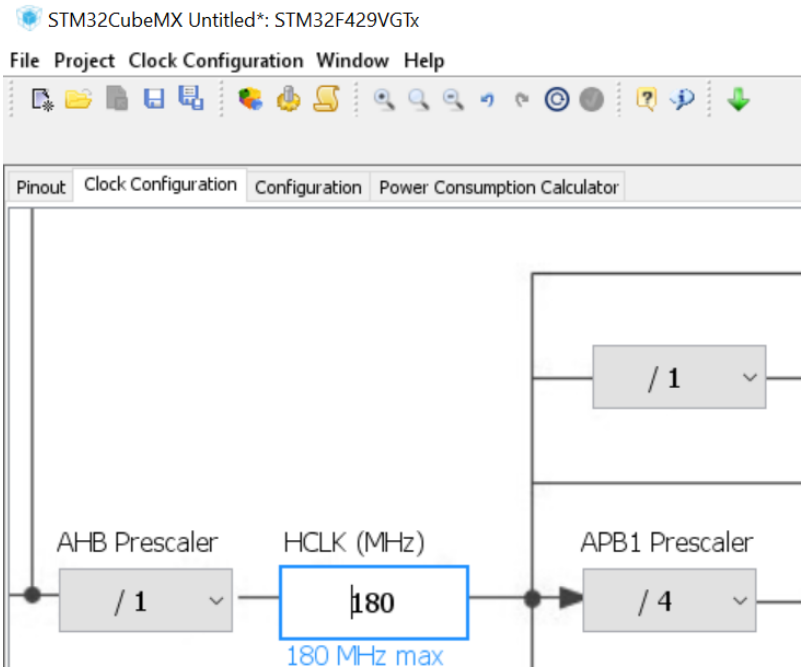

Ilustración 7: Ajuste de las frecuencias de los clocks

En la pestaña “Configuration” nos interesa configurar el Timer 6.

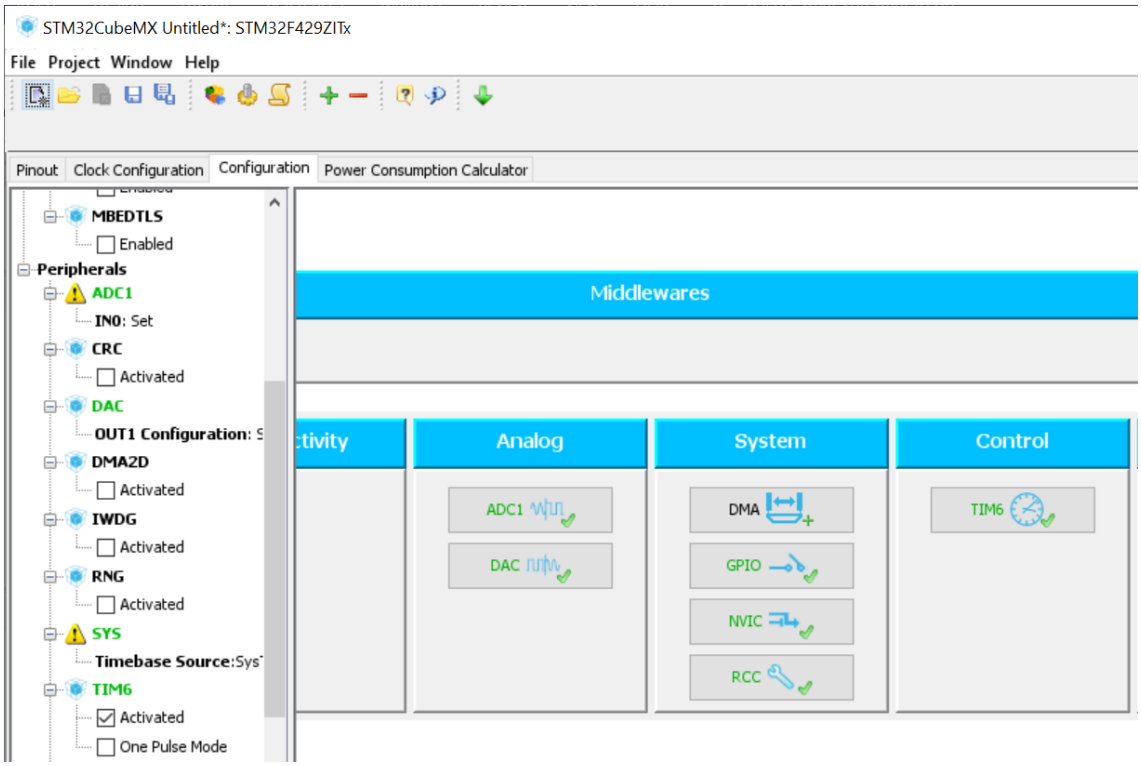

Ilustración 8: Pestaña "Configuration"

Hacemos click en TIM6 y configuramos nuestro temporizador para que tenga la frecuencia adecuada:

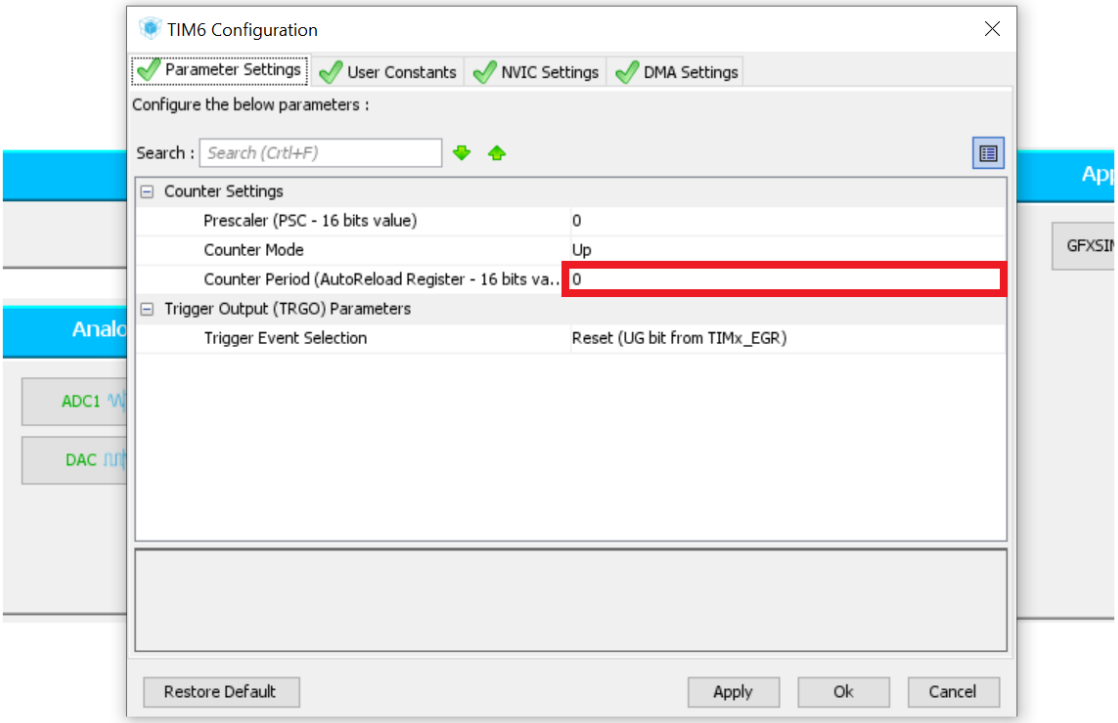

Ilustración 9: Configuración del temporizador

El campo “Counter Period” nos permite determinar la frecuencia del temporizador. Dicha frecuencia está determinada por la siguiente fórmula:

$$f_{INTERRUPCIÓN} = \frac{f_{CLK\_CNT}}{TIM\_ARR + 1}$$

Ilustración 10: Fórmula para hallar "Counter Period" deseado

El valor de Fclk_cnt es la frecuencia que le llega al Timer 6. Sabemos su valor de la siguiente manera:

1) El Timer 6 está en el bus APB1, según la página 67 del Reference Manual:

Table 1. STM32F4xx register boundary addresses (continued)

Boundary address	Peripheral	Bus	Register map
0x4000 7400 - 0x4000 77FF	DAC		*Section 14.5.15: DAC register map on page 453*
0x4000 7000 - 0x4000 73FF	PWR		*Section 5.6: PWR register map on page 149*
0x4000 6800 - 0x4000 6BFF	CAN2		*Section 32.9.5: bxCAN register map on page 1118*
0x4000 6400 - 0x4000 67FF	CAN1		
0x4000 5C00 - 0x4000 5FFF	I2C3		*Section 27.6.11: I2C register map on page 872*
0x4000 5800 - 0x4000 5BFF	I2C2		
0x4000 5400 - 0x4000 57FF	I2C1		
0x4000 5000 - 0x4000 53FF	UART5		*Section 30.6.8: USART register map on page 1018*
0x4000 4C00 - 0x4000 4FFF	UART4		
0x4000 4800 - 0x4000 4BFF	USART3		
0x4000 4400 - 0x4000 47FF	USART2		
0x4000 4000 - 0x4000 43FF	I2S3ext		*Section 28.5.10: SPI register map on page 925*
0x4000 3C00 - 0x4000 3FFF	SPI3 / I2S3		
0x4000 3800 - 0x4000 3BFF	SPI2 / I2S2	APB1	
0x4000 3400 - 0x4000 37FF	I2S2ext		
0x4000 3000 - 0x4000 33FF	IWDG		*Section 21.4.5: IWDG register map on page 712*
0x4000 2C00 - 0x4000 2FFF	WWDG		*Section 22.6.4: WWDG register map on page 719*
0x4000 2800 - 0x4000 2BFF	RTC & BKP Registers		*Section 26.6.21: RTC register map on page 836*
0x4000 2000 - 0x4000 23FF	TIM14		*Section 19.5.12: TIM10/11/13/14 register map on page 694*
0x4000 1C00 - 0x4000 1FFF	TIM13		
0x4000 1800 - 0x4000 1BFF	TIM12		*Section 19.4.13: TIM9/12 register map on page 684*
0x4000 1400 - 0x4000 17FF	TIM7		*Section 20.4.9: TIM6 and TIM7 register map on page 707*
0x4000 1000 - 0x4000 13FF	TIM6		
0x4000 0C00 - 0x4000 0FFF	TIM5		*Section 18.4.21: TIMx register map on page 648*
0x4000 0800 - 0x4000 0BFF	TIM4		
0x4000 0400 - 0x4000 07FF	TIM3		

Ilustración 11: Correspondencia entre buses y temporizadores según el datasheet de la placa

2) La pestaña “Clock Configuration” de CubeMX nos dice que a los timers del bus APB1 les llega una señal de reloj de 90MHz:

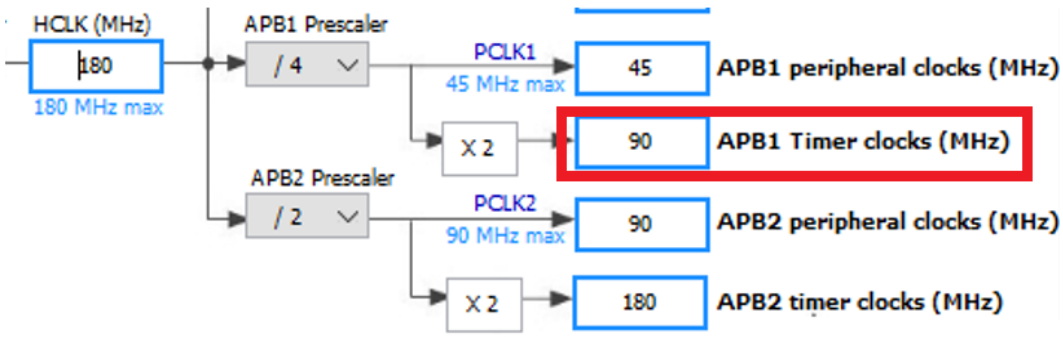

Ilustración 12: Seleccionando frecuencias para el Timer 6 en CubeMX

Por tanto, al timer 6 le llegan 90Mhz; el valor que ponemos en “Counter Period” determina cada cuántos “ticks” que recibe el temporizador, el temporizador emite una señal. Es decir, si ponemos 2249 en Counter Period, como le llegan 90MHz: 90000000/(2249 + 1) = 40Khz. El temporizador le indicará al ADC que muestree 40000 veces por segundo.

Entender el significado de “Counter Period” es útil para más tarde modificar el periodo de muestreo fácilmente desde dentro del código, poniendo el que nosotros queramos. Ahora estamos estableciendo este valor a través de CubeMX, pero cuando hayamos generado el código con esta herramienta podremos modificarlo modificando el valor de “htim6.Init.Period” (por ejemplo, si quisiésemos un muestreo de 40KHz, le daríamos un valor de 2249) dentro de la función encargada de configurar el timer 6, “MX_TIM6_Init”.

En la siguiente imagen vemos la parte del código que proporciona CubeMX para configurar el timer 6. Aún no se ha terminado de utilizar CubeMX para generar el código, pero esta es la parte del código que CubeMX nos proporcionará:

```
static void MX_TIM6_Init(void)
{

  TIM_MasterConfigTypeDef sMasterConfig = {0};

  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 0;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = 2249;
  htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    Error_Handler();
  }
```

Ilustración 13: Parte del código generado por CubeMX para configurar el Timer 6

En nuestra aplicación, este temporizador determina la frecuencia con la que se ejecuta la función que:

- Recoge un dato del ADC y lo pone al final del vector que va guardando la señal de entrada.
- Elimina el dato más antiguo del vector.
- Genera una señal de salida a partir de dicho vector (filtrado).

Es decir, determina la frecuencia de muestreo. Como la frecuencia de muestreo es algo que puede ser elegida por el usuario a través de la pantalla, en lugar de un valor fijo, "htim6.Init.Period" ha de poder ser modificado por el usuario. Para ello se ha modificado la función "MX_TIM6_Init" generada por CubeMX, añadiéndose un parámetro de entrada a la función:

```
static void MX_TIM6_Init(int TIM_ARR)
{

  TIM_MasterConfigTypeDef sMasterConfig = {0};

  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 0;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = TIM_ARR; // TIM_ARR depende de la frec. de muestreo
  htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    Error_Handler();
  }
```

Ilustración 14: Configuración modificada del Timer 6

El valor de TIM_ARR se obtiendrá mediante la siguiente función, que lo que hace es obtener el parámetro de entrada de "MX_TIM6_Init" a partir de la frecuencia de muestreo, utilizando la fórmula de la Ilustración 10, y teniendo en cuenta que al Timer 6 le llegan 90MHz:

```
int CalculaTIM_ARR(int FrecMuest) {
  int TIM_ARR;
  TIM_ARR = (90000000/FrecMuest)-1;
  return TIM_ARR;
}
```

Ilustración 15: Función que calcula TIM_ARR

Una vez explicada la relevancia de la propiedad "Counter Period" del Timer 6 en CubeMX, cómo afecta a la aplicación y cómo se puede modificar el el código C final, terminamos nuestro proyecto de CubeMX:

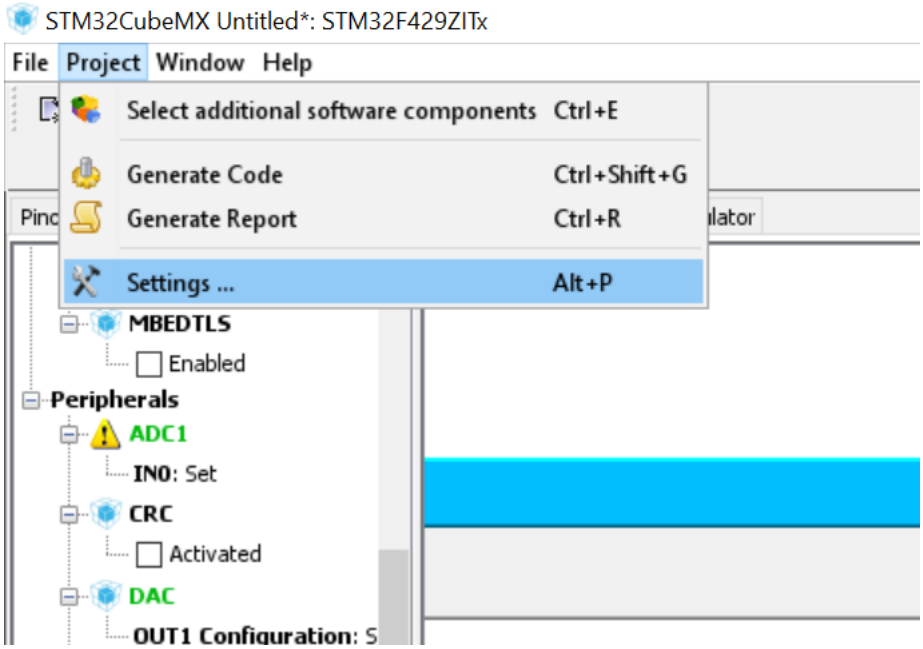

Ilustración 16: Inicio de la generación del código

En nuestro caso, elegimos la opción MDK-ARM V5 porque queremos un proyecto con el que podamos trabajar en Keil MDK:

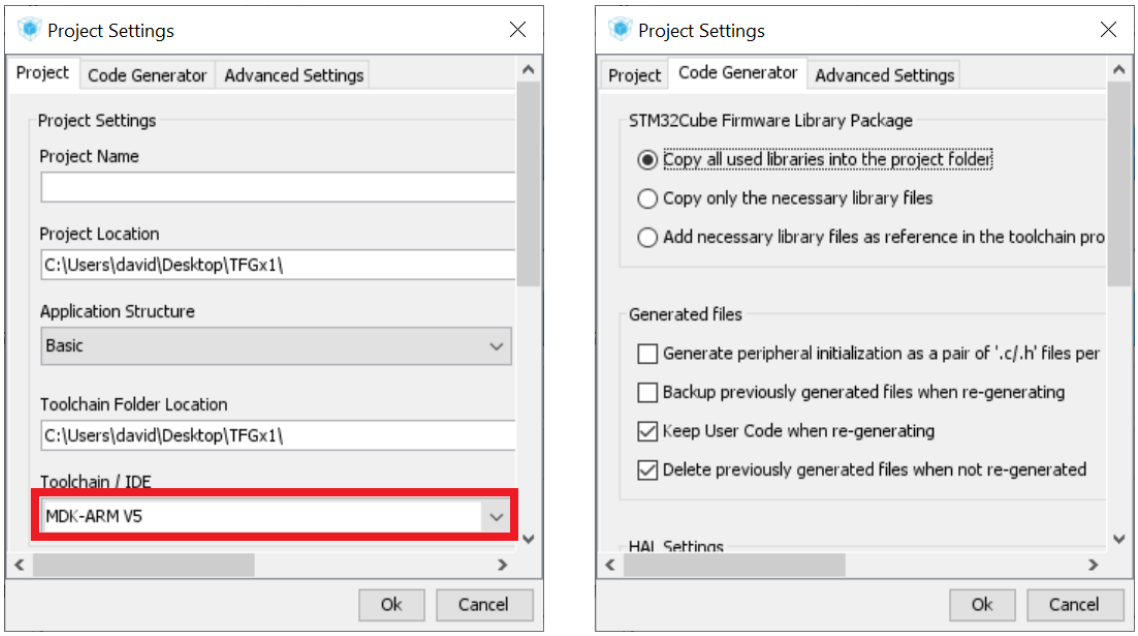

Ilustración 17: Seleccionando preferencias para el código

Una vez hecho el proyecto, ya tenemos el código C generado por CubeMX, que configura todos los aspectos necesarios para poder utilizar la placa como queremos (ADC, DAC, Timer, etc.). Este código es la base para crear nuestra aplicación. Ya no utilizaremos más CubeMX.

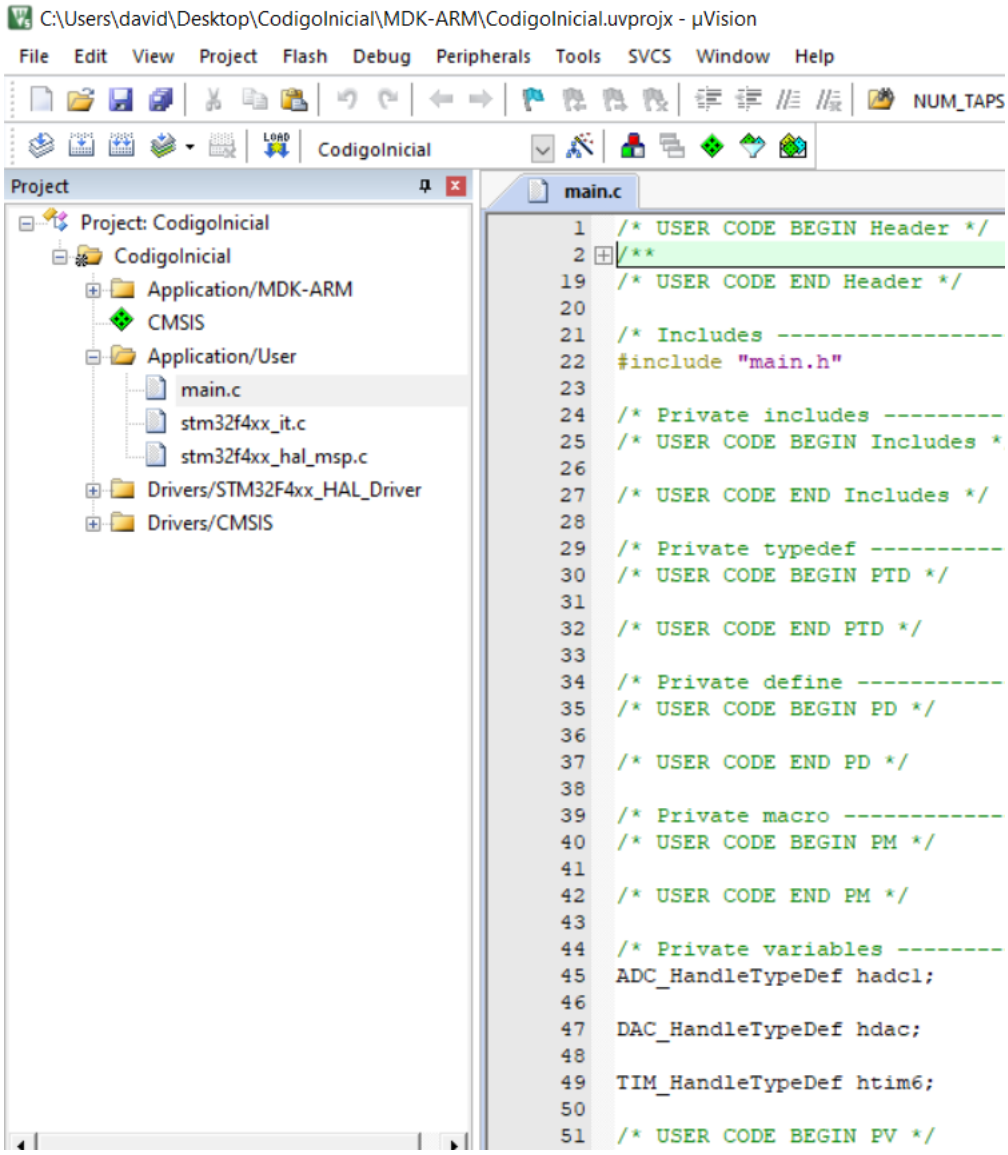

Ilustración 18: Código generado por CubeMX

Algoritmo de filtrado y su implementación en coma flotante

Para realizar el filtrado se realizará un filtro FIR (Finite Impulse Response). El valor de la secuencia de salida "*y*" de un filtro FIR es la suma ponderada de los valores de entrada "*x*" más recientes:

$$y[n] = \sum_{i=o}^{N} h[i]x[n-i]$$

Donde "*h*" es el conjunto de coeficientes del filtro. Diseñar el filtro consiste en encontrar los coeficientes.

Se pueden utilizar varios enventanados diferentes para calcular estos coeficientes. En este caso, se ha elegido diseñar un filtro de ventana rectangular, por ser el más sencillo.

Los coeficientes del filtro de ventana rectangular "*h*" se calculan de la siguiente manera:

$$h(n) = w(n)\, h_d(n)$$

Como $w(n) = 1$ para todo "*n*", por ser rectangular, queda que:

$$h(n) = h_d(n)$$

Y para calcular $h_d(n)$:

$$h_d(n) = \begin{cases} \dfrac{\sin(\omega_c(n-M))}{\pi(n-M)} & n \neq \dfrac{N}{2} \\ \dfrac{\omega_c}{\pi} & n = \dfrac{N}{2} \end{cases}$$

$$\omega_c = \frac{2\pi f_c}{f_s}$$

Donde:

- f_c es la frecuencia de corte
- f_s es la frecuencia de muestreo
- N es igual al nº de coeficientes del filtro menos uno

El valor de N afecta a la calidad del filtrado. A mayor N, mejor será la calidad, pero más tiempo lleva realizar todas las operaciones necesarias para filtrar, por lo que la frecuencia de muestreo máxima se ve reducida.

En este caso, el valor de N se ha introducido en duro en el programa con un valor de 31. Es un valor arbitrario que es lo bastante grande como para que el filtrado dé resultados aceptables, y lo bastante pequeño como para que nuestro procesador pueda filtrar hasta unos 80 KHz (se ha comprobado experimentalmente).

Puesto que en esta aplicación el usuario introduce los valores de f_c y f_s , es necesario crear una función que calcule los coeficientes del $h(n)$ del filtro

```
#define OrdenFiltro 30
```

Ilustración 19: Definición de OrdenFiltro

```
void CalculaCoeficientes(struct Inputs_Usuario Inputs){
  float h_float_temp[OrdenFiltro+1];
  float Wc;
  int n=0;

  Wc = (2*Pi*Inputs.FrecuenciaCorte)/Inputs.FrecuenciaMuestreo;

  //calculo de los coeficientes del filtro en coma flotante
  for (n=0; n<=OrdenFiltro; n++){
    h_float[n]=((float)sin(Wc*(n-OrdenFiltro/2)))/(Pi*(n-OrdenFiltro/2));
  }
  //calculo del coeficiente central del filtro en coma flotante
  if (OrdenFiltro % 2 == 0){
      h_float[OrdenFiltro/2] = Wc/Pi;
  }
```

$\omega_c = \frac{2\pi f_c}{f_s}$

$\frac{\sin(\omega_c(n-M))}{\pi(n-M)}$

$\frac{\omega_c}{\pi}$

Ilustración 20: Cálculo de los coeficientes

La parte que se ocupa de aplicar el filtro a la señal de entrada de la función que se ejecuta en cada período de muestreo (el nombre de la función es "HAL_TIM_PeriodElapsedCallback"; se puede ver la función entera en el anexo) es la siguiente:

```
1  SalidaDAC_float = 0;
   for (int i=0; i<=OrdenFiltro; i++){
     SalidaDAC_float = SalidaDAC_float + ADC_DATA[i]*h_float[OrdenFiltro+1-i];
   }

2  for (int i=0;i<=OrdenFiltro-1;i++){
     ADC_DATA[i] = ADC_DATA[i+1];
   }

3  HAL_ADC_Start(&hadc1);
   HAL_ADC_PollForConversion(&hadc1,100);
   ADC_DATA[OrdenFiltro] = HAL_ADC_GetValue(&hadc1);

4  HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
   HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, SalidaDAC_float);
```

Ilustración 21: Filtrado

La Ilustración 21 muestra el algoritmo de filtrado en coma flotante; la parte en punto fijo no se muestra aquí, porque el objetivo de esta parte del informe es mostrar la idea general del funcionamiento de la aplicación. Más adelante se muestra y se explica la parte del programa que hace el filtrado en punto fijo.

La parte 1 de la Ilustración 21 aplica la filtro FIR con la fórmula mostrada anteriormente:

$$y[n] = \sum_{i=o}^{N} h[i]x[n-i]$$

La parte 2 de la Ilustración 21 desplaza los valores guardados en la variable ADC_DATA una posición a la izquierda, dejando la última posición libre para que se pueda guardar en esta última posición el valor recién muestreado sin que pise datos:

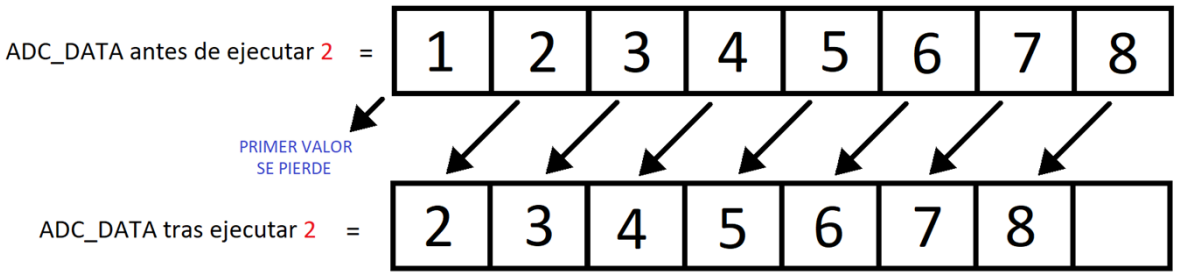

Ilustración 22: Vector ADC_DATA antes y después de ejecutar el código 2

La parte 3 coge un dato a través de ADC y lo guarda en la última posición del vector ADC_DATA:

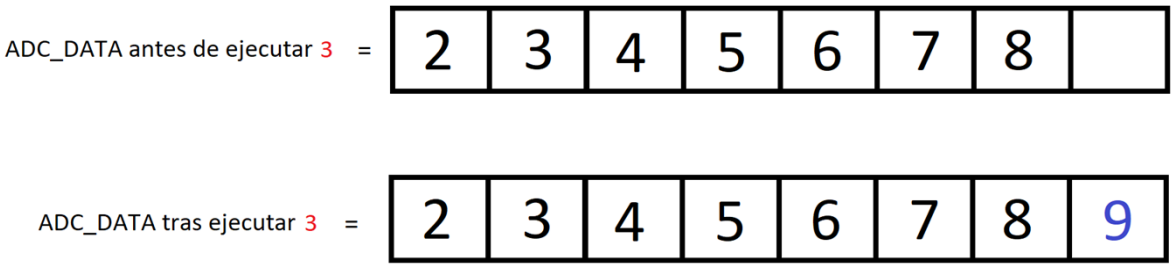

Ilustración 23: Vector ADC_DATA antes y después de ejecutar el código 3

La parte 4 de la Ilustración 21 emite por el DAC el valor obtenido tras aplicar el filtro al vector de muestras ADC_DATA.

Pantalla

Se aprovechará el display de la placa para que el usuario pueda introducir si desea que los cálculos se realicen en coma flotante o punto fijo, la frecuencia de corte y la frecuencia de muestreo. El código que implementa estos menús es largo, por lo que en lugar de mostrarse aquí se muestra en el anexo.

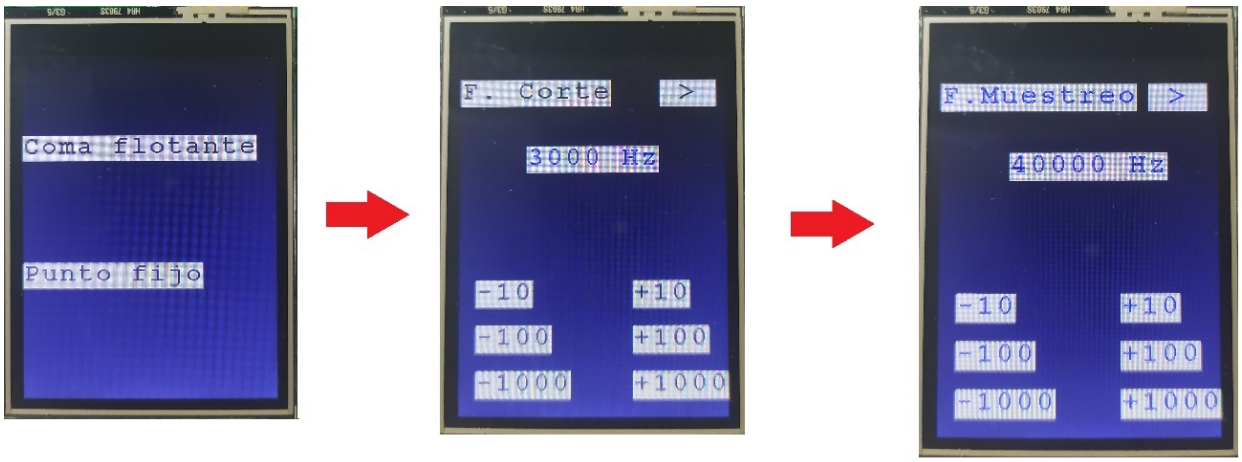

Ilustración 24: Los tres menús

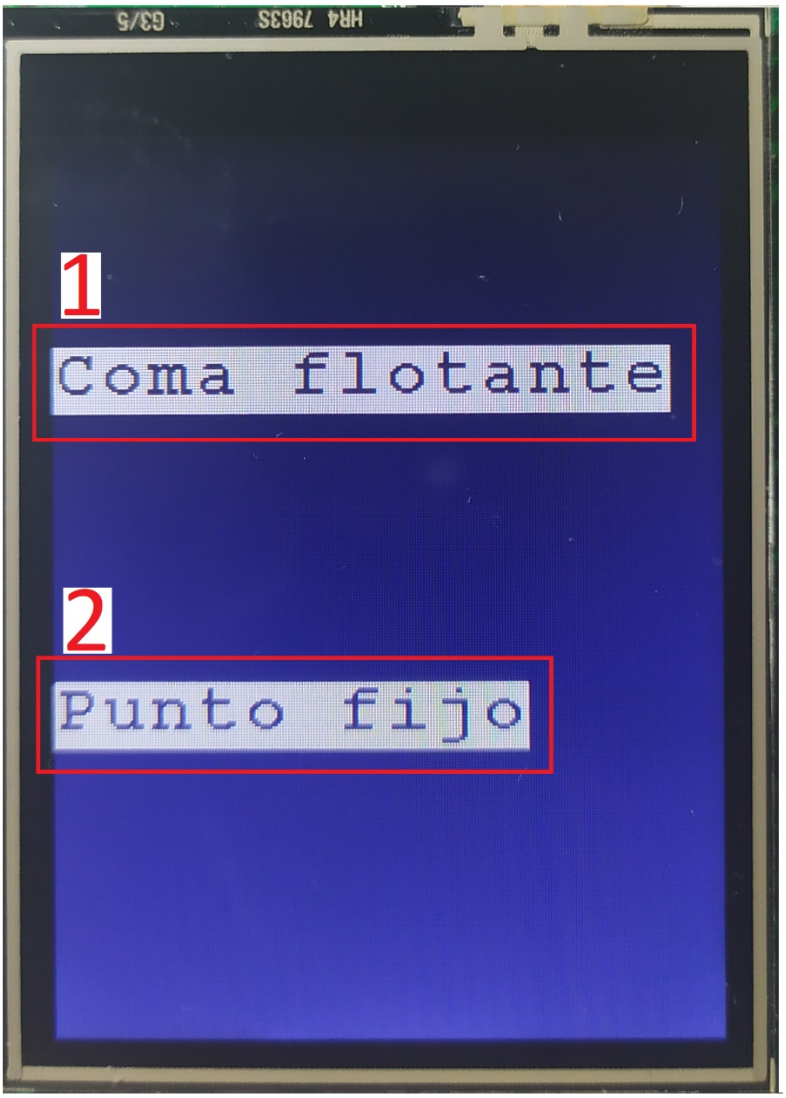

Ilustración 25: Primer menú

Tenemos dos botones. El botón 1 (“Coma flotante”) indica que los cálculos del filtrado se realizarán en coma flotante. Los coeficientes del filtro FIR y el resto de valores involucrados en los cálculos de la señal filtrada están definidos en coma flotante (tipo “float” en el lenguaje C).

Si pulsamos el botón 2 los valores estarán en punto fijo. Tras seleccionar “Coma flotante” o “Punto fijo” especificaremos la frecuencia de corte.

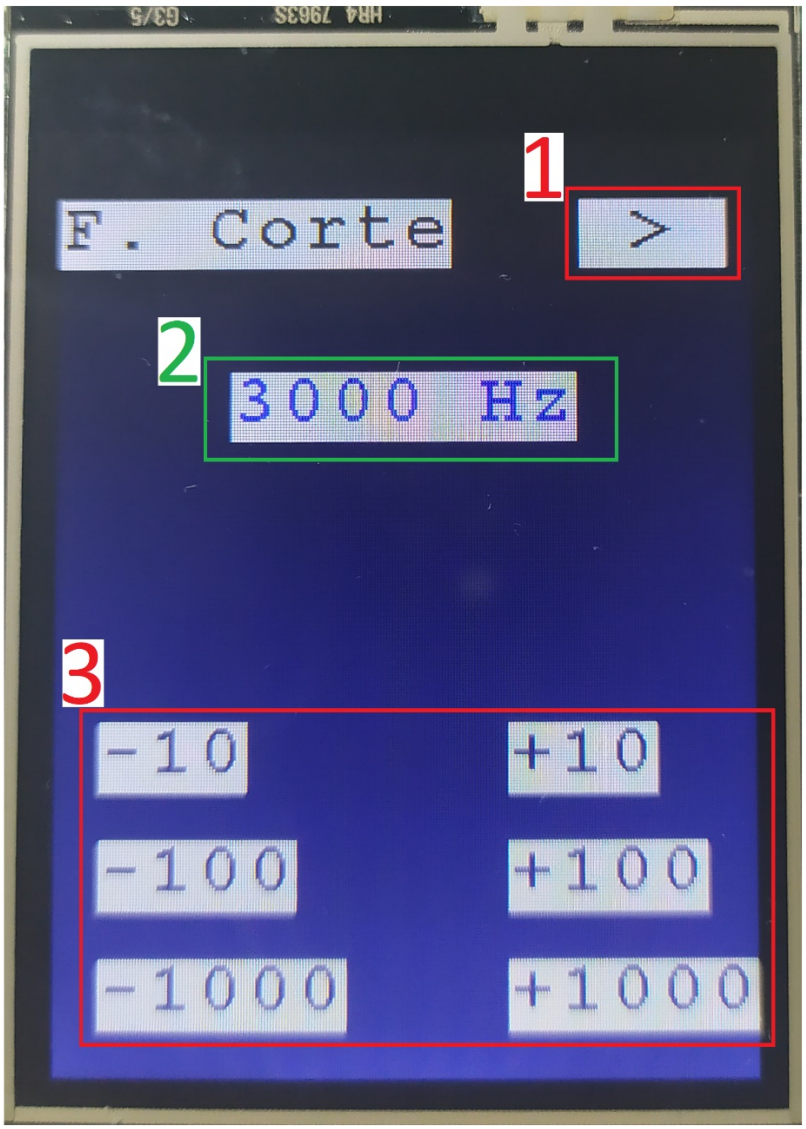

Ilustración 26: Segundo menú

Tras seleccionar “Coma flotante” o “Punto fijo”, podemos determinar la frecuencia de corte mediante los botones 3. Este grupo de 6 botones permite variar la frecuencia indicada en 2, donde se muestran la frecuencia de corte (en herzios).

Por ejemplo, al pulsar el botón “-10”, la frecuencia de corte pasaría a ser 2090 Hz.

Una vez se ha terminado de ajustar la frecuencia deseada, pulsando 1 se pasaría al siguiente menú, que permite ajustar la frecuencia de muestreo.

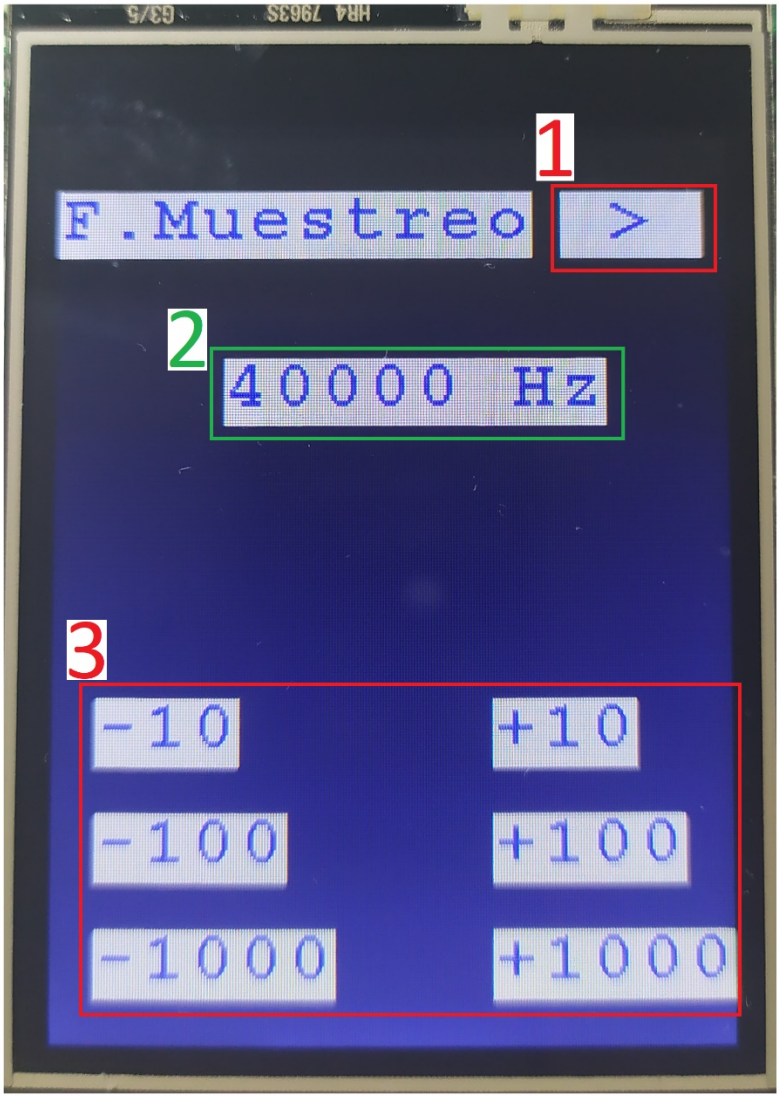

Ilustración 27: Tercer menú

Este menú funciona de la misma manera que el anterior.

El grupo de botones 3 sirve para modificar la frecuencia indicada en 2, que en este caso muestra la frecuencia de muestreo, no la de corte.

Una vez se ha elegido la frecuencia deseada, pulsando 1 comenzará el filtrado.

Generador de ondas

Para comprobar que el filtrado funciona correctamente necesitamos generar de alguna manera una onda que pueda ser leída por el ADC de nuestra placa STM32F429.

Podemos utilizar los conocimientos adquiridos para programar estas placas para hacer un generador de ondas utilizando otra placa de STMicroelectronics: la STM32F407.

La STM32F407 lleva un Cortex-M4 al igual que la STM32F429 que estamos utilizando pera hacer el filtrado, pero no tiene pantalla. La STM32F407 generará la onda que será filtrada por la STM32F429.

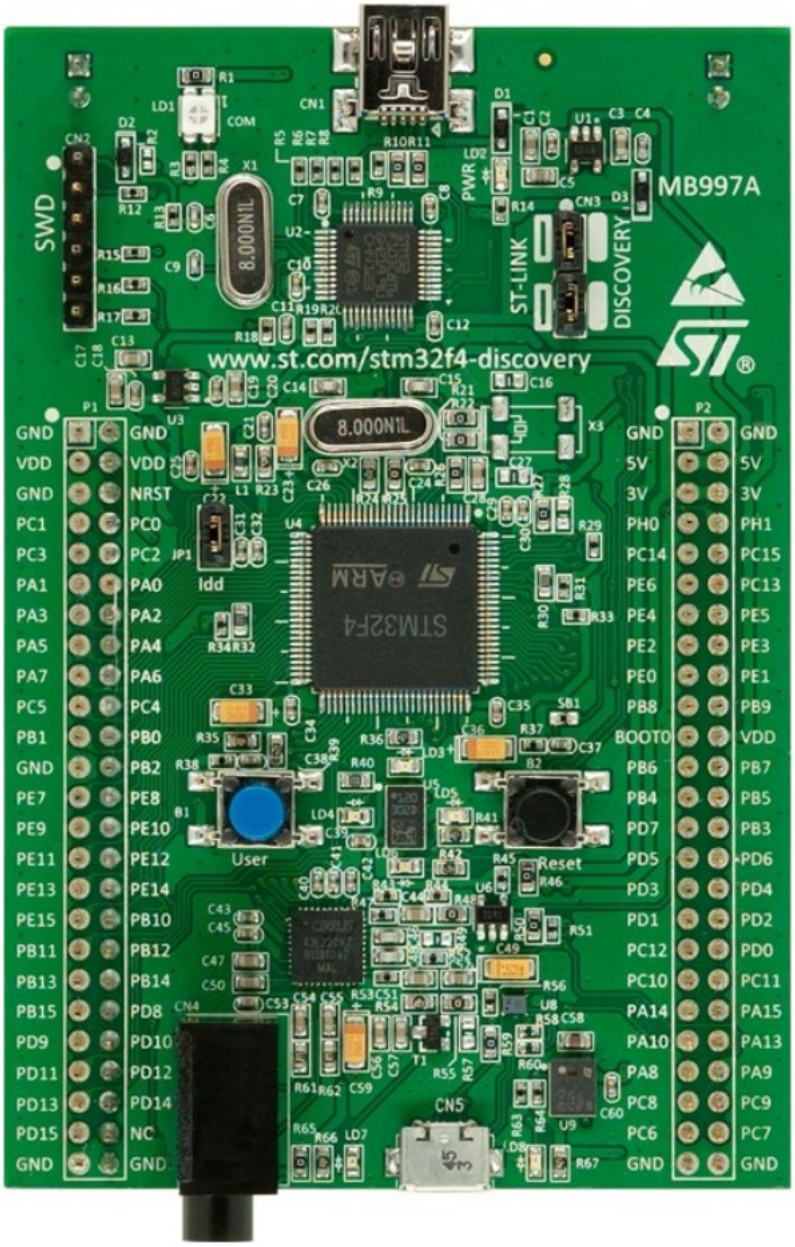

Ilustración 28: Placa STM32F407

Para utilizar esta placa como generador de señales utilizaremos su DAC. Emitirá valores de manera cíclica, generando una señal que será la suma de dos senoidales, una de ellas con una frecuencia diez veces mayor a la anterior.

Para crear dicha señal podemos utilizar Matlab. En la siguiente ilustración, la variable "y" es un vector que contiene los valores de la nueva onda. Estos valores se encuentran entre 0 y 4095, para que se ajusten a los valores mínimo y máximo que puede emitir el DAC de la placa STM32F407.

```
for index = 23:224
    x(index)=index;
    y(index)=sin(0.0625*index)+sin(0.625*index)+2;
    y(index)= round((4095/4)*y(index));
end

plot(x,y)
```

Ilustración 29: Código de MATLAB que para crear la señal de salida de la placa STM32F407. Se suma 2 para que no haya valores negativos.

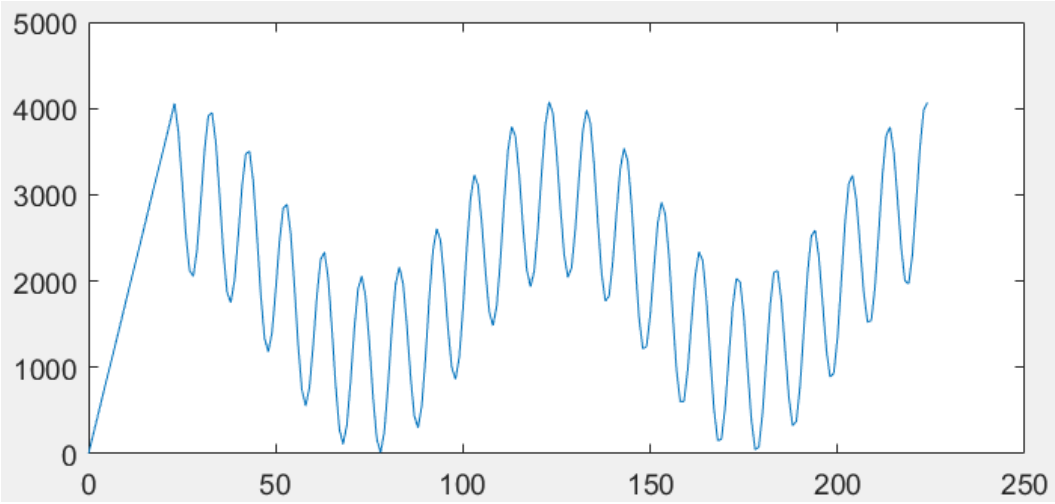

Ilustración 30: Visualización de la onda en Matlab utilizando "plot(x,y)"

Ahora que tenemos en “y” un vector con los valores entre 0 y 4095 adecuados para obtener la señal deseada, podemos coger estos valores y meterlos en el código de la placa.

Para poder empezar a escribir el programa en Keil tenemos que seguir el mismo proceso que hemos seguido con la placa STM32F429 (la principal, la que tiene pantalla y realiza el filtrado). Es decir, tenemos que utilizar CubeMX para obtener el código C de partida.

En el caso anterior, como queríamos realizar una aplicación que recibía una señal, la procesaba y emitía otra señal, necesitábamos un ADC, un DAC y un temporizador. Ahora sólo necesitamos un DAC y un temporizador (para que active el DAC en el intervalo de tiempo deseado); no se necesita ADC puesto que esta aplicación no recibe ninguna señal, sólo la emite. Por tanto, en la pestaña “Pinout” de CubeMx sólo hay que activar un DAC y un Timer (en este caso se ha elegido el Timer 6).

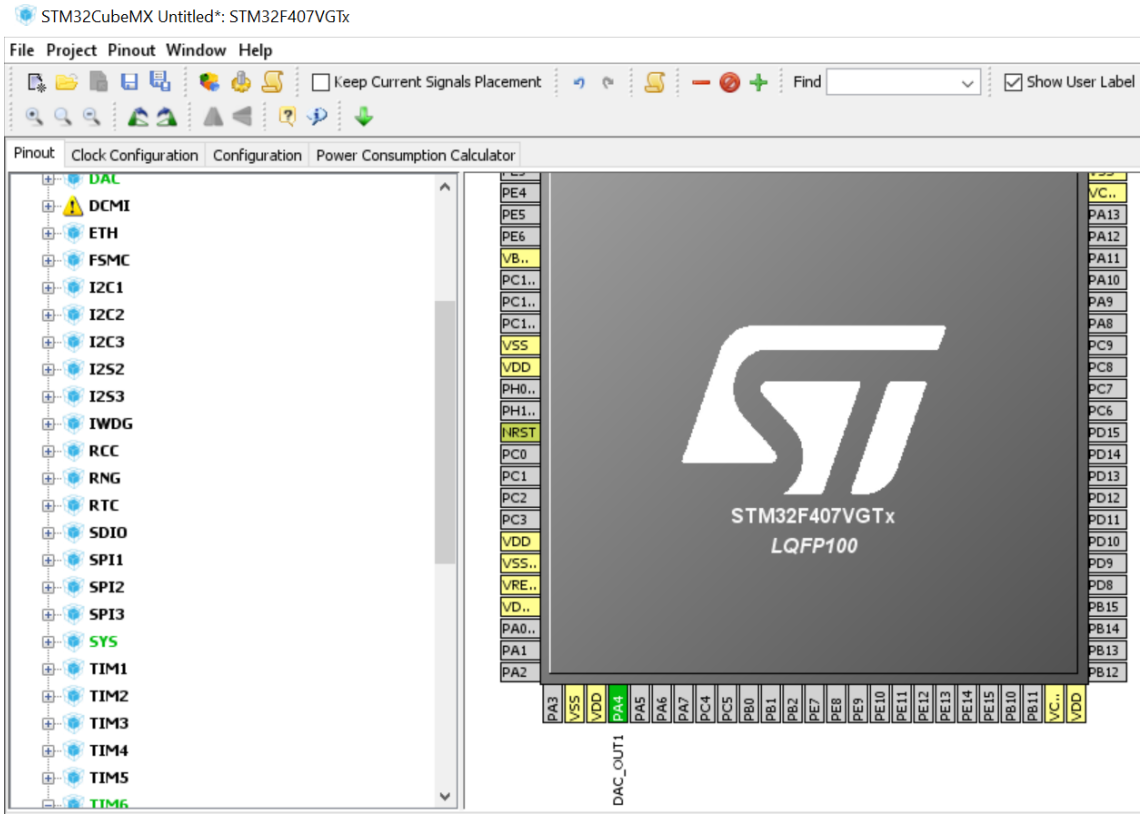

Ilustración 31: Configuración con CubeMX de la placa STM32F407

La configuración de la pestaña “Clock configuration” queda así, con la máxima frecuencia posible de 168MHz:

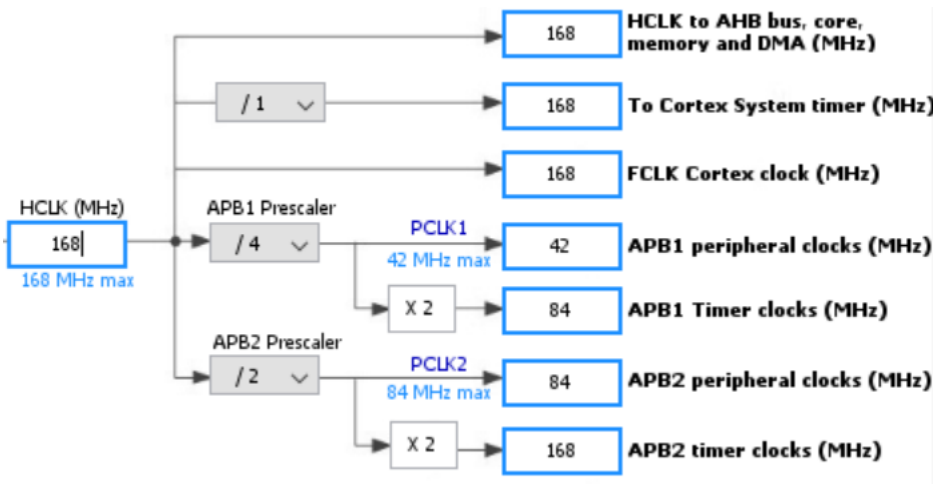

Ilustración 32: Configuración clocks

Hecho esto, podemos generar el código C como hicimos anteriormente con CubeMX.

Abrimos el programa de partida en C y guardamos el vector que obtuvimos en Matlab, el que contiene los valores resultado de sumar dos senoides con frecuencia una 10 veces mayor que la otra, con valores comprendido entre 0 y 4095. Copiamos estos valores (que en el script de Matlab eran la variable “y”) y los guardamos en un vector en C que se ha llamado “values”, que tiene una longitud de 200 posiciones:

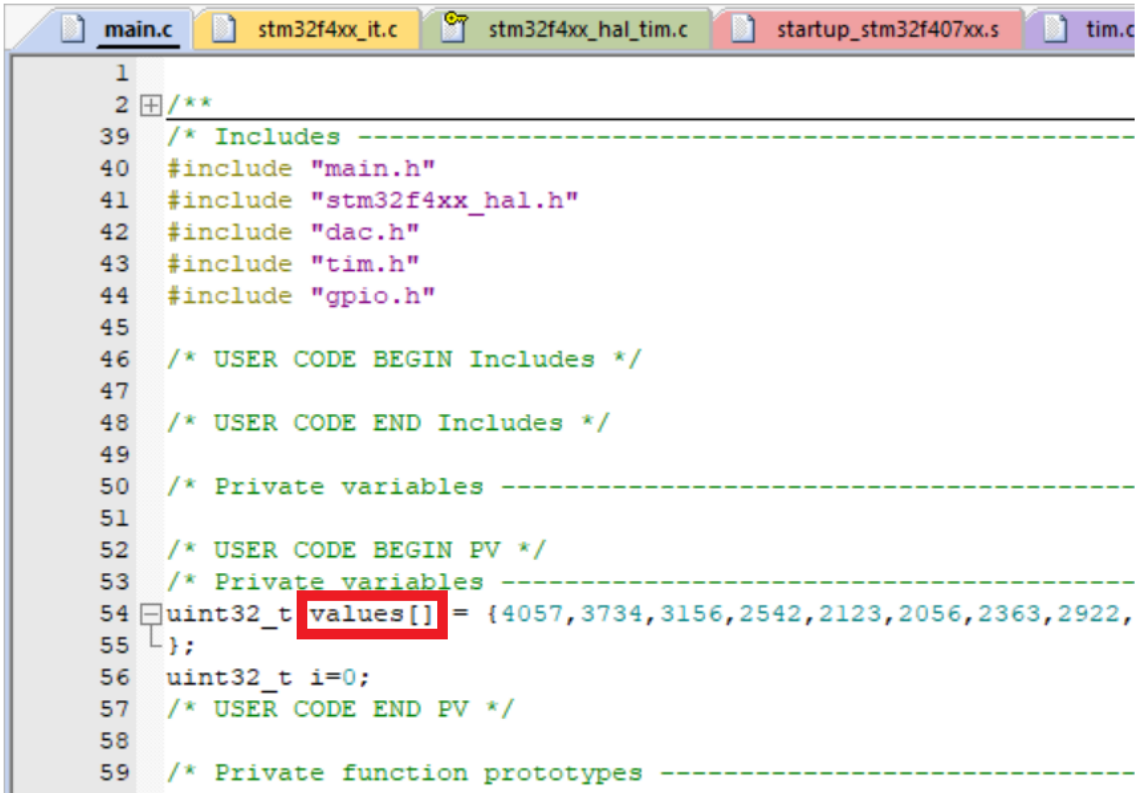

Ilustración 33: Insertar valores obtenidos con MATLAB en el código de la placa

Una vez tenemos los valores que queremos emitir cíclicamente guardados en duro en el vector "values", hay que escribir la función que hace que estos valores se emitan por el DAC. Esta función es el callback del Timer 6, la función que se ejecuta cada vez que el Timer 6 manda la señal.

```
/* USER CODE BEGIN 0 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  HAL_DAC_Start(&hdac,  DAC_CHANNEL_1);
  HAL_DAC_SetValue(&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, values[i]);

  i++;
  if(i>=200){
    i=0;
  }
}
/* USER CODE END 0 */
```

Ilustración 34: Función "HAL_TIM_PeriodElapsedCallback": código que se ejecuta cada vez que el Timer 6 realiza una interrupción

Lo que esta función hace es emitir el valor del vector "values" que se encuentra en la posición "i" cada vez que se ejecuta la función. Este valor será de entre 0 y 3,3 voltios (0 se corresponde con 0 voltios, y 4095 con 3,3 voltios).

Tras ejecutar esta función 200 veces se habrán emitido los 200 valores del vector “values”. Para que este vector se emita cíclicamente, se incluye un “if” (marcado en color rojo en la ilustración anterior) que hace que el valor del índice “i” vuelva a cero una vez llega a 200. Así, tras emitir “values[200]” siempre se emitirá “values[0]”.

Esto funciona porque la variable “i” no es local de la función; es global, por lo que su valor se mantiene entre llamadas a la función “HAL_TIM_PeriodElapsedCallback”.

Sabemos que el vector “values[]” contiene valores que son el resultado de sumar dos senoidales, una con una frecuencia diez veces superior a la otra; pero para saber qué frecuencia tienen al ser emitidas por el DAC hay que tener en cuenta la configuración del temporizador.

Al igual que para el código de la placa STM32F429, el valor de “htim6.Init.Period” dentro de la función “MX_TIM6_Init” (proporcionada por CubeMX) determina la frecuencia de salida:

```
void MX_TIM6_Init(void)
{
  TIM_MasterConfigTypeDef sMasterConfig;

  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 0;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = 1000;
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}
```

Ilustración 35: Configuración temporizador para placa STM32F407

Si le damos un valor de 1000, por ejemplo, y utilizando la fórmula de la Ilustración 10, obtenemos cada cuánto tiempo se ejecuta la función “HAL_TIM_PeriodElapsedCallback”, y por tanto, cada cuánto se emite un valor por el DAC:

$$f_{interrución} = \frac{f_{clk\_cnt}}{TIM\_ARR\ + 1} = \frac{84000000}{1000 + 1} = 83916$$

Es decir, con un valor de “htim6.Init.Period” 1000, se emitirían 83916 valores por segundo.

Si el vector “values” tiene 200 valores, y contiene dos periodos de la onda de menor frecuencia, entonces un periodo se corresponde con 100 valores:

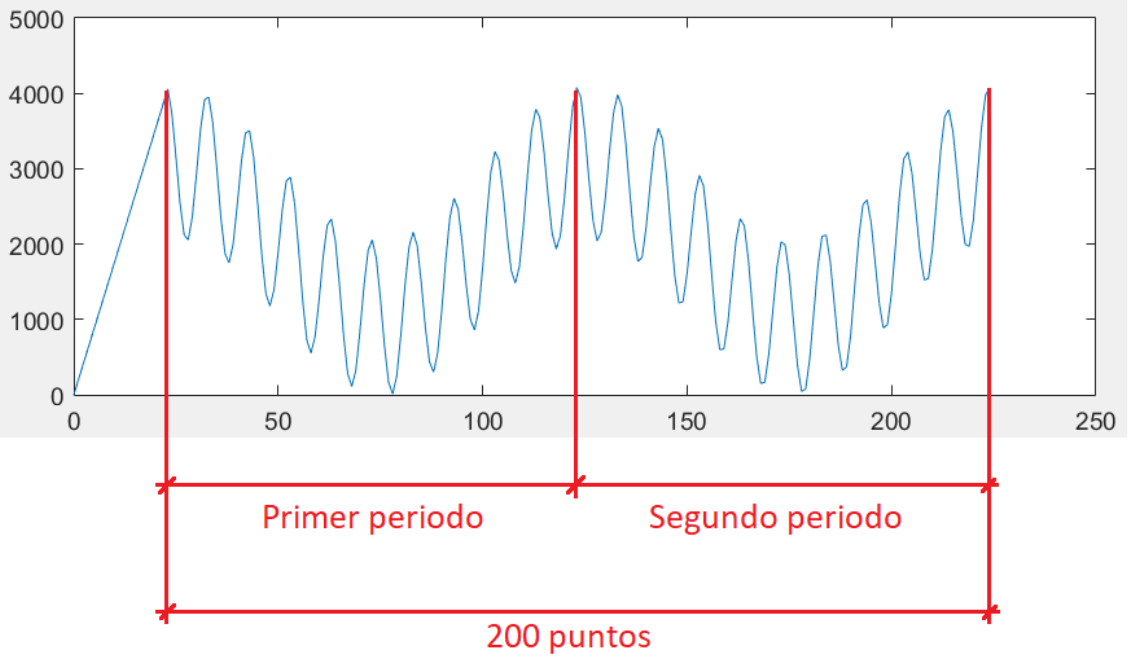

Ilustración 36: Vector "values"

Si un periodo se corresponde con 100 valores, y se emiten 83916 valores por segundo, entonces la frecuencia de la senoidal de menor será de 839 Hz. Como la otra senoidal tiene una frecuencia 10 veces mayor, tendrá una frecuencia de aproximadamente 8 KHz.

Una vez hecho todo esto, ya sólo faltaría escribir la línea que hace que comiencen las interrupciones del temporizador y la placa empiece a emitir valores por el DAC:

```c
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM6_Init();
  MX_DAC_Init();
  /* USER CODE BEGIN 2 */
  HAL_TIM_Base_Start_IT(&htim6);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */

  }
  /* USER CODE END 3 */

}
```

Ilustración 37: Llamada a la función HAL_TIM_Base_Start_IT(&htim6), para que comiencen las interrupciones

Tras cargar el programa en la placa y encenderla, si medimos el voltaje del DAC respecto a tierra con un osciloscopio (pin PA4 para el DAC) vemos la siguiente señal:

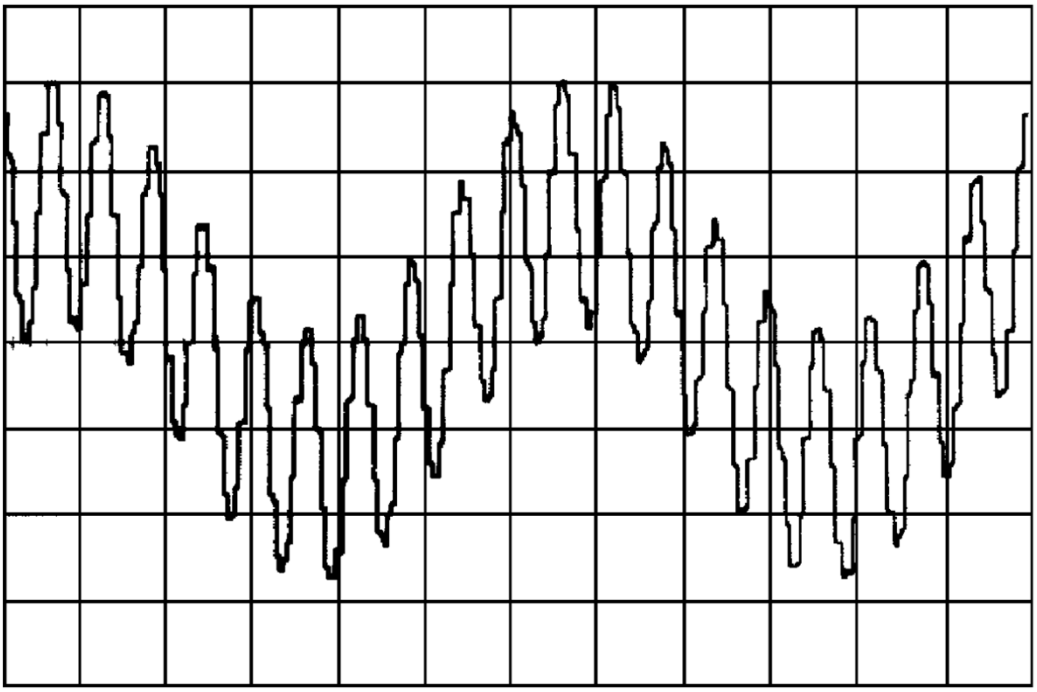

Ilustración 38: Onda de salida de STM32F407 (entrada de STM32F429) vista con un osciloscopio

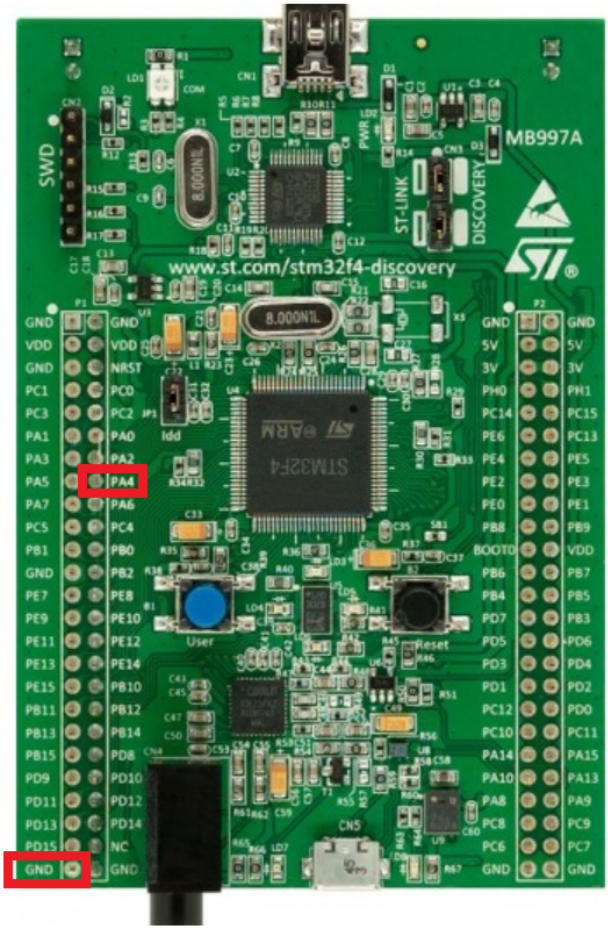

Ilustración 39: Pines PA4 y GND

Cada división abarca 0,2 milisegundos; de pico a pico de la onda de menor frecuencia hay unas 6 divisiones, por lo que:

$$Periodo = \ 0.0002 * 6 = 0.0012 \ s$$

$$Frecuencia = \frac{1}{0.0012 \ s} = 833.33 \ Hz$$

La frecuencia que habíamos calculado anteriormente teóricamente (en función del código) era 839 Hz, y al medir con el osciloscopio, obtenemos un valor muy próximo, 833 Hz (no es igual a 839 Hz porque no son 6 divisiones exactamente). Por tanto, parece razonable pensar que este método de realizar un generador de señales funciona bien.

El procedimiento que se acaba de describir se puede utilizar para generar cualquier onda (senoidal, diente de sierra, o cualquier otra) con la frecuencia deseada. Utilizando una placa como la STM32F407 es fácil diseñar un generador de ondas, utilizando un vector que contenga el patrón a emitir deseado.

Conexión de las dos placas

Una vez hemos comprobado con el osciloscopio que el generador de ondas hecho con la placa STM32F407 funciona, podemos utilizarlo para comprobar que la aplicación de filtrado realizada en la placa STM32F429 también funciona correctamente.

Para ello se conectarán las placas de la siguiente manera:

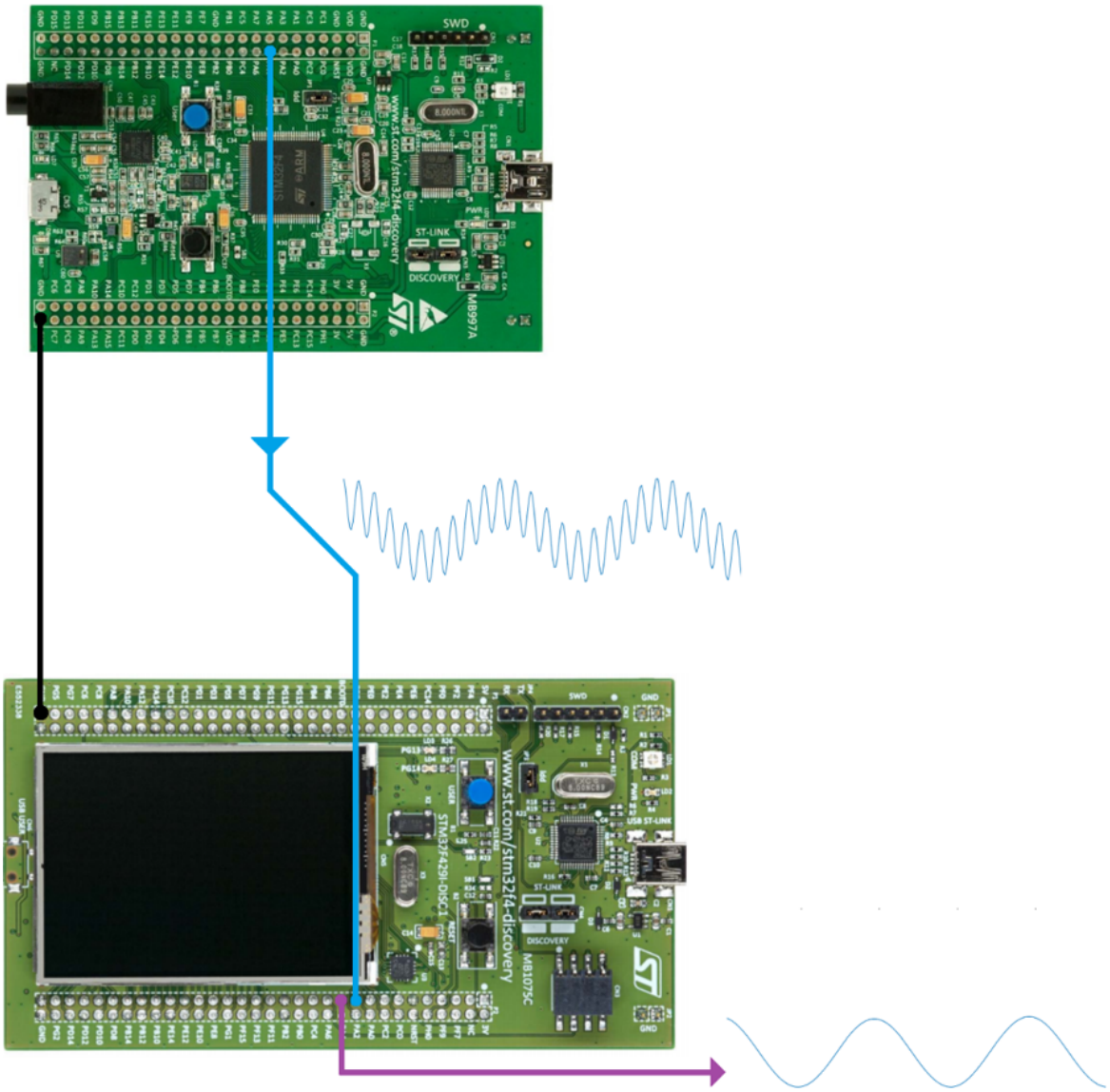

Ilustración 40:Esquema de la conexión de las dos placas

La señal a filtrar es la suma de dos senoidales de 800 Hz y 8 Khz; se intentará eliminar la componente de mayor frecuencia pero no la de menor frecuencia, para lo que se ha de seleccionar una frecuencia de corte entre 800 Hz y 8 KHz; en este caso, se selecciona 3 KHz mediante la pantalla táctil. La frecuencia de muestreo elegida es 40 KHz.

Las ondas de entrada y salida de la placa STM32F429:

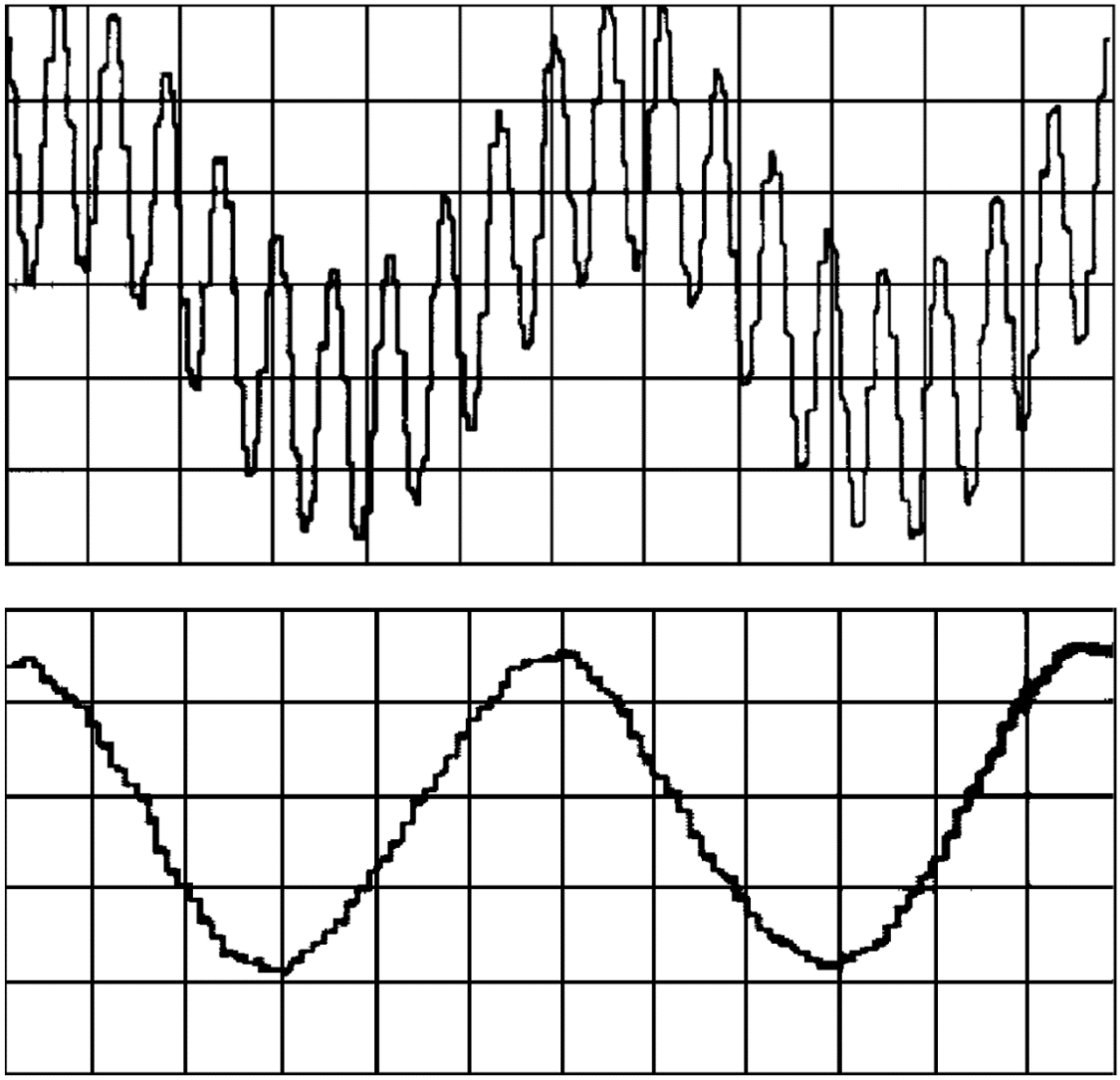

Ilustración 41: Ondas de entrada (superior) y salida (inferior) de STM32F429

La onda de salida tiene la forma deseada, porque ya sólo se aprecia la senoidal de menor frecuencia. La aplicación funciona correctamente.

Formas de representar números

En esta sección se explica las diferencias entre dos formas de representar cantidades: coma flotante y punto fijo.

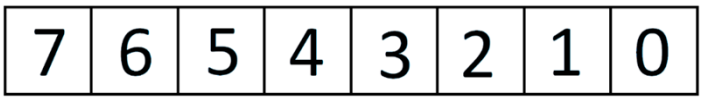

Ilustración 42: Conjunto de cifras que representan un valor numérico

Si dicho el número de la imagen anterior estuviera en punto fijo, un número determinado de bits se dedicaría para la parte entera, y los bits restantes para la parte fraccionaria. Si estuviera en coma flotante, una parte se dedicaría a la mantisa y otra parte al exponente.

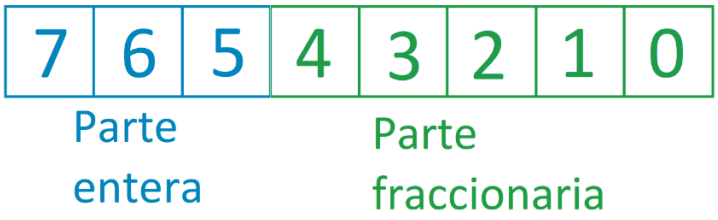

Ilustración 43: Número representado en punto fijo

Ilustración 44: Número representado en coma flotante

Características de punto fijo y coma flotante

Cada tipo de representación tiene sus ventajas y sus inconvenientes. Para entender por qué, hay que estudiar las características de coma flotante y punto fijo.

1) Para un mismo número de bits, coma flotante proporciona un rango mayor.
2) Los números en punto fijo son equidistantes; los números en coma flotante no lo son:

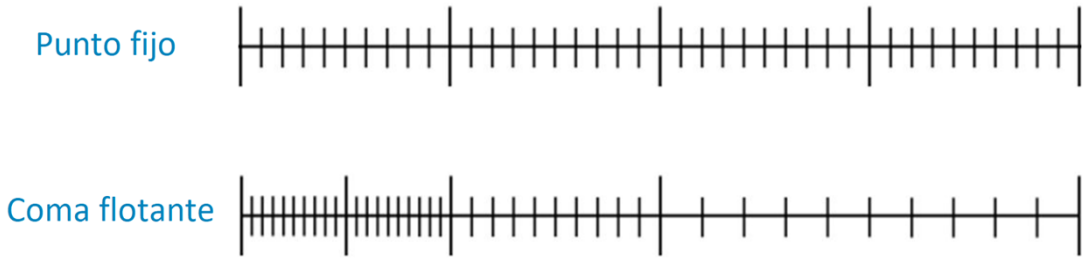

Ilustración 45: Distancia entre número en punto fijo, y en coma flotante

Para ver claramente este hecho podemos estudiar un ejemplo. Supongamos que tenemos 5 bits para representar números. El formato en punto fijo reservará 5 bits para la parte entera y 0 para la fraccionaria, y el formato en coma flotante reservará 2 bits para la mantisa y 3 para el exponente, tal que:

Ilustración 46: Ejemplo de representación de números en punto fijo y coma flotante.

Para coma flotante, definimos la norma para obtener el exponente (que está codificado en los tres primeros bits) de la siguiente forma:

$$\text{Exponente} = (\text{valor de los tres primeros bits}) - 3$$

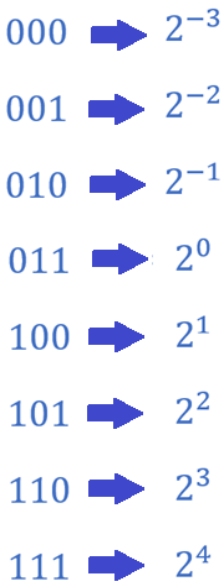

Ilustración 47: Codificación del exponente en coma flotante

Comparemos qué ocurre si los representamos en punto fijo, y si utilizamos coma flotante.

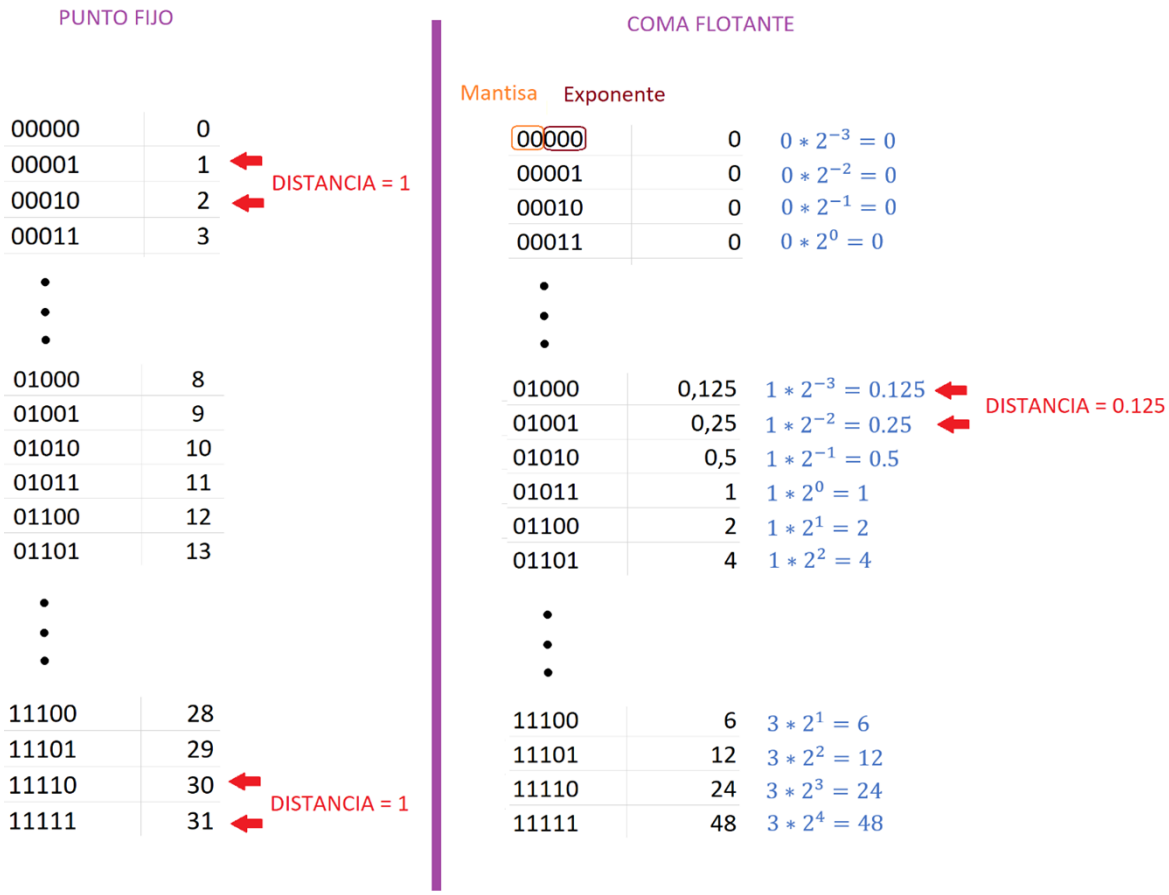

Ilustración 48: Diferencias entre el espaciado de los números en punto fijo y coma flotante

Vemos que la distancia entre cada número en punto fijo siempre es la misma: en este caso, 1, ya que no hemos dejado ningún bit para la parte fraccionaria.

En coma flotante, sin embargo, la distancia entre puntos es variable. La menor distancia que vemos en el ejemplo es de 0.125 (menor que 1). Cuanto más nos alejamos de cero, mayores se hacen las distancias entre los puntos de coma flotante.

Esto es necesario para que sea cierta la propiedad 1), que se mencionó anteriormente. Si para el mismo número de bits coma flotante ofrece mayor rango, y además la distancia entre algunos números es menor que la distancia entre los números en punto fijo, entonces dicha distancia tendrá que ser mayor para otros números, ya que el número máximo de valores para ambas representaciones (punto fijo y coma flotante) es el mismo.

Visualmente, los números en punto fijo y coma flotante están espaciados de la siguiente manera:

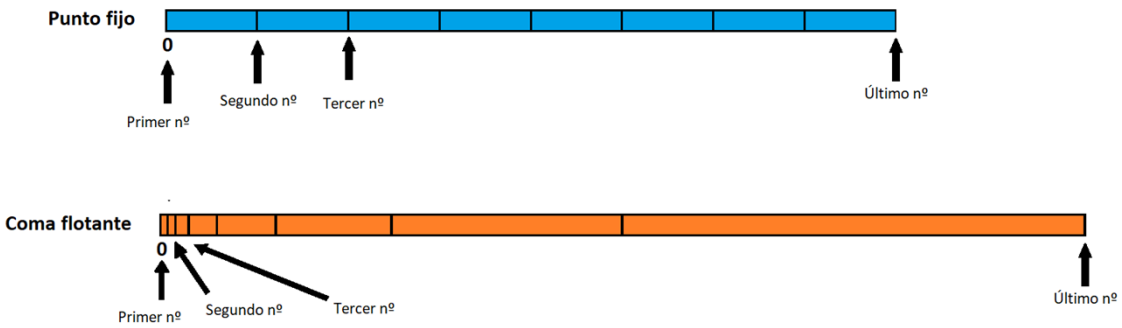

Ilustración 49: Diferencias entre el espaciado de los números en punto fijo y coma flotante

Representar números en coma flotante y punto fijo en C

Hacer programas en los que los valores se representen en coma flotante es sencillo, en C se puede hacer declarando las variables con tipo “float”.

Cuando declaramos una variable tipo float en Keil, el compilador almacena en memoria un número de 32 bits representado en coma flotante según el estándar IEEE 754, de la siguiente forma:

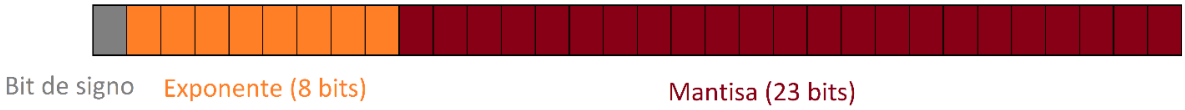

Ilustración 50: Reparto de bits entre bit de signo, exponente y mantisa en el formato IEEE 754

Para este formato, el rango es del orden de -10^{-38} a 10^{-38}. Convertir un número representado en coma flotante según el estándar IEEE 754 a decimal requiere hallar el exponente, el cual se halla restando 127 al exponente (bits 24 – 31).

Podemos comprobar que nuestro compilador guarda el valor de variables declaradas como tipo float mediante este convertidor. Por ejemplo, si escribimos un programa en Keil y damos el valor de -3.2:

```
float variable;

variable = -3.2;
```

Ilustración 51: Crear variable tipo float en Keil

Compilamos el código, lo ejecutamos y miramos el valor en el debugger, obteniendo el siguiente resultado:

Ilustración 52: Visualizar variable en el debugger

Este valor en hexadecimal se corresponde con la representación en binario:

1100 0000 0100 1100 1100 1100 1100 1101

Este valor en binario codifica el valor -3.200000047 en el formato IEEE 754, porque no es posible representar el valor -3.2 en binario. El número 0xC04CCCD no es exactamente igual a -3.2 en este formato, debido a la precisión limitada como resultado de tener un número finito de bits para representar el número.

Representar números en punto fijo en C puede hacerse declarando variables de tipo int16_t, int32_t, etc.

Para int16_t, si no hacemos ninguna manipulación, tendríamos 16 bits (en complemento a 2) para representar la parte entera, y 0 para la parte fraccionaria.

Si queremos hacer operaciones con fracciones utilizando números enteros, es necesario utilizar un factor de escala.

Supongamos que queremos sumar X = 1.5 más Y = 0.5 en el lenguaje C, pero sólo podemos declarar variables de tipo entero. Una forma de hacerlo sería la siguiente:

```
int16_t X = 15;
int16_t Y = 5;
int16_t Resultado = X + Y;
```

Ilustración 53: Código en Keil

El valor de la variable Resultado será 20. Hemos multiplicado tanto X como Y por 10; por tanto, el factor de escala es 10. Así para obtener el resultado correcto lo que habría que hacer es dividir el resultado por dicho factor: 20/10 = 2.

En el caso de una suma se divide el resultado por el factor de escala; en el caso de una multiplicación, habría que dividir el resultado por el factor de escala al cuadrado.

Esto implica que operar en punto fijo conlleva estar al tanto de qué resultados se obtienen en cada paso, de qué manera están escalados y qué hay que hacer para obtener el resultado deseado, deshaciendo el factor de escala.

En el ejemplo anterior se utilizó un factor de escala igual a 10, pero en realidad conviene elegir valores para el factor de escala que sean potencia de 2 (2, 4, 8…) ya que para el procesador es menos costoso multiplicar/dividir por una potencia de dos, que sólo tiene que desplazar el valor en binario el número de posiciones deseado a la derecha (multiplicación) o izquierda (división).

Esto introduce una dificultad más para el programador de la aplicación en punto fijo, ya que a la hora de depurar puede ser más difícil ver con claridad que los valores se parecen a lo esperado, al estar multiplicados o divididos por una potencia de 2 en lugar de por una potencia de 10, lo cual hace que los valores de las variables no se “parezcan” a su valor correspondiente no escalado. Por ejemplo: se ve fácilmente que 15 es 1.5 multiplicado por 10, pero cuesta más ver que 48 es 1.5 multiplicado por 32 (2^5).

Otro posible problema a tener en cuenta es que el rango de las variables en punto fijo es muy limitado en comparación con coma flotante, por lo que hay que asegurarse de que no tenemos problemas de overflow (el resultado de una operación es demasiado grande como para poder almacenarlo en la variable).

En punto fijo habrá que almacenar el resultado de una operación en una variable que tenga un número suficiente de bits para guardarlo por completo, ya que, si no, se perderán los bits más significativos y obtendremos un resultado completamente erróneo. Por ejemplo, si intentáramos realizar la siguiente operación, en la cual tanto los operandos como el resultado están almacenados en variables de tipo int16_t:

```
volatile int16_t X = 2999;
volatile int16_t Y = 670;
volatile int16_t Resultado;
```

Ilustración 54: Código en Keil

```
Resultado = X * Y;
```

Ilustración 55: Código en Keil

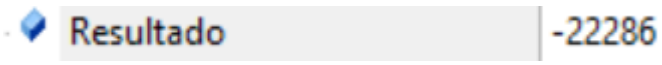

Ilustración 56: Debugger

El debugger nos muestra un valor incorrecto. El resultado correcto de la operación 2999*670 es 2009330, no -22286. Este problema está ocurriendo porque es imposible almacenar 2009330 en 16 bits.

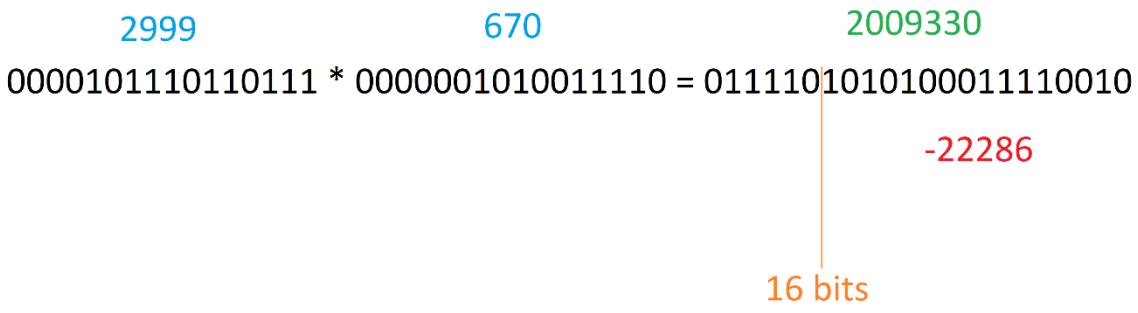

Ilustración 57: Límite de 16 bits causa problemas

Para almacenar el resultado correcto es necesario tener como mínimo 22 bits en la variable Resultado, pero al tener sólo 16, se pierden los 6 (22 – 16 = 6) bits más significativos, y por lo tanto el resultado es incorrecto. El número sale negativo porque el primer bit es un 1, y está representado en complemento a 2.

Para asegurar que el resultado de una multiplicación no produzca un overflow hay que hacer que la variable que guarda el resultado tenga un número de bits igual o superior a la suma de bits de las variables que se van a multiplicar. Esta precaución no es necesaria cuando operamos con números en coma flotante.

Otro posible problema a tener en cuenta al operar en punto fijo es underflow. Underflow ocurre cuando un número cercano a 0 se redondea a 0. En el caso de nuestra aplicación esto es algo que también hay que considerar, ya que los coeficientes h(n) del filtro FIR son valores entre 0 y 1, por lo que hay que considerar cómo tratar este problema para que los coeficientes del filtro no sean iguales a 0.

Efectos de tener o no tener FPU al hacer operaciones en coma flotante

La unidad de coma flotante (FPU) permite realizar operaciones matemáticas en hardware directamente.

Para ver la diferencia en el código compilado cuando se tiene FPU y cuando no se tiene, veamos un ejemplo de una operación aritmética en coma flotante:

```
float resultado;
float x = 12315483243.45634635;
float y = 7235.893453;

resultado = x/y;
```

Por ejemplo, calcular el valor de la variable “resultado” en el procesador utilizando la FPU, la línea de código “resultado = x/y” se ejecutaría en sólo 5 instrucciones, gracias a las instrucciones que la FPU permite utilizar:

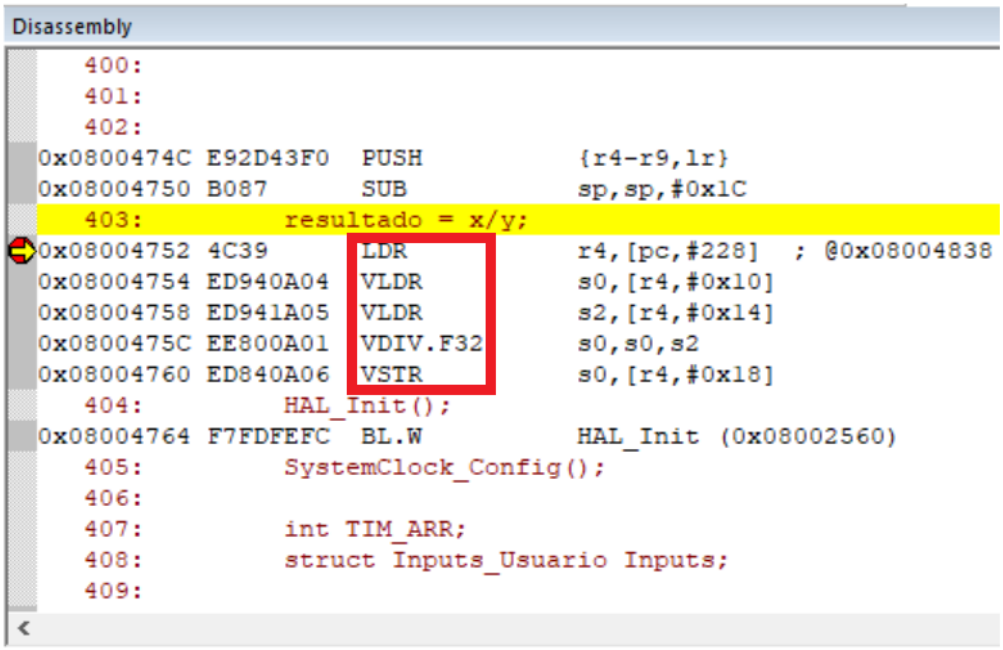

Ilustración 58: Instrucciones para ejecutar "resultado = x/y" con FPU

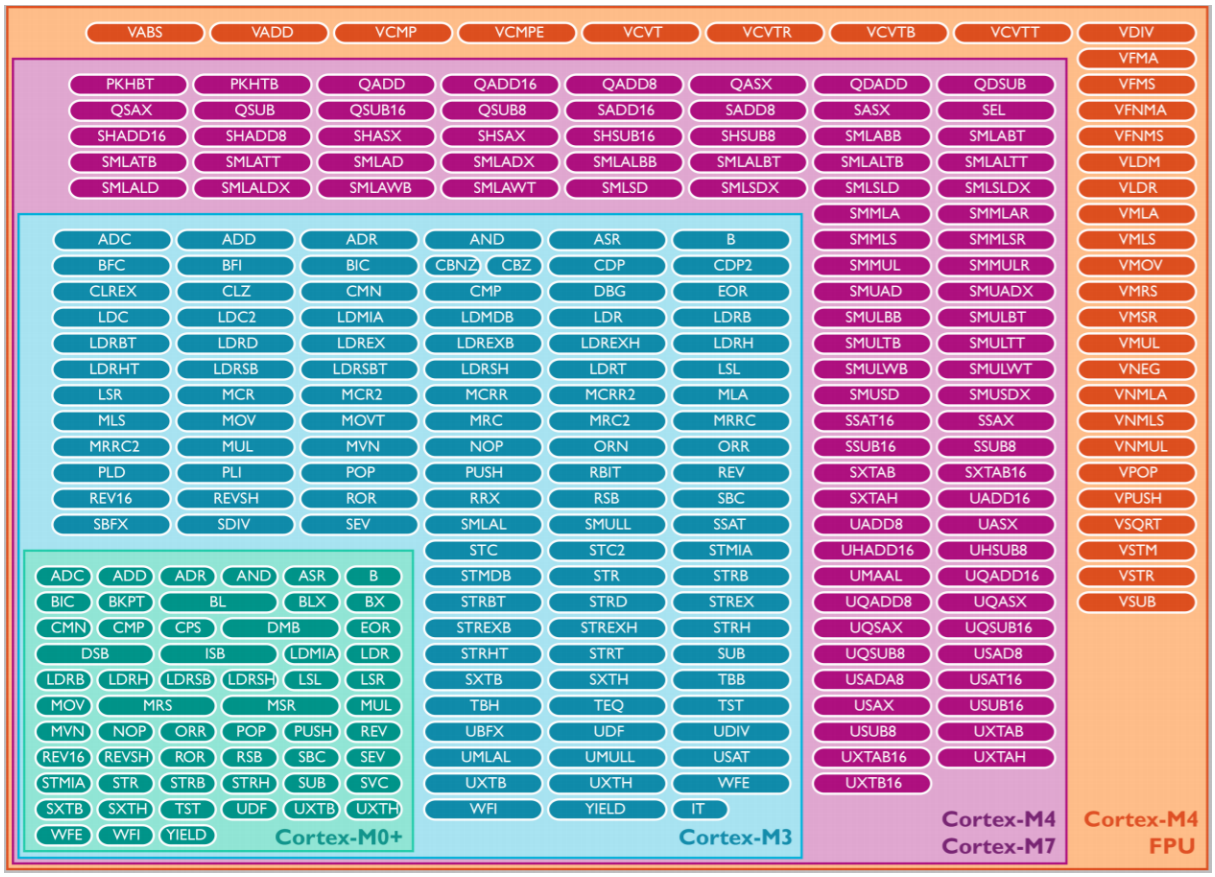

Ilustración 59: Las instrucciones de color naranja sólo pueden ser utilizadas por Cortex-M4 que tengan FPU

Hacer la misma operación (“resultado = x/y”) sin utilizar la FPU requiere de la ejecución de muchas más instrucciones, ya que no hay un hardware específico para realizar este tipo de operaciones aritméticas en coma flotante, por lo que han de ser realizadas en software.

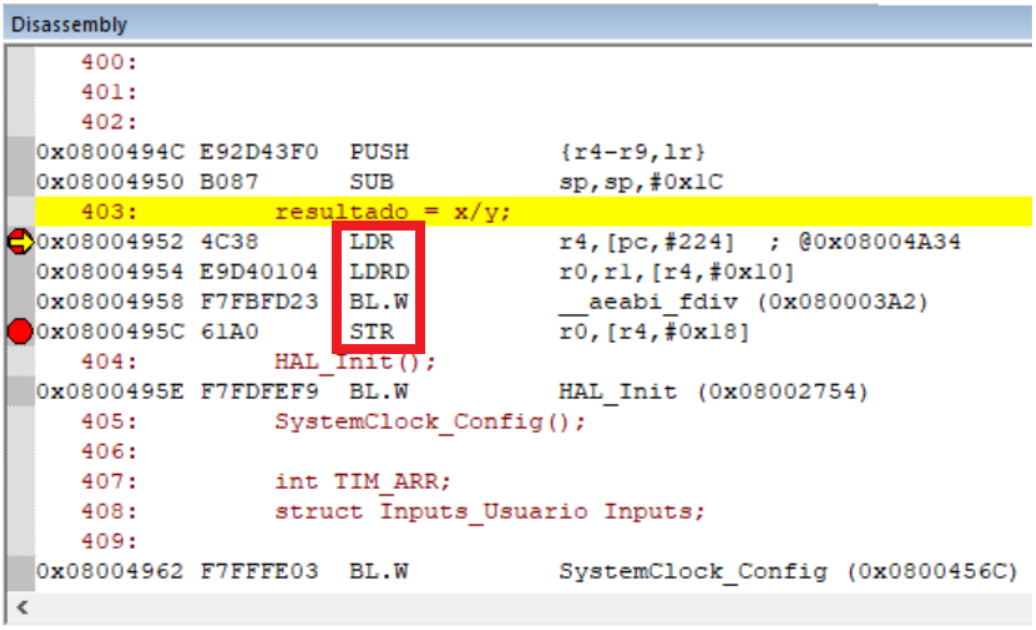

Ilustración 60: Instrucciones para ejecutar "resultado = x/y" sin FPU

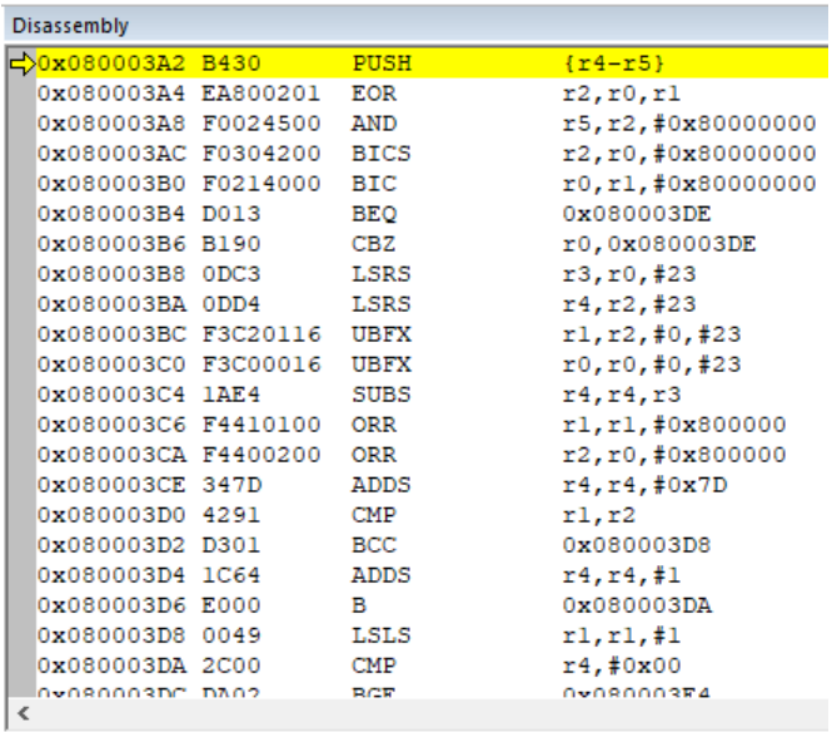

Ilustración 61: Parte de las instrucciones ejecutadas tras BL.W

Como consecuencia del gran número de instrucciones necesarias, el tiempo requerido para hacer esta división es mucho mayor; tarda 4 veces más sin FPU que con FPU.

Modificaciones del algoritmo para adaptarlo a punto fijo

Primero hemos de pasar los coeficientes que calculamos en coma flotante a punto fijo.

El vector “h_float” almacena los valores del filtro en coma flotante, calculados como se explicó anteriormente. Cada uno de estos valores está entre 0 y 1, por lo que antes de poder almacenarlos en una variable de tipo entero habrá que multiplicarlos por alguna cantidad determinada, para que sean mayores que 1 y no tener el problema del underflow al redondear al pasar la variable de “float” a “int”.

```
//calculo de los coeficientes del filtro en punto fijo
for (n=0; n<=OrdenFiltro; n++){
  h_float_temp[n]=(h_float[n])*131072; // Ddesplazar 17 posiciones en binario 2^17 = 131072
}

for (n=0;n<=OrdenFiltro;n++){  // h_PFint contiene los valores del filtro para punto fijo
  h_int[n] = (int16_t)h_float_temp[n];
}
```

Ilustración 62: Código Keil

En la ilustración anterior se muestra cómo se crea la variable “h_int”, un vector de enteros que contiene los parámetros del filtro que se utiliza para realizar el filtrado en punto fijo.

El vector “h_int” se obtiene a partir del vector “h_float”; “h_int” contiene los valores de “h_float” multiplicados por un factor de escala adecuado. En este caso se ha elegido 131072 (que es igual a 2^17).

Así, por ejemplo, si un coeficiente “i” determinado del filtro fuera h(i) = 0.0015492014, al multiplicar por 131072 se obtiene 203.0569259, y en punto fijo quedaría así: h(i) = 203.

Se está añadiendo error, ya que 203 no es igual a 203.0569259. Si deshacemos el factor de escala: 203/131072 = 0.00154876709, que no es igual a 0.0015492014.

Para minimizar el error hay que hacer que la variable entera contenga más cifras decimales del valor en coma flotante; para ello habría que usar un factor de escala mayor. El problema de usar un factor de escala muy grande es que puede

que el resultado de multiplicar el número en coma flotante por el factor de escala no sea demasiado grande como para poder almacenarlo en la variable sin que haya overflow.

Una vez obtenidos los coeficientes en punto fijo se aplica el filtrado. Al filtrar hay que tener en cuenta la posibilidad de overflow; para ello es necesario asegurarse de que la variable que guardará el resultado de realizar la operación tenga los suficientes bits como para almacenar el resultado.

Dicha variable en nuestro código se llama SalidaDAC_intTEMP. La parte "int" hace referencia a que es en punto fijo, y "TEMP" hace referencia a que es temporal, en el sentido de que no es el resultado final que se emitirá a través del conversor digital a analógico debido a que hay que deshacer el factor de escala.

En este caso, para que el valor de SalidaDAC_intTEMP se ajuste al valor de salida del DAC (12 bits, es decir, valores entre 0 y 4095) se desplaza 17 lugares a la derecha y se almacena su valor en SalidaDAC_int, la cual almacena el valor entre 0 y 4095 que se emitirá por el DAC.

```
SalidaDAC_intTEMP = 0;
for (int i=0; i<=OrdenFiltro; i++){
  SalidaDAC_intTEMP += ((int16_t)(ADC_DATA[i])*(int16_t)h_int[OrdenFiltro+1-i]);
}

SalidaDAC_int = (int32_t)(SalidaDAC_intTEMP>>17);

for (int i=0;i<=OrdenFiltro-1;i++){
  ADC_DATA[i] = ADC_DATA[i+1];
}

HAL_ADC_Start(&hadc1);
HAL_ADC_PollForConversion(&hadc1,100);
ADC_DATA[OrdenFiltro] = HAL_ADC_GetValue(&hadc1);

HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, SalidaDAC_int);
```

Ilustración 63: Algoritmo de filtrado en punto fijo. La línea resaltada deshace el factor de escala.

La ilustración anterior muestra la parte del código que realiza el filtrado en punto fijo. Es el mismo algoritmo que para el filtrado en coma flotante mostrado en la ilustración 21, con las siguientes diferencias:

1) Los parámetros h del filtro que se utilizan para hacer el filtrado son los contenidos en “h_int”, en lugar de los de “h_float”

2) Se revierte el factor de escala; una vez se ha hecho el filtrado, hay que tener en cuenta que cada uno de parámetros del filtro había sido multiplicado previamente por 131072 (2^17), por lo que tras haber usado los parámetros para obtener el resultado del filtrado, hay que dividir este resultado por 2^17 para que sea correcto.

Si el procesador fuera de menos de 32 bits, mantener la misma capacidad de filtrado (mismo orden de filtro y misma frecuencia de muestreo máxima), habría que reducir la precisión de las operaciones. Si el procesador fuese de 16 bits, por ejemplo, tanto “ADC_DATA” como “h_int” podrían tener que ser de 8 bits, para evitar el desbordamiento de “SalidaDAC_intTEMP”.

Análisis temporal del algoritmo de filtrado

Podemos comprobar cuánto tarda en realizarse el filtrado en cada iteración del filtro observando los temporizadores que nos ofrece el debugger de Keil MDK y utilizando breakpoints en nuestro código.

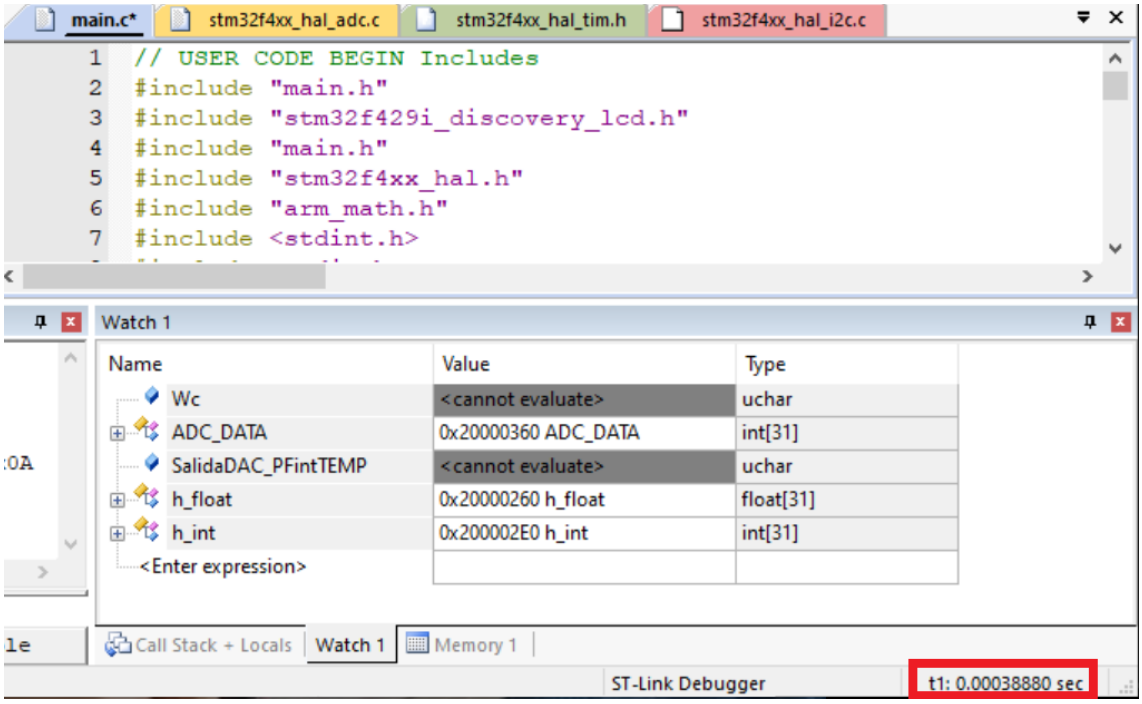

Ilustración 64: Estudio de tiempo de ejecución utilizando el debugger de Keil

Podemos comparar cuánto se tarda en realizar el filtrado con 3 configuraciones diferentes:

1) En coma flotante (variables tipo float) y utilizando la FPU
2) En coma flotante (variables tipo float) sin utilizar la FPU
3) En punto fijo (variables tipo entero)

Keil nos permite seleccionar si al compilar se obtendrá un código compilado que incluye instrucciones que hacen uso de la FPU o no. Si seleccionamos la opción “Not Used”, la FPU no se utilizará.

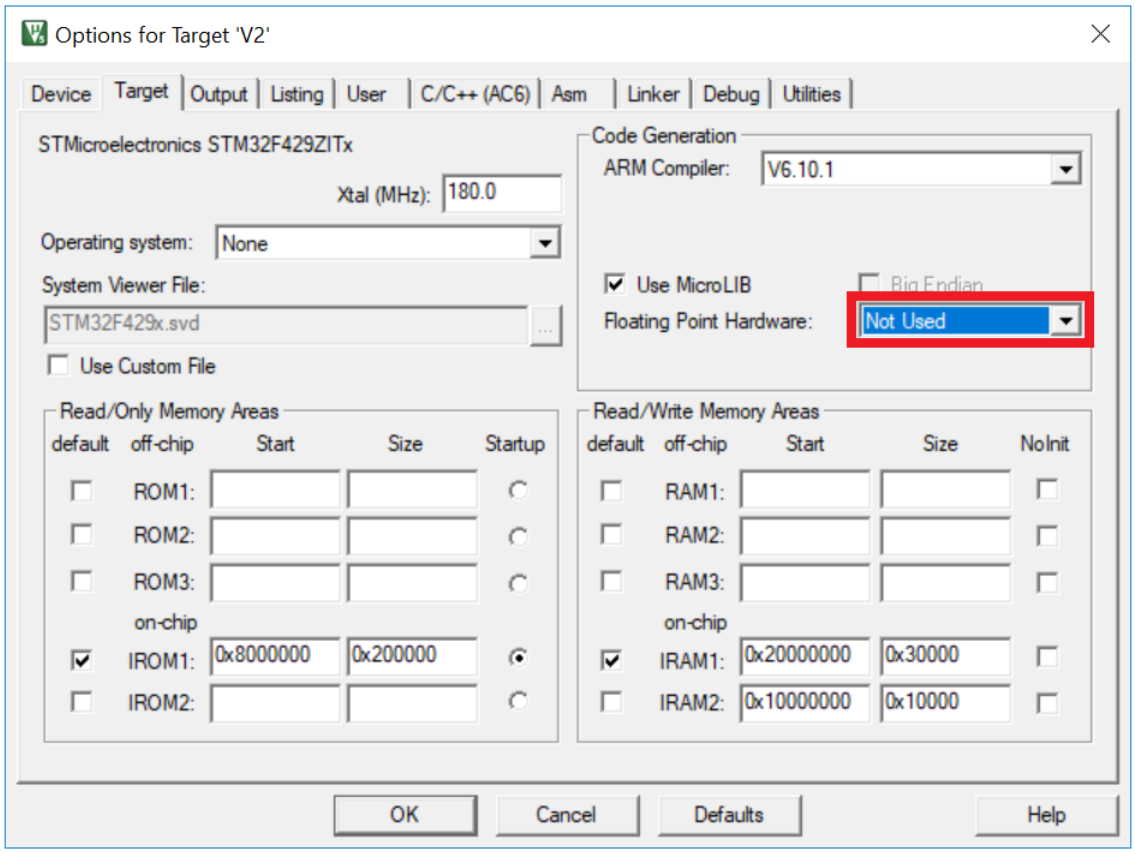

Ilustración 65: Configuración de Keil para no utilizar la unidad de coma flotante

Si seleccionamos un filtro de 30 taps y ejecutamos el código de la Ilustración 66 entre los dos breakpoints, el tiempo de ejecución es de 14.79 microsegundos.

```
350     SalidaDAC_float = 0;
351     for (int i=0; i<=OrdenFiltro; i++){
352       SalidaDAC_float += ADC_DATA[i]*h_float[OrdenFiltro+1-i];
353     }
354
355     for (int i=0;i<=OrdenFiltro-1;i++){
356       ADC_DATA[i] = ADC_DATA[i+1];
357     }
358
359     HAL_ADC_Start(&hadc1);
360     HAL_ADC_PollForConversion(&hadc1,100);
361     ADC_DATA[OrdenFiltro] = HAL_ADC_GetValue(&hadc1);
362
363     HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
364     HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, SalidaDAC_float);
```

Ilustración 66: Algoritmo de filtrado en coma flotante

Si compilamos el código con la opción “Single Precision”, al ejecutar el código se utilizará la FPU. Si utilizáramos un procesador Cortex-M7 en lugar del M4, además de las opciones “Not Used” y “Single Precision” tendríamos la opción “Double Precision”, que permite a la FPU tratar números en coma flotante de 64 bits.

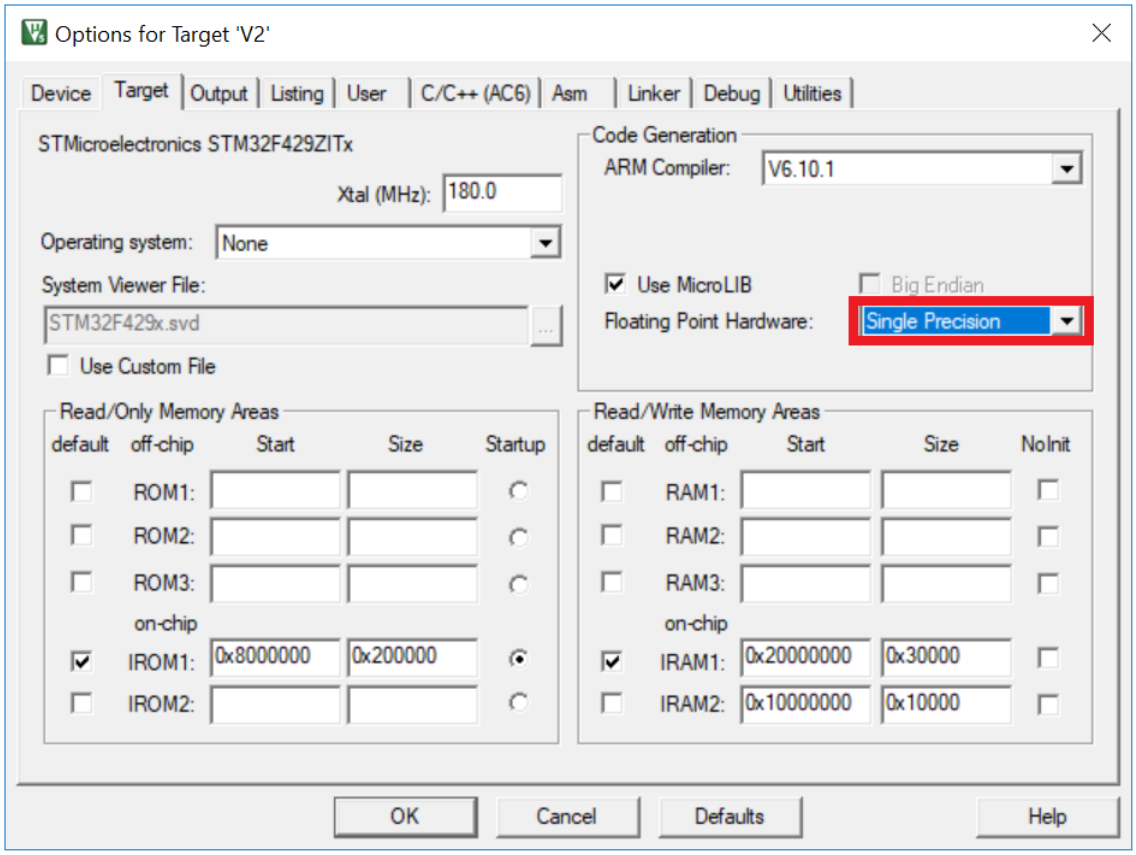

Ilustración 67: Configuración de Keil para uitlizar la unidad de coma flotante

Al ejecutar el código anterior en las mismas condiciones (orden 30, breakpoints en las mismas líneas, frecuencia del reloj igual), el tiempo de ejecución es de 5.46 microsegundos.

En comparación con la prueba anterior, vemos que si no activamos la FPU el código tarda más de cinco veces más en ejecutarse (28.9/5.46 = 5.29).

Si en lugar de un filtro de orden 30 hubiéramos puesto uno de orden 100, compilando sin la FPU activada tardaría 93 microsegundos, y con la FPU, 18 microsegundos.

Esto implica que sería imposible realizar un filtrado con una frecuencia de 40 KHz si no se utiliza la FPU, ya que 1/40 KHz = 25 microsegundos. El tiempo máximo que tendríamos es 25 microsegundos, y como 93 > 25, el filtrado no se puede realizar. Si no tuviéramos FPU, la única manera de hacer un filtrado de orden 100 y frecuencia de muestreo de 40 KHz sería utilizando punto fijo, no coma flotante.

Una vez medidos los tiempos de ejecución del algoritmo en coma flotante sin FPU y con FPU, podemos medir el tiempo de ejecución del algoritmo en punto fijo (tiempo de ejecución de las líneas entre los breakpoints de la Ilustración 68). El resultado es 5,15 microsegundos.

```
369     SalidaDAC_intTEMP = 0;
370     for (int i=0; i<=OrdenFiltro; i++){
371       SalidaDAC_intTEMP += ((int32_t)(ADC_DATA[i])*(int32_t)h_int[OrdenFiltro+1-i]);
372     }
373
374     SalidaDAC_int = (int32_t)(SalidaDAC_intTEMP>>17);
375
376     for (int i=0;i<=OrdenFiltro-1;i++){
377       ADC_DATA[i] = ADC_DATA[i+1];
378     }
379
380     HAL_ADC_Start(&hadc1);
381     HAL_ADC_PollForConversion(&hadc1,100);
382     ADC_DATA[OrdenFiltro] = HAL_ADC_GetValue(&hadc1);
383
384     HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
385     HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, SalidaDAC_int);
```

Ilustración 68: Algoritmo de filtrado en punto fijo

La siguiente tabla resume los resultados obtenidos:

Tipo de filtrado	Tamaño filtro	Tiempo ejecución (µs)
Coma flotante sin FPU	30	28,9
Coma flotante con FPU	30	5,46
Punto Fijo	30	5,15

Ilustración 69: Resultados

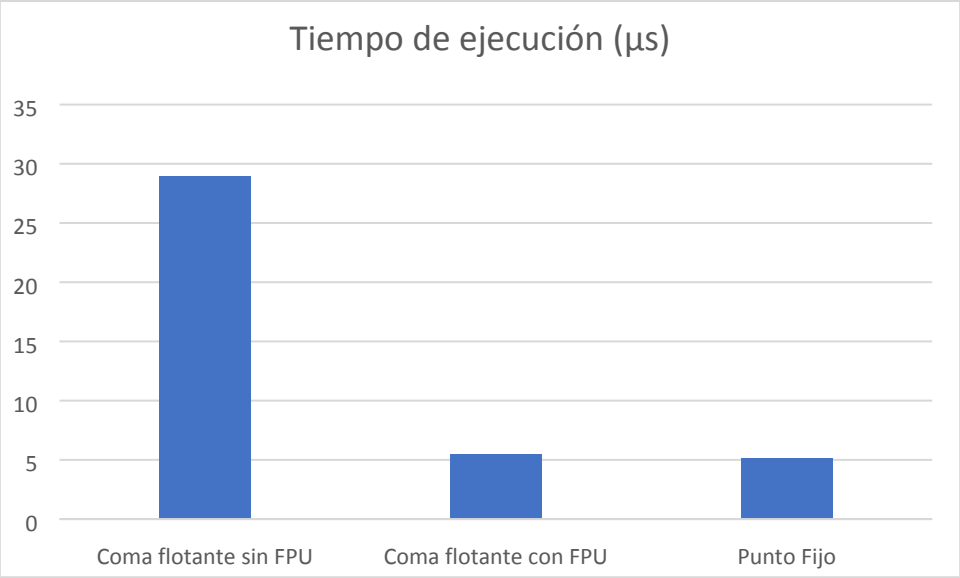

Ilustración 70: Visualización de los resultados

Además de estas tres situaciones (coma flotante sin FPU, coma flotante con FPU, y punto fijo) también se ha medido el tiempo que se tarda en realizar el filtrado utilizando las funciones específicas proporcionadas por Keil, que permiten no tener que programar nosotros a mano el bucle "for" que hace el filtrado.

La librería de DSP de Keil nos permite implementar varios filtros a través de diferentes funciones. Por ejemplo, la función "arm_fir_f32" hace los cálculos correspondientes a las líneas 351-353 de nuestro código (ver Ilustración 66), en coma flotante de 32 bits.

Si se quiere hacer un filtrado con formato en punto fijo q15, existe "arm_fir_q15". Hay disponibles más funciones para diferentes formatos numéricos. Sin embargo, estas funciones de filtrado en punto fijo no se ocupan de evitar problemas de overflow; seguimos teniendo que ocuparnos nosotros de que eso no ocurra.

La documentación de estas funciones se puede encontrar en el apartado "Bibliografía y otros recursos" de este informe.

El resultado de probar con estas funciones es que tardan un tiempo similar en realizar el filtrado, y algunas, como "arm_fir_f32", tardan más. No he podido comprobar por qué tardan más que el bucle "for" de nuestra aplicación (ilustración

71), ya que Keil no permite ver cómo están definidas estas funciones de las librerías de DSP.

```
SalidaDAC_float = 0;
for (int i=0; i<=OrdenFiltro; i++){
  SalidaDAC_float += ADC_DATA[i]*h_float[OrdenFiltro+1-i];
}
```

Ilustración 71: Filtrado sin utilizar funciones proporcionadas por Arm/Keil

Costes Cortex-M4 con y sin FPU

La diferencia de precio en la actualidad entre un Cortex-M4 sin FPU y uno con FPU es grande: los que tienen FPU cuestan entre 2 y 3 veces más que los que no la tienen.

En las siguientes imágenes se muestran una serie de chips con sus precios. Estos precios se han obtenido de Digikey; la diferencia de precios es similar para otros vendedores.

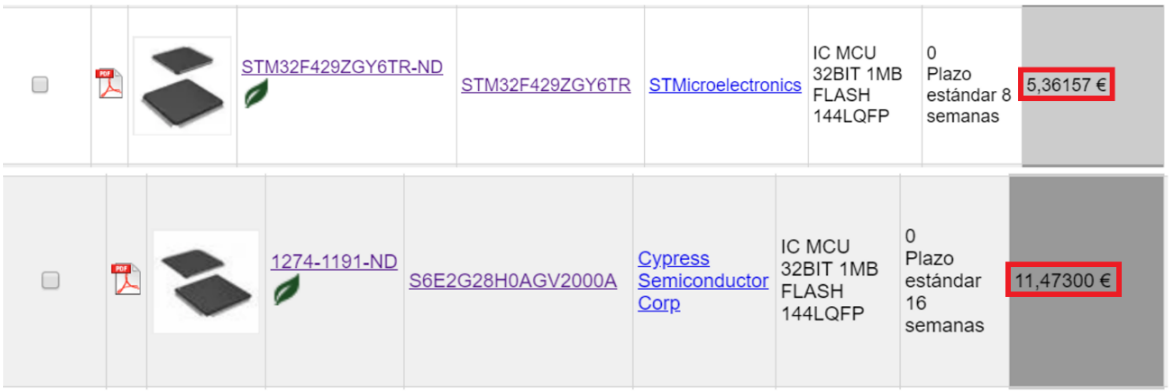

Ilustración 72: Coste chip con Cortex-M4 sin FPU (arriba) y con FPU (abajo), con características muy similares excepto por la presencia de FPU

Comparar piezas		Imagen	Número de pieza de Digi-Key	Número de pieza del fabricante	Fabricante	Descripción	Cantidad disponible	Precio unitario EUR
☐								
☐		Digi-Key Foto no disponible	497-19356-2-ND	STM32F446MCY6TR	STMicroelectronics	IC MCU 32BIT 256KB FLASH 81WLCSP	0 Plazo estándar 8 semanas	2,93863 €
☐			497-15812-2-ND	STM32F446MEY6TR	STMicroelectronics	IC MCU 32BIT 512KB FLASH 81WLCSP	5.000 - Inmediata	3,75634 €
☐			497-19357-2-ND NEW	STM32F446RCT6TR	STMicroelectronics	IC MCU 32BIT 256KB FLASH 64LQFP	0 Plazo estándar 8 semanas	3,89074 €

Ilustración 73: Chips con Cortex-M4 sin FPU; los más baratos cuestan unos 3 ó 4 euros

Comparar piezas		Imagen	Número de pieza de Digi-Key	Número de pieza del fabricante	Fabricante	Descripción	Cantidad disponible	Precio unitario EUR
☐								
☐			428-3717-ND	S6E2GH8H0AGV2000A	Cypress Semiconductor Corp	IC MCU 32BIT 1MB FLASH 144LQFP	0 Plazo estándar 16 semanas	7,06925 €
☐			1274-1195-ND	S6E2GM6HHAGV2000A	Cypress Semiconductor Corp	IC MCU 32BIT 512KB FLASH 144LQFP	0 Plazo estándar 20 semanas	9,93333 €
☐			428-3729-ND	S6E2GM8J0AGV2000A	Cypress Semiconductor Corp	IC MCU 32BIT 1MB FLASH 176LQFP	0 Plazo estándar 16 semanas	10,94325 €

Ilustración 74: Chips con Cortex-M4 sin FPU; los más baratos se acercan a los 10 euros

Caso real de dispositivo que utiliza procesador sin FPU para aplicaciones de DSP

Para realizar el TFG compré un pequeño osciloscopio digital que me permitió comprobar que la aplicación de filtrado funcionaba correctamente. Al abrirlo pude ver que utiliza un ATMEGA64A-AU, sin FPU. Este aparato se compró en 2019; es un ejemplo de que pese a que se van abaratando los procesadores con FPU, sigue siendo relevante considerar el desarrollo en punto fijo para poder reducir costes, sobretodo para dispositivos de precio bajo como éste (Ilustración 75), para los que el procesador representa un porcentaje significativo del coste total de los materiales del producto.

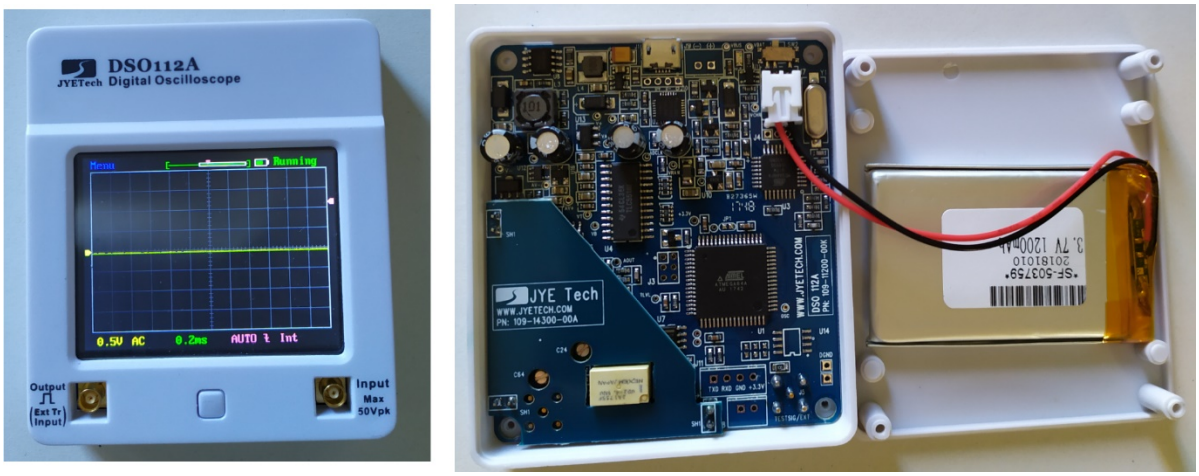

Ilustración 75: Osciloscopio

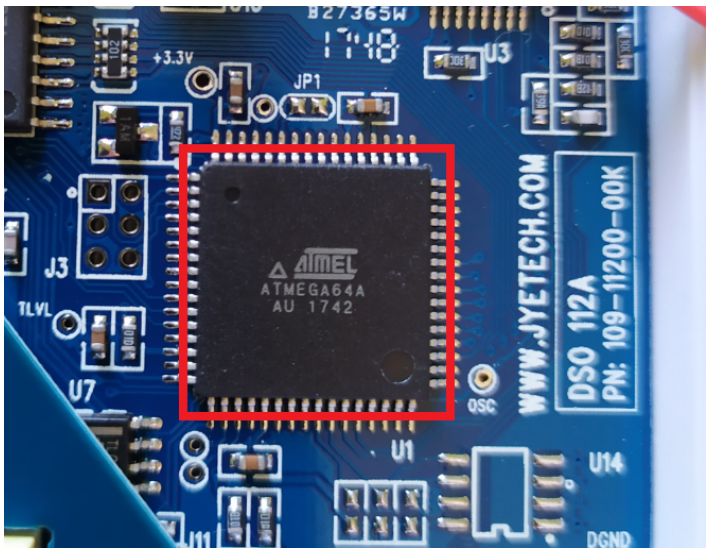

Ilustración 76: ATMEGA64A del osciloscopio

Conclusiones

Hemos visto que una vez se aprende a utilizar CubeMX y nos familiarizamos con las herramientas que nos proporciona el fabricante, programar en estas placas es bastante sencillo una vez se han superado las dificultades iniciales de familiarizarse con las herramientas de desarrollo proporcionadas por el fabricante. El uso de tutoriales que se pueden encontrar en Internet ayuda bastante a iniciarse.

Respecto a si para un proyecto en concreto conviene usar un Cortex-M4 con FPU o sin FPU, todo depende del análisis económico de la situación, y de si es viable hacer la aplicación en punto fijo.

Si sí se puede hacer en punto fijo, entonces habría que estudiar el coste de ambas opciones. Si se hace en punto fijo el desarrollo de la aplicación llevará más tiempo y por tanto costará más dinero, pero permitirá ahorrar en cada unidad vendida ya que los chips serán más baratos. Si se hace en coma flotante, cada unidad será más cara pero desarrollar la aplicación costará menos dinero.

Parece que escoger hacerlo sin FPU conviene cuando no se van a fabricar muchas unidades, ya que en ese caso el tiempo de desarrollo puede ser un mayor coste que el coste de las piezas. Si se van a fabricar muchas unidades, ahorrar en el hardware eliminando la FPU puede suponer una gran reducción de costes.

Se trata de un problema de optimización en el cual se quiere minimizar el coste total de producción.

Para ayudarnos a pensar en el problema, se puede utilizar esta “ecuación”:

$$\boldsymbol{Coste\ de\ Producción = Coste\ de\ Desarrollo + Coste\ Materiales}$$

El coste de producción (**CdP**) es el coste total de implementar el microcontrolador a todas las unidades de un producto determinado.

Por ejemplo, si una empresa quiere producir 5000 unidades de un aparato que contiene un Cortex-M4, CdP sería el coste de comprar y programar 5000 Cortex-M4.

El coste de desarrollo (**CdD**) es el coste del tiempo de los ingenieros que se encargan de escribir el software.

El coste material (**CM**) es el coste de comprar todos los Cortex-M4.

CdD es independiente del número de unidades a fabricar, y CM aumenta con el número de unidades a fabricar ("n") y con el coste de cada microcontrolador ("Coste_Unitario"):

$$CdP = CdD + CdF$$

$$CdP = CdD + n * Coste\_Unitario$$

El CdF es función de "n" y "Coste_Unitario", pero no tiene por qué ser exactamente igual a $n * Coste\_Unitario$ (cuanto mayor es "n", menor suele ser "Coste_Unitario", ya que los proveedores suelen ofrecer precios menores al comprar lotes grandes), pero es una buena aproximación para ayudarnos a pensar en la idea general de la situación.

Representando en un gráfico los costes de producción en función del número de unidades producidas:

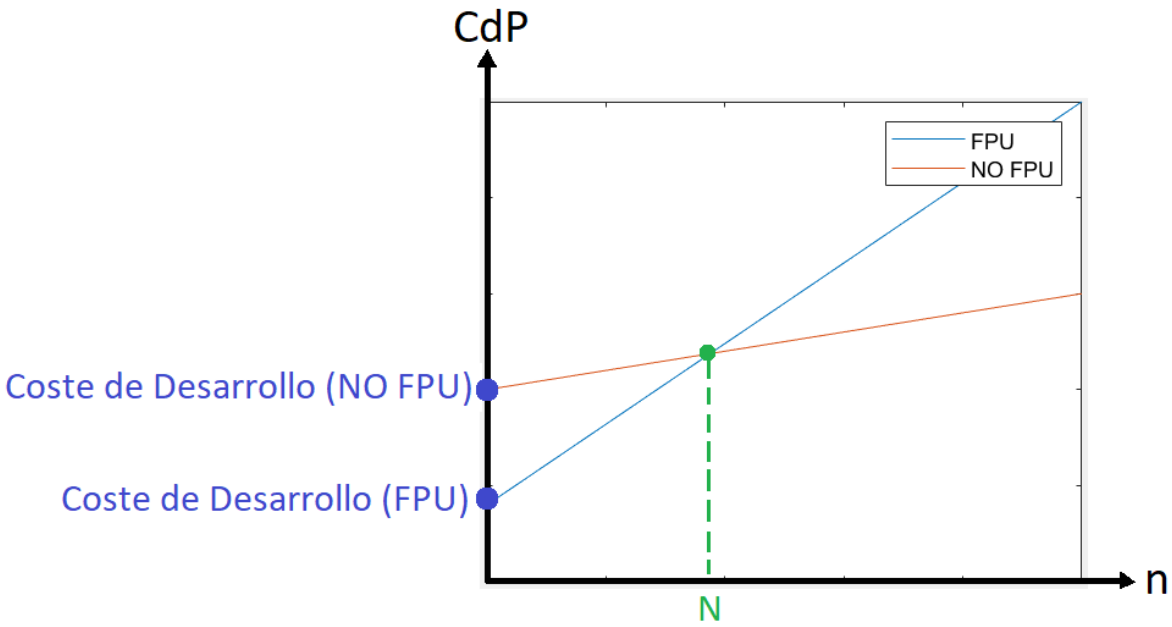

Ilustración 77: Coste de producción en función del número de unidades producidas

La línea roja representa los costes de producción cuando se utilizan microprocesadores sin FPU y se programa en punto fijo, y la línea azul cuando tienen FPU y se programa en coma flotante. Estas rectas vienen de la ecuación:

$$CdP = CdD + n * Coste\_Unitario$$

El coste de desarrollo cuando se trabaja en coma flotante es menor que el coste de desarrollo cuando se trabaja sin FPU; por otro lado, el coste unitario de cada microprocesador es mayor si tienen FPU.

La conclusión es que hay un número "N" de unidades que determina qué se tipo de procesador se debe comprar; si el número de unidades es inferior a "N", se debe elegir uno con FPU, y si es mayor a "N", con FPU.

El motivo por el que el coste de desarrollo es menor cuando se programa en coma flotante es que el trabajo tiende a ser más sencillo y lleva menos tiempo, porque normalmente no tendrá que preocuparse por problemas como overflow y underflow.

Por ejemplo, en el caso de la aplicación de filtrado que se ha hecho en este TFG, me encontré con las siguientes dificultades:

1) Como los parámetros del filtro están entre 0 y 1, es necesario multiplicarlos por un factor de escala para que sean mayores que uno y así evitar underflow. Tras utilizar estos parámetros para realizar el filtrado tengo que deshacer el factor de escala.

2) Al multiplicar los parámetros del filtro por el factor de escala podría haber problemas si algún resultado se sale del rango de la variable que lo guarda.

   La variable "h_int" es un vector de tipo int16_t; el valor máximo que puede guardar es 32,767. Si cada uno de estos parámetros está entre 0 y 1 y los multiplico por 2^17, por ejemplo, es posible que el resultado de alguno de ellos exceda el rango y tenga un overflow.

   Tengo que tener cuenta que una de las propiedades de este filtro es que la suma de todos sus parámetros es aproximadamente 1; por tanto, cuantos más parámetros tenga el filtro (cuanto mayor sea su orden), menores serán los valores de los parámetros, lo que permite un factor de escala mayor sin que haya overflow.

Esto dificulta las cosas, porque implica que el orden del filtro, la precisión que le puedo dar a sus parámetros (factor de escala) y el tipo de los valores del vector del filtro “h_int” (int16_t, int32_t,...) están relacionados.

En un principio pensé dar al usuario la opción de elegir la orden del filtro a través de la pantalla, en lugar de fijarla en 30. Pero tras hacer esto me di cuenta de que entonces es posible que el usuario eligiera un orden lo bastante bajo como para que al multiplicar los parámetros del filtro por el orden de escala ocurra un overflow. Para evitar esto tendría dos opciones: o reducir el factor de escala (empeorando así la calidad del filtrado porque sus coeficientes son menos parecidos a su valor ideal, al tener menos cifras), o eligiendo un tipo que ocupa más memoria, como int32_t.

Todos estos ajustes debidos al uso de punto fijo hacen que sea más difícil asegurar que todo vaya a funcionar bien, ya que parece que sea más probable que uno haya cometido algún error al programar, y que dicho error no se pueda detectar hasta que sea demasiado tarde (bugs cuando el producto ya está en el mercado). También resultarían en un código más largo y más difícil de leer, con más líneas y más complejidad en general, lo que haría que costase más que otra persona lo entendiese con facilidad si quisiera modificarlo.

La aplicación de filtrado que se ha hecho es muy sencilla, y aun así he tenido dificultades. Si el programa a realizar fuera más complejo, creo que es posible que fuera prácticamente imposible de realizar.

Por otro lado, una de las ventajas de utilizar punto fijo (aparte de permitir ahorrar dinero en la FPU) es poder determinar la precisión de las operaciones; al utilizar coma flotante el programador no puede determinar la precisión de las operaciones, ya que si se trabaja con una precisión determinada (“single precision” para números representados en coma flotante en 32 bits, “double precision” para 64 bits, etc.), ésa es la precisión que se tiene, puesto que el tamaño de la mantisa está predeterminado. En punto fijo el ingeniero puede decidir qué precisión se utiliza, determinando el número de bits de los operandos. La precisión de coma

flotante será suficiente para muchas aplicaciones, pero es posible que no lo sea para algunas otras.

Bibliografía y recursos

Libros

[1] YIU, JOSEPH. The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors. Tercera edición. Estados Unidos: Newnes, 2014.

[2] MARTIN, TREVOR. The Designer's Guide to the Cortex-M Processor Family. Segunda edición. Estados Unidos: Newnes, 2016.

[3] FISHER, MARK. ARM Cortex M4 Cookbook. Estados Unidos: Packt publishing, 2016.

[4] PRABHU, K. M. M. Window Functions and Their Applications in Signal Processing. Florida, Estados Unidos: CRC Press, 2014.

Páginas web y otros documentos

(2019) https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4 - Permite consultar información y descargar software en relación con el procesador Cortex-M4: información técnica, herramientas de desarrollo, guías de usuario, etc.

(2019) https://www2.keil.com/mdk5 - Descargar Keil MDK

(2019) https://www.st.com/en/development-tools/stm32cubemx.html - Descargar CubeMX

(2019) https://www.keil.com/pack/doc/STM32Cube/General/html/index.html - Manual para crear proyectos utilizando CubeMX y Keil MDK.

(2019) https://www.st.com/en/evaluation-tools/32f429idiscovery.html- Información acerca de la placa STM32F429I-Discovery utilizada en este proyecto

(2019)https://www.keil.com/pack/doc/CMSIS/DSP/html/group__FIR.html#ga0cf008f650a75f5e2cf82d10691b64d9 – Funciones de la librería de DSP de Arm para implementar filtros FIR

(2019) https://www.youtube.com/watch?v=EYwyIwClsGc – Canal de Youtube de Arm

Anexo: Código

En esta sección se expone el código utilizado para realizar la aplicación. Esta sección se divide en los siguientes apartados:

1) Definiciones previas
2) Función “Main”
3) Función “Pantalla”
4) Función “CalculaCoeficientes”
5) Función “CalculaTIM_ARR”
6) MX_TIM6_Init
7) Función “HAL_TIM_PeriodElapsedCallback”

Esta aplicación requiere mucho más código para poder funcionar, proporcionado por el fabricante; aquí sólo se expone el código escrito por mí, el alumno.

1) Definiciones iniciales

Estas líneas de código se encuentran en main.c, antes de la función main. Se ponen los include necesarios, se define “Pi” como 3.1416, y se establece que el orden del filtro será de 30 taps. Este valor se puede modificar si se desea un número de taps diferente.

Después se define el struct “Inputs_Usuario”para almacenar las frecuencias de corte y de muestreo seleccionadas por el usuario. La función “pantalla” devuelve un struct de tipo “Inputs_Usuario”.

La variable “ts” ha de definirse para poder utilizar la pantalla táctil. Las variables “Modo”, h_FLOAT y h_PFint, que contienen los valores de los filtros en coma flotante y punto fijo respectivamente, se definen como globales porque los valores almacenados en estas variables se utilizan en varias funciones y una de ellas (“HAL_TIM_PeriodElapsedCallback”) está declarada por el fabricante, y no he conseguido modificar su prototipo para que acepte más entradas. Las variables “Modo”, “ADC_DATA”, “SalidaDAC_float”, “SalidaDAC_int” y “SalidaDAC_intTEMP” son globales porque si se declarasen en la función que las utiliza (“HAL_TIM_PeriodElapsedCallback”), serían declaradas miles de veces por segundo, una vez por cada período de muestreo.

```
// USER CODE BEGIN Includes
#include "main.h"
#include "stm32f429i_discovery_lcd.h"
#include "main.h"
#include "stm32f4xx_hal.h"
#include "arm_math.h"
#include <stdint.h>
#include <stdio.h>
#include "stm32f429i_discovery_ts.h"
#include "stm32f429i_discovery.h"
// USER CODE END Includes

// Private variables
#define Pi 3.1416
#define OrdenFiltro 30

struct Inputs_Usuario {
  int FrecuenciaCorte;
  int FrecuenciaMuestreo;
};

int Modo=0; // 0 si coma flotante, 1 si punto fijo

float h_float[OrdenFiltro+1];  // coeficientes filtro en coma flotante
int16_t h_int[OrdenFiltro+1]; // coeficientes filtro en punto fijo
int16_t ADC_DATA[OrdenFiltro+1]; // vector valores leídos por el ADC
float SalidaDAC_float = 0; // valor de salida del DAC en coma flotante
volatile int SalidaDAC_int=0; // valor de salida del DAC en punto fijo
int32_t SalidaDAC_intTEMP = 0; // valor anterior sin ajustar escala

TS_StateTypeDef  ts; // variable necesaria para utilizar el display
```

2) Main

Es la función principal de la aplicación, la primera en ejecutarse y la que llama a las demás funciones del programa.

Primero llama a la función pantalla, que devuelve los parámetros elegidos por el usuario a través de la pantalla táctil.

Estos parámetros son la información necesaria para llamar a las funciones CalculaCoeficientes, CalculaTIM_ARR y HAL_TIM_Base_Start_IT. El cometido de cada una de estas funciones se explica en su apartado.

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();

    int TIM_ARR;
    struct Inputs_Usuario Inputs;

    Inputs = pantalla();
    CalculaCoeficientes(Inputs);
    TIM_ARR = CalculaTIM_ARR(Inputs.FrecuenciaMuestreo);

    MX_GPIO_Init();
    MX_ADC1_Init();
    MX_DAC_Init();
    MX_TIM6_Init(TIM_ARR);

    HAL_TIM_Base_Start_IT(&htim6);

  while (1)
  {

  }
}
```

3) Pantalla

Recibe los inputs introducidos por el usuario a través de la pantalla táctil. Estos inputs son:

- Coma flotante o punto fijo
- Frecuencia de corte
- Frecuencia de muestreo

En función de los mismos da valores a la variable global “Modo” (vale 0 si es coma flotante y 1 si es punto fijo) y a la variable “Inputs”, la cual es un struct de tipo Inputs_usuario y es devuelta a main. “Inputs” contiene dos campos, “FrecuenciaCorte” y “FrecuenciaMuestreo”.

La idea general es la siguiente: se comprueba cuál fue la última posición de la pantalla tocada por el usuario con la siguiente línea:

```
BSP_TS_GetState(&ts);
```

Cada vez que dicha línea se ejecuta, la variable “ts” pasa a guardar las coordenadas (en píxeles) de la posición que tocó el usuario. Dicha variable es un struct con el campo “.X”, que guarda la coordenada X de la posición tocada por el usuario, y el campo “.Y”, que guarda la coordenada Y. La pantalla tiene una resolución de 320 x 240 píxeles, por lo que si, por ejemplo, ts.Y valiese 160 y ts.X valiese 120, esto implicaría que el usuario tocó en el centro de la pantalla.

Debido a esto, en la función “Pantalla” se repite este patrón:

```
if (ts.X > Valor1 && ts.Y < Valor2 && ts.X > Valor3 && ts.Y < Valor4){
        Código
}
```

Es decir: si el usuario tocó es una región determinada de la pantalla (acotada por “Valor1”, “Valor2”, “Valor3” y “Valor4”, que denotan coordenadas en píxeles), ejecutar el código del “if”.

```
struct Inputs_Usuario pantalla(){
        struct Inputs_Usuario Inputs;

        BSP_SDRAM_Init();
        BSP_LCD_Init();//Inicializamos LCD
        BSP_TS_Init(240, 320);

        BSP_LCD_Init();
        BSP_LCD_LayerDefaultInit(1, SDRAM_DEVICE_ADDR);
        BSP_LCD_DisplayOn();

        //MENU PARA ELEGIR PUNTO FIJO O COMA FLOTANTE
        BSP_LCD_SelectLayer(1);
        BSP_LCD_Clear(LCD_COLOR_BLACK);
        BSP_LCD_DisplayStringAtLine(3,"Coma flotante");
        BSP_LCD_DisplayStringAtLine(8,"Punto fijo");

        int aux=1;
        Inputs.FrecuenciaCorte=3000;
        Inputs.FrecuenciaMuestreo=40000;
        ts.Y=0;
        while(aux==1){
            BSP_TS_GetState(&ts);
            if (ts.Y>10&&ts.Y<160){
                BSP_LCD_Clear(LCD_COLOR_WHITE);
                BSP_LCD_DisplayStringAt(10, 150, (uint8_t *)"Coma Flotante", LEFT_MODE);
                HAL_Delay(1500);
                aux=0;
                Modo=0; //indica al resto del código que se desea filtrar en coma
flotante
            }

            if (ts.Y>200){
                BSP_LCD_Clear(LCD_COLOR_WHITE);
                BSP_LCD_DisplayStringAt(10, 150, (uint8_t *)"Punto Fijo", LEFT_MODE);
                HAL_Delay(1500);
                aux=0;
                Modo=1; //indica al resto del código que se desea filtrar en punto fijo
            }
        }

        //MENU PARA ELEGIR FRECUENCIA DE CORTE
        BSP_LCD_Clear(LCD_COLOR_BLACK);
        BSP_LCD_DisplayStringAt(0, 20, (uint8_t *)"F. Corte", LEFT_MODE);
        BSP_LCD_DisplayStringAt(180, 20, (uint8_t *)" > ", LEFT_MODE);
        BSP_LCD_DisplayStringAt(15, 200, (uint8_t *)"-10", LEFT_MODE);
        BSP_LCD_DisplayStringAt(155, 200, (uint8_t *)"+10", LEFT_MODE);
        BSP_LCD_DisplayStringAt(15, 240, (uint8_t *)"-100", LEFT_MODE);
        BSP_LCD_DisplayStringAt(155, 240, (uint8_t *)"+100", LEFT_MODE);
        BSP_LCD_DisplayStringAt(15, 280, (uint8_t *)"-1000", LEFT_MODE);
        BSP_LCD_DisplayStringAt(155, 280, (uint8_t *)"+1000", LEFT_MODE);

        char cadena[10];
        sprintf(cadena, "%d Hz", Inputs.FrecuenciaCorte);
        BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
        BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);

        aux=1;
        while(aux==1){
                ts.X=0;
                ts.Y=0;
                BSP_TS_GetState(&ts);

                BSP_LCD_SetTextColor(LCD_COLOR_BLUE);

                if (ts.X>10&&ts.X<120&&ts.Y>170&&ts.Y<240){
                    if (Inputs.FrecuenciaCorte >10){
                        Inputs.FrecuenciaCorte = Inputs.FrecuenciaCorte-10;
```

```
        }
        sprintf(cadena, "%d Hz", Inputs.FrecuenciaCorte);
        BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
        BSP_LCD_FillRect(60, 80,220,60);
        BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
        BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
        HAL_Delay(100);
    }
    if (ts.X>10&&ts.X<120&&ts.Y>240&&ts.Y<280){
        if (Inputs.FrecuenciaCorte >100){
            Inputs.FrecuenciaCorte = Inputs.FrecuenciaCorte-100;
        }
        sprintf(cadena, "%d Hz", Inputs.FrecuenciaCorte);
        BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
        BSP_LCD_FillRect(60, 80,220,60);
        BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
        BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
        HAL_Delay(100);
    }
    if (ts.X>10&&ts.X<120&&ts.Y>280&&ts.Y<320){
        if (Inputs.FrecuenciaCorte >1000){
            Inputs.FrecuenciaCorte = Inputs.FrecuenciaCorte-1000;
        }
        sprintf(cadena, "%d Hz", Inputs.FrecuenciaCorte);
        BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
        BSP_LCD_FillRect(60, 80,220,60);
        BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
        BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
        HAL_Delay(100);
    }

    if (ts.X>120&&ts.X<240&&ts.Y>170&&ts.Y<240){
        if (Inputs.FrecuenciaCorte < 39990){
            Inputs.FrecuenciaCorte = Inputs.FrecuenciaCorte+10;
        }
        sprintf(cadena, "%d Hz", Inputs.FrecuenciaCorte);
        BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
        BSP_LCD_FillRect(60, 80,200,60);
        BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
        BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
        HAL_Delay(100);
    }
    if (ts.X>120&&ts.X<240&&ts.Y>240&&ts.Y<280){
        if (Inputs.FrecuenciaCorte < 39900){
            Inputs.FrecuenciaCorte = Inputs.FrecuenciaCorte+100;
        }
        sprintf(cadena, "%d Hz", Inputs.FrecuenciaCorte);
        BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
        BSP_LCD_FillRect(60, 80,200,60);
        BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
        BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
        HAL_Delay(100);
    }
    if (ts.X>120&&ts.X<240&&ts.Y>280&&ts.Y<320){
        if (Inputs.FrecuenciaCorte < 39000){
            Inputs.FrecuenciaCorte = Inputs.FrecuenciaCorte+1000;
        }
        sprintf(cadena, "%d Hz", Inputs.FrecuenciaCorte);
        BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
        BSP_LCD_FillRect(60, 80,200,60);
        BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
        BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
        HAL_Delay(100);
    }
    if (ts.Y>5&&ts.Y<100&&ts.X>180){
        HAL_Delay(100);
        BSP_LCD_Clear(LCD_COLOR_WHITE);
        HAL_Delay(500);
```

```
            aux=0;
        }
    HAL_Delay(150);
}

HAL_Delay(150);

//MENU PARA ELEGIR FRECUENCIA DE MUESTREO
BSP_LCD_Clear(LCD_COLOR_BLACK);
BSP_LCD_DisplayStringAt(0, 20, (uint8_t *)"F.Muestreo", LEFT_MODE);
BSP_LCD_DisplayStringAt(180, 20, (uint8_t *)" > ", LEFT_MODE);
BSP_LCD_DisplayStringAt(15, 200, (uint8_t *)"-10", LEFT_MODE);
BSP_LCD_DisplayStringAt(155, 200, (uint8_t *)"+10", LEFT_MODE);
BSP_LCD_DisplayStringAt(15, 240, (uint8_t *)"-100", LEFT_MODE);
BSP_LCD_DisplayStringAt(155, 240, (uint8_t *)"+100", LEFT_MODE);
BSP_LCD_DisplayStringAt(15, 280, (uint8_t *)"-1000", LEFT_MODE);
BSP_LCD_DisplayStringAt(155, 280, (uint8_t *)"+1000", LEFT_MODE);

sprintf(cadena, "%d Hz", Inputs.FrecuenciaMuestreo);
BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);

aux = 1;
while(aux==1){
        ts.X=0;
        ts.Y=0;
        BSP_TS_GetState(&ts);
        BSP_LCD_SetTextColor(LCD_COLOR_BLUE);

        if (ts.X>10&&ts.X<120&&ts.Y>170&&ts.Y<240){
            if (Inputs.FrecuenciaMuestreo >10){
                Inputs.FrecuenciaMuestreo = Inputs.FrecuenciaMuestreo-10;
            }
            sprintf(cadena, "%d Hz", Inputs.FrecuenciaMuestreo);
            BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
            BSP_LCD_FillRect(60, 80,220,60);
            BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
            BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
            HAL_Delay(100);
        }
        if (ts.X>10&&ts.X<120&&ts.Y>240&&ts.Y<280){
            if (Inputs.FrecuenciaMuestreo >100){
                Inputs.FrecuenciaMuestreo = Inputs.FrecuenciaMuestreo-100;
            }
            sprintf(cadena, "%d Hz", Inputs.FrecuenciaMuestreo);
            BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
            BSP_LCD_FillRect(60, 80,220,60);
            BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
            BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
            HAL_Delay(100);
        }
        if (ts.X>10&&ts.X<120&&ts.Y>280&&ts.Y<320){
            if (Inputs.FrecuenciaMuestreo >1000){
                Inputs.FrecuenciaMuestreo = Inputs.FrecuenciaMuestreo-1000;
            }
            sprintf(cadena, "%d Hz", Inputs.FrecuenciaMuestreo);
            BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
            BSP_LCD_FillRect(60, 80,220,60);
            BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
            BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
            HAL_Delay(100);
        }
        if (ts.X>120&&ts.X<240&&ts.Y>170&&ts.Y<240){
            if (Inputs.FrecuenciaMuestreo < 39990){
                Inputs.FrecuenciaMuestreo = Inputs.FrecuenciaMuestreo+10;
            }
            sprintf(cadena, "%d Hz", Inputs.FrecuenciaMuestreo);
            BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
```

```
                    BSP_LCD_FillRect(60, 80,200,60);
                    BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
                    BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
                    HAL_Delay(100);
                }
                if (ts.X>120&&ts.X<240&&ts.Y>240&&ts.Y<280){
                    if (Inputs.FrecuenciaMuestreo < 39900){
                        Inputs.FrecuenciaMuestreo = Inputs.FrecuenciaMuestreo+100;
                    }
                    sprintf(cadena, "%d Hz", Inputs.FrecuenciaMuestreo);
                    BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
                    BSP_LCD_FillRect(60, 80,200,60);
                    BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
                    BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
                    HAL_Delay(100);
                }
                if (ts.X>120&&ts.X<240&&ts.Y>280&&ts.Y<320){
                    if (Inputs.FrecuenciaMuestreo < 39000){
                        Inputs.FrecuenciaMuestreo = Inputs.FrecuenciaMuestreo+1000;
                    }
                    sprintf(cadena, "%d Hz", Inputs.FrecuenciaMuestreo);
                    BSP_LCD_SetTextColor(LCD_COLOR_BLACK);
                    BSP_LCD_FillRect(60, 80,200,60);
                    BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
                    BSP_LCD_DisplayStringAt(60, 80, (uint8_t *)cadena, LEFT_MODE);
                    HAL_Delay(100);
                }
                if (ts.Y>5&&ts.Y<100){
                    HAL_Delay(100);
                    BSP_LCD_Clear(LCD_COLOR_WHITE);
                    BSP_LCD_DisplayStringAt(40, 150, (uint8_t *)"Filtrando", LEFT_MODE);
                    HAL_Delay(500);
                    aux=0;
                }
            HAL_Delay(150);
        }
return Inputs;
}
```

4) CalculaCoeficientes

El propósito de esta función es dar valores a las variables globales “h_float” y “h_int”, que contienen los valores de los filtros en coma flotante y en punto fijo respectivamente.

```
void CalculaCoeficientes(struct Inputs_Usuario Inputs){
    float h_float_temp[OrdenFiltro+1];
    float Wc;
    int n=0;

    Wc = (2*Pi*Inputs.FrecuenciaCorte)/Inputs.FrecuenciaMuestreo;

    //calculo de los coeficientes del filtro en coma flotante
    for (n=0; n<=OrdenFiltro; n++){
        h_float[n]=((float)sin(Wc*(n-OrdenFiltro/2)))/(Pi*(n-OrdenFiltro/2));
    }
    //calculo del coeficiente central del filtro en coma flotante
    if (OrdenFiltro % 2 == 0){
            h_float[OrdenFiltro/2] = Wc/Pi;
    }

    //calculo de los coeficientes del filtro en punto fijo
    for (n=0; n<=OrdenFiltro; n++){
        h_float_temp[n]=(h_float[n])*131072; // Multiplicar por 131072 es como
desplazar 17 posiciones en binario (2^17)
    }

    for (n=0;n<=OrdenFiltro;n++){  // h_PFint contiene los valores del filtro en
formato int, para punto fijo
        h_int[n] = (int16_t)h_float_temp[n];
    }
}
```

5) CalculaTIM_ARR

Recibe la frecuencia de muestreo seleccionada por el usuario y devuelve TIM_ARR, un valor que se pasa como parámetro de entrada a la función que configura el temporizador (“MX_TIM6_Init”) y que determina la frecuencia con la que se ejecuta la función “HAL_TIM_PeriodElapsedCallback”, la que realiza el filtrado.

```
int CalculaTIM_ARR(int FrecMuest) {
	int TIM_ARR;
	TIM_ARR = (90000000/FrecMuest)-1;
	return TIM_ARR;
}
```

6) MX_TIM6_Init

Configura el temporizador que hace que se ejecute la función "HAL_TIM_PeriodElapsedCallback" (función que realiza el filtrado) a intervalos regulares. Es decir, "MX_TIM6_Init" determina la frecuencia de muestreo en nuestra aplicación.

"MX_TIM6_Init" es creada por CubeMX, y la única modificación que se ha hecho es incluir el parámetro de entrada "TIM_ARR", para que la frecuencia del temporizador sea modificable por el usuario al introducir la frecuencia de muestreo a través de la pantalla táctil mediante la línea "htim6.Init.Period = TIM_ARR". Ésta es la única línea que ha sido modificada, el resto del código de esta función es el proporcionado por CubeMX.

```
static void MX_TIM6_Init(int TIM_ARR)
{
  /* USER CODE BEGIN TIM6_Init 0 */

  /* USER CODE END TIM6_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM6_Init 1 */

  /* USER CODE END TIM6_Init 1 */
  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 0;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = TIM_ARR; // TIM_ARR depende de la frec. de muestreo
  htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_ENABLE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM6_Init 2 */

  /* USER CODE END TIM6_Init 2 */
}
```

7) HAL_TIM_PeriodElapsedCallback

Función que realiza el filtrado. La frecuencia de muestreo seleccionada indica cuántas veces se ejecuta por segundo.

La función comprueba si la variable global “Modo” vale 0 ó 1. Si vale 0, ejecuta el filtrado en coma flotante; si vale 1, en punto fijo.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){

    if (Modo==0){

        SalidaDAC_float = 0;

        for (int i=0; i<=OrdenFiltro; i++){
            SalidaDAC_float = SalidaDAC_float + ADC_DATA[i]*h_float[OrdenFiltro+1-i];
        }

        for (int i=0;i<=OrdenFiltro-1;i++){
            ADC_DATA[i] = ADC_DATA[i+1];
        }

        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1,100);
        ADC_DATA[OrdenFiltro] = HAL_ADC_GetValue(&hadc1);

        HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
        HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, SalidaDAC_float);
    }

    if (Modo==1){

        SalidaDAC_intTEMP = 0;

        for (int i=0; i<=OrdenFiltro; i++){
             SalidaDAC_intTEMP = (int32_t)SalidaDAC_intTEMP +
             ((int16_t)(ADC_DATA[i])*(int16_t)h_int[OrdenFiltro+1-i]);
        }

        SalidaDAC_int = (int32_t)(SalidaDAC_intTEMP>>17);

        for (int i=0;i<=OrdenFiltro-1;i++){
            ADC_DATA[i] = ADC_DATA[i+1];
        }

        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1,100);
        ADC_DATA[OrdenFiltro] = HAL_ADC_GetValue(&hadc1);

        HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
        HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, SalidaDAC_int);
    }
}
```