MASB | STM32
ADCs

#stm32 #c #biomedical #microcontroladores #programacion

Albert Álvarez Carulla
hello@thealbert.dev
https://thealbert.dev/

25 de febrero de 2020

Índice

1. ADCs

En la primera parte de esta práctica hemos *jugado* con el ADC en Arduino. En esta segunda parte vamos a **operar el ADC con las HAL** en STM32CubeIDE. Una vez más, más ~~dificultad~~ complejidad aporta una **mayor flexibilidad**.

Vamos a ver cómo operar el **ADC con funciones bloqueantes** (funciones basadas en la técnica de *polling*), cómo realizar un **barrido en más de un canal** de entrada del ADC y cómo combinar el ADC con un periférico nuevo: el **DMA (*Direct Memory Access*)**.

1.1. Objetivos

- Introducción a los ADCs en STM32CubeIDE.
- Operación del ADC con funciones bloqueantes.
- Barrido de conversión de canales del ADC.
- Combinación del DMA con el ADC.
- Conversiones periódicas con el *timer*.
- Medición de temperatura con el canal interno de un ADC.
- Primer vistazo a la comunicación serie.

1.2. Procedimiento

1.2.1. Lectura de la AD2

En este primer proyecto, vamos a una señal analógica generada por la AD2. Tenéis libertad para escoger la señal que queráis utilizar (ECG, seno, triangulo, etc.), pero que siempre esté entre los 3.3 y 0 V (no queremos tensiones negativas ni por encima de la alimentación del microcontrolador). Realizada la lectura de la señal, **la enviaremos al ordenador** mediante una comunicación serie UART (*Universal Asynchronous Receiver-Transmitter*).

1.2.1.1. Creación del proyecto Vamos a crear una rama nueva de desarrollo a partir de la rama de `master`. En la carpeta `stm32cube` creamos un proyecto llamado `masb-p04`.

1.2.1.2. Configuración del ADC Empezamos con la configuración del pin donde está conectado la AD2.

> La conexión de la AD2 se realiza como en la primera parte de la práctica.

El **pin** al que está conectado es el `PA0`. Buscamos ese pin en el `Pinout view` y **lo configuramos como `ADC1_IN0` con el *label* `RDIV`**. La nomenclatura `ADC1_IN0` indica que se trata del **canal 0 del ADC 1.** Seguidamente, vamos al menú lateral y, dentro del desplegable `Analog`, clicamos sobre `ADC1`. De este modo, nos dirigiremos al **formulario de configuración del único ADC** del que dispone este microcontrolador.

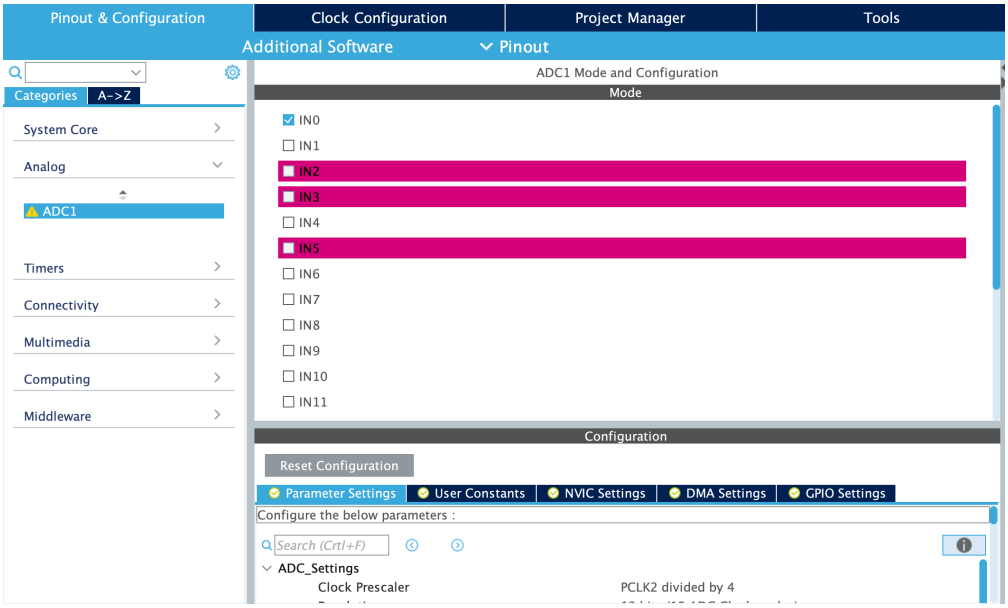

Figura 1: Configuración del ADC.

En la parte superior, llamada `Mode`, aparecen los canales del ADC. **Un canal del ADC es una entrada a este**. El ADC dispone de 16 canales más un par internos. Uno de esos canales internos está conectado a un sensor de temperatura interno del microcontrolador.

> **ADC solo hay uno**. Pero ese único ADC puede convertir la señal de cada uno de sus canales **alternando** entre ellos. **No podrá leer todos los canales simultáneamente**.

Actualmente, tenemos habilitado el canal `IN0` puesto que hemos configurado previamente el pin de ese canal.

En la parte inferior del formulario de configuración del ADC `Configuration`, configuramos los **parámetros del ADC** en sí: frecuencia de operación, resolución, alineamiento de los datos, (des)habilitación de diferentes modos de conversión, etc. Os dejo la arda tarea de leer la documentación del microcontrolador sobre la operación del ADC para saber qué es cada cosa (página 213). De momento **lo dejamos todo por defecto**.

> Hay un botón super útil llamado `Reset Configuration` que os reinicia al estado por defecto la configuración del ADC si tocáis algo.

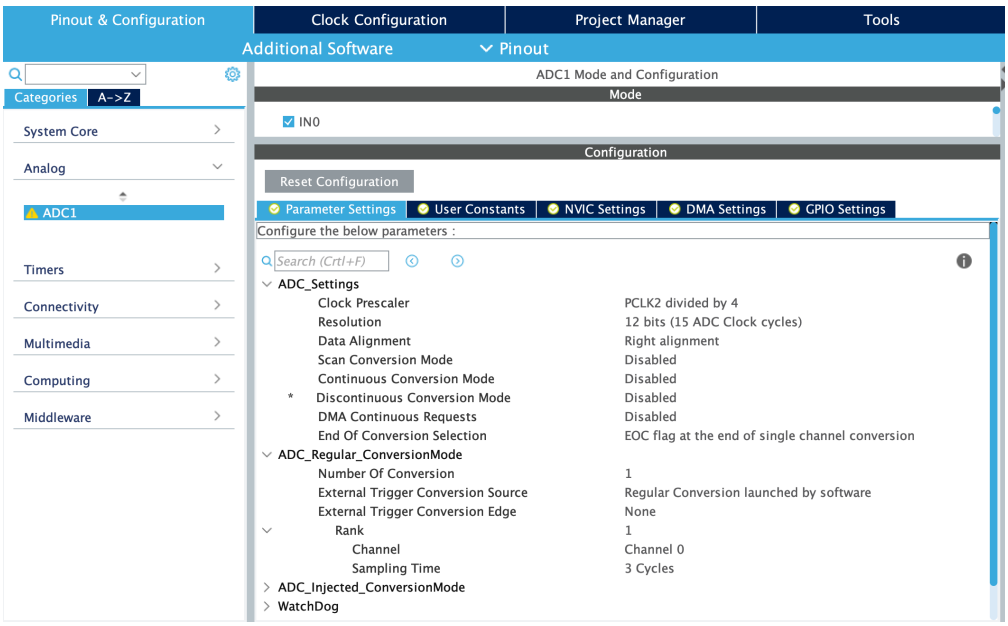

Figura 2: Configuración del ADC.

Guardamos el archivo de configuración y generamos el código. Vámonos al `main.c`.

1.2.1.3. Operación bloqueante del ADC

Ahora veremos cómo leer una señal analógica con el ADC con funciones bloqueantes. Se llama **bloqueante** porque primeramente iniciaremos la conversión y después utilizaremos ***polling*** para leer todo el rato el estado de la conversión y esperar a que esta finalice para leer el resultado obtenido.

Las funciones que utilizaremos són: `HAL_ADC_Start`, `HAL_ADC_PollForConversion` y `HAL_ADC_GetValue`. La primera función se encarga de **iniciar una conversión del ADC**. La segunda función hace que el **código pare su ejecución hasta que finalice la conversión** iniciada. La última función sirve para **obtener el valor resultante** de la conversión. Lo que haremos será iniciar la conversión, esperar a que finalice y guardaremos el resultado en una variable `uint32_t` llamada `ad2ADC`. La función `main` quedaría del siguiente modo:

```
1  ...
2
3    /* USER CODE BEGIN 2 */
4
5    uint32_t ad2ADC = 0; // no hace falta que sea global, es local
```

```
6
7       /* USER CODE END 2 */
8
9     /* Infinite loop */
10    /* USER CODE BEGIN WHILE */
11    while (1)
12    {
13      HAL_ADC_Start(&hadc1); // iniciamos la conversion
14      HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que finalice
            la conversion
15          ad2ADC = HAL_ADC_GetValue(&hadc1);  // leemos el resultado de
                la conversion y lo
16                                              // guardamos en ad2ADC
17      /* USER CODE END WHILE */
18
19      /* USER CODE BEGIN 3 */
20    }
21    /* USER CODE END 3 */
22
23  ...
```

Si compilamos el proyecto, **nos aparecerá el siguiente *warning***:

```
1  variable 'ad2ADC' set but not used [-Wunused-but-set-variable]
```

El compilador nos está diciendo que **hemos creado una variable que luego no utilizamos**. Es decir, guardamos un valor en ella, pero luego no hacemos nada con este valor. Es un *warning* que tal y como se configura el compilador por defecto, no da problemas; pero es un *warning* importante ya que, si no fuera por la configuración por defecto, **el compilador entiende que no la usamos, que no la necesitamos y por tanto no creará la variable y la borrará**.

¿Cómo podemos evitar este *warning*? Hay dos modos. El primer es añadir `volatile` en la declaración de la variable. El compilador no optimiza esta variable y por tanto no la elimina si no se usa. Pero sabemos que utilizar la *keyword* `volatile` tiene un efecto sobre cómo se lee/escribe en memoria esa variable.

El segundo modo es **llamar esa variable para no hacer nada**. Esto lo hacemos con `(void)ad2ADC;`. Llamamos a la variable, pero no hacemos con ella. Para que esta nueva instrucción no nos moleste, podemos **añadirla justo después del *while loop*** puesto que **la ejecución del programa nunca irá más allá de ese bucle**. La función `main` quedaría:

```
1  ...
2
3    /* USER CODE BEGIN 2 */
4
5    uint32_t ad2ADC = 0; // no hace falta que sea global, es local
6
7      /* USER CODE END 2 */
```

```
 8
 9    /* Infinite loop */
10    /* USER CODE BEGIN WHILE */
11    while (1)
12    {
13      HAL_ADC_Start(&hadc1); // iniciamos la conversion
14      HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que finalice
            la conversion
15          ad2ADC = HAL_ADC_GetValue(&hadc1);  // leemos el resultado de
                la conversion y lo
16                                              // guardamos en ad2ADC
17      /* USER CODE END WHILE */
18
19      /* USER CODE BEGIN 3 */
20    }
21      (void)ad2ADC; // evitar warning
22    /* USER CODE END 3 */
23
24  ...
```

Ya sin errores ni *warnings* durante la compilación, iniciamos la depuración del programa.

Una vez iniciado el programa, **añadimos un *breakpoint* en la instrucción `HAL_ADC_Start(&hadc1)`**. A la derecha, tenemos la ventana llamada `Variables` donde aparece la variable `ad2ADC` y su valor. **Cada vez que reanudamos la ejecución del programa y se detiene en el *breakpoint*, el valor de la variable en la ventana `Variables` se actualiza.**

> Ignorad que la variable aparece como `potADC`. Es como se llamaba antes al utilizarse un potenciómetro en lugar de la AD2.

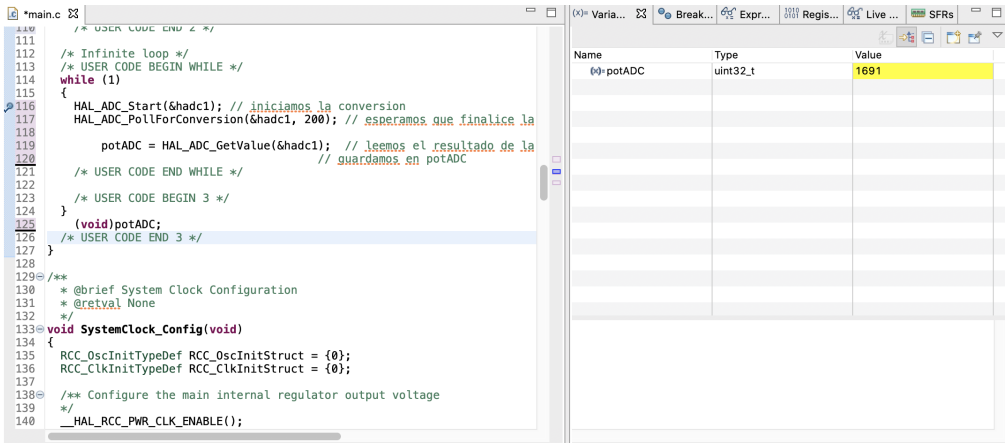

Figura 3: Ventana de variables.

1.2.2. Comunicación serie

Hay herramientas avanzadas de depuración que dejan visualizar la memoria del microcontrolador en tiempo real. Esto permitiría comprobar el funcionamiento del ADC. Pero, de momento, nos quedan un poco grande esas herramientas. Sin embargo, vamos a utilizar la comunicación serie para **enviar el resultado del ADC al ordenador** y, **mediante CoolTerm visualizar esos datos**.

No es objeto de esta práctica ver la comunicación serie. Tampoco nos hace falta configurar nada puesto que al crear el proyecto hemos inicializado al estado por defecto los periféricos del microcontrolador; quedando el periférico UART configurado. Vamos a ver directamente cómo enviar el valor al ordenador.

La función que utilizaremos es la `HAL_UART_Transmit`. A esta función le pasaremos la configuración pertinente, un puntero a un *buffer*, el número de elementos que enviaremos de ese *buffer* y el número de milisegundos máximos para ejecutar esta instrucción. No veremos en esta práctica la parte de comunicación. Simplemente, añadid el código pertinente para que la función `main` quede del siguiente modo:

```
1  ...
2
3  /* Private includes
      ----------------------------------------------------------*/
4  /* USER CODE BEGIN Includes */
5  #include <stdio.h>
6  /* USER CODE END Includes */
7
8  ...
9
10     /* USER CODE BEGIN 2 */
11
12   uint32_t ad2ADC = 0; // no hace falta que sea global, es local
13   char txBuffer[32] = { 0 }; // buffer para transmitir
14   uint8_t txBufferSize = 0; // tamano a enviar
15
16   /* USER CODE END 2 */
17
18   /* Infinite loop */
19   /* USER CODE BEGIN WHILE */
20   while (1)
21   {
22         HAL_ADC_Start(&hadc1); // iniciamos la conversion
23         HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que
              finalice la conversion
24         ad2ADC = HAL_ADC_GetValue(&hadc1);  // leemos el resultado de
              la conversion y lo
25                                             // guardamos en ad2ADC
26
```

```
27          txBufferSize = sprintf(txBuffer, "%lu\n", ad2ADC); //
               codificamos el numero a texto
28          HAL_UART_Transmit(&huart2, &txBuffer, txBufferSize, 1000); //
               enviamos el dato
29
30      /* USER CODE END WHILE */
31
32      /* USER CODE BEGIN 3 */
33    }
34    /* USER CODE END 3 */
35
36  ...
```

Aparte del código añadido a la función `main`, daros cuenta que hemos añadido un `#include` en la parte superior del `main.c` (para que tener disponible la función `sprintf`) y hemos eliminado `(void)ad2ADC;` después del *while loop* puesto que ahora sí que utilizamos la variable `ad2ADC` y no nos saltará el *warning*.

Compilamos y depuramos. Con el programa corriendo, **vamos a CoolTerm**, **configuramos el puerto** serie como en la primera parte de la práctica, y **un baudrate de 115200 baudio.** Delante de nosotros tienen que mostrarse los resultados del ADC enviados por el microcontrolador al ordenador.

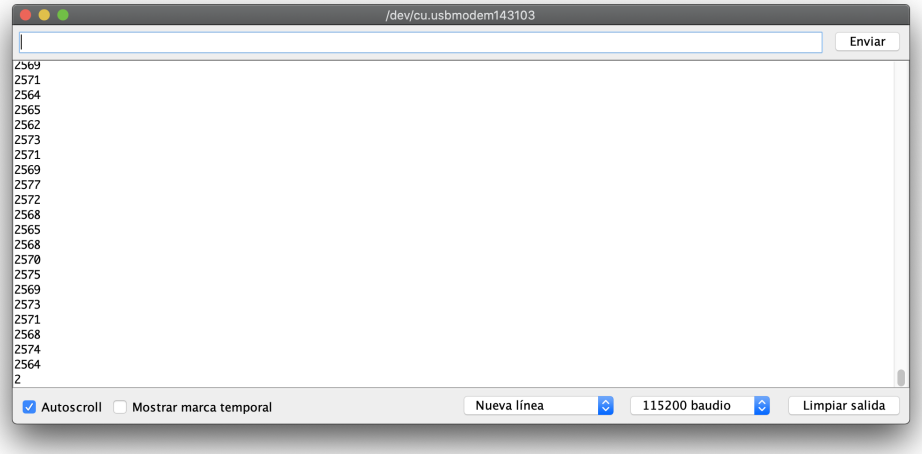

Figura 4: Monitor serie.

Abrimos el chart y.. Bueno… Pasa lo que ya sabíamos: que se autoescala el eje de ordenadas. Para solucionarlo, vamos a enviar al ordenador las constantes 0 y 4095.

> Esta vez enviamos 4095 y no 1023 puesto que el ADC realiza una conversión de 12 bits. En Arduino está capado a 10 bits.

Para hacer esto, **modificamos la instrucción justo antes de `HAL_UART_Transmit`**. Quedaría así:

```
...

    /* USER CODE BEGIN 2 */

  uint32_t ad2ADC = 0; // no hace falta que sea global, es local
  char txBuffer[32] = { 0 }; // buffer para transmitir
  uint8_t txBufferSize = 0; // tamano a enviar

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
        HAL_ADC_Start(&hadc1); // iniciamos la conversion
        HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que
           finalice la conversion
        ad2ADC = HAL_ADC_GetValue(&hadc1);  // leemos el resultado de
           la conversion y lo
                                        // guardamos en ad2ADC

        txBufferSize = sprintf(txBuffer, "0,4095,%lu\n", ad2ADC); //
           codificamos el numero a texto
        HAL_UART_Transmit(&huart2, &txBuffer, txBufferSize, 1000); //
           enviamos el dato

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */

...
```

Compilamos, depuramos, iniciamos la ejecución del programa y nos vamos al `Chart` de CoolTerm. Ahora sí tenemos lo que queríamos.

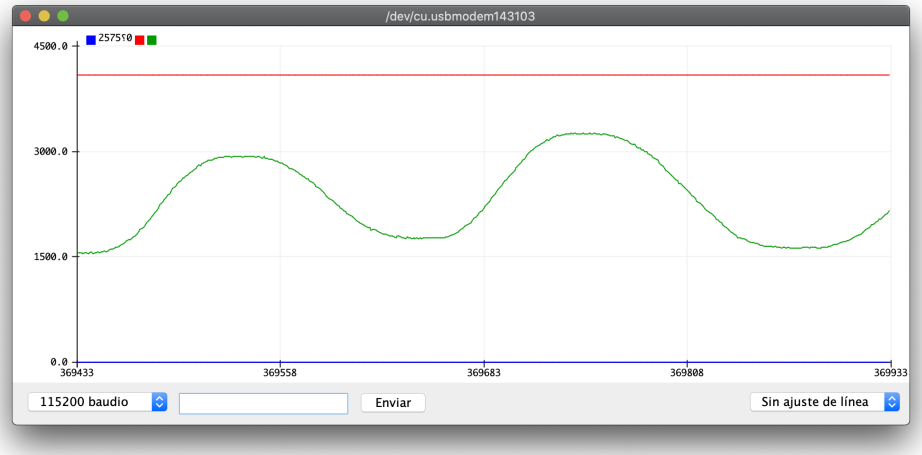

Figura 5: Serial plotter.

Vamos a ver ahora cómo leer dos canales del ADC.

1.2.3. Modo escaneo

El nombre **"escaneo" viene del hecho que en este modo el ADC escanea diferentes canales** que nosotros configuremos de tal modo que el ADC va alternando de canal a canal para hacer una conversión en cada uno de ellos. En esta parte vamos a leer el canal 0 al que está conectada la AD2 y el canal interno conectado al sensor de temperatura.

1.2.3.1. Configuración del microcontrolador **Abrimos el archivo `.ioc`** de nuestro proyecto y vamos al formulario de configuración del ADC. Aparte del canal `IN0` que ya tenemos habilitado, **habitamos en canal `Temperature Sensor Channel`**. En el subformulario inferior `Configuration`, **habilitamos `Scan Conversión Mode` y `Discontinuos Conversión Mode`.** En `ADC_Regular_ConversionMode`, configuramos el `Number Of Conversion` a 2. A continuación, **en `Rank` 2, seleccionamos `Channel Temperature Sensor` en el campo `Channel`**. Lo que hemos hecho es habilitar el modo escaneo de manera discontinua. ¿Qué quiere decir? Que no hará automáticamente la conversión de ambos canales consecutivamente; sino que **tendremos que leer el ADC para leer el primer canal y volver a leer el ADC para leer el segundo**. Hay otros modos de operar, pero aquí utilizaremos este. El orden de escaneo de los canales viene definido por los *ranks*. **Se empieza con los *ranks* de numeración inferior a superior**. Hemos configurado el *rank* 2 como el canal del sensor de temperatura. Al haber habilitado dos canales, se sobreentiende que el canal

`IN0` será el *rank* 1 y por ello STM32CubeMX no nos pide configurarlo expresamente. El formulario de configuración del ADC quedaría del siguiente modo.

Figura 6: Configuración ADC.

Guardamos la configuración y generamos código.

1.2.3.2. Medición de la temperatura El proceso para convertir la señal analógica de dos canales tal y como lo tenemos configurado no difiere mucho del anterior. Simplemente, **debemos de realizar el proceso anterior dos veces seguidas**: una para el canal `IN0` y otra para el canal del sensor de temperatura. Creamos una **variable llamada `tempADC`** para guardar el resultado de la conversión del canal del sensor de temperatura. La función `main` quedaría del siguiente modo:

```
1  ...
2
3    /* USER CODE BEGIN 2 */
4
5    uint32_t ad2ADC = 0; // no hace falta que sea global, es local
6    uint32_t tempADC = 0; // no hace falta que sea global, es local
7
8    /* USER CODE END 2 */
9
10   /* Infinite loop */
11   /* USER CODE BEGIN WHILE */
12   while (1)
13   {
14     HAL_ADC_Start(&hadc1); // iniciamos la conversion para el divisor
           de tension
```

```
15      HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que finalice
            la conversion
16      ad2ADC = HAL_ADC_GetValue(&hadc1);
17
18      HAL_ADC_Start(&hadc1); // iniciamos la conversion para el sensor de
             temperatura
19      HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que finalice
            la conversion
20      tempADC = HAL_ADC_GetValue(&hadc1);  // leemos el resultado de la
            conversion y lo
21                                           // guardamos en tempADC
22
23      /* USER CODE END WHILE */
24
25      /* USER CODE BEGIN 3 */
26    }
27    (void)ad2ADC; // evitar warning
28    (void)tempADC; // evitar warning
29    /* USER CODE END 3 */
30
31  ...
```

Compilamos, depuramos e iniciamos el programa. Añadimos un ***breakpoint* al primer `HAL_ADC_Start`** y en la ventana `Variables` comprobamos que se actualizan los valores de las variables `ad2ADC` y `tempADC` cada vez que reanudamos/pausamos la ejecución del código. **El valor de la variable `tempADC` no nos dice mucho...** Vamos a calcular la temperatura real en función del valor de `tempADC`.

Cómo calcular la temperatura a partir del valor de `tempADC` nos lo dice el manual de referencia del microcontrolador en la página 226. Allí se nos dice que la fórmula a utilizar es:

$$\text{Temp(Celsius)} = \frac{\text{V}_{\text{señal}} - \text{V}_{25}}{\text{Avg\_Slope}} + 25$$

Figura 7: Fórmula temperatura.

`V25` y `Avg_Slope` son dos constantes que podemos encontrar en el *datasheet* del microcontrolador. `Vseñal` es la conversión de `tempADC` a tensión. Tenemos la fórmula que relaciona el valor digital del ADC con la tensión de la señal de entrada en la introducción de la primera parte de esta práctica. ¿Os atrevéis a montar la expresión que necesitamos para obtener la temperatura a partir de `tempADC`? Es cero complicado y deberíais de hacer el esfuerzo de deducirlo, pero sigamos adelante...

Para calcular la temperatura, primeramente crearemos una variable que acepte decimales. Esa **variable será de tipo `float` y la llamaremos `temp`**. Para implementar las constantes de la fórmula que seguro que habéis intentado deducir, utilizaremos constantes. La función `main`quedaría:

```
...

  /* USER CODE BEGIN 2 */

  uint32_t ad2ADC = 0; // no hace falta que sea global, es local
  uint32_t tempADC = 0; // no hace falta que sea global, es local

    float temp = 0; // variable para la temperatura
  const float sense = 3.3/4095, // factor de conversion de ADC a
     tension
                    v25 = 0.76, // tension de referencia a 25 C
                        avgSlope = 0.0025; // variacion de vsenal con
                           la temp

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    HAL_ADC_Start(&hadc1); // iniciamos la conversion para el divisor
       de tension
    HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que finalice
       la conversion
    ad2ADC = HAL_ADC_GetValue(&hadc1);

    HAL_ADC_Start(&hadc1); // iniciamos la conversion para el sensor de
        temperatura
    HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que finalice
       la conversion
    tempADC = HAL_ADC_GetValue(&hadc1);  // leemos el resultado de la
       conversion y lo
                                         // guardamos en tempADC

    temp = ((tempADC*sense-v25)/avgSlope)+25; // calculo de la
       temperatura

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  (void)ad2ADC; // evitar warning
    (void)temp; // evitar warning
  /* USER CODE END 3 */

...
```

Compilad, depurad e iniciad la ejecución del programa. Con un ***breakpoint* en el primer `HAL_ADC_Start`**, comprobad como ahora tenemos la variable `temp` nos da un valor de tempera-

tura. **Podéis echar el aliento sobre el microcontrolador y soplarle para subir y bajar algo la temperatura, respectivamente.**

Quedaría más espectacular con el `Chart` de CoolTerm… Va. Añadid el siguiente código al `main` para que podáis ver una gráfica de temperatura.

```
...

  /* USER CODE BEGIN 2 */

  uint32_t ad2ADC = 0; // no hace falta que sea global, es local
  uint32_t tempADC = 0; // no hace falta que sea global, es local

    float temp = 0; // variable para la temperatura
  const float sense = 3.3/4095, // factor de conversion de ADC a
      tension
                  v25 = 0.76, // tension de referencia a 25 C
                      avgSlope = 0.0025; // variacion de vsenal con
                         la temp

    char txBuffer[32] = { 0 }; // buffer para transmitir
  uint8_t txBufferSize = 0; // tamano a enviar

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    HAL_ADC_Start(&hadc1); // iniciamos la conversion para el divisor
       de tension
    HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que finalice
       la conversion
    ad2ADC = HAL_ADC_GetValue(&hadc1);

    HAL_ADC_Start(&hadc1); // iniciamos la conversion para el sensor de
        temperatura
    HAL_ADC_PollForConversion(&hadc1, 200); // esperamos que finalice
       la conversion
    tempADC = HAL_ADC_GetValue(&hadc1);  // leemos el resultado de la
       conversion y lo
                                         // guardamos en tempADC

    temp = ((tempADC*sense-v25)/avgSlope)+25; // calculo de la
       temperatura

    txBufferSize = sprintf(txBuffer, "10,30,%.3f\n", temp); //
       codificamos el numero a texto
        HAL_UART_Transmit(&huart2, &txBuffer, txBufferSize, 1000); //
           enviamos el dato

```

```
36      HAL_Delay(20); // una media cada 20 ms, sino el grafico avanza muy
            rapido
37                                  // pero los delays son caca, no hay
                                       que usarlos
38
39      /* USER CODE END WHILE */
40
41      /* USER CODE BEGIN 3 */
42    }
43    (void)ad2ADC; // evitar warning
44    /* USER CODE END 3 */
45
46  ...
```

Nos aparecerá es un error tal como este:

```
1  The float formatting support is not enabled, check your MCU Settings
       from "Project Properties > C/C++ Build > Settings > Tool Settings",
       or add manually "-u _printf_float" in linker flags.
```

Simplemente, **hacemos lo que nos pide**. Vamos a las propiedades del proyecto desde `Project > Properties` o clic derecho sobre el archivo de proyecto `masb-p04` en el `Project Explorer` a la derecha y vamos a `Properties`. En la ventana emergente, desplegamos `C/C++ Build` en el menú lateral y vamos a `Settings`. Allí, en el formulario de la derecha, vamos a la pestaña `Tool Settings` y, teniendo seleccionado `MCU Settings` en el submenú de la izquierda, marcamos la casilla `Use float with printf from newlib-nano (-u _printf_float)`. Una imagen vale más que mil palabras. Os dejo una captura de las propiedades del proyecto.

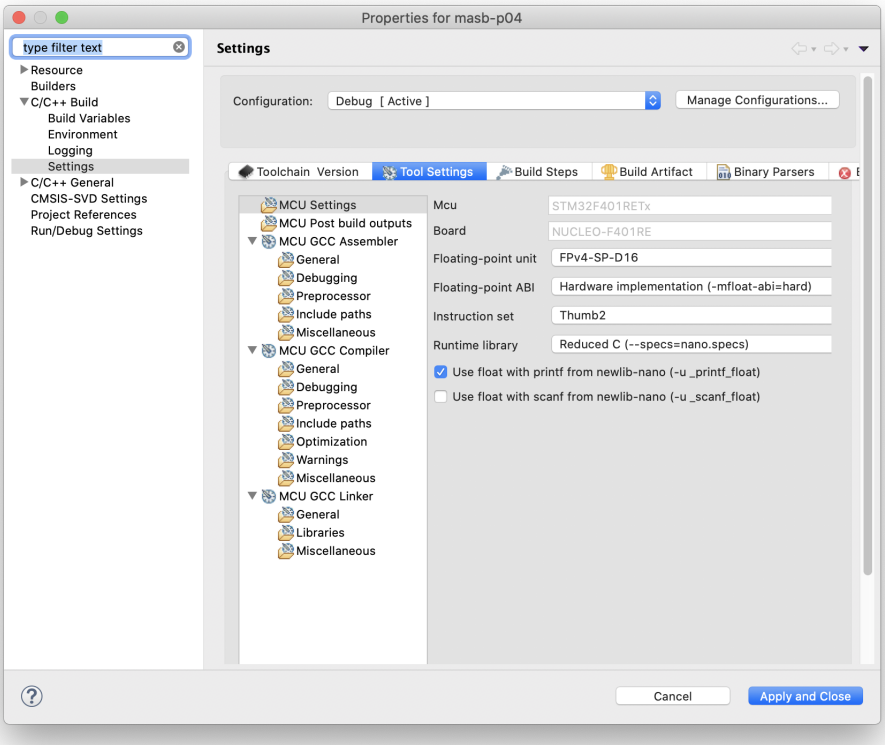

Figura 8: Propiedad proyecto.

Ahora ya **podemos ir al `Chart`** de CoolTerm y ver una gráfica de temperatura.

> No esperéis milagros del sensor de temperatura de dentro el microcontrolador. No es muy preciso y tiene mucho ruido, pero no está pensado para un control fino de la temperatura.

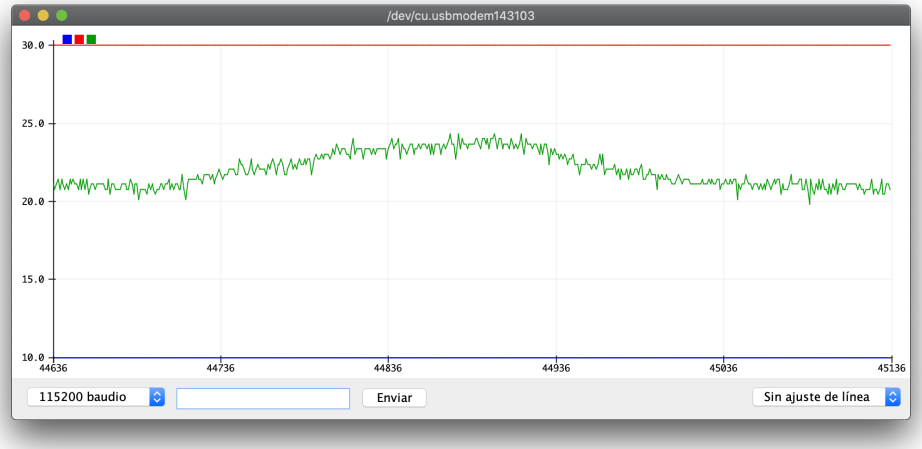

Figura 9: Gráfico de temperatura.

1.2.4. *Direct Memory Access*

Madre mía la de cosas que veréis hoy… Pero esto será rápido. Empecemos por, ¿qué es el **DMA**? No es más que un **circuito que interactúa directamente entre los periféricos y la memoria del microcontrolador para ofrecer un canal de comunicación directo**. Ejemplo rápido: con el DMA, podemos hacer que los resultados del ADC vayan directamente a la memoria sin que tenga que intervenir la CPU.

> A efectos prácticos, **los resultados del ADC se nos guardarán directamente en una variable sin que hagamos nosotros nada**. Útil, ¿verdad?

Además, en esta parte haremos que el *timer* 2 active la conversión en el ADC de manera periódica cada unos 24 µs. O lo que es lo mismo, 42 kSps.

1.2.4.1. Configuración del microcontrolador

Vamos al formulario de configuración del ADC y, para empezar todos desde 0, **inicializamos el formulario pulsando el botón `Reset Configuration`**. En el subformulario de canales, nos **aseguramos que solo el canal `IN0` esté marcado**.

En el apartado `ADC_Regular_ConversionMode`, seleccionamos `Timer 2 Trigger Out event` en el campo `External Trigger Conversion Source`. Ahora, en este mismo subformulario, nos vamos a la pestaña `DMA Settings`. Allí clicamos sobre el botón `Add`. Se nos habrá creado una nueva línea. En la primera columna, seleccionamos `ADC1` en el desplegable. Configuramos, en la parte de abajo, `Circular` en el campo `Mode` y `Half World` en los campos `Data Width`. Volvemos a la pestaña `Parameters Settings` y allí habilitamos el `DMA Continous Requests`.

Las configuraciones del ADC y el DMA quedarían del siguiente modo:

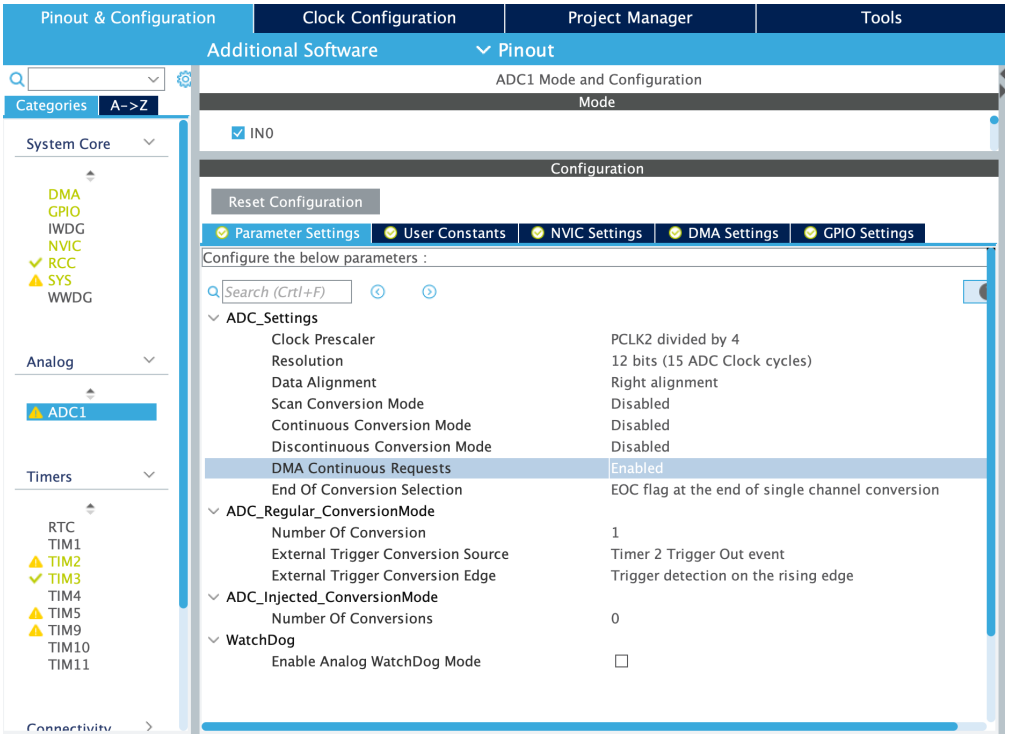

Figura 10: Configuración del ADC.

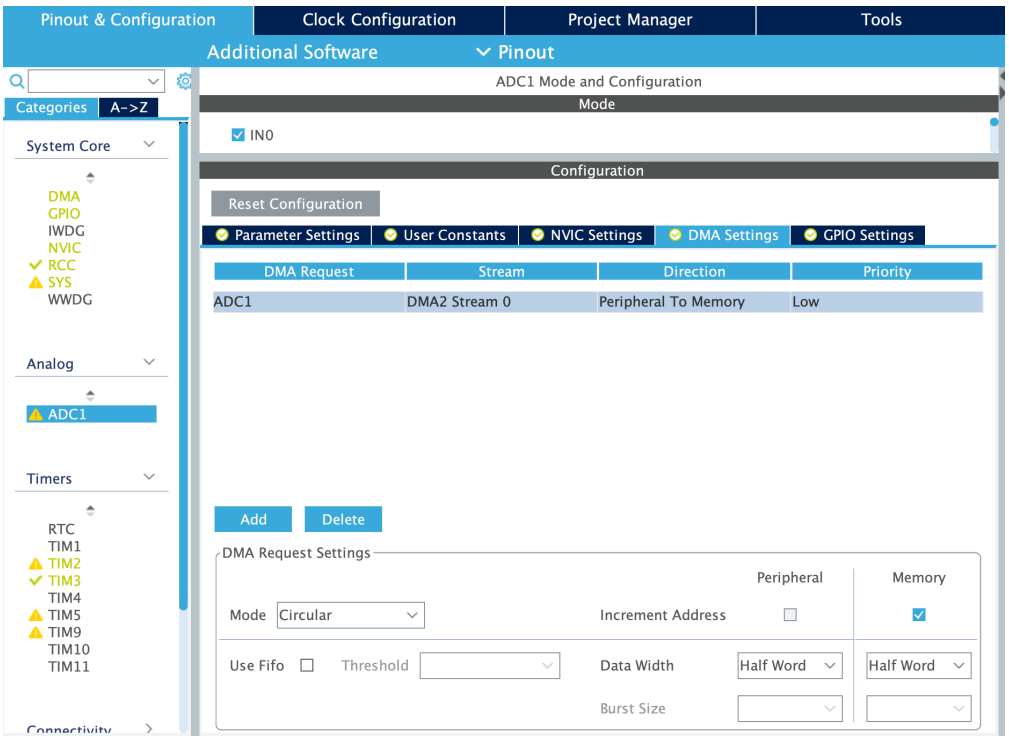

Figura 11: Configuración del DMA.

Ahora vamos a **configuración el *timer* 2**. Nos vamos a su pestaña de configuración. Seleccionamos `Internal Clock` en el campo `Clock Source`. Seguidamente, establecemos un periodo de 2,000 ciclos de reloj en el campo `Counter Period`. Por último, seleccionamos `Update Event` en el campo `Trigger Event Selection`. Está sería la configuración:

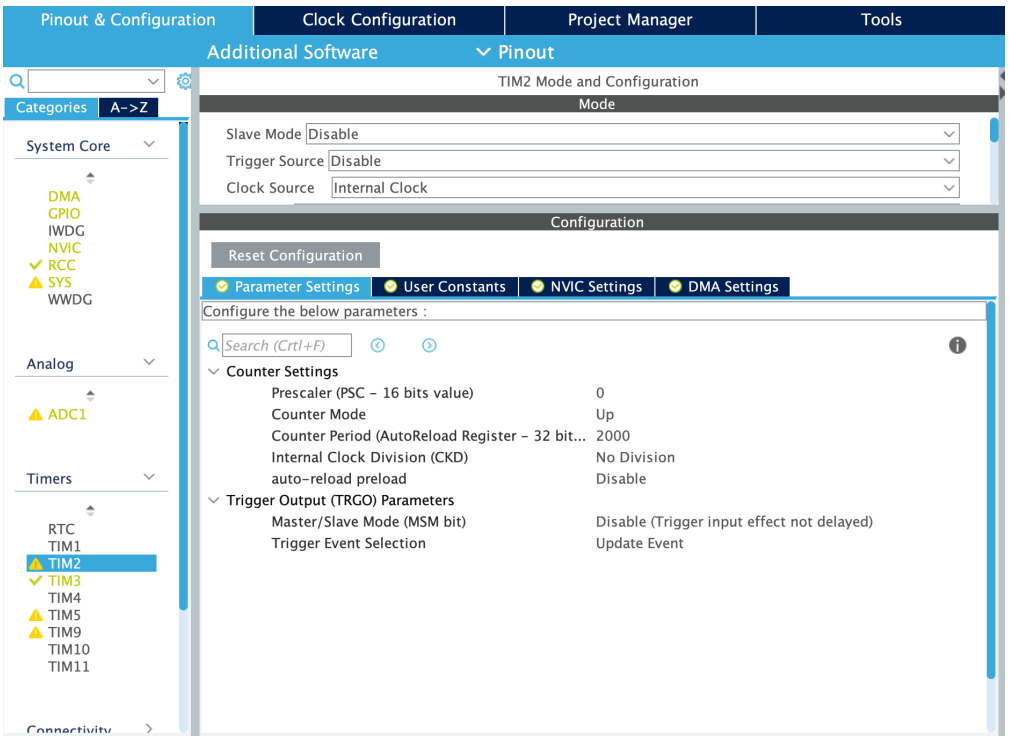

Figura 12: Configuración del Timer.

Ya hemos terminado la configuración. Guardamos el archivo y generamos código.

1.2.4.2. Conversiones periódicas con DMA Vamos a hacer que *mágicamente* nuestro ADC realice una conversión cada 24 µs y se guarde el resultado en la variable `ad2ADC`. Primeramente, nos vamos a nuestro archivo `main.c`. **Vamos a crear nuestra variable `ad2ADC`, pero esta vez de tipo `uint16_t`.** Esta vez es de 16 bits y no de 32 porque en el DMA hemos configurado que mueva valores del ADC a la memoria de con un tamaño de *half-word* (16 bits). Ahora, simplemente, **arrancamos el DMA** para estar preparado cuando empiecen las conversiones del ADC, e **iniciamos el *timer* 2** para que ejecute una conversión periódicamente. Para lo primero, utilizamos la función `HAL_ADC_Start_DMA`. Para lo segundo, ya la sabéis, `HAL_TIM_Base_Start`. El código quedaría del siguiente modo:

```
...

  /* USER CODE BEGIN 2 */

    uint16_t ad2ADC = 0; // no hace falta que sea global, es local

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
```

```
11
12      HAL_ADC_Start_DMA(&hadc1, (uint32_t*)&ad2ADC, 1); // iniciamos DMA
13      HAL_TIM_Base_Start(&htim2); // iniciamos timer --> conversiones
14      while (1)
15      {
16      /* USER CODE END WHILE */
17
18      /* USER CODE BEGIN 3 */
19      }
20    /* USER CODE END 3 */
21
22  ...
```

Compilamos y no nos debería generar ningún error. Vamos a añadir el código que falta para que se envíe por comunicación serie al ordenador.

```
1   ...
2
3     /* USER CODE BEGIN 2 */
4
5       uint16_t ad2ADC = 0; // no hace falta que sea global, es local
6
7       char txBuffer[32] = { 0 }; // buffer para transmitir
8       uint8_t txBufferSize = 0; // tamano a enviar
9
10    /* USER CODE END 2 */
11
12    /* Infinite loop */
13    /* USER CODE BEGIN WHILE */
14
15      HAL_ADC_Start_DMA(&hadc1, (uint32_t*)&ad2ADC, 1); // iniciamos DMA
16      HAL_TIM_Base_Start(&htim2); // iniciamos timer --> conversiones
17      while (1)
18      {
19
20        txBufferSize = sprintf(txBuffer, "0,4095,%lu\n", ad2ADC); //
              codificamos el numero a texto
21        HAL_UART_Transmit(&huart2, &txBuffer, txBufferSize, 1000); //
              enviamos el dato
22
23      /* USER CODE END WHILE */
24
25      /* USER CODE BEGIN 3 */
26      }
27    /* USER CODE END 3 */
28
29  ...
```

Compilamos, depuramos e iniciamos la ejecución del programa. Vamos al `Chart` de CoolTerm y vamos como se envía el valor del ADC correctamente. Todo ello sin tener que hacer en ningún momento una

conversión a mano o guardar el resultado en ninguna variable. Se hace todo automáticamente.

1.3. Reto

Beber dos litros de agua al día. Eso sí es un reto. Para esta práctica no tenéis reto con el microcontrolador. Podéis tomaros un respiro.

1.4. Evaluación

1.4.1. Entregables

Estos son los elementos que deberán de estar disponibles para el profesorado de cara a vuestra evaluación.

- ☐ **Commits**

Se os deja a vuestro criterio cuándo realizar los *commits*. No hay número mínimo/máximo y se evaluará el uso adecuado del control de versiones (creación de ramas, *commits* en el momento adecuado, mensajes descriptivos de los cambios, …).

- ☐ ~~**Reto**~~
- ☐ **Informe**

Informe con el mismo formato/indicaciones que en prácticas anteriores. Guardad el informe con el nombre `REPORT.md` en la misma carpeta que este documento. Guardadlo en vuestra rama y no en la del reto.

El informe debe de contener:

- ☐ **Especificaciones del ADC:** Haremos un mini-ejercicio de búsqueda de información en la documentación. Simplemente indicad las especificaciones básicas del ADC que aparecen en el *datasheet* del microcontrolador de nuestra EVB. Esas especificaciones deben ser:
  - ☐ Alimentación mínima y máxima del ADC.
  - ☐ Tensión de referencia mínima y máxima del ADC.
  - ☐ Tiempo de conversión para una fADC de 30 MHz y una resolución de 12 bits.
- ☐ **Diferencia con Arduino:** En la parte anterior también iniciábamos conversiones del ADC con el *timer* en Arduino. ¿Qué diferencia hay entre cómo lo hicimos en la primera parte y cómo lo hemos hecho en esta?

1.4.2. Pull Request

Finalizados todos los entregables, acordaos de hacer el *push* pertinente y cread un *Pull Request* (PR) de vuestra rama a la master. Acordaos de ponerme como *Reviewer*.

1.4.3. Rúbrica

La rúbrica que utilizaremos para la evaluación la podéis encontrar en el CampusVirtual. Os recomendamos que le echéis un vistazo para que sepáis exactamente qué se evaluará y qué se os pide.

Se empezará a evaluar la limpieza y legibilidad del código (sentencias alineadas, sentencias anidadas con una tabulación delante, claves alineadas, comentarios, etc.).

1.5. Conclusiones

Madre mía todo lo que hemos visto del ADC. En esta práctica hemos visto **cómo operar de manera sencilla/rudimentaria el ADC con funciones bloqueantes** y activando la conversión y la lectura del resultado por *software*. También hemos visto una breve, breve, **breve pincelada de comunicación serie** para enviar los resultados de la conversión al ordenador. Hemos hecho la **conversión de más de un canal** y, finalmente, hemos hecho que la **conversión se haga automáticamente de manera periódica** y que **el resultado se guarde en memoria también automáticamente**. Esto lo hacemos activando la conversión con un *timer* y moviendo los datos del periférico a la memoria con el DMA.

En la siguiente práctica… **¡Comunicación serie!**