MASB | Arduino ADCs

#stm32 #c #biomedical #microcontroladores #programacion

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

25 de febrero de 2020

Índice

1. ADCs

ADC o *Analog-to-Digital Converter* es un periférico fundamental en circuitos AMS (*Analog Mixed Signal*) o circuitos de señal mixta. Con el **ADC** seremos capaces de **convertir una señal de tensión continua a una señal digital** realizando una **discretización** y una **cuantificación** de la **señal continua**.

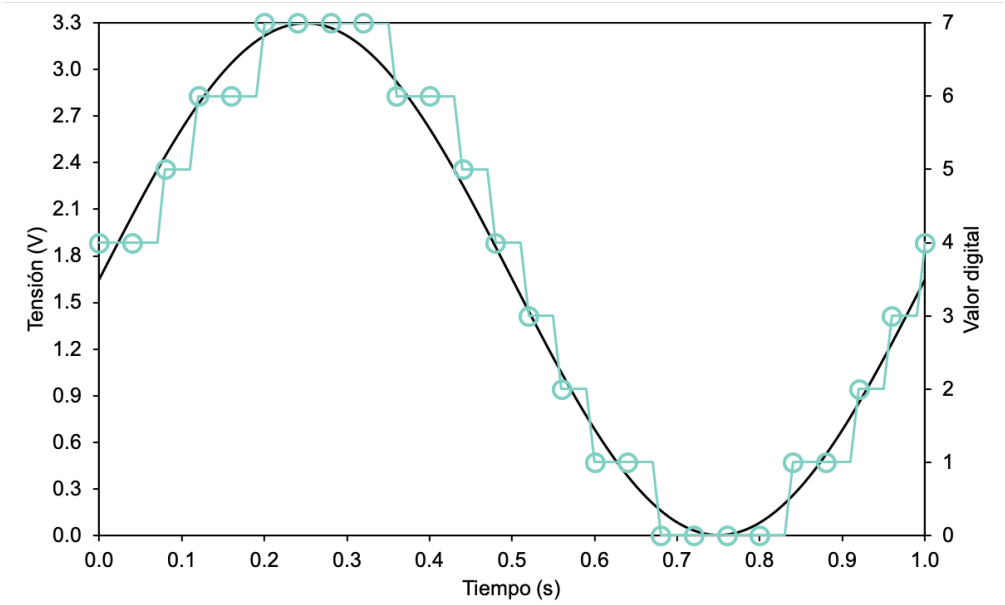

Figura 1: Conversión ADC.

> En la imagen: discretización y cuantificación de una señal de tensión analógica. Características de la señal analógica: amplitud 3.3 Vp-p, frecuencia de 1 Hz y *offset* de 1.65 V. Características de la conversión: resolución de 3 bits, tensión de referencia de 3.3 V y *samplig rate* de 25 Sps.

Hacemos un breve repaso de ADCs recordando qué son la tensión de referencia, los bits de resolución y el *sampling rate*:

- **Bits de resolución:** es el número de bits de los que dispone el ADC para generar un valor digital a partir de un valor analógico de una señal. Por ejemplo, si un ADC tiene 10 bits, el ADC podrá generar hasta 210 valores (de 0 a 1023 en base decimal). Otro ejemplo, el de la imagen, si un ADC tiene 3 bits, podrá generar hasta 23 valores (de 0 a 7 en base decimal). Junto a la tensión de referencia, el número de bits de un ADC establece la resolución del ADC.
- **Tensión de referencia:** valor de tensión fijo de referencia (Vref) utilizado para compararlo con la señal analógica de entrada (Vseñal) que está siendo cuantificada. Por ejemplo, suponiendo un ADC de 10 bits, si la señal de entrada es igual a la tensión de referencia, el ADC dará como valor

digital 1023, el más alto. Si la señal de entrada es un tercio de la tensión de referencia, el ADC dará 341 (un tercio de 1023). La fórmula de conversión sería:

$$\mathrm{ADC}_{\mathrm{valor}} = \frac{\mathrm{V}_{\mathrm{señal}}}{\mathrm{V}_{\mathrm{ref}}}(2^{\mathrm{bits}} - 1)$$

Figura 2: Fórmula conversión ADC.

- ***Sampling rate*:** número de muestras por segundo que tomamos de la señal analógica. Puede darse en hercios o Sps (*Samples per second*). Aquí debemos de acordarnos de aplicar el criterio de Nyquist.

Hala. Resumidos los ADCs en cuatro párrafos y medio. Ahora en serio, los ADCs es todo un mundo y aquí veremos solo cómo operar con ellos con el microcontrolador. Con las pinceladas que hemos dado será suficiente.

Antes de entrar en materia, veremos que lo que ofrece **Arduino para trabajar con ADCs es muy básico**. Se podrían hacer cosas más avanzadas y retorcer un poco Arduino, pero el nivel de complejidad para hacerlo haría que hacerlo a nivel de registros directamente fuera más eficiente. Por ello, **en esta práctica incorporaremos el trabajo por primera vez** en la asignatura. Como reto haremos un programa/proyecto común entre tu compañero y tú. *Let's go!*

1.1. Objetivos

- Introducción a los ADCs en Arduino.
- Combinación de *timers* y ADC en Arduino para *sampling rates* fijos.
- Introducción a los *workflows* basados en Git para el trabajo en equipo.

1.2. Procedimiento

1.2.1. Conversión con una AD2

Vamos a ver cómo leer una señal analógica procedente de una Analog Discovery 2. Para hacerlo más interesante, haremos la conversión de una señal que emula un ECG.

1.2.1.1. Conexión de la AD2 ¿Te gusta tu ordenador? ¿Te gustaría que durara muchos años? Pues **asegurate de seguir bien estos pasos** ya que conectaremos la Analog Discovery 2 a la EVB que a su vez está conectada al puerto USB de tu ordenador. Si haces un cortocircuito…Ahí lo dejo.

> En realidad, **la EVB tiene un sistema de seguridad** que protege el puerto USB de posibles cortocircuitos. Pero lo deseable y recomendado es **no tentar a la suerte**.

Lo que haremos será conectar **uno de los terminales `Ground` de la AD2 a un `GND` de la EVB** y el **terminal `W1` de la AD2 al pin `A0` de la EVB**.

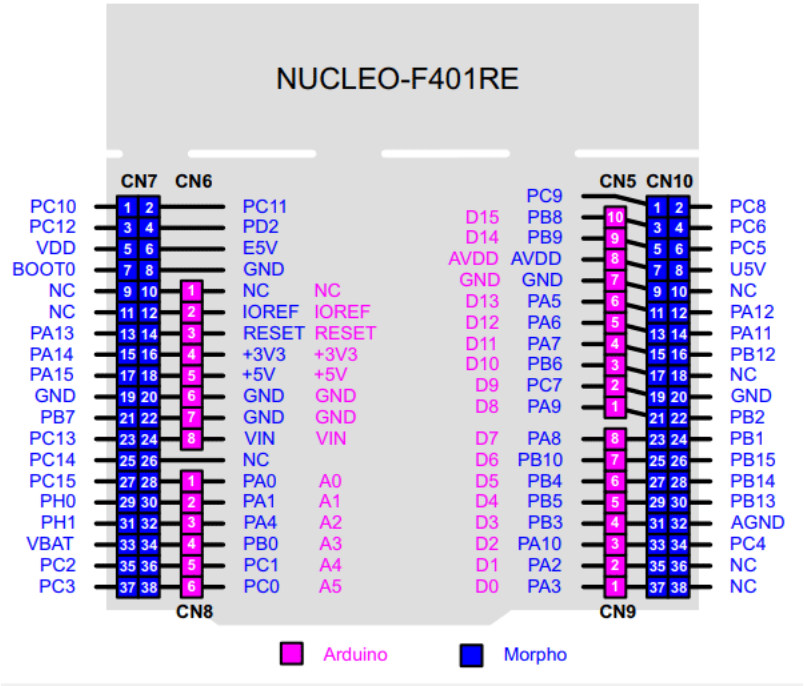

Figura 3: Pinout de la EVB.

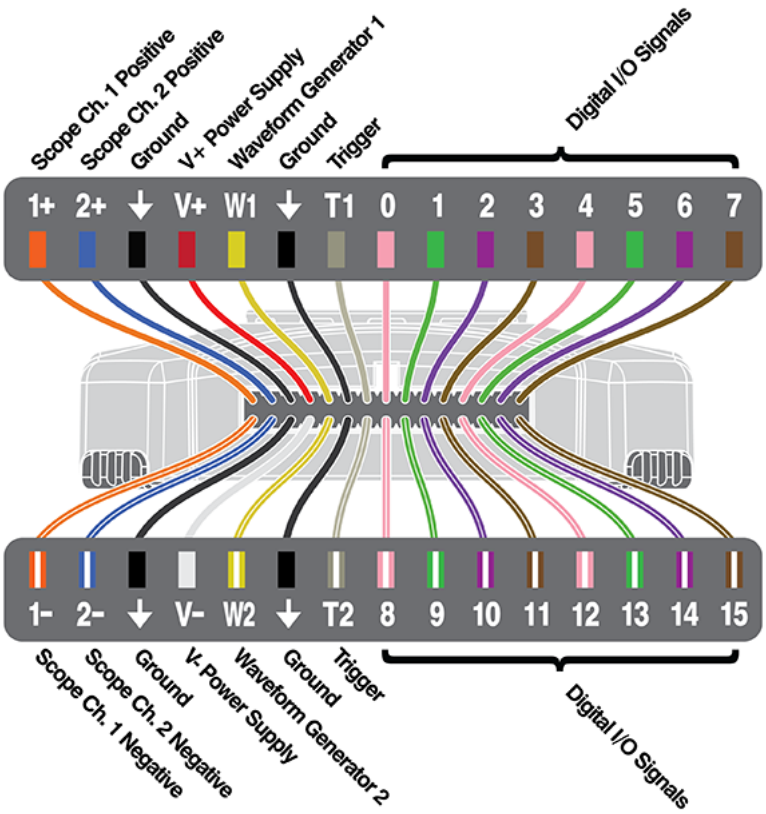

Figura 4: Pinout de la AD2.

Abrimos el programa **Waveforms** para controlar la AD2, clicamos en “Wavegen” a la izquierda y seguidamente clicamos al icono de engranaje que hay al lado de “Type”. Seleccionamos “Import” e importamos el siguiente archivo con los datos de un ECG extraídos de:

> Moody GB, Mark RG. The impact of the MIT-BIH Arrhythmia Database. IEEE Eng in Med and Biol 20(3):45-50 (May-June 2001). (PMID: 11446209); DOI:10.13026/C2F305

Finalmente, configuramos la siguiente ventana tal y como aparece en la siguiente imagen:

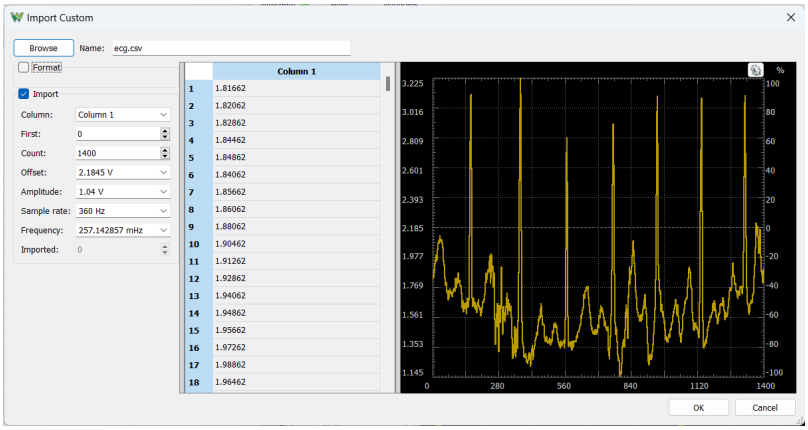

Figura 5: Configuración de la AD2 para la generación de un ECG.

Clicamos Ok y una vez iniciemos el generador tendremos un ECG conectado a nuestro microcontrolador.

> Importante destacar que el ECG no correspondería directamente a la señal electrica obtenida de los electrodos, sino que sería la señal obtenida previa amplificación de la misma.

1.2.1.2. Lectura de una señal analógica

Para **leer una señal analógica en Arduino** solo hay que utilizar la **función `analogRead`** indicando el pin donde se ha conectado la señal a leer. Ya está. Nada más. No hay que configurar el pin ni nada. Ya podemos recoger e irnos a casa.

¡No, no, no! Es broma. Vamos a trastear un poco ya que es tan fácil operar con los ADCs en Arduino.

Vamos a crearnos un *sketch* llamado `masb-p04` en la carpeta `arduino` de nuestro repositorio local. En el programa, **crearemos una variable numérica tipo `int`** llamada adcVal para guardar el valor resultante de la conversión del ADC. El programa quedaría del siguiente modo:

```
1  int adcVal = 0; // variable para almacenar la conversion
2
3  void setup() {
4    // put your setup code here, to run once:
5
6  }
7
8  void loop() {
9    // put your main code here, to run repeatedly:
10
11   adcVal = analogRead(A0); // leemos el valor de la conversion
12 }
```

Compilamos y subimos el programa. ¿Funciona? Ni idea. **No tenemos manera de ver si la conversión está teniendo lugar o no**, y **no tenemos disponibles herramientas de depuración, como los *break-***

***points*, que sí tenemos en STM32CubeIDE**. Por ello, vamos a dar el paso a introducir la comunicación serie. Pero nada, solo la puntita del pie.

1.2.1.3. Comunicación serie

Lo veremos en detalle en prácticas posteriores, pero básicamente lo que haremos será **establecer una comunicación serie ente nuestro ordenador y el microcontrolador** y hacer que este último envíe los valores de la conversión del ADC para ver que todo funciona.

Hasta que lo veamos en prácticas anteriores, simplemente seguid estos pasos.

1. Inicializamos la comunicación serie en la función `setup` con la instrucción `Serial.begin(9600)`, donde 9600, como veremos en otra práctica, es el ***baudrate***.
2. Utilizamos la función `Serial.println` para **enviar un texto/número** y después **un retorno de carro** que haga que en el ordenador nos aparezca los números que vamos enviando en líneas distintas.

El código quedaría del siguiente modo:

```
1  int adcVal = 0; // variable para almacenar la conversion
2
3  void setup() {
4    // put your setup code here, to run once:
5    Serial.begin(9600); // configuramos la comunicacion serie
6  }
7
8  void loop() {
9    // put your main code here, to run repeatedly:
10
11   adcVal = analogRead(A0); // leemos el valor de la conversion
12   Serial.println(adcVal); // enviamos el valor de la conversion al PC
13 }
```

Compilamos, subimos y… Seguimos sin ver nada.

Nos falta **abrir el terminal serie** donde poder ver los datos que nos está enviando el microcontrolador. Para ello, primeramente **configuramos el puerto COM** donde nos está enviando información el microcontrolador. Vamos a `Herramientas > Puerto` y seleccionamos `COMX`, donde `Xs` será un número. Si te aparece más de un puerto `COM`, deberás de ir probando hasta ver cuál es el de nuestra placa. (Se puede ver cuál es nuestro puerto `COM` desde el `Administrador de dispositivos` de Windows). Luego, vamos a `Herramientas > Monitor Serie` y se nos abrirá un terminal donde irán apareciendo los valores generados por el ADC y que el microcontrolador está enviando al ordenador.

> Para los usuarios de los productos de Cupertino, en lugar de `COMX`, los puertos os aparecerán del estilo `/dev/cu.usbmodem143103`. Puede variar un poco, tipo la numeración, por ejemplo.

Figura 6: Monitor serie.

El terminal es extremadamente útil, pero para ver un valor que se actualiza de manera continua… puede ser poco práctico. Podemos utilizar otro programa llamado CoolTerm que nos permite graficar los datos enviados. Para configurarlos, abrimos la aplicación y lo primero que haremos es clicar al **icono** `Options`. En esta ventana podemos configurar múltiples parámetros de la comunicación y el terminal, pero solo configuraremos una cosa de momento: el puerto. **Seleccionamos el puerto** pertinente de nuestra EVB. Si no os aparece vuestra EVB, clicad en `Re-Scan Serial Ports`.

> Acordaos que en Windows tomará el valor `COMX` y en Mac/Linux `usbmodemXXXX`

Una vez seleccionado, dejamos la configuración por defecto.

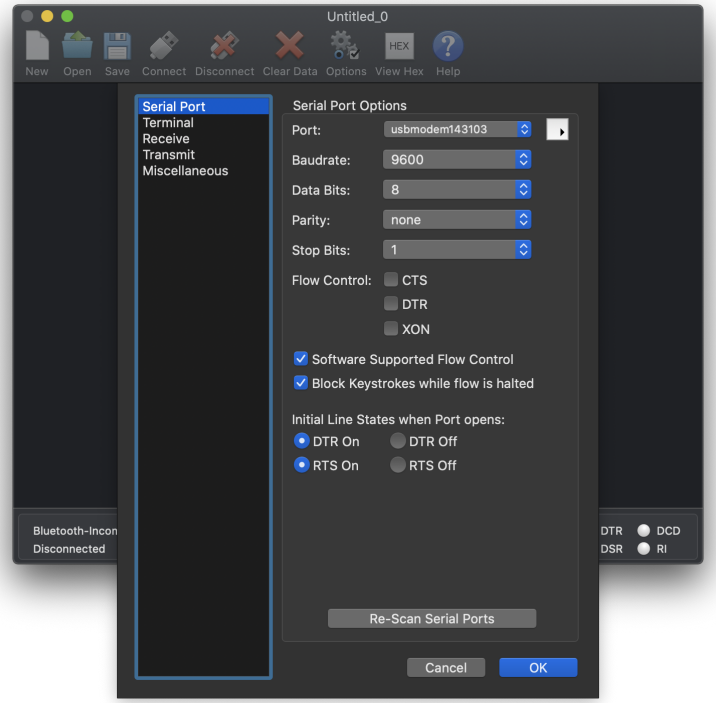

Figura 7: Configuración CoolTerm.

Podemos clicar en el icono `Save` para guardar la configuración. Con todo configurado, **clicamos en el botón `Connect`**. Deberíamos de ver como van apareciendo los datos en pantalla, pero si clicamos en `View` > `Chart`...

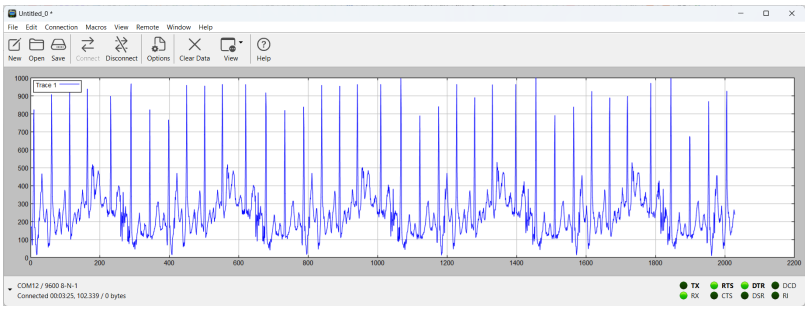

Figura 8: Graficador del puerto serie.

Tachán! Aquí está nuestro ECG. Como podemos ver, **el eje de las ordenadas (Y) se autoescala** a los valores mostrados en el gráfico. Para **forzar al gráfico con un mínimo y un máximo** en el eje de ordenadas, lo que haremos será **enviar tres valores por serie**: el **mínimo** (0), el **valor del ADC** y el **máximo** (1023). Los números, para que los sepa interpretar correctamente el *Serial Plotter*, deben de

estar separados por una coma. Lo haríamos del siguiente modo:

> Aprovechamos para añadir un `Delay` en el programa y así permitir que el ECG se vea mejor.

```
int adcVal = 0; // variable para almacenar la conversion

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600); // configuramos la comunicacion serie
}

void loop() {
  // put your main code here, to run repeatedly:
  delay(10);
  adcVal = analogRead(A0); // leemos el valor de la conversion

  Serial.print(0); // enviamos el minimo
  Serial.print(","); // coma para separar los numeros
  Serial.print(adcVal); // enviamos el valor de la conversion al PC
  Serial.print(","); // coma para separar los numeros
  Serial.println(1023); // enviamos el maximo con retorno de carro
}
```

Cuidado, que ahora **hemos utilizado otra función para enviar el valor del ADC**. No utilizamos `println` (con retorno de carro), sino `print`. `println` la utilizamos para enviar el último valor enviado: el máximo.

Compilamos y subimos. Ahora sí, se nos están enviando tres valores/señales: dos señales constantes para establecer la escala del eje de ordenadas y una señal que es el valor del ADC. Como veis, ha sido rápido. Vamos a ver el reto.

1.3. Reto: Control del *sampling rate*

En las aplicaciones donde se captura/convierte una señal analógica, **es muy importante establecer un *sampling rate* conocido y que sea estable**. Tal y como hemos hecho el programa anterior, esto no ocurre. Si por lo que fuera pusiéramos más código en la función `loop` (que sería lo más normal), la conversión tendría lugar al llegar a la instrucción `analogRead`, pero antes se habrá ejecutado todo el código que pueda haber antes o después en la función `loop`. Esto hace que no tengamos un *sampling rate* de nuestro gusto puesto que no podemos establecer cuánto ha de tardar el microprocesador en ejecutar el resto de instrucciones.

> Por ejemplo, en el caso anterior no podemos decir que queremos un *sampling rate* de 1 Sps.

> (Podríamos hacer la trampa del `delay(10)`, pero es eso: una trampa que nos garantiza que como mínimo habrán 10 ms entre muestra y muestra, pero no que haya un mayor tiempo).

Tampoco podemos garantizar que el *sampling rate* sea constante ya que si hay alguna sentencia condicional (**`if-else`**) o algún bucle con iteraciones variables, el tiempo de ejecución del código de la función `loop` será diferente para cada ejecución. Tenemos que lograr que la instrucción `analogRead` se ejecute con una periodicidad constante de nuestro gusto. ¿Se os ocurre algo? Sí. Los *timers*.

Ahora veremos un ejemplo de **proyecto** con una **conversión ADC** a un ***sampling rate* constante de 10 Sps**. Además, **activaremos y desactivaremos la conversión a nuestra voluntad con el pulsador**.

Para hacer las pruebas, generad con la AD2 un seno con una amplitud de 1.25 V, un offset de 1.5 V y una frecuencia de 1 Hz. Nunca, nunca, nunca, nunca, nunca, nunca, nunca, nunca, nunca, NUNCA apliquéis una tensión negativa a una entrada de un microcontrolador.

1.3.0.1. *Workflow* básico de trabajo en equipo

Este reto lo haremos por parejas. Por primera vez, trabajaremos en un **proyecto de Arduino utilizando más de un archivo**. **Cada miembro trabajará en archivos distintos**. A continuación, os dejo la estructura de ficheros que tendrá la carpeta del proyecto.

```
arduino
 - masb-p04
    - adc.ino
    - gpio.ino
    - masb-p04.ino
    - timer.ino
```

Arduino IDE toma el archivo `.ino` con el mismo nombre que la carpeta como el archivo principal (sería como el `main.c` de STM32CubeIDE). Por lo que el fichero `masb-p04.ino` será el archivo principal. El resto de archivos son, a efectos prácticos, una extensión de este archivo principal. En lugar de tenerlo todo junto en un solo archivo, **podemos separarlo en múltiples archivos** facilitando la **legibilidad** del código y el **trabajo en equipo**.

Las **tareas están repartidas** entre el miembro `A`del equipo y el miembro `B`del equipo.

> Ahora es cuando os tiráis corriendo hacia al teléfono a preguntar a vuestro compañero si quiere ser/hacer `A` o `B`.

El miembro `A` deberá de ocuparse de los ficheros:

- `masb-p04.ino`
- `adc.ino`

El miembro `B` deberá de ocuparse de los ficheros:

- `gpio.ino`
- `timer.ino`

Ahora separaremos el flujo de trabajo de los dos miembros en dos apartados distintos. Tu **ocupate de realizar tu trabajo**, pero **es recomendable que te leas también el apartado de tu compañero** para saber qué ha hecho.

1.3.0.2. Flujo de trabajo del miembro "A"

1.3.0.2.1. Rama de desarrollo Lo primero que vamos a hacer es **ir a la rama `master`** e **importar** todos los posibles **cambios**/actualizaciones que hayan podido haber en el **repositorio remoto**.

```
git checkout master
git pull
```

Seguidamente, nos **creamos nuestra rama de desarrollo**. La llamaremos un pelín diferente a lo habitual. **Añadiremos la *keyword* `reto` en el nombre**.

```
git checkout -b develop/A-reto-<tu-nombre-sin-espacios-ni-caracteres-
    raros>
```

1.3.0.2.2. Desarrollo de masb-p04.ino Vamos a Arduino IDE y guardamos un ***sketch* como `masb-p04`** en la **carpeta `arduino`** de nuestro repositorio local. En este *sketch*, aparte de inicializar la comunicación serie, llamaremos a dos funciones que **inicializarán el GPIO y el *timer***. "¡Pero si eso lo hace el miembro `B`!" El hará la implementación, nosotros solo llamaremos a las funciones que él tiene que hacer. Esas funciones para inicializar el GPIO y el *timer* les llamaremos `GPIO_Init` y `TIMER_Init`, respectivamente.

> En un entorno profesional, las funciones se llamarían así porque lo habéis acordado tu compañero y tú o porque directamente vuestro superior o el arquitecto de software así os lo ha dicho. La cosa es que los dos miembros tenéis que utilizar la misma nomenclatura.

El código quedaría del siguiente modo:

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600); // configuramos la comunicacion serie

  TIMER_Init(); // inicializamos el timer
  GPIO_Init(); // inicializamos el gpio
}

```

```
 9 void loop() {
10   // put your main code here, to run repeatedly:
11
12 }
```

Ya hemos acabado el desarrollo planificado para el archivo `masb-p04.ino`. Solo inicializaremos los periféricos indicados y no haremos nada en la función `loop` ya que lo haremos todo con interrupciones.

Aquí sería un buen momento para **hacer un *commit* y *push***.

1.3.0.2.3. Desarrollo de adc.ino Teniendo abierto el archivo `masb-p04.ino`, para crear este archivo hacemos clic en la **flecha a la derecha** del todo que está **alineada con el nombre del archivo** y seleccionamos `Nueva Pestaña`.

Figura 9: Nuevo archivo de proyecto.

Nos pedirá un nombre. Ponemos `adc`. Con esto, se nos habrá creado un fichero `adc.ino` en la carpeta `masb-p04`. En este fichero **crearemos una función llamada `conversionADC`** que será la que llame nuestro compañero desde el *timer*.

> Nuestro compañero sabrá que la función se llama `conversionADC` porque así lo hemos acordado o nos lo ha indicado nuestro superior o el arquitecto de *software*.

En el archivo empezaremos creando una **macro** para hacer referencia al pin que utilizaremos para el ADC. Seguidamente, **crearemos dos variables**: `valADC` y `onADC`. La primera nos servirá para almacenar el **valor resultante de la conversión** del ADC y la segunda la utilizaremos para configurar el ADC como **apagado o encendido**. El código quedaría del siguiente modo:

```
#define ANALOG_PIN    A0

int valADC = 0; // variable para almacenar el valor de la conversion
bool onADC = false; // variable que nos dice si el ADC esta apagado o
    encendido

// realizamos una conversion si esta encendido el ADC y lo enviamos por
     puerto serie
void conversionADC(void) {

}
```

Nos falta implementar la función `conversionADC`. En esta función miraremos si el ADC está encendido o no y, si lo está, realizamos una conversión ADC y lo mostramos por el *Serial Plotter*. Nuestro compañero controlará el valor de `onADC` desde la ISR del pulsador. El código final quedaría del siguiente modo:

```
#define ANALOG_PIN    A0

int valADC = 0; // variable para almacenar el valor de la conversion
bool onADC = false; // variable que nos dice si el ADC esta apagado o
    encendido

// realizamos una conversion si esta encendido el ADC y lo enviamos por
     puerto serie
void conversionADC(void) {

  if (onADC) { // si esta encendido

    valADC = analogRead(ANALOG_PIN); // realizamos una conversion

    Serial.print(0); // enviamos el minimo
    Serial.print(","); // espacio para separar los numeros
    Serial.print(valADC); // enviamos un elemento del bufferADC
    Serial.print(","); // espacio para separar los numeros
    Serial.println(1023); // enviamos el maximo con retorno de carro
  }
}
```

Guardamos y hacemos un ***commit***. También haced un ***push*** para subir los cambios al servidor.

1.3.0.2.4. *Pull Request* y *Review* Nuestro trabajo ya habría acabado. O casi. **Nos faltan dos cosas**. Primero de todo, **hacer un *Pull Request* de esta rama hacia la *master* y poner, importante, como *reviewer* a nuestro compañero `B`**. Él revisará nuestro código y, si lo ve `OK`, tú o él podéis realizar el `merge` hacia la rama máster. Si no os da el `OK`, deberéis de atender las peticiones de cambio que os pide, hacer las modificaciones pertinentes, subir los cambios y realizar una segunda petición de revisión o *re-request*.

Aquí tampoco acaba tu trabajo, porque ahora **deberás de ir al *Pull Request* de tu compañero**, **revisar que todo está `OK`** y, si es así, aprobarle el *Pull Request*. Si no estás conforme con su desarrollo, házselo saber y proponle cambios. Una vez los aplique, aprueba el *Pull Request* y tú o él haced el `merge` a la rama `master`.

Os dejo un enlace a la documentación de ayuda de GitHub donde podéis ver en detalle todo lo que podéis hacer durante la revisión de los *Pull Requests*.

> En este proyecto, es indiferente quien hace primero el *merge* a la rama `master`.

1.3.0.3. Flujo de trabajo del miembro "B"

1.3.0.3.1. Rama de desarrollo Lo primero que vamos a hacer es **ir a la rama `master`** e **importar** todos los posibles **cambios**/actualizaciones que hayan podido haber en el **repositorio remoto**.

```
git checkout master
git pull
```

Seguidamente, nos **creamos nuestra rama de desarrollo**. La llamaremos un pelín diferente a lo habitual. **Añadiremos la *keyword* `reto` en el nombre**.

```
git checkout -b develop/A-reto-<tu-nombre-sin-espacios-ni-caracteres-
    raros>
```

1.3.0.3.2. Desarrollo de gpio.ino Lo primero que vamos a hacer es crear un ***sketch* nuevo** en Arduino IDE y lo **guardamos como `gpio` en la carpeta `arduino`** de nuestro repositorio local. En este archivo **vamos a crear dos funciones**: `GPIO_Init` y `toogleStateADC`. En la primera función **inicializaremos el GPIO** (modo `INPUT` y configuración de una ISR) y la segunda función será la **ISR que se ejecute al pulsar el pulsador**. La primera función, `GPIO_Init`, la llamará vuestro compañero desde la función `setup` del archivo `masb-p04.ino`.

> Tu compañero sabría el nombre de tus funciones para llamarlas ya que, en un entorno profesional, así lo habéis acordado tu compañero y tú o porque directamente vuestro superior o el arquitecto de software así os lo ha dicho. La cosa es que los dos miembros del equipo tenéis que utilizar la misma nomenclatura.

También crearíamos una **macro** para facilitar la lectura del código. Este quedaría así:

```
#define PUSH  23

// configuracion del gpio
void GPIO_Init(void) {
```

```
5
6  }
7
8  void toogleStateADC(void) {
9    // ISR del PULSADOR
10
11 }
```

En `GPIO_Init` configuramos el GPIO y su ISR.

```
1  #define PUSH  23
2
3  // configuracion del gpio
4  void GPIO_Init(void) {
5
6    pinMode(PUSH, INPUT); // configuramos el PUSH como GPIO de entrada
7    // anadimos una interrupcion al pin del PUSH para una transicion de
        HIGH to LOW (falling)
8    attachInterrupt(digitalPinToInterrupt(PUSH), toogleStateADC, FALLING)
        ;
9
10 }
11
12 void toogleStateADC(void) {
13   // ISR del PULSADOR
14
15 }
```

En la ISR, conmutamos el valor de la variable `onADC`. `onADC` es una variable que ha declarado nuestro compañero en el archivo `adc.ino` para poder configurar si el ADC está apagado o encendido. Tú sabes el nombre de esta variable porque así has quedado con él que la llamaríais o porque vuestro superior así os lo ha mandado. El código quedaría del siguiente modo:

```
1  #define PUSH  23
2
3  // configuracion del gpio
4  void GPIO_Init(void) {
5
6    pinMode(PUSH, INPUT); // configuramos el PUSH como GPIO de entrada
7    // anadimos una interrupcion al pin del PUSH para una transicion de
        HIGH to LOW (falling)
8    attachInterrupt(digitalPinToInterrupt(PUSH), toogleStateADC, FALLING)
        ;
9
10 }
11
12 void toogleStateADC(void) {
13   // ISR del PULSADOR
14
15   onADC = !onADC; // alternamos el estado del ADC
```

```
16
17  }
```

Tenemos el archivo `gpio.ino` finiquitado. Sería un buen momento para un ***commit*** y ***push***.

1.3.0.3.3. Desarrollo de timer.ino Teniendo abierto el archivo `gpio.ino`, para crear este archivo hacemos clic en la **flecha a la derecha** del todo que está **alineada con el nombre del archivo** y seleccionamos `Nueva Pestaña`.

Figura 10: Nuevo archivo de proyecto.

> En la imagen pone que estamos en el archivo `masb-p04`. Eso da igual. Es solo para ilustrar donde debemos de clicar.

Nos pedirá un nombre. Ponemos `timer`. Con esto, se nos habrá creado un fichero `timer.ino` en la carpeta `gpio`. En este archivo, **crearemos dos funciones**: `TIMER_Init` y `startAdcConversion`. La primera la utilizaremos para **configurar el *timer*** y nuestro compañero la llamará desde la función `setup`. La segunda función será la **ISR que ejecutemos cada 100 ms**. Creamos las dos funciones y configuramos el *timer* 3 para que ejecute la ISR `startAdcConversion` cada 100 ms. El código quedaría así:

```
1  void TIMER_Init(void) {
2
3    // configuracion del timer 3
4    HardwareTimer *MyTim = new HardwareTimer(TIM3);
5
6    MyTim->setMode(2, TIMER_OUTPUT_COMPARE); // configuramos el modo del
        timer sin salida por ningun pin
7    MyTim->setOverflow(100000, MICROSEC_FORMAT); // configuramos el
        periodo del timer a 100 ms
8    MyTim->attachInterrupt(startAdcConversion); // indicamos el nombre de
         la ISR a ejecutar
9    MyTim->resume(); // iniciar el timer
10 }
11
12 // ISR del timer
13 void startAdcConversion(HardwareTimer*) {
14
15 }
```

Finalmente, **en la ISR, llamamos a la función `conversionADC`** que nuestro compañero ha implementado para realizar una conversión del ADC si este está encendido.

```
1  void TIMER_Init(void) {
2
3    // configuracion del timer 3
4    HardwareTimer *MyTim = new HardwareTimer(TIM3);
5
6    MyTim->setMode(2, TIMER_OUTPUT_COMPARE); // configuramos el modo del
        timer sin salida por ningun pin
7    MyTim->setOverflow(10000, MICROSEC_FORMAT); // configuramos el
        periodo del timer a 100 ms
8    MyTim->attachInterrupt(startAdcConversion); // indicamos el nombre de
         la ISR a ejecutar
9    MyTim->resume(); // iniciar el timer
10 }
11
12 // ISR del timer
13 void startAdcConversion(HardwareTimer*) {
14
15   // tratamos de iniciar conversion
16   conversionADC();
17
```

```
18 }
```

Con esto tendríamos el archivo finalizado. Es el momento ideal para hacer ***commit*** y ***push*** de nuestros cambios.

1.3.0.3.4. Cambio de nombre de la carpeta **Arduino IDE, por defecto, nos crea una carpeta con el mismo nombre del *sketch* que hemos creado** y allí dentro guarda el archivo `.ino`. Eso es porque, tal y como hemos comentado antes, para Arduino IDE el archivo principal es aquel con el mismo nombre que la carpeta que lo contiene. Pero este no es nuestro caso. **Nuestro archivo principal se llama `masb-p04.ino` y lo está desarrollando nuestro compañero**. Por esto, **vamos a ir a la carpeta `arduino`** de nuestro repositorio local **y vamos a cambiar el nombre de la carpeta de `gpio` a `masb-p04`**.

Una vez hecho el cambio, hacemos ***commit*** y ***push***.

1.3.0.3.5. *Pull Request* y *Review* Nuestro trabajo ya habría acabado. O casi. **Nos faltan dos cosas**. Primero de todo, **hacer un *Pull Request* de esta rama hacia la *master* y poner, importante, como *reviewer* a nuestro compañero `A`**. Él revisará nuestro código y, si lo ve `OK`, tú o él podéis realizar el `merge` hacia la rama máster. Si no os da el `OK`, deberéis de atender las peticiones de cambio que os pide, hacer las modificaciones pertinentes, subir los cambios y realizar una segunda petición de revisión o *re-request*.

Aquí tampoco acaba tu trabajo, porque ahora **deberás de ir al *Pull Request* de tu compañero, revisar que todo está `OK`** y, si es así, aprobarle el *Pull Request*. Si no estás conforme con su desarrollo, házselo saber y proponle cambios. Una vez los aplique, aprueba el *Pull Request* y tú o él hacer el `merge` a la rama `master`.

Os dejo un enlace a la documentación de ayuda de GitHub donde podéis ver en detalle todo lo que podéis hacer durante la revisión de los *Pull Requests*.

> En este proyecto, es indiferente quien hace primero el *merge* a la rama `master`.

1.3.0.4. Resumen Y con esto, ¡habéis hecho vuestro **primer proyecto juntos**! Ahora yo iré a la rama `master` y veré vuestro proyecto.

Como habéis podido comprobar, a veces llamamos funciones que no son nuestras y cuyo nombre parece que no deberíamos de saber en un principio. **Los flujos de trabajo basado en git**, u otras herramientas, posibilitan poder desarrollar en paralelo y poder trabajar en equipo en un mismo proyecto. Pero **no evitan las más que necesarias reuniones de equipos** donde **se acuerda** qué arquitectura *software* debe de tener un proyecto, cuál va a ser la estructura de directorios del proyecto,

como se llamarán las funciones que implementaremos, etc. Y normalmente, además, **todo esto se recoge en la documentación técnica del proyecto** para que esté disponible para todos los miembros del equipo.

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:

- ☐ **Pregunta:** Ignorando aspectos de procesado de la señal (Nyquist y demás), no se me ocurre ningún motivo por el cual no pueda hacer que el *sampling rate* sea tan pequeño como yo quiera (una conversión cada 1 segundo, 1 hora, 1 día,…); pero… ¿creéis que **hay algún límite para establecer un *sampling rate* máximo**? Si es así, ¿cuál sería **un posible motivo**?
- ☐ **Con tus palabras:** explícame porque no podíamos lograr un *sampling rate* dado con el primer caso y porque tampoco estaba garantizado que el *sampling rate* fuese constante.

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* (menos en el del reto).

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

En esta práctica hemos aprovechado que **el ADC en Arduino es muy sencillo** (solo hay que utilizar la función `analogRead`) para empezar a ver cómo podemos hacer un **proyecto de manera conjunta** con nuestro compañero. Hemos visto **cómo desarrollar en paralelo** y cómo **luego unirlo todo en una sola rama**. También hemos visto cómo **git no sustituye la necesidad de realizar reuniones de equipo donde se acuerden los aspectos técnicos del proyecto** a los que estarán sujetos todos los miembros del equipo de desarrollo. Por último, en el proyecto hemos visto **cómo fijar un *sampling rate* constante**. Algo vital en aplicaciones basadas en ADCs.