MASB | Arduino
Comunicación serie I

#stm32 #c #biomedical #microcontroladores #programacion

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

6 de marzo de 2020

Índice

1. Comunicación serie - Parte I

> **IMPORTANTE:** En lugar de utilizar `develop/A-<tu-nombre>` para hacer tu parte de la práctica, utiliza `lab/A-<tu-nombre>`.

En la práctica anterior hemos utilizado una comunicación serie para enviar datos del microcontrolador al ordenador, pero lo hemos hecho a modo "salto de fe". Sin saber cómo funciona o qué estábamos haciendo. En esta práctica empezaremos a ver **cómo operar** con las comunicaciones serie y **qué es lo que estamos enviando**.

Existen múltiples tipos de comunicaciones serie: I2C (*Inter-Integrated Circuit*), SPI (*Serial Peripheral Interface*), CAN (*Controller Area Network*), UART (*Universal Asyncronous Receiver-Transmitter*), PCIe (*Peripheral Component Interconnect Express*), USB (*Universal Serial Bus*), Ethernet, … De todas ellas, en esta práctica, **veremos la comunicación serie UART**. Esta es una de las comunicaciones serie más sencillas y de las primeras que se suele introducir. Es común, y es nuestro caso, que una EVB también contenga un convertidor UART a USB de tal modo que, **mediante UART, podemos enviar datos a un ordenador a través de una conexión USB**. Por ello, suele ser de las primeras comunicaciones serie a aprender. Eso, y por ser una de las comunicaciones más conocidas.

1.0.1. Serie vs Paralelo

Vale, vale,… Pero empecemos por ver **qué es una comunicación serie** y en qué se diferencia del otro tipo de comunicación llamada paralelo. Pues muy sencillo. El calificativo serie y paralelo hacen referencia al **modo en el que se envían los bits entre dispositivos**. En el caso de la **comunicación paralela**, **más de un bit es enviado de manera simultanea** de un dispositivo a otro. Esto conlleva que se necesitan, como mínimo, una línea de comunicación por bit (mayor uso de pines). Pero tiene como ventaja una mayor velocidad de transmisión al poderse enviar más de un bit a la vez.

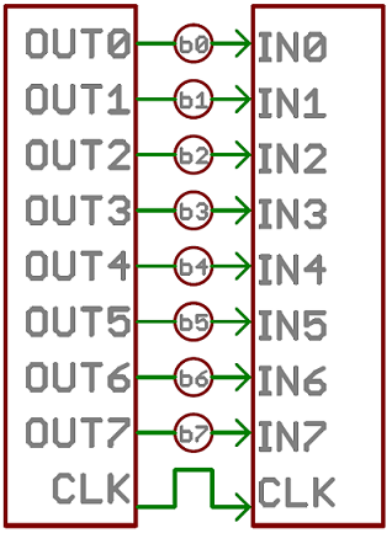

Figura 1: Comunicación paralelo.

> Imagen de una comunicación paralelo. [Fuente]

Por otro lado, en una **comunicación serie enviamos los bits individualmente de manera secuencial**. Uno detrás de otro. De este modo, reducimos el número de líneas necesarias (y por consiguiente, el número de pines necesarios), pero a costa de una menor velocidad de comunicación.

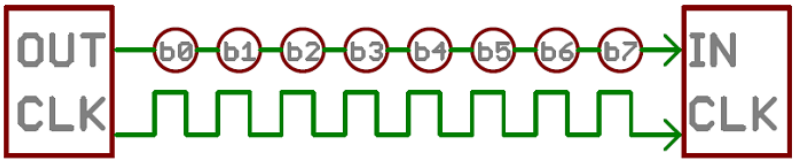

Figura 2: Comunicación serie.

> Imagen de una comunicación serie. [Fuente]

1.0.2. Síncrona vs asíncrona

Otra clasificación para las comunicaciones es si son síncronas o asíncronas. ¿La diferencia entre ellas? La **diferencia es si se utiliza o no una señal de reloj para sincronizar el envío de los bits**. Una comunicación **síncrona**, aparte de enviar los bits, también **enviará una señal de reloj** para sincronizar la escritura/lectura de los bits entre dispositivos. Una comunicación **asíncrona no necesitará** de esa **señal de reloj** y nos ahorramos un pin. Obviamente, a nivel de electrónica (que no a nivel de

programación, que es lo que nosotros veremos) implica una mayor complejidad que será transparente para nosotros.

1.0.3. UART

Antes hemos mencionado **UART como una comunicación serie** cuando **no es así estrictamente hablando**. **UART es el periférico** encargado de llevar a cabo la comunicación serie. De hecho, en nuestro microcontrolador el módulo "UART" se llama **USART** (*Universal Synchronous/Asynchronous Receiver Transmitter*) ya que el periférico puede configurarse para realizar **comunicaciones síncronas y asíncronas**. Así que en realidad lo que veremos, simplemente, es una comunicación serie asíncrona. "¿Y porque nos engañas? ¿¡Qué te hemos hecho!?" Nooo… Tranquilos. Haremos referencia a la comunicación serie asíncrona como UART ya que es muy extendido hacer referencia a esta comunicación por su periférico y, de este modo, podréis encontrar más fácilmente documentación e información sobre esta comunicación. En esta práctica, cuando hablemos de UART, comunicación serie o, simplemente, comunicación, estaremos hablando de la comunicación serie asíncrona.

1.0.3.1. Conexionado El conexionado es sencillo. Cada dispositivo dispondrá de **tres pines dedicados a la comunicación**:

- **TX:** Pin dedicado al envío de bits.
- **RX:** Pin dedicado a la recepción de bits.
- **GND:** Tensión de referencia que deben de compartir los dispositivos.

Para la comunicación entre dos dispositivos, **el TX de uno debe de estar conectado al RX** del otro y **viceversa**. La conexión quedaría del siguiente modo:

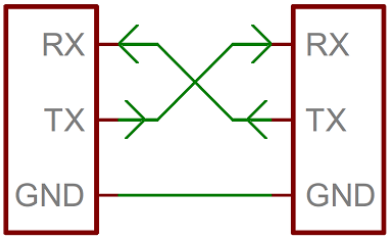

Figura 3: Conexionado de una comunicación serie asíncrona.

> Imagen de interconexión para una comunicación serie. [Fuente]

Pese a que es posible conectar **más de dos dispositivos** en un mismo bus de comunicación serie asíncrona, **no suele ser recomendable** puesto que se requiere tomar una serie de medidas fuera del "estándar" para que no haya conflictos de comunicación en el bus.

> Un bus de datos es simplemente la agrupación lógica de todas las señales que intervienen en la comunicación.

1.0.3.2. Parámetros de la comunicación y *bit framing* Sobre la comunicación serie asíncrona debemos de quedarnos con cuatro parámetros esenciales:

- **Número de bits de datos:** son el número de bits de datos que enviaremos por cada paquete de bits. Estos pueden ser de 5 a 9 siendo 8 bits (1 byte) lo más normal.
- **Número de bits de sincronización:** número de bits que se utilizan para detectar el inicio y fin de un paquete de bits. Siempre habrá un bit inicial de sincronización. Nosotros deberemos de escoger si habilitamos uno o dos bits de sincronización al final del paquete, también conocidos como bits de stop. Suele ser común utilizar solo un bit de stop. La línea TX siempre está en nivel alto cuando no se está haciendo ninguna transmisión. En este estado se dice que se está en estado IDLE. Los bits de sincronización se encargan de realizar una transición de IDLE a `0` cuando se inicie la transmisión de un paquete de bits y de `0` a IDLE cuando finalice.
- **(Des)Habilitación del bit de paridad:** se puede habilitar o no un bit de paridad que añade a la comunicación un sencillo control de errores. Este parámetro se puede configurar en *none*, *even* o *odd*. Si se configura en *none*, no habrá bit de paridad. Cuando lo configuremos en *even*, si el número de `1` en los bits de datos es par, el bit de paridad será 1. Cuando lo configuremos en *odd*, si el número de `1` en los bits de datos es impar, el bit de paridad será 1.
- ***Baud rate*:** velocidad de transmisión de un símbolo. En este caso, tomamos como símbolo el bit, por lo que el *baud rate* y el *bit rate* coinciden. Esta velocidad, indicada como bps (*bit per second*) puede tomar cualquier valor. Aun así, suelen ser comunes las velocidades de: 1200, 2400, 4800, 9600, 19200, 38400, 57600 y 115200 bps. La más común suele ser 9600 bps. La máxima también suele ser 115200 bps (más allá, el número de bits que se pueden perder por el camino puede ser considerable y sería necesario una gestión de errores). Sabiendo el *baud rate*, podemos calcular cuánto tardaría la transmisión de un paquete de bits.

Figura 4: Bit framing.

> Imagen de *bit framing* para una comunicación serie. [Fuente]

Suele ser común referenciar una comunicación serie asíncrona con una nomenclatura generada a partir de los parámetros de la comunicación. Por ejemplo, 9600 8N1 significa: un *baud rate* de 9600,

con 8 bits de datos, sin paridad y un bit de stop. Otro ejemplo sería 115200 5O2 que correspondería a: un baud rate de 115200 bps, 5 bits de datos, un bit de paridad *odd* y dos bits de stop.

> En esta práctica hablaremos bastante de paquetes o *framing* de bits. Importante recalcar lo de **bits** porque en la siguiente práctica hablaremos de paquetes o *framing* de **bytes**, que es muy distinto.

Con toda la teoría repasada, ¡vamos a trabajar con la comunicación serie asíncrona!

1.1. Objetivos

- Introducción a la comunicación serie en Arduino.
- Codificación ASCII.
- Envío de datos RAW.
- Sentencia *switch*.

1.2. Procedimiento

1.2.1. Instalación de CoolTerm

Antes de empezar, nos vamos a **descargar e instalar** la aplicación **CoolTerm** en nuestro ordenador. Podéis encontrarla **aquí**. Esta aplicación, disponible tanto en Windows, Mac y Linux, es un terminal serie como el de Arduino IDE, pero con alguna función interesante más que nos ayudará a ver qué estamos enviando en la comunicación serie.

1.2.2. Codificación ASCII

Vamos a ver un aspecto básico que utilizamos cada día, pero que queda enmascarado al enviar datos mediante texto: la codificación ASCII.

1.2.2.1. "Hola, soy STM32"

Antes de entrar al detalle de qué es, vamos a crear un pequeño *sketch* en Arduino IDE que nos envíe por **comunicación serie** un texto tipo **"Hola, soy STM32."**. Para ello, nos creamos un *sketch* con el **nombre** `masb-p05` en la **carpeta** `arduino` de nuestro repositorio local.

Para realizar una comunicación serie en Arduino, disponemos de la **librería/objeto** `Serial`. Para configurar el ***baud rate* de la comunicación**, utilizamos la función `Serial.begin()` donde se indica como parámetro el *baud rate*. Por defecto, la comunicación viene configurada como XXXX 8N1, donde XXXX es el *baud rate* que hemos configurado. Vamos a **inicializar** la comunicación serie a **9600 8N1**.

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600); // 9600 8N1
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Ahora, vamos a hacer que se envíe el **mensaje** "Hola, soy STM32." **cada vez que pulsemos el pulsador**. El código quedaría del siguiente modo:

```
#define PULSADOR 23

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600); // 9600 8N1

  pinMode(PULSADOR, INPUT); // modo input
  // configuracion de la ISR
  attachInterrupt(digitalPinToInterrupt(PULSADOR), hola, FALLING);
}

void loop() {
  // put your main code here, to run repeatedly:

}

void hola(void) {
  // ISR pulsador
  Serial.print("Hola, soy STM32."); // enviamos mensaje
}
```

Con la función `Serial.print()` enviamos un mensaje/texto por comunicación serie. Compilamos y subimos el programa al microcontrolador. Vamos a configurar CoolTerm para que reciba esos mensajes.

1.2.2.2. Configuración de CoolTerm

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.

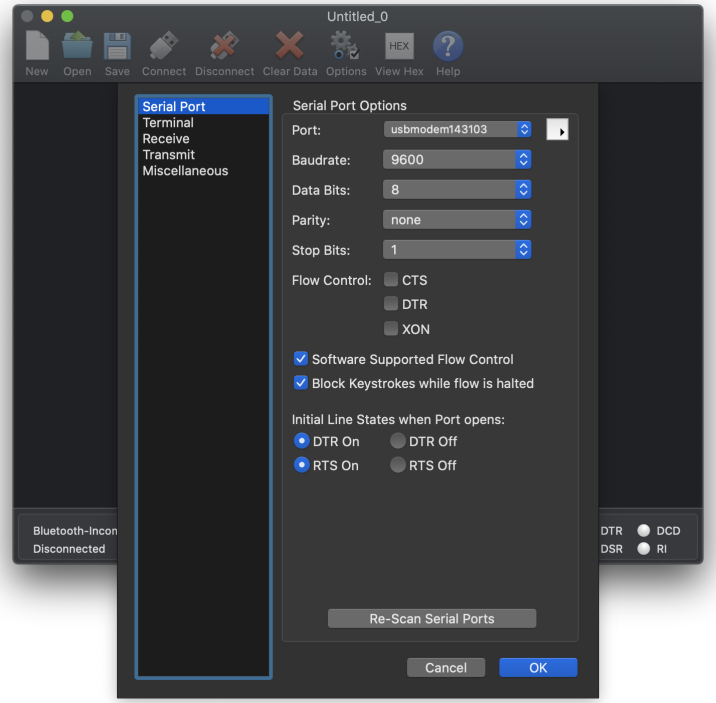

Figura 5: Configuración CoolTerm”

Podemos clicar en el icono `Save` para guardar la configuración. Con todo configurado, **clicamos en el botón `Connect`**. Probamos a **enviar un mensaje pulsando el pulsador** de nuestra EVB y **deberíamos de recibir un “Hola, soy STM32.”** por cada pulsación.

Podemos comprobar que **se concatena el texto** sin que se nos creen nuevas líneas para cada mensaje. Eso es porque hemos usado la función `Serial.print()` que envía el texto tal cual. Si utilizamos la función **`Serial.println()` nos envía el texto** pertinente y después de este **añade dos caracteres más**: un **retorno de carro** y un carácter de **línea nueva**. El primero corresponde al carácter `\r` y el segundo al carácter `\n`. Vamos a sustituir la función `Serial.print()` por `Serial.println()` en nuestro *sketch*. Quedaría del siguiente modo:

```
1  #define PULSADOR 23
2
3  void setup() {
4    // put your setup code here, to run once:
5    Serial.begin(9600); // 9600 8N1
6
7    pinMode(PULSADOR, INPUT); // modo input
8    // configuracion de la ISR
9    attachInterrupt(digitalPinToInterrupt(PULSADOR), hola, FALLING);
10 }
```

```
11
12 void loop() {
13   // put your main code here, to run repeatedly:
14
15 }
16
17 void hola(void) {
18   // ISR pulsador
19   Serial.println("Hola, soy STM32."); // enviamos mensaje
20 }
```

Compilamos, subimos y pulsamos el pulsador. Ahora sí, **CoolTerm identifica el fin de un mensaje** al encontrar los carácteres `\r` y `\n` y **crea una nueva línea**.

Lo que acabamos de usar son caracteres de delimitación o *term char*. Estos nos indican dónde acaba un mensaje.

1.2.2.3. Pero... ¿Qué estamos enviando?

CoolTerm tiene una funcionalidad muy interesante que nos deja ver **qué estamos enviando en realidad**. Nos deja ver los **datos en crudo o *raw***. Esta funcionalidad la activamos clicando sobre el **icono `View HEX`**. Al clicar, el terminal nos muestra tres columnas.

Empezamos por la **segunda columna**. Esta muestra **en hexadecimal los paquetes de bits enviados**. Es lo que realmente estamos enviando. Son los **datos en raw**. Por defecto, muestra 16 paquetes de bits por fila.

La **primera columna** es simplemente una **numeración que hace CoolTerm** para identificar cada paquete de bits enviado. El valor de la columna identifica el primer paquete de bits de los 16 paquetes representados en la segunda columna de esa fila.

Por último, la **tercera columna** es la **codificación ASCII** de los valores *raw* realmente enviados y que aparecen en la segunda columna. Y aquí llegamos al meollo del asunto: **al enviar *strings* desde el microcontrolador, no estamos enviando texto, estamos enviando números que són codificados en ASCII por el ordenador**. En la siguiente tabla podéis ver la codificación ASCII de los 128 primeros caracteres.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Figura 6: Codificación ASCII"

> Codificación ASCII. [Fuente]

Así que cuando enviamos una `A`, en realidad estamos enviando un `65`. Cuando enviamos una `f` enviamos un `102`. Y así consecutivamente. Por lo tanto, para enviar "Hola, soy STM32.", estamos enviando `72 111 108 97 44 32 115 111 121 32 83 84 77 51 50 46` en decimal o `0x48 0x6F 0x6C 0x61 0x2C 0x20 0x73 0x6F 0x79 0x20 0x53 0x54 0x4D 0x33 0x32 0x2E` en hexadecimal.

> Al programar microcontroladores, para indicar que un número está siendo indicado utilizando hexadecimal, se coloca `0x` delante del número.
>
> Si alguien no sabe qué es o cómo se obtiene un número en base hexadecimal, puede encontrar más info aquí.

"Muy bien, ¿y?" Vale. Imaginemos que queremos enviar un número almacenado en una variable de un byte, es decir, una variable tipo `uint8_t`. Esa variable puede tomar un valor máximo de 255. Si enviásemos este número en codificación ASCII tendríamos que enviar `0x32 0x35 0x35`; tres bytes. Si lo enviamos en *raw* tendríamos que enviar `0xFF`; un solo byte. ¿Veis qué ha ocurrido? Por querer

utilizar la codificación ASCII hemos tenido que utilizar tres bytes en lugar de solo uno. **La codificación ASCII nos añade *overhead*** de dos bytes. Esto nos **limita el ancho de banda de la comunicación** notablemente y será inviable la comunicación en aplicaciones donde necesitemos una comunicación rápida. Así que, ***rule of thumb***:

- Utilizar **siempre comunicación con datos *raw*** y nunca tendréis problemas ni nadie os mirará mal.
- Si vuestra aplicación no precisa de una transmisión rápida o de muchos datos (y os da igual lo que piense la gente de vosotros…) podéis usar ASCII.

Pensad, que en aplicaciones reales no enviaremos nunca texto, siempre enviaremos valores numéricos e instrucciones o variables de *status* codificadas en valores numéricos, así que poco o ningún sentido tiene utilizar ASCII para enviar datos.

> Cuando hablo de codificación de instrucciones o variables de *status* codificadas en valores numéricos hablo que, por ejemplo, cuando quiera decirle, desde el ordenador, a un dispositivo que inicie una medición, no le enviaré "Por favor, cuando te venga bien, inicia la medición del pH de esta muestra.", sino que enviaré, por ejemplo `0x01`. Y si quiero que pare le enviaré un `0x02` y no un "Para, para, que me la estas liando.".
>
> **La elección de cómo se codifica cada instrucción es nuestra y forma parte del proceso de diseño del *set* de instrucciones de nuestro dispositivo.**

1.2.3. Datos *raw*

Hemos visto que la codificación **ASCII es el mal** y debemos de alejarnos de ella. Ni siquiera gastaremos tiempo en ver un ejemplo de cómo enviar datos del ordenador al microcontrolador en ASCII. Lo veremos directamente en *raw*.

1.2.3.1. Enviar datos *raw*

Para enviar datos en *raw* nos olvidamos de la función `Serial.print()` y utilizamos **la función `Serial.write()`**. El *sketch*, para enviar un 1, quedaría del siguiente modo:

```
#define PULSADOR 23

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600); // 9600 8N1

  pinMode(PULSADOR, INPUT); // modo input
  // configuracion de la ISR
  attachInterrupt(digitalPinToInterrupt(PULSADOR), hola, FALLING);
}
```

```
11
12 void loop() {
13   // put your main code here, to run repeatedly:
14
15 }
16
17 void hola(void) {
18   // ISR pulsador
19   Serial.write(0x01); // enviamos un 1
20 }
```

Compilamos y subimos el programa al microcontrolador. Antes de enviar nada, vamos a **limpiar el terminal de CoolTerm** clicando sobre el icono `Clear Data`. Pulsamos el botón y vemos como en *raw* recibimos un `0x01`, mientras que la codificación ASCII (que nos da igual) sería "inicio de cabecera" que aparece como un punto.

Par ver que nos os engaño, podéis probar de enviar primero un 255 en texto con la función `Serial.print()` y luego el número 255 con la función `Serial.write()`. Veréis claramente que el uso de ASCII nos penaliza gravemente. (¡Hacedlo!)

1.2.3.2. Recibir datos *raw*

Vamos a hacer que nuestro microcontrolador **reciba datos del ordenador**. Para ello, haremos dos ejemplos: uno en el que el microcontrolador devolverá al ordenador lo que ha recibido (*echo*) y otro en el que cuando el ordenador envíe un 1 el LED de la EVB se encienda y al recibir un 2 el LED de la EVB se apague.

1.2.3.3. *Echo*

Vamos a **enviar datos** del ordenador al microcontrolador y haremos que este **responda con los mismos datos que ha recibido**. Primeramente, modificamos el *sketch* para hacer que el microcontrolador esté pendiente de si se han recibido datos. Esto se hace con la **función `Serial.available()`**. Esta función devuelve el número de bytes recibidos y que aún no hemos leído. Después, leeremos un byte del objeto `Serial` con la función `Serial.read()`. El *sketch* quedaría del siguiente modo:

```
1  void setup() {
2    // put your setup code here, to run once:
3    Serial.begin(9600); // 9600 8N1
4  }
5
6  void loop() {
7    // put your main code here, to run repeatedly:
8    if (Serial.available() > 0) { // si quedan bytes pendientes de leer
         ...
9      Serial.write(Serial.read()); // enviamos por puerto serie lo que
           nos acaban de enviar
10   }
```

```
11 }
```

Compilamos y subimos. Ahora nos vamos a CoolTerm y vamos a **`Connection > Send String...`**. Se nos abrirá una ventana donde podemos escribir lo que deseamos enviar al microcontrolador. En la ventana podemos escoger el método de entrada: ASCII o Hex. ASCII es el mal. Escogemos Hex. Escribimos un valor hexadecimal de nuestro gusto y clicamos Send. Si todo está correcto, el microcontrolador recibirá esos datos y los devolverá al ordenador, así que **esos mismos datos os deben de aparecer en el terminal de CoolTerm**.

1.2.3.4. Control de un LED Vamos a hacer un último ejemplo para controlar un LED. Modificaremos el *sketch* de Arduino para que al recibir un **`0x01` el LED se encienda** y al recibir un **`0x02` se apague**. Para ello, utilizaremos la **sentencia `switch`**. Esta sentencia lo que hace es comparar el valor o variable que le pasemos con diferentes casos. Podéis encontrar más info aquí. Además, siempre que enviemos una instrucción al microcontrolador, este nos devolverá un byte de *status* de tal modo que **si la instrucción es correcta nos devolverá un `0x00`** indicando que no hay ningún error, y **si enviamos algo distinto a `0x01` o `0x02` devolverá un `0x01`** indicando que se ha enviado una instrucción inexistente. El código sería el siguiente:

```
1  #define LED 13
2
3  void setup() {
4    // put your setup code here, to run once:
5    Serial.begin(9600); // 9600 8N1
6    pinMode(LED, OUTPUT); // pin de salida
7    digitalWrite(LED, LOW); // apagado por defecto
8  }
9
10 void loop() {
11   // put your main code here, to run repeatedly:
12   byte rxBuffer; // buffer para guardar los datos recibidos
13
14   if (Serial.available() > 0) { // si quedan bytes pendiente de leer...
15     rxBuffer = Serial.read();
16
17     switch(rxBuffer) { // comparamos el valor de rxBuffer con
           diferentes valores
18       case 0x01: // encender LED
19         digitalWrite(LED, HIGH);
20         Serial.write(0x00); // devolvemos que no ha habido ningun error
21         break;
22       case 0x02: // apagar LED
23         digitalWrite(LED, LOW);
24         Serial.write(0x00); // devolvemos que no ha habido ningun error
25         break;
26       default: // si enviamos algo distinito, enviamos un 1 para
             alertar de que el comando
```

```
27                  // enviado no existe
28          Serial.write(0x01);
29          break;
30      }
31    }
32  }
```

Compilamos, subimos y nos vamos a CoolTerm. Limpiamos el terminal y enviamos un `0x01` al microcontrolador. Si está todo correcto, vemos que el LED se enciende y el microcontrolador nos contesta un `0x00` indicándonos que no hay ningún error. Ahora enviamos un `0x02` y logramos apagar el LED sin error. Si por lo que fuera enviamos un valor distinto, como por ejemplo `0x03`, el microcontrolador nos devuelve un `0x01` indicándonos que hemos enviado una instrucción inexistente. ¡Perfectísimo!

1.2.4. Problema...

No se si se os ha venido solo a la cabeza, pero vamos a ver un problema que tiene el uso de datos en *raw*. Vamos a presentar el problema mediante un ejemplo.

En el ejemplo anterior con el LED, solo hay dos instrucciones y son de un solo byte. Muy sencillo. Pero no suele ser lo normal. Normalmente se hay multitud de instrucciones disponibles, muchas de ellas aceptan parámetros (como por ejemplo enciende el LED 1, 2, 3,...) y además, muchas veces (por no decir siempre), se suele añadir un byte o más para realizar una gestión de errores (lo veremos en la siguiente práctica). Esto hace que no solo enviemos un solo byte, sino que enviamos un *array* de bytes. Ahora imaginaros que sois el microcontrolador y yo os envío lo siguiente:

---

48 6F 6C 61 2C 20 73 6F 79 20 53 54 4D 33 32 2E 41 64 69 F3 73 2E 4D 65 20 6C 6C 61 6D 6F 20 4D 61 6E 6F 6C 6F 2C 65 6E 20 73 65 72 69 6F 20 68 61 73 20 74 72 61 64 75 63 69 64 6F 20 65 73 74 6F 3F 66 72 69 6B 69 2E

Es imposible que sepáis dónde empieza y dónde acaba una instrucción. Y esto le pasa también al microcontrolador. De hecho, era lo que pasaba en el primer ejemplo en el que usábamos `Serial.print()` donde en el CoolTerm nos aparecía todo el texto concatenado.

Lo que hemos hecho para solucionarlo es utilizar el comando `Serial.println()` para enviar datos de tal modo que se nos añadía a nuestro mensaje un *term char*. Ese *term char* era un retorno de carro `\r` (`0x0D` en ASCII) y un carácter de nueva línea `\n` (`0x0A` en ASCII). De este modo, CoolTerm sabe cuando acaba un mensaje o paquete y crea una línea nueva. Aplicando esto al mensaje anterior quedaría:

---

48 6F 6C 61 2C 20 73 6F 79 20 53 54 4D 33 32 2E **0D 0A** 41 64 69 F3 73 2E **0D 0A** 4D 65 20 6C 6C 61 6D 6F 20 4D 61 6E 6F 6C 6F 2C **0D 0A** 65 6E 20 73 65 72 69 6F 20 68 61 73 20 74 72 61 64 75 63 69 64 6F 20 65 73 74 6F 3F **0D 0A** 66 72 69 6B 69 2E **0D 0A**

Ahora, buscando los carácteres `0x0D` y `0x0A` podemos saber dónde finaliza un mensaje/instrucción. Parece obvio que, para que esto funcione, el propio mensaje no puede contener los carácteres `0x0D` y `0x0A` para no leer un falso fin de instrucción.

Esto en ASCII no hay problema, pero en *raw*… Imaginaros que escogemos que el carácter de final de instrucción es el `0x00`. Eso quiere decir que podemos enviar un 0 o se detectaría un falso fin de instrucción. Tampoco sirve de nada utilizar otro número como *term char* ya que no podemos garantizar que ese número nunca se vaya a enviar como dato. ¡No hay manera de indicar el fin de una instrucción en *raw*!

O eso parece a simple vista… Hay una manera y es aplicando una codificación del paquete de bytes para lograr que el *term char* que utilicemos no aparezca en los datos que enviamos. Eso lo veremos en la siguiente práctica.

1.3. Reto

Vamos a hacer un proyecto en equipo, pero esta vez “sin guía”. Solo os diré qué tenéis que hacer y de qué se tiene que encargar el miembro `A` y el miembro `B`.

1.3.1. Objetivo

En el reto debéis de implementar un *set* de instrucciones para el microcontrolador. Este *set* de instrucciones es el siguiente:

Instr	Descripción
0x01	Iniciar conversión periódica del ADC. Al recibir esta instrucción el microcontrolador debe de convertir el valor del divisor de tensión (ya sabéis como conectarlo) cada 1 segundo y enviarlo al ordenador cada vez. El ADC devuelve un valor de 10 bits, por lo que deberéis de enviar 2 bytes. Para hacerlo, utilizad el código que tendréis a continuación.

Instr	Descripción
0x02	Detenemos la conversión del ADC.

Cada vez que enviemos una instrucción, el microcontrolador nos tiene que devolver un valor de *status*. Si la instrucción es correcta devolveremos `0x00`. Si la instrucción que enviemos no existe, devolveremos el código `0x01`.

```
// Codigo para enviar una variable de 2 bytes

int valAdc = analogRead(A0); // guardamos el valor del ADC en una
    variable

// valAdc esta compuesto por 2 bytes: los primeros 8 bits (MSB) y los 8
     ultimos (LSB)
Serial.write(valAdc); // enviamos primero LSB
Serial.write(valAdc >> 8); // shifteamos/movemos los 8 bits de MSB a
    LSB y enviamos LSB
```

1.3.2. Reparto de tareas

El proyecto debe de desarrollarse en una rama propia siguiendo las directrices de la práctica anterior. Luego se deben de combinar los desarrollos de los dos miembros del equipo en la rama `master` mediante los pertinentes *Pull Requests* que deben de ser aprobados y *merged* por el otro compañero.

La estructura de directorios/archivos debe de ser la siguiente:

```
arduino
  - masb-p05-reto-a
    - adc.ino
    - serial.ino
    - masb-p05-reto-a.ino
    - timer.ino
```

Intentad seguir el esquema de la práctica anterior como guía, ¡pero no dudéis en modificar lo que necesitéis para adaptaros al nuevo reto! Leed las tareas del otro miembro para saber qué funciones debéis de llamar desde vuestros archivos. También os adjunto un diagrama de flujo (¡primera vez que vemos uno!) de cómo debe de ser la ejecución del programa.

1.3.2.1. Miembro A El miembro `A` se encargará de gestionar los siguientes archivos con las siguientes funciones:

- masb-p05-reto-a.ino

- `void setup(void)` En esta función se llaman todas las funciones de configuración de los diferentes periféricos.

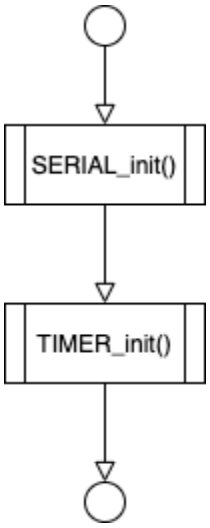

Figura 7: Diagrama de flujo de la función setup.

> Diagrama de flujo de la función `setup()`.

- `void loop(void)`

  Se llama a la función `SERIAL_check()` del archivo `serie.ino`.

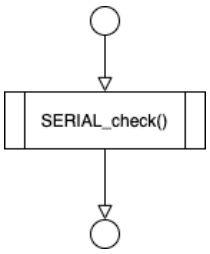

Figura 8: Diagrama de flujo de la función loop.

> Diagrama de flujo de la función `loop()`.

- serial.ino
  - `void SERIAL_init(void)` Inicializa la comunicación serie con la configuración 9600 8N1.

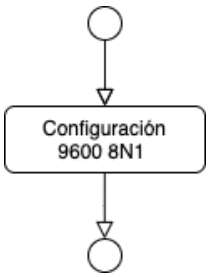

Figura 9: Diagrama de flujo de la función SERIAL_init.

> Diagrama de flujo de la función `SERIAL_init()`.

- `void SERIAL_check(void)` Comprueba si se ha recibido datos serie y se realizan las acciones pertinentes según la instrucción.

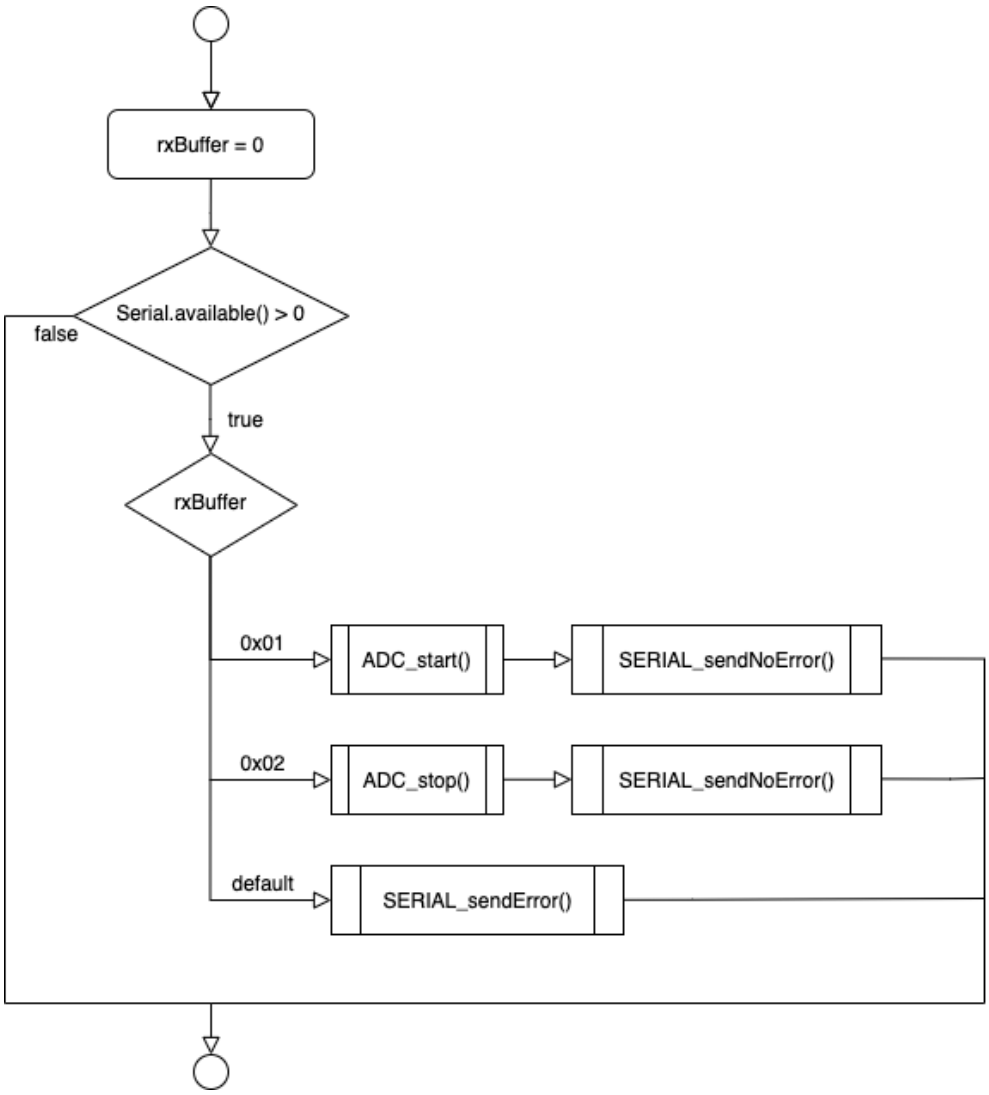

Figura 10: Diagrama de flujo de la función SERIAL_check.

> Diagrama de flujo de la función `SERIAL_check()`.

- `void SERIAL_sendAdc(int value)` Función utilizada para enviar el valor del ADC. Acepta como parámetro el valor a enviar.

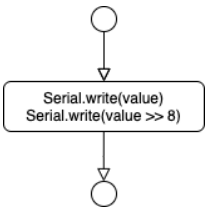

Figura 11: Diagrama de flujo de la función SERIAL_sendAdc.

> Diagrama de flujo de la función `SERIAL_sendAdc(int value)`.

- `void SERIAL_sendNoError(void)` Función utilizada para enviar el *status* sin error.

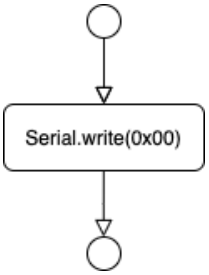

Figura 12: Diagrama de flujo de la función SERIAL_sendNoError.

> Diagrama de flujo de la función `SERIAL_sendNoError()`.

- `void SERIAL_sendError(void)` Función utilizada para enviar el *status* con error.

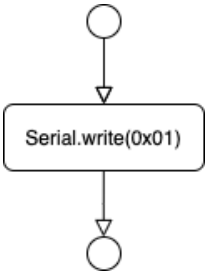

Figura 13: Diagrama de flujo de la función SERIAL_sendError.

> Diagrama de flujo de la función `SERIAL_sendError()`.

1.3.2.2. Miembro B

El miembro `B` se encargará de gestionar los siguientes archivos:

- adc.ino

- `void ADC_convert(void)` Función desde la que se inicia una conversión ADC y posteriormente se envía el dato mediante `SERIAL_sendAdc`.

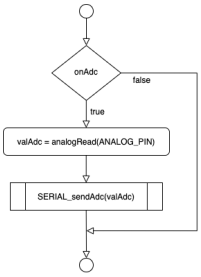

Figura 14: Diagrama de flujo de la función ADC_convert.

> Diagrama de flujo de la función `ADC_convert()`.

- `void ADC_stop(void)` Función que se utilizará desde `SERIAL_check` para detener la conversión periódica con el ADC.

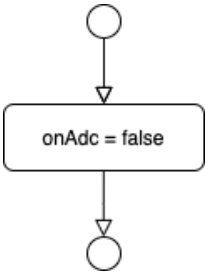

Figura 15: Diagrama de flujo de la función ADC_stop.

> Diagrama de flujo de la función `ADC_stop()`.

- `void ADC_start(void)` Función que se utilizará desde `SERIAL_check` para iniciar la conversión periódica con el ADC.

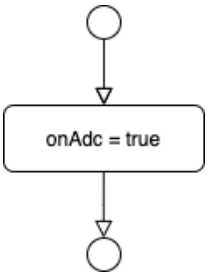

Figura 16: Diagrama de flujo de la función ADC_start.

> Diagrama de flujo de la función `ADC_start()`.

- timer.ino
  - **`void`** `TIMER_init(`**`void`**`)` Inicializa el *timer*.

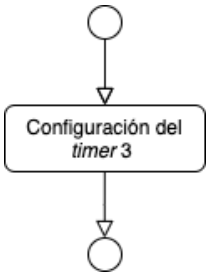

Figura 17: Diagrama de flujo de la función TIMER_init.

> Diagrama de flujo de la función `TIMER_init()`.

- **`void`** `TIMER_ISR(HardwareTimer*)` ISR para realizar la conversión ADC con la función `ADC_convert`.

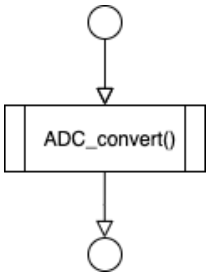

Figura 18: Diagrama de flujo de la función TIMER_ISR.

> Diagrama de flujo de la función `TIMER_ISR(HardwareTimer*)`.

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:

- ☐ **Decodificación ASCII** Decodifica el siguiente texto en ASCII a hexadecimal: “STM32F401”.
- ☐ **Sentencia *switch*** Describe con tus palabras el funcionamiento de la sentencia *switch*.
- ☐ **Parámetros de comunicación** Crea una tabla donde se indique el *baud rate*, número de bits de datos, paridad y bits de stop de la siguientes configuraciones: 115200 6E1, 4800 7E2, 1200 5O2, 19200 9N1 y 57600 8N2.
- ☐ **Velocidad de transmisión** ¿Cuánto tiempo tardaríamos en enviar 8 bytes en una comunicación serie asíncrona con una configuración 9600 8N1? ¿Y con una configuración 115200 8E2?

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.

1.5. Conclusiones

Tenemos clara una cosa: **ASCII es odiado en las comunicaciones serie** (vale, me he pasado, pero se debe de evitar). En la medida de lo posible, utilizaremos el envío de **datos en *raw* para evitar el *overhead*** añadido por la codificación ASCII. Hemos visto como enviar y recibir datos en *raw* con CoolTerm para poder **controlar las acciones del microcontrolador**. También hemos visto los aspectos básicos de una comunicación serie asíncrono y los diferentes **parámetros** que la configuran.

Por último, hemos visto un **problema asociado** con la comunicación serie asíncrona de **datos en *raw***: la gestión del *term char*. Hemos de evitar que el *term char* aparezca en los datos del mensaje. Esto lo veremos en la siguiente práctica donde trabajaremos el *framing* de paquetes de bytes.