MASB | STM32
Comunicación serie II

#stm32 #c #biomedical #microcontroladores #programacion

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

18 de marzo de 2020

Índice

1. Comunicación serie - Parte II

Empieza la segunda parte. Como siempre, ahora **haremos en STM32CubeIDE lo mismo que hicimos en Arduino IDE** en la primera parte. En este caso, **las funciones COBS de (de)codificación no tienen dependencias** (no utilizan librerías específicas de Arduino ni HAL de STM32F4). Por ello, las funciones son directamente compatibles entre Arduino y STM32CubeIDE y **la migración es directa**. Dado que ya hemos visto cómo implementar las funciones de (de)codificación COBS, **vamos a implementar esas funciones junto con la comunicación UART**. De este modo, estaremos más cerca de tener hecho lo que luego deberemos de implementar en el proyecto. Ya solo nos quedaría implementar un juego de instrucciones, pero qué es eso y cómo hacerlo lo veremos en la siguiente práctica.

Esta práctica la haremos **individualmente** y la dividiremos en **tres partes**. En la primera de ellas, prepararemos el proyecto y a ese proyecto **migraremos las funciones de (de)codificación COBS** que implementamos en la primera parte de la práctica. La segunda parte consistirá en **enviar un paquete codificado en COBS** a través de la UART cada vez que pulsemos el pulsador y ver lo enviado mediante CoolTerm. En esta segunda parte aprovecharemos para hablar sobre el ***endianness*** y las **uniones en C**. La tercera y última parte consistirá en **recibir un paquete de datos** des de CoolTerm y decodificarlo.

¡Empecemos!

1.1. Objetivos

- Envío de paquetes de datos codificados en COBS.
- Recepción de paquetes de datos codificados en COBS.
- Utilizar uniones en C.
- Distinguir/identificar *endiannless*.

1.2. Procedimiento

Nuestra aplicación final seguirá el diagrama de flujo de más abajo. Básicamente, cada vez que se pulse el pulsador se codificará en COBS un paquete de bytes (en este caso, un valor decimal almacenado en una variable `float`) y, a su vez, también estaremos pendientes de la recepción de paquetes de bytes codificados en COBS y los decodificaremos. Vamos a empezar preparando el proyecto.

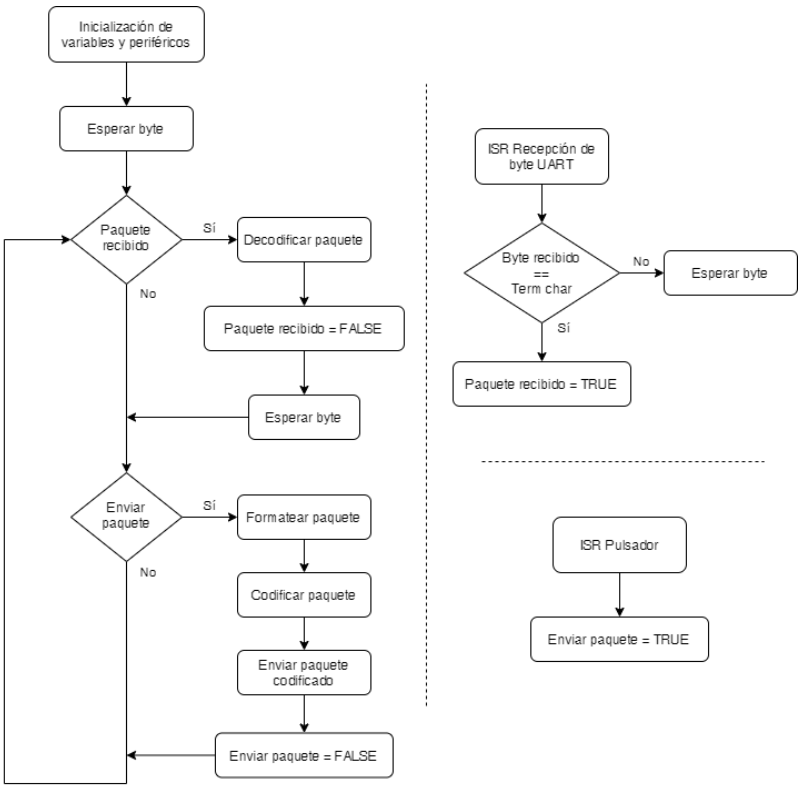

Figura 1: Diagrama de flujo de la aplicación.

1.2.1. Preparación del proyecto base

1.2.1.1. Creación y configuración del proyecto Vamos a crearnos una **rama para nuestro desarrollo** y, en esa rama, vamos a crear un **proyecto en STM32CubeIDE** con el **nombre** `masb-p06` en la **carpeta** `stm32cube` de nuestro repositorio local.

En ese nuevo proyecto, **vamos a dejar la configuración por defecto de los periféricos** y únicamente **activaremos las interrupciones del GPIO** del pulsador y de la **UART2**.

1.2.1.2. Migración de las funciones de (de)codificación Vamos a **crear las carpetas** `components` en las carpetas `Core` > `Inc` y `Core` > `Src`.

> ¿Por qué la llamamos `components`? No hay una razón específica. Podéis llamarla como queráis (por ejemplo, `programitas`). En este caso las llamamos `components` porque yo suelo llamarlas así debido a que programo de tal modo que cada archivo actúa como un componente individual

> que se encarga de una única función especifica dentro del programa. Pero no hay razón que prohíba llamar esas carpetas de otro modo.

En la carpeta `Core` > `Src` > `components` **creamos el archivo `cobs.c`**. En ese archivo, **añadimos las funciones que desarrollamos en la primera parte**. Una vez añadidas, haremos unas **ligeras modificaciones**. Simplemente, **modificamos los tipos de variables** para hacerlos coherentes con el resto de tipos de variables utilizados en el resto de archivos del proyecto en STM32CubeIDE. Esas modificaciones consisten en **sustituir** `unsigned long` por `uint32_t`, `byte` por `uint8_t` y `int` por `uint16_t`. Por último, **al inicio del archivo `cobs.c`**, añadimos un `include` **al archivo `components/cobs.h`** que crearemos a continuación. El archivo `cobs.c` quedaría del siguiente modo (he eliminado los comentarios para ahorrar espacio en el informe, ¡pero dejad los comentarios que hayáis hecho en el código!):

```c
#include "components/cobs.h"

// codifica un mensaje en COBS
uint32_t COBS_encode(uint8_t *decodedMessage, uint32_t lenght, uint8_t
    *codedMessage) {

  uint32_t read_index = 0,
                write_index = 1,
                code_index = 0;

  uint8_t code = 0x01;

  while(read_index < lenght) {
    if (decodedMessage[read_index] == 0x00) {
      codedMessage[code_index] = code;
      code_index = write_index;
      write_index++;
      code = 0x01;
    } else {
      codedMessage[write_index] = decodedMessage[read_index];
      write_index++;
      code++;
      if (code == 0xFF) {
        codedMessage[code_index] = code;
        code_index = write_index;
        write_index++;
        code = 0x01;
      }
    }
    read_index++;
  }
  codedMessage[code_index] = code;

  return write_index;
}
```

```
35
36 // decodifica un mensaje en COBS
37 uint32_t COBS_decode(uint8_t *codedMessage, uint32_t lenght, uint8_t *
       decodedMessage) {
38
39   uint32_t read_index = 0,
40            write_index = 0;
41
42   while(read_index < lenght) {
43     uint8_t code = codedMessage[read_index];
44     read_index++;
45     for (uint16_t i = 1; i < code; i++) {
46       decodedMessage[write_index] = codedMessage[read_index];
47       write_index++;
48       read_index++;
49     }
50     if (code < 0xFF && read_index < lenght) {
51       decodedMessage[write_index] = 0x00;
52       write_index++;
53     }
54   }
55
56   return write_index;
57 }
```

Ahora vamos a **crear el archivo `cobs.h`** en la carpeta `Core > Inc > components`. Recordemos, que a diferencia que en Arduino, hemos de crear un archivo de cabecera o *header file* para prototipar allí las funciones del archivo `cobs.c` que queramos que estén disponibles des de otros archivos. El archivo `cobs.h` tomaría la siguiente forma:

```
1  #ifndef INC_COMPONENTS_COBS_H_
2  #define INC_COMPONENTS_COBS_H_
3
4  // Include de los header files del microcontrolador.
5  #include "stm32f4xx_hal.h"
6
7  // Prototipos de funciones (funciones disponibles en el resto de
       archivos que hagan un include de este archivo).
8  uint32_t COBS_encode(uint8_t *decodedMessage, uint32_t lenght, uint8_t
       *codedMessage);
9  uint32_t COBS_decode(uint8_t *codedMessage, uint32_t lenght, uint8_t *
       decodedMessage);
10
11 #endif /* INC_COMPONENTS_COBS_H_ */
```

Podéis ver cómo también **hemos añadido un `include` al archivo `stm32f4xx_hal.h`** donde se definen los tipos de variables utilizados (`uint8_t`, `int8_t`, `uint16_t`, etc.).

Con esto, ya hemos migrado nuestras funciones de Arduino a STM32Cube.

1.2.1.3. Preparación del stm32main

Vamos a replicar la filosofía que aplicamos en la práctica anterior creando las funciones `setup` y `loop` en un archivo `stm32main.c` para imitar lo que hace Arduino y así evitar tener que ir toqueteando el archivo `main.c`.

Primeramente, **creamos el archivo `stm32main.c`** en la carpeta `Core` > `Src` > `components`. En ese archivo **añadimos las definiciones de las funciones `setup` y `loop`** que, de momento, estarán vacías. Como siempre, **hacemos un `include` al archivo de cabecera de este archivo**. El archivo `stm32main.c` quedaría del siguiente modo:

```c
#include "components/stm32main.h"

// Funcion ejecutada antes del while loop (solo se ejecuta una vez).
void setup(void) {

}

// Funcion ejecutada en el while loop.
void loop(void) {

}
```

Seguidamente, **creamos el archivo de cabecera `stm32main.h`** en la carpeta `Core` > `Inc` > `components`. En ese archivo, **hacemos un `include`** tanto del archivo `stm32f4xx_hal.h` como del archivo `cobs.h`. Haciendo este último `include`, el compilador no se quejará si utilizamos en el archivo `stm32main.c` alguna de las funciones del archivo `cobs.c`. En este archivo, ya solo falta **añadir los prototipos de las funciones** `setup` y `loop`. Esta sería la pinta del archivo `stm32main.h`.

```c
#ifndef INC_COMPONENTS_STM32MAIN_H_
#define INC_COMPONENTS_STM32MAIN_H_

#include "stm32f4xx_hal.h" // Include de los header files del
    microcontrolador.
#include "cobs.h" // Include para tener acceso a las funciones del
    archivo cobs.c.

// Prototipos de funciones (funciones disponibles en el resto de
    archivos que hagan un include de este archivo).
void setup(void);
void loop(void);

#endif /* INC_COMPONENTS_STM32MAIN_H_ */
```

1.2.1.4. Preparación del main

Ya solo queda **añadir las funciones `setup` y `loop`**, que de momento no hacen nada, **a la función `main` del archivo `main.c` y añadir el `include` al archivo `components/stm32main.h`** para que el compilador no se queje. El archivo quedaría del siguiente modo:

```
...

/* Private includes
   ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "components/stm32main.h"
/* USER CODE END Includes */

...

/* USER CODE BEGIN 2 */
  setup(); // Funcion que se ejecutara una sola vez.
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      loop(); // Funcion que se ejecutara continuamente.
    /* USER CODE END WHILE */

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

...
```

Ya hemos migrado las funciones de (de)codificación COBS de Arduino a STM32CubeIDE y tenemos el proyecto base preparado para la siguiente parte.

1.2.2. Envío de un paquete de bytes codificado

En esta parte vamos a hacer que cada vez que pulsemos el pulsador se envíe un valor numérico decimal almacenado en un **`float`** por UART habiéndolo codificado previamente en COBS.

1.2.2.1. Uniones en C Para realizar ese envío, vamos a hacer uso de las uniones en C. ¿Por qué? Os hago una pregunta: en los paquetes de bytes solo pueden enviarse bytes (obvio), ¿cómo envías una variable que ocupa cuatro bytes? Fácil, uno a uno. La cosa está en cómo lograr pasar un **`float`** de cuatro bytes a un *array* de bytes de cuatro elementos. Con las uniones se torna fácil. Vamos a seguir el camino a la inversa y vamos a ver primero el código y luego lo explicamos.

```
union floatBytesConverter_U{
  float f;
  char b[4];
} floatBytesConverter;
```

Una **unión** no es más que una **estructura de almacenamiento de datos** donde **todas las variables que la forman (miembros) comparten el mismo espacio de memoria**. Ese espacio de memoria tendrá el tamaño necesario para poder almacenar el miembro de mayor tamaño de la unión. En el ejemplo de arriba, dentro de una unión tipo `floatBytesConverter_U` llamada `floatBytesConverter`, se define una variable **`float`** llamada `f` y un *array* tipo **`char`** llamado `b`, y ambas variables ocupan el mismo espacio en memoria. ¿Qué supone eso? Pues que si yo escribo en la variable `floatBytesConverter.f` y luego escribo en la variable `floatBytesConverter.b` estaré sobreescribiendo el valor previamente escrito, y viceversa. Pero también supone que si yo escribo un valor en la variable `floatBytesConverter.f`, luego puedo acceder/leer ese espacio de memoria mediante `floatBytesConverter.b` y extraer su contenido como un *array* tipo **`char`**. Es como tener variables de diferente tipo con un mismo puntero. Podéis encontrar más información sobre la unión en C aquí.

Así pues, para realizar una conversión de un **`float`** a un *array* de cuatro elementos, podemos hacer:

```
1  float decimalNumber = 34.52;
2  char floatBytes[4] = { 0 };
3
4  union floatBytesConverter_U{
5    float f;
6    char b[4];
7  } floatBytesConverter;
8
9  floatBytesConverter.f = decimalNumber;
10
11 floatBytes[0] = floatBytesConverter.b[0]; // BYTE0
12 floatBytes[1] = floatBytesConverter.b[1]; // BYTE1
13 floatBytes[2] = floatBytesConverter.b[2]; // BYTE2
14 floatBytes[3] = floatBytesConverter.b[3]; // BYTE3
```

Una cosa que nos queda pendiente de ver/conocer es: ¿el byte 0… es el MSB (*Most Significant Byte*) o el LSB (*Least Significant Byte*)? Es importante saberlo de cara a luego seguir el convenio que establezcamos en la comunicación serie. Ese convenio fijará si enviamos primero los MSB o los LSB. Estamos hablando del *endianness*.

1.2.2.2. *Endianness*

El ***endianness* no es más el orden en el que se almacenan los bytes de una variable en la memoria**. En *little endian*, el byte menos significativo de una variable se almacena en el espacio de memoria con la dirección de memoria más pequeña. *Big endian*, justo lo contrario. El byte más significativo se almacena en la posición de memoria con la dirección de memoria más pequeña. Si no os ha quedado claro, podéis ampliar información aquí.

En nuestro caso, podemos comprobar, en el manual de referencia de nuestro microcontrolador (página 38), que nuestro microcontrolador almacena los datos en formato *little endian*.

1.2.2.3. Envío de un *float* Ahora ya sabemos cómo hacer una conversión de *floats* a *arrays* con uniones. También sabemos cómo operar con la UART de nuestro microcontrolador con las HAL para enviar un paquete de datos. Las interrupciones de nuestros GPIOs: controladas. Y hemos migrado las funciones COBS para codificar un paquete antes de enviarlo. Lo tenemos todo.

Primeramente, vamos a crear el *callback* de las interrupciones del pulsador. Cuando se pulse, se alternará el valor de una variable de bandera llamada `sendData`.

> Una variable de bandera sirve para almacenar un estado. En este caso, utilizaremos una bandera para almacenar el evento "pulsador pulsado".

También crearemos dos *buffers*/*arrays*. Uno para almacenar el mensaje que queremos enviar y otro para almacenar ese mensaje codificado en COBS y que será lo que finalmente enviaremos. El código de nuestro archivo `stm32main.c` quedaría del siguiente modo (tenéis los detalles de la implementación el propio código):

```
1  #include "components/stm32main.h"
2
3  /*
4   * Declaracion de una variable externa.
5   * De este modo, se le dice al compilar que en algun lado de nuestro
       programa existe
6   * una variable llamada huart2 de tipo UART_HandleTypeDef que podemos
       usar. (Esta
7   * definida/declarada en el archivo main.c). Es una manera de acceder a
       variables
8   * de otros archivos haciendo "trampas".
9   */
10 extern UART_HandleTypeDef huart2;
11
12 _Bool sendData     = FALSE; // Bandera que indica que se quiere enviar
       un paquete.
13
14 uint8_t txBuffer[UART_BUFFER_SIZE]  = { 0 }, // Buffer de transmision
       de la UART.
15          txDecoded[UART_BUFFER_SIZE] = { 0 }; // Buffer con el paquete
               decodificado a transmitir.
16
17 uint32_t txEncodedLenght = 0; // Tamano del buffer codificado a
       transmitir.
18
19 // Funcion ejecutada antes del while loop (solo se ejecuta una vez).
20 void setup(void) {
21
22 }
23
24 // Funcion ejecutada en el while loop.
25 void loop(void) {
```

```
26
27      // Si se quiere enviar un paquete...
28      if (sendData) {
29
30          // Ejemplo float a char/byte array:
31          // float = 4 bytes ( BYTE3 | BYTE2 | BYTE1 |BYTE0 )
32          float decimalNumber = 34.52;
33
34          // Una union comparte el mismo espacio de memoria.
35          union floatBytesConverter_U{
36              float f;
37              char b[4];
38          } floatBytesConverter;
39
40          // Escribimos en el espacio de memoria de la union como float.
41          floatBytesConverter.f = decimalNumber;
42
43          // Leemos el espacio de memoria de la union como char array.
44          // Sabemos el orden de los bytes (endian) porque lo indica el
                manual de referencia del microcontrolador.
45          txDecoded[0] = floatBytesConverter.b[0]; // BYTE0
46          txDecoded[1] = floatBytesConverter.b[1]; // BYTE1
47          txDecoded[2] = floatBytesConverter.b[2]; // BYTE2
48          txDecoded[3] = floatBytesConverter.b[3]; // BYTE3
49
50          txEncodedLenght = COBS_encode(txDecoded, 4, txBuffer); //
                Codificamos el paquete a enviar. Guardamos el tamano del
                buffer resultante.
51          txBuffer[txEncodedLenght] = UART_TERM_CHAR; // Anadimos el term
                 char
52          txEncodedLenght++; // Los indices de los vectores se incian en
                0. Incrementamos en 1 para lograr el largo.
53          HAL_UART_Transmit_IT(&huart2, txBuffer, txEncodedLenght); //
                Enviamos
54
55          sendData = FALSE; // Desactivamos la bandera.
56      }
57
58  }
59
60  // ISR del pulsador.
61  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
62
63      sendData = TRUE; // Activamos la bandera para enviar un paquete.
64
65  }
```

Hay un par de *defines*/macros que los definimos en el archivo `stm32main.h`. Esas macros son `UART_BUFFER_SIZE` y `UART_TERM_CHAR`. El primero lo he definido como 128 y el segundo como `0x00`.

Compilamos y depuramos. Abrimos CoolTerm, si no lo teníamos abierto ya, y configuramos la conexión en 115200 8N1 y conectamos. Pulsamos el pulsador del microcontrolador y comprobamos que recibimos los datos esperados. ¿Cómo podéis comprobarlo? Utilizad los *breakpoints* y comprobad que los pasos se realizan correctamente viendo los valores que toma el *buffer* de antes y después de la codificación.

1.2.3. Recepción de paquetes de bytes codificados

Ahora vamos a preparar nuestro microcontrolador para que pueda recibir paquetes de bytes codificados en COBS. Una vez recibidos, los decodificaremos. Utilizaremos, una vez más, una variable de bandera llamada `dataReceived`. Esa bandera la activaremos al recibir un *term char*. Este caso es mucho más sencillo que el anterior y comentamos el funcionamiento directamente en el código. Dejo solo los comentarios de lo que hemos añadido:

```
1  #include "components/stm32main.h"
2
3  extern UART_HandleTypeDef huart2;
4
5  _Bool sendData     = FALSE;
6  _Bool dataReceived = FALSE; // Bandera que indica que se ha recibido un
       paquete.
7
8  uint8_t rxBuffer[UART_BUFFER_SIZE]  = { 0 }, // Buffer de recepcion de
       la UART.
9          rxDecoded[UART_BUFFER_SIZE] = { 0 }; // Buffer con el paquete
               receibido decodificado.
10
11 uint32_t rxDecodedLenght = 0; // Tamano del buffer codificado a
       transmitir.
12
13 uint8_t txBuffer[UART_BUFFER_SIZE]  = { 0 },
14         txDecoded[UART_BUFFER_SIZE] = { 0 };
15
16 uint32_t txEncodedLenght = 0;
17
18 uint32_t rxIndex = 0; // Indice del buffer de recepcion de la UART.
19
20 // Funcion ejecutada antes del while loop (solo se ejecuta una vez).
21 void setup(void) {
22
23     HAL_UART_Receive_IT(&huart2, &rxBuffer[rxIndex], 1); // Esperamos
           la recepcion del primer byte.
24     (void)rxDecodedLenght; // "Truco" para hacer que el compilador no
           elimine esta variable puesto que nunca la usamos. Recordad de
           practicas anteriores.
25 }
26
```

```
27 // Funcion ejecutada en el while loop.
28 void loop(void) {
29
30     // Si se ha recibido un paquete...
31     if (dataReceived) {
32
33         rxDecodedLenght = COBS_decode(rxBuffer, rxIndex, rxDecoded); //
               Decodificamos el paquete.
34
35         // Aqui pondriamos todo lo que deberia de realizar el
              microcontrolador en funcion del paquete recibido:
36         // obedecer ordenes, configurar variables, etc.
37         // En esta practica no hacemos nada.
38
39         rxIndex = 0; // Reiniciamos puntero.
40
41         dataReceived = FALSE; // Desactivamos la bandera.
42         HAL_UART_Receive_IT(&huart2, &rxBuffer[rxIndex], 1); //
              Volvemos a activar la recepcion.
43     }
44
45     if (sendData) {
46
47         float decimalNumber = 34.52;
48
49         union floatBytesConverter_U{
50             float f;
51             char b[4];
52         } floatBytesConverter;
53
54         floatBytesConverter.f = decimalNumber;
55
56         txDecoded[0] = floatBytesConverter.b[0];
57         txDecoded[1] = floatBytesConverter.b[1];
58         txDecoded[2] = floatBytesConverter.b[2];
59         txDecoded[3] = floatBytesConverter.b[3];
60
61         txEncodedLenght = COBS_encode(txDecoded, 4, txBuffer);
62         txBuffer[txEncodedLenght] = UART_TERM_CHAR;
63         txEncodedLenght++;
64         HAL_UART_Transmit_IT(&huart2, txBuffer, txEncodedLenght);
65         sendData = FALSE;
66     }
67 }
68
69 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
70
71     sendData = TRUE;
72
73 }
74
```

```
75 // ISR para la recepcion de 1 byte por UART.
76 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
77 {
78
79     if (rxBuffer[rxIndex] == UART_TERM_CHAR) { // Si hemos recibido el
       term char...
80         dataReceived = TRUE; // Activamos la bandera que indica la
           recepcion de un paquete.
81     } else { // Si no...
82         rxIndex++; // Incrementamos el indice del buffer de recepcion
           ...
83         HAL_UART_Receive_IT(&huart2, &rxBuffer[rxIndex], 1); // y
           esperamos recibir un byte mas.
84     }
85 }
```

Compilamos e iniciamos el programa en el microcontrolador. Vamos a CoolTerm y enviamos en hexadecimal un mensaje codificado en COBS (por lo que no puede tener bytes aleatorios ni contener `0x00`) y añadimos el *term char* final `0x00`. Usad *breakpoints* para ver qué se ha recibido en el microcontrolador y el mensaje decodificado.

Con este último caso, ya hemos implementado el diagrama de flujo del inicio del guion.

1.3. Reto

No hay reto.

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**~~ No hay 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.

El informe debe de contener:

- ☐ **Uniones en C:** Explica **con tus propias palabras** qué son las uniones en C y cómo podemos usarlas para realizar las conversiones.
- ☐ ***Endianness***: Explica **con tus propias palabras** qué es el *endianness* y sus dos opciones: *big* y *little endian*.

1.4.2. Pull Request

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.

1.5. Conclusiones

En esta práctica hemos implementado la **codificación COBS junto** al **envío** y **recepción** de paquetes de bytes con la UART. Para ello, hemos visto qué son y cómo utilizamos **las uniones en C** para realizar conversiones de *floats* a *array*. También hemos visto qué es el ***endianness*** y cómo afecta al almacenamiento de los datos en memoria.

En la **siguiente práctica**, ya finalizaremos la comunicación UART. **Implementaremos un juego de instrucciones**. Ya veremos lo que es.