MASB | Arduino
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

Henos aquí en una nueva práctica sobre comunicaciones serie. De la práctica anterior **tenemos clarísimo en qué consiste una comunicación serie asíncrona**, **qué es la codificación ASCII** y **qué significa el envío de datos en *raw* y sus beneficios**.

> ¿Qué? ¿Cómo? ¿Que no lo tienes claro? Ya estás volviendo inmediatamente a la práctica anterior a revisarla y a volverla a hacer si hace falta. No dudes tampoco en **preguntar lo que sea necesario con tal de entender completamente todo lo visto** hasta ahora sobre comunicación serie. Piensa que este es un apartado **muy importante en el ámbito de los microcontroladores** y que todo lo que veamos en esta práctica, y la siguiente, se apoyará en los conceptos de la práctica anterior. **¡Asegúrate de haber integrado todos los conceptos vistos!**

Un aspecto muy importante que vimos sobre la comunicación serie es el **uso de *term chars* para indicar el final de un mensaje o instrucción**. En codificación ASCII, utilizamos dos *term chars*: el retorno de carro (`\r` o `0x0D` en *raw*) y el carácter de línea nueva (`\n` o `0x0A` en *raw*). De este modo, el terminal serie de Arduino sabía cuándo finalizaba una instrucción o mensaje para proceder a dibujar los datos en el *Serial Plotter* o mostrar la instrucción o mensaje en una nueva línea en el *Serial Monitor*.

En ***raw*** vimos que la cosa se complicaba. ¿Qué carácter fijamos como *term char*? Y más importante: una vez fijado el *term char* (`0x00` por ejemplo), ¿cómo nos aseguramos que ese carácter/byte no aparece en el cuerpo del mensaje o instrucción y es detectado como un **falso *term char***? Un ejemplo. Imaginaros que fijamos el *term char* como `0x00` y que el mensaje que queremos enviar es `0xA1` `0x12` `0x01` `0xFF` `0x00` `0x23` `0x32` `0xAF` `0x05`. Añadiendo el *term char*, la instrucción/mensaje quedaría así:

0xA1 0x12 0x01 0xFF **0x00** 0x23 0x32 0xAF 0x05 **0x00**

¿Cómo sabe el microcontrolador o el ordenador que la instrucción acaba en el último `0x00` y no en el primero? (Pensad que antes y después de esta instrucción estarían los bytes de otros mensajes ya enviados y por enviar.) La solución es **codificar el mensaje para que en él no aparezca el carácter que hayamos fijado como *term char***. En este caso, evitar que se envíe un `0x00`.

Existen infinidad de este tipo de protocolos de codificación y escogeremos uno u otro en función de, entre otros criterios, el ***overhead*** (o número de bytes añadidos para realizar la codificación). En esta

asignatura utilizaremos el **protocolo COBS (*Consistent Overhead Byte Stuffing*) por su sencillez y bajo *overhead***.

En la práctica veremos directamente cómo implementar el COBS, pero **os daré la receta**. La explicación completa de **su implementación la tenéis que leer directamente de un artículo de sus creadores Stuart Cheshire y Mary Baker**. Nuestro amigo Stuart ha colgado su artículo en su página web personal, así que tenemos acceso a su *paper by the face*. **Lo podéis encontrar aquí**. Leerse **el *paper* es parte de la práctica** y deberéis de contestar *un petit questionnaire* en el report de la práctica. Los autores han hecho un magnifico trabajo y esfuerzo articulando un escrito que evita florituras técnicas, por lo que, aunque obviamente es un *paper* sobre ingeniería y habrá tecnicismos, **una persona que no sea ingeniero informático puede comprender de qué se está hablando perfectamente**.

> **Voy a repetirlo. Ejem, ejem. "En el guion solo pondré la receta. En el *paper*, tenéis que leer la explicación del algoritmo".**

¿Cómo haremos nuestra práctica en pijama desde nuestros hogares? **Cada miembro del equipo hará trabajos distintos**, pero relacionados. El **miembro A** creará una **función para codificar** un paquete de bytes en COBS. El **miembro B** creará una **función para decodificar** un paquete de bytes en COBS. Repito, uno se encarga de la codificación y otro de la decodificación. Sencillo, ¿verdad?

Bien. Para **comprobar que hemos implementado correctamente la (de)codificación** en COBS, implementaremos una serie de **tests unitarios** de una manera simple y rudimentaria. **Los tests unitarios son programas que llaman a nuestras funciones y comprueban su correcto funcionamiento**. La manera usual de implementarlos es dar a la función a testear un valor y comprobar qué devuelve. El valor que se le envía a la función a testear se llama **"vector de test"**. Puesto que conocemos el vector de test ya que nosotros lo fijamos a voluntad, **conocemos qué nos debe de devolver la función de antemano**. **Comparando ese valor conocido de antemano con lo que realmente nos devuelve la función** a testear, podemos validar el correcto funcionamiento de la función.

Entre infinidad de opciones, hay **dos filosofías** en la implementación de tests unitarios: 1) **uno mismo** se implementa los tests unitarios de sus funciones, o 2) **un equipo** se encarga de programar y **un segundo** equipo se encarga solo de implementar los tests unitarios. Cada filosofía tiene sus ventajas y desventajas, que quedan fuera del *scope* de la asignatura. En esta parte de la práctica os doy los tests hechos para que veáis un ejemplo. No tendréis que hacerlos. **Pero mirároslos y haced un esfuerzo para entenderlos bien.**

Todo lo haremos de manera muy pautada y guiada. Así que sentémonos, hagamos la práctica y a disfrutar.

¡Al ataque!

1.1. Objetivos

- (De)codificación de paquetes de bytes en COBS.
- Primer contacto con tests unitarios.
- Resolución de conflictos en git. (Sí, habrá conflictos.)

1.2. Procedimiento

¿Os habéis leído ya el *paper*? Anda. Leedlo primero.

> Super importante entender la codificación COBS explicada en el *paper*. Si realizáis la práctica directamente sin leer/entender el *paper*, todo os sonará a chino. Avisados…

Ahora sí. Vamos primero a desgranar cuál será la **estructura de carpetas/archivos** a seguir, **qué deberá de hacer cada miembro** del equipo, cuál será la **nomenclatura establecida** para nuestras funciones y cuál será el ***workflow* en git**.

> Cómo siempre, leed también las tareas de vuestro compañero para saber qué hace y no perderos ningún aspecto de la práctica.

1.2.1. Estructura de directorios/archivos

La estructura de directorios os la encontráis ya hecha, junto con los archivos a desarrollar. La estructura será la siguiente:

```
arduino
- README.md
- QUESTIONS.md
- masb-p06
  - cobs.ino
  - masb-p06.ino
  - test.ino
```

1.2.2. Tareas de cada miembro

Los dos miembros trabajaréis sobre el archivo `cobs.ino`. En algún momento se os pedirá modificar el archivo `masb-p06.ino` para que podáis ejecutar los tests, pero los cambios no debéis de guardarlos en git. **Solo tenéis que hacer *add*/*commit* del archivo `cobs.ino`.**

Tareas:

- El **miembro A** desarrollará la **función que codificará** los datos *raw* en COBS.
- El **miembro B** desarrollará la **función que decodificará** los datos en COBS a *raw*. También se encargará de inicializar el árbol git creando la rama `develop` desde la rama `master`.

1.2.3. Nomenclatura

- **COBS_encode**

Parámetro	Valor
Nombre de la función	COBS*encode
Descripción de la función	Codifica un _buffer* de datos *raw* en COBS y lo almacena en un segundo *buffer*.
Parámetros	**byte *decodedMessage:** Puntero al *buffer* donde están los datos a codificar. **unsigned long lenght:** Número de elementos que contiene el *buffer* a codificar. **byte *codedMessage:** Puntero al buffer donde se guardarán los datos codificados.
Devuelve	**unsigned long:** Número de elementos que forman el *buffer* codificado.

- **COBS_decode**

Parámetro	Valor
Nombre de la función	COBS_decode
Descripción de la función	Decodifica un *buffer* de datos en COBS y lo almacena en un segundo *buffer*.
Parámetros	**byte *codedMessage:** Puntero al *buffer* donde están los datos a decodificar. **unsigned long lenght:** Número de elementos que contiene el *buffer* a decodificar.
Devuelve	**unsigned long:** Número de elementos que forman el *buffer* decodificado.

1.2.4. *Workflow* en git

Para el desarrollo de la práctica utilizaremos el *workflow* de la práctica anterior:

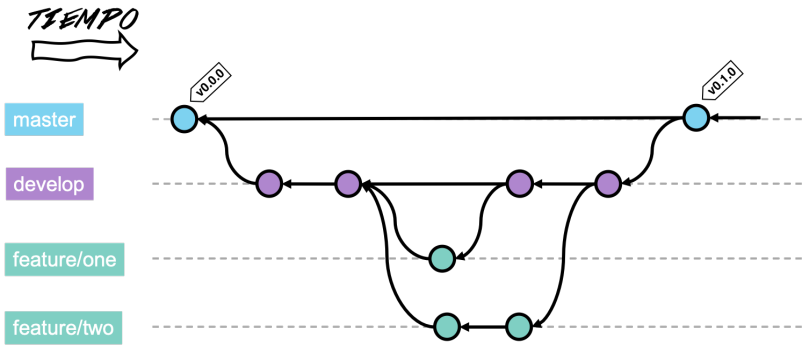

Figura 1: Git tree del repositorio.

- **master:** rama que contiene el código de producción. Se entiende como código de producción aquel que se le puede entregar a un cliente y funciona 100 %.
- **develop:** rama que contiene nuestro desarrollo. En esta rama se agrupan todos los desarrollos de los miembros del equipo y se testean. Una vez validado su correcto funcionamiento, los contenidos de la rama `develop` se vuelcan a la rama `master` para entregar al cliente.
- **feature/***:** rama que contiene el desarrollo individual o colectivo de una funcionalidad. Los asteriscos deben de ser sustituidos por un término descriptivo de la funcionalidad desarrollada en esa rama. Los contenidos de esa rama se vuelcan en la rama `develop` una vez han sido testeados.

El **miembro `A`** trabajará sobre una **rama llamada `feature/cobs-encode`** que debe de **nacer de la rama `develop` que creará el miembro `B`**.

El **miembro `B`** trabajará sobre una **rama llamada `feature/cobs-decode`** que debe de **nacer de la rama `develop` que creará el mismo**.

Una vez todas las funcionalidades deseadas se han incorporado a la rama `develop` y se ha comprobado que **todo funciona 100 %**, se hace un *Pull Request* a la rama `master` y se envía al cliente (a mí).

1.2.5. *Workflow* del miembro A

1.2.5.1. Creación de la rama *feature* Vamos a **crear nuestra rama de desarrollo `feature/cobs-encode`**, pero esta debe de **nacer de la rama `develop`** que debe de **crear nuestro compañero**. Si aún no la ha creado, ¡fríele el móvil a llamadas para que la cree y puedas empezar! Una vez exista la rama `develop` hacemos:

```
git checkout develop
git pull
git checkout -b feature/cobs-encode
```

Ya tenemos nuestra rama y podemos empezar.

1.2.5.2. Desarrollo de la función COBS_encode

No te voy a engañar. Te ha tocado la función "difícil". Pongo entre comillas lo de "difícil" porque en realidad… difícil no es, pero si no tienes claro cómo se realiza la codificación COBS, te será imposible entender qué estamos haciendo (no sabrás qué es un *code byte*, qué es un *code block*, etc.). Por ello, **si ves que te pierdes, vuelve al *paper*** e intenta darle una vuelta a cómo se realiza la codificación.

La función la implementaremos **en el archivo `cobs.ino`**. Lo abrimos desde Arduino IDE. De la tabla del apartado Nomenclatura, sabemos cómo se llamará la función y sus parámetros. Podemos empezar por ello. El código del archivo `cobs.ino` tomaría la siguiente forma:

```
// codifica un mensaje en COBS
unsigned long COBS_encode(
  byte *decodedMessage, // puntero al buffer donde estan los datos a codificar
  unsigned long lenght, // numero de elementos que contiene el mensaje a codificar
  byte *codedMessage // puntero al buffer donde se guardaran los datos codificados
  ) {

}
```

> Como ves, he puesto los parámetros en líneas distintas para hacer la función más legible.

Los comentarios son bastante claros sobre qué es cada parámetro. Si tienes dudas sobre cómo operar con punteros/*arrays*, **puedes ver más info aquí.** (Échale un vistazo a esta página y mírate también los conceptos que aparecen en la tabla al final de la misma. Te ayudará a entender mejor cómo operar con los *arrays*.)

> ¿Te has mirado ya la página que te acabo de recomendar? Si no lo hacéis, lo echaréis en falta en un par de párrafos más abajo.

Bien. Empezaremos creando tres variables. Las tres son índices para los *arrays* `decodedMessage` y `codedMessage`. El **índice `read_index` lo utilizamos para acceder al elemento del *array* `decodedMessage` que estamos leyendo**, el **puntero `code_index` indica la posición donde se guardará el *code byte* en el *array* `codedMessage`**, y el **puntero `write_index` indica la posición en**

el *array* `codedMessage` donde se guardarán los bytes del *code block* que vayamos leyendo del `decodedMessage`.

La **variable** `code` la utilizaremos para **calcular el *code byte***, que no es más que un **contador**.

> Vuelvo a hacer **hincapié** en que **no explicaré el algoritmo de codificación COBS**. Eso **está en el *paper***. Por ejemplo, el `code byte` al que hago referencia, se explica en el *paper* que indica el número de bytes consecutivos hasta encontrar un `0x00` (este último incluido).
>
> **Esta aclaración es de regalo para que veáis que hay que leerse/entender el *paper* antes de ponerse con el código.**

El código quedaría del siguiente modo:

```
1  // codifica un mensaje en COBS
2  unsigned long COBS_encode(
3    byte *decodedMessage, // puntero al buffer donde estan los datos a
       codificar
4    unsigned long lenght, // numero de elementos que contiene el mensaje
       a codificar
5    byte *codedMessage // puntero al buffer donde se guardaran los datos
       codificados
6    ) {
7
8    unsigned long read_index = 0, // indice al elemento del buffer sin
       codificar que estamos leyendo
9                  write_index = 1, // indice al elemento del buffer
                     codificado donde escribiremos la codificacion
10                 code_index = 0; // indice al elemento del buffer
                     codificado donde escribiremos el code byte
11
12   byte code = 0x01; // code byte
13
14 }
```

Ahora vamos a implementar un **bucle *while*** de tal modo que este **se ejecute hasta que finalicemos de procesar todos los elementos** del *buffer* a codificar.

```
1  // codifica un mensaje en COBS
2  unsigned long COBS_encode(
3    byte *decodedMessage, // puntero al buffer donde estan los datos a
       codificar
4    unsigned long lenght, // numero de elementos que contiene el mensaje
       a codificar
5    byte *codedMessage // puntero al buffer donde se guardaran los datos
       codificados
6    ) {
7
```

```
8   unsigned long read_index = 0, // indice al elemento del buffer sin
        codificar que estamos leyendo
9                 write_index = 1, // indice al elemento del buffer
                      codificado donde escribiremos la codificacion
10                code_index = 0; // indice al elemento del buffer
                      codificado donde escribiremos el code byte
11
12  byte code = 0x01; // code byte
13
14  while(read_index < lenght) { // mientras no finalizamos la conversion
        de los paquetes...
15
16    read_index++; // incrementamos el indice al buffer de datos a
          codificar
17
18  }
19 }
```

En cada iteración de ese bucle *while* comprobaremos el valor del byte que estamos tratando (apuntado por el índice `read_index`). Si el byte sin codificar es `0x00`, hemos encontrado un *term char* que hay que hacer que desaparezca. Si no es `0x00`, seguimos con la búsqueda.

```
1  // codifica un mensaje en COBS
2  unsigned long COBS_encode(
3    byte *decodedMessage, // puntero al buffer donde estan los datos a
       codificar
4    unsigned long lenght, // numero de elementos que contiene el mensaje
       a codificar
5    byte *codedMessage // puntero al buffer donde se guardaran los datos
       codificados
6    ) {
7
8    unsigned long read_index = 0, // indice al elemento del buffer sin
       codificar que estamos leyendo
9                 write_index = 1, // indice al elemento del buffer
                      codificado donde escribiremos la codificacion
10                code_index = 0; // indice al elemento del buffer
                      codificado donde escribiremos el code byte
11
12   byte code = 0x01; // code byte
13
14   while(read_index < lenght) { // mientras no finalizamos la conversion
         de los paquetes...
15
16     if (decodedMessage[read_index] == 0x00) { // si encontramos un 0x00
           ...
17
18     } else { // si no ...
19
20     }
```

```
21
22      read_index++; // incrementamos el indice al buffer de datos a
           codificar
23
24    }
25  }
```

Al encontrar un `0x00`, finalizamos la codificación del *code block* e iniciamos la codificación del siguiente.

```
1  // codifica un mensaje en COBS
2  unsigned long COBS_encode(
3    byte *decodedMessage, // puntero al buffer donde estan los datos a
        codificar
4    unsigned long lenght, // numero de elementos que contiene el mensaje
        a codificar
5    byte *codedMessage // puntero al buffer donde se guardaran los datos
        codificados
6    ) {
7
8    unsigned long read_index = 0, // indice al elemento del buffer sin
        codificar que estamos leyendo
9                  write_index = 1, // indice al elemento del buffer
                      codificado donde escribiremos la codificacion
10                 code_index = 0; // indice al elemento del buffer
                      codificado donde escribiremos el code byte
11
12   byte code = 0x01; // code byte
13
14   while(read_index < lenght) { // mientras no finalizamos la conversion
         de los paquetes...
15
16     if (decodedMessage[read_index] == 0x00) { // si encontramos un 0x00
           ...
17
18       // finalizamos bloque
19       codedMessage[code_index] = code; // guardamos el codigo en el
            buffer codificado
20       code_index = write_index; // apuntamos al siguiente byte del
            buffer codificado donde guardaremos el codigo
21
22       // empezamos un bloque
23       write_index++; // incrementamos el indice de codedMessage ya que
            la primera posicion sera para el codigo
24       code = 0x01; // reinicializamos el codigo/contador
25
26     } else { // si no ...
27
28     }
29
30     read_index++; // incrementamos el indice al buffer de datos a
```

```
          codificar
31
32    }
33  }
```

Si no encontramos un `0x00`, vamos al siguiente elemento e incrementamos el *code byte*. Si este toma el valor de `0xFF` finalizamos el *code block* e iniciamos el siguiente.

```
1  // codifica un mensaje en COBS
2  unsigned long COBS_encode(
3    byte *decodedMessage, // puntero al buffer donde estan los datos a
       codificar
4    unsigned long lenght, // numero de elementos que contiene el mensaje
       a codificar
5    byte *codedMessage // puntero al buffer donde se guardaran los datos
       codificados
6    ) {
7
8    unsigned long read_index = 0, // indice al elemento del buffer sin
       codificar que estamos leyendo
9                  write_index = 1, // indice al elemento del buffer
                     codificado donde escribiremos la codificacion
10                 code_index = 0; // indice al elemento del buffer
                     codificado donde escribiremos el code byte
11
12   byte code = 0x01; // code byte
13
14   while(read_index < lenght) { // mientras no finalizamos la conversion
         de los paquetes...
15
16     if (decodedMessage[read_index] == 0x00) { // si encontramos un 0x00
           ...
17
18       // finalizamos bloque
19       codedMessage[code_index] = code; // guardamos el codigo en el
           buffer codificado
20       code_index = write_index; // apuntamos al siguiente byte del
           buffer codificado donde guardaremos el codigo
21
22       // empezamos un bloque
23       write_index++; // incrementamos el indice de codedMessage ya que
           la primera posicion sera para el codigo
24       code = 0x01; // reinicializamos el codigo/contador
25
26     } else { // si no ...
27
28       codedMessage[write_index] = decodedMessage[read_index]; //
           movemos el byte del buffer sin codificar al codificado
29       write_index++; // incrementamos el indice de codedMessage
30       code++; // incrementamos el codigo/contador
31
```

```
32        if (code == 0xFF) { // si el contador ha llegado a 0xFF sin
              encontrar ningun 0x00 ...
33
34          // finalizamos bloque
35          codedMessage[code_index] = code; // guardamos el codigo en el
                buffer codificado
36          code_index = write_index; // apuntamos al siguiente byte del
                buffer codificado donde guardaremos el codigo
37
38          // empezamos un bloque
39          write_index++; // incrementamos el indice de codedMessage ya
                que la primera posicion sera para el codigo
40          code = 0x01; // reinicializamos el codigo/contador
41
42        }
43      }
44
45      read_index++; // incrementamos el indice al buffer de datos a
            codificar
46
47    }
48  }
```

Finalmente, una vez finalizados todos los elementos, añadimos el último *code byte* y devolvemos el número de elementos generados en la codificación.

```
1  // codifica un mensaje en COBS
2  unsigned long COBS_encode(
3    byte *decodedMessage, // puntero al buffer donde estan los datos a
         codificar
4    unsigned long lenght, // numero de elementos que contiene el mensaje
         a codificar
5    byte *codedMessage // puntero al buffer donde se guardaran los datos
         codificados
6    ) {
7
8    unsigned long read_index = 0, // indice al elemento del buffer sin
         codificar que estamos leyendo
9                  write_index = 1, // indice al elemento del buffer
                       codificado donde escribiremos la codificacion
10                 code_index = 0; // indice al elemento del buffer
                       codificado donde escribiremos el code byte
11
12   byte code = 0x01; // code byte
13
14   while(read_index < lenght) { // mientras no finalizamos la conversion
         de los paquetes...
15
16     if (decodedMessage[read_index] == 0x00) { // si encontramos un 0x00
           ...
17
```

```
18        // finalizamos bloque
19        codedMessage[code_index] = code; // guardamos el codigo en el
              buffer codificado
20        code_index = write_index; // apuntamos al siguiente byte del
              buffer codificado donde guardaremos el codigo
21
22        // empezamos un bloque
23        write_index++; // incrementamos el indice de codedMessage ya que
              la primera posicion sera para el codigo
24        code = 0x01; // reinicializamos el codigo/contador
25
26      } else { // si no ...
27
28        codedMessage[write_index] = decodedMessage[read_index]; //
              movemos el byte del buffer sin codificar al codificado
29        write_index++; // incrementamos el indice de codedMessage
30        code++; // incrementamos el codigo/contador
31
32        if (code == 0xFF) { // si el contador ha llegado a 0xFF sin
              encontrar ningun 0x00 ...
33
34          // finalizamos bloque
35          codedMessage[code_index] = code; // guardamos el codigo en el
                buffer codificado
36          code_index = write_index; // apuntamos al siguiente byte del
                buffer codificado donde guardaremos el codigo
37
38          // empezamos un bloque
39          write_index++; // incrementamos el indice de codedMessage ya
                que la primera posicion sera para el codigo
40          code = 0x01; // reinicializamos el codigo/contador
41
42        }
43      }
44
45      read_index++; // incrementamos el indice al buffer de datos a
            codificar
46
47    }
48
49    // finalizamos bloque
50    codedMessage[code_index] = code; // guardamos el codigo en el buffer
          codificado
51
52    return write_index; // devolvemos cuantos elementos se han escrito en
          el buffer codificado
53
54  }
```

1.2.5.2.1. Mini-resumen de COBS_encode Os voy a resumir todo lo que habéis hecho/codificado en la práctica para darle un poco de "perspectiva":

```
unsigned long COBS_encode(byte *decodedMessage, unsigned long lenght,
    byte *codedMessage) {

  unsigned long read_index = 0,
                write_index = 1,
                code_index = 0;

  byte 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;

}
```

Ya está. Este bloque de código de arriba (27 líneas de código) **es todo lo que hemos hecho en toda la práctica**. Os he añadido comentarios y espacios para hacerlo más entendible y os he puesto varias veces el mismo código para ir paso a paso, pero el código que hemos hecho tiene pocas líneas. ¡No hay

excusa para no dedicarle un tiempo a **entender 100 % que hace**! ¡Dadle duro!

> Si os cuesta entender la función, no menospreciéis la poderosa opción de usar papel y lápiz para dibujar los *arrays* e ir siguiendo lo que va haciendo. No es coña.

1.2.5.3. Test de la función COBS_encode

¡Fiu! Ya hemos **comprendido** e implementado la función para codificar en COBS. Vamos a testear que funciona. Simplemente, id al **archivo `masb-p06.ino`** y **descomentad la instrucción `TEST_COBS_encode_start();`**. Abrid el ***Serial Monitor*** de Arduino IDE y **compilad y subid el código** a la EVB. Si todo está correcto, en el terminal os debe de indicar que se han hecho cuatro tests y todos se han pasado.

```
----------------------------------------
 Test init
----------------------------------------

Test COBS ENCODE (0): OK
Test COBS ENCODE (1): OK
Test COBS ENCODE (2): OK
Test COBS ENCODE (3): OK
TEST ENCODE Finished --------------------
```

Si hay alguno que de error… revisa tú implementación de la función `COBS_encode`.

> Te recomiendo que eches un vistazo al archivo `test.ino` para que veas qué es lo que hace.

1.2.5.4. *Pull Request* a la rama develop

Una vez comprobado que funciona correctamente tu función, haz un *add*/*commit*/*push* **solo del archivo `cobs.ino`**.

```
git add arduino/masb-p06/cobs.ino
git commit -m "Función COBS_encode implementada."
git push
```

Seguidamente, ves a GitHub y crea un ***Pull Request* hacia la rama `develop`**. **No hagas el *merge***. Pon de ***reviewer* a tu compañero** y él hará el *merge* si aprueba tu desarrollo. Si no, te pedirá los cambios necesarios. No te olvides que tu compañero te pondrá como *reviewer* de su *Pull Request* y deberás aprobar los cambios y hacer el *merge* o requerirle que aplique los cambios que creas convenientes.

> ¿Te han aparecido conflictos al querer hacer el *merge* del *Pull Request* de tu compañero? Salta al apartado Resolución de conflictos en git.

1.2.6. *Workflow* del miembro B

1.2.6.1. Creación de las ramas *develop* y *feature*

Eres el responsable de **crear la rama `develop`** a partir de la rama `master`. **Tu compañero** creará su rama `feature` desde la rama `develop`, así que **no puede empezar hasta que la crees**.

```
git checkout master
git pull
git checkout -b develop
git push
```

Seguidamente, crea tu rama `feature` desde `develop`.

```
git checkout develop
git pull
git checkout -b feature/cobs-decode
```

1.2.6.2. Desarrollo de la función COBS_decode

¡Que suerte has tenido! Te ha tocado la función "fácil". Cosas del destino. Igualmente, mira la función que realiza el otro miembro del grupo porque es importante que entiendas qué hace para cuando tengas que revisar su código en el *Pull Request*.

Tu función, llamada `COBS_decode`, **se encargará de decodificar los datos en COBS a *raw***. Lo hará siguiendo el algoritmo indicado en el *paper*: leyendo el *code byte* y los siguientes bytes de datos del *code block* para después añadir un `0x00`. Empezamos la función:

```
// decodifica un mensaje en COBS
unsigned long COBS_decode(
  byte *codedMessage, // puntero al buffer donde estan los datos a
     decodificar
  unsigned long lenght, // numero de elementos que contiene el mensaje
     a decodificar
  byte *decodedMessage // puntero al buffer donde se guardaran los
     datos decodificados
  ) {

}
```

Seguidamente, creamos dos índices: `read_index` y `write_index`. El primero indicará el elemento del *buffer* codificado a leer y el segundo el elemento del *buffer* decodificado a escribir.

```
// decodifica un mensaje en COBS
unsigned long COBS_decode(
  byte *codedMessage, // puntero al buffer donde estan los datos a
     decodificar
  unsigned long lenght, // numero de elementos que contiene el mensaje
     a decodificar
```

```
5    byte *decodedMessage // puntero al buffer donde se guardaran los
        datos decodificados
6    ) {
7
8    unsigned long read_index = 0, // indice al elemento del buffer
        codificado que estamos leyendo
9                  write_index = 0; // indice al elemento del buffer
                       codificado donde escribiremos la codificacion
10
11 }
```

Luego, utilizamos un *while loop* para recorrer todos los elementos de *buffer* `codedMessage`. En cada iteración del *loop* aplicaremos el algoritmo de leer el *code byte* y leer/copiar el resto de bytes del *code block*.

```
1  // decodifica un mensaje en COBS
2  unsigned long COBS_decode(
3    byte *codedMessage, // puntero al buffer donde estan los datos a
        decodificar
4    unsigned long lenght, // numero de elementos que contiene el mensaje
        a decodificar
5    byte *decodedMessage // puntero al buffer donde se guardaran los
        datos decodificados
6    ) {
7
8    unsigned long read_index = 0, // indice al elemento del buffer
        codificado que estamos leyendo
9                  write_index = 0; // indice al elemento del buffer
                       codificado donde escribiremos la codificacion
10
11   while(read_index < lenght) { // mientras no finalizamos la conversion
         de los paquetes...
12
13   }
14 }
```

Guardamos el *code byte* y empezamos a leer/copiar el resto de elementos del *code block*.

```
1  // decodifica un mensaje en COBS
2  unsigned long COBS_decode(
3    byte *codedMessage, // puntero al buffer donde estan los datos a
        decodificar
4    unsigned long lenght, // numero de elementos que contiene el mensaje
        a decodificar
5    byte *decodedMessage // puntero al buffer donde se guardaran los
        datos decodificados
6    ) {
7
8    unsigned long read_index = 0, // indice al elemento del buffer
        codificado que estamos leyendo
```

```
9                 write_index = 0; // indice al elemento del buffer
                     codificado donde escribiremos la codificacion
10
11    while(read_index < lenght) { // mientras no finalizamos la conversion
          de los paquetes...
12
13      int code = codedMessage[read_index]; // el primer elemento es el
           codigo
14      read_index++; // vamos al siguiente elemento codificado
15
16      for (int i = 1; i < code; i++) { // movemos el numero de elementos
           indicados por code del buffer codificado al no codificado ...
17        decodedMessage[write_index] = codedMessage[read_index]; //
             copiamos valores
18        write_index++; // vamos al siguiente elemento decodificado
19        read_index++; // vamos al siguiente elemento codificado
20      }
21    }
22  }
```

Si el código no era `0xFF` ni estábamos al final de la trama de paquetes, añadimos el `0x00` que se eliminó en la codificación.

```
1   // decodifica un mensaje en COBS
2   unsigned long COBS_decode(
3     byte *codedMessage, // puntero al buffer donde estan los datos a
         decodificar
4     unsigned long lenght, // numero de elementos que contiene el mensaje
         a decodificar
5     byte *decodedMessage // puntero al buffer donde se guardaran los
         datos decodificados
6     ) {
7
8     unsigned long read_index = 0, // indice al elemento del buffer
         codificado que estamos leyendo
9                   write_index = 0; // indice al elemento del buffer
                       codificado donde escribiremos la codificacion
10
11    while(read_index < lenght) { // mientras no finalizamos la conversion
          de los paquetes...
12
13      int code = codedMessage[read_index]; // el primer elemento es el
           codigo
14      read_index++; // vamos al siguiente elemento codificado
15
16      for (int i = 1; i < code; i++) { // movemos el numero de elementos
           indicados por code del buffer codificado al no codificado ...
17        decodedMessage[write_index] = codedMessage[read_index]; //
             copiamos valores
18        write_index++; // vamos al siguiente elemento decodificado
19        read_index++; // vamos al siguiente elemento codificado
```

```
20      }
21
22      if (code < 0xFF && read_index < lenght) { // si el codigo era menor
            a 0xFF ...
23        decodedMessage[write_index] = 0x00; // anadimos el 0x00 que se
              elimino en la codificacion
24        write_index++; // y vamos al siguiente elemento decodificado
25      }
26    }
27  }
```

Finalmente, devolvemos el número de elementos resultantes de la descodificación.

```
1  // decodifica un mensaje en COBS
2  unsigned long COBS_decode(
3    byte *codedMessage, // puntero al buffer donde estan los datos a
         decodificar
4    unsigned long lenght, // numero de elementos que contiene el mensaje
         a decodificar
5    byte *decodedMessage // puntero al buffer donde se guardaran los
         datos decodificados
6    ) {
7
8    unsigned long read_index = 0, // indice al elemento del buffer
         codificado que estamos leyendo
9                  write_index = 0; // indice al elemento del buffer
                       codificado donde escribiremos la codificacion
10
11   while(read_index < lenght) { // mientras no finalizamos la conversion
         de los paquetes...
12
13     int code = codedMessage[read_index]; // el primer elemento es el
           codigo
14     read_index++; // vamos al siguiente elemento codificado
15
16     for (int i = 1; i < code; i++) { // movemos el numero de elementos
           indicados por code del buffer codificado al no codificado ...
17       decodedMessage[write_index] = codedMessage[read_index]; //
             copiamos valores
18       write_index++; // vamos al siguiente elemento decodificado
19       read_index++; // vamos al siguiente elemento codificado
20     }
21
22     if (code < 0xFF && read_index < lenght) { // si el codigo era menor
           a 0xFF ...
23       decodedMessage[write_index] = 0x00; // anadimos el 0x00 que se
             elimino en la codificacion
24       write_index++; // y vamos al siguiente elemento decodificado
25     }
26   }
27
```

```
28    return write_index; // devolvemos el numero de bytes generados en la
          decodificacion
29
30  }
```

1.2.6.2.1. Mini-resumen de COBS_decode Como en el caso de `COBS_encode`, aquí tenéis el código sin espacios de más ni comentarios:

```
1  unsigned long COBS_decode(byte *codedMessage, unsigned long lenght,
       byte *decodedMessage) {
2
3    unsigned long read_index = 0,
4                  write_index = 0;
5
6    while(read_index < lenght) {
7
8      int code = codedMessage[read_index];
9      read_index++;
10
11     for (int i = 1; i < code; i++) {
12       decodedMessage[write_index] = codedMessage[read_index];
13       write_index++;
14       read_index++;
15     }
16
17     if (code < 0xFF && read_index < lenght) {
18       decodedMessage[write_index] = 0x00;
19       write_index++;
20     }
21   }
22
23   return write_index;
24
25 }
```

Como ves, la función no es muy larga/complicada. Dedica un tiempo a asegurarte que comprendes 100 % su funcionamiento.

1.2.6.3. Test de la función COBS_decode Ya hemos **comprendido** e implementado la función para decodificar en COBS. Vamos a testear que funciona. Simplemente, id al **archivo** **`masb-p06.ino`** y **descomentad la instrucción** **`TEST_COBS_decode_start();`**. Abrid el ***Serial Monitor*** de Arduino IDE y **compilad y subid el código** a la EVB. Si todo está correcto, en el terminal os debe de indicar que se han hecho cuatro tests y todos se han pasado.

```
1  ----------------------------------------
2   Test init
```

```
3  ---------------------------------------
4
5  Test COBS DECODE (0): OK
6  Test COBS DECODE (1): OK
7  Test COBS DECODE (2): OK
8  Test COBS DECODE (3): OK
9  TEST DECODE Finished -------------------
```

Si hay alguno que dé error… revisa tú implementación de la función `COBS_decode`.

> Te recomiendo que eches un vistazo al archivo `test.ino` para que veas qué es lo que hace.

1.2.6.4. *Pull Request* a la rama develop

Una vez comprobado que funciona correctamente tu función, haz un *add*/*commit*/*push* **solo del archivo `cobs.ino`**.

```
1  git add arduino/masb-p06/cobs.ino
2  git commit -m "Función COBS_decode implementada."
3  git push
```

Seguidamente, ves a GitHub y crea un ***Pull Request* hacia la rama `develop`**. **No hagas el *merge***. Pon de ***reviewer* a tu compañero** y él hará el *merge* si aprueba tu desarrollo. Si no, te pedirá los cambios necesarios. No te olvides que tu compañero te pondrá como *reviewer* de su *Pull Request* y deberás aprobar los cambios y hacer el *merge* o requerirle que aplique los cambios que creas convenientes.

> ¿Te han aparecido conflictos al querer hacer el *merge* del *Pull Request* de tu compañero? Salta al apartado Resolución de conflictos en git.

1.2.7. Resolución de conflictos en git

Al llegar a este apartado, existen dos posibilidades:

- Eres el primero que ha hecho el *merge* del *Pull Request* de tu compañero, no has tenido ningún problema y no sabes de qué conflicto te estoy hablando.
- Eres el último en intentar hacer el *merge* del *Pull Request* de tu compañero y te ha saltado un mensaje diciendo que existen conflictos.

Para el primer caso, quédate, que verás qué son y a qué se deben los conflictos en git y cómo resolverlos. Si tu caso es el segundo, vamos a resolver los conflictos. Es fácil.

En la siguiente imagen tenéis un **ejemplo de un *Pull Request* con conflictos**. Ignorad el nombre del repositorio, mensajes de los *commits* y los *paths* de los archivos. Es un ejemplo.

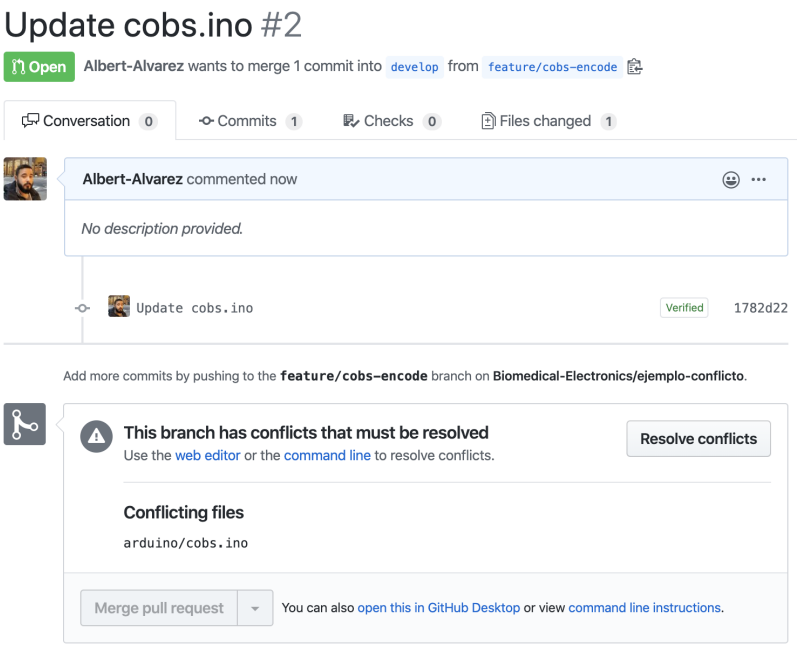

Figura 2: Pull Request con conflictos.

Un conflicto se debe, básicamente, **a que se ha editado un mismo archivo desde ramas distintas**. Por ello, **git no sabe cuál de las dos versiones debe de prevalecer sobre la otra** y **nos pide a nosotros que explicitemos qué queremos que se guarde**.

Podríamos resolver los conflictos en nuestro ordenador, pero GitHub nos posibilita hacerlo desde la web. Para ello, **clicamos sobre el botón `Resolve conflicts`**. GitHub nos llevará al/los archivo/s que presenta/n conflictos y nos mostrará su contenido.

Git (y no GitHub, importante) indica los conflictos añadiendo lo siguiente directamente en el contenido del archivo:

```
...

<<<<<<< rama1
  ...<codigo_rama1>...
=======
  ...<codigo_rama1>...
>>>>>>> rama2

...
```

Git nos indica el contenido que se ha añadido en cada una de las ramas delimitando esos contenidos del modo que se indica arriba. **Nosotros deberemos de solucionar el conflicto dejando el código que queramos que prevalezca y eliminando el que no.** Por último, deberemos de **eliminar los**

marcadores de conflicto de git.

En el caso que nos ocupa, tenemos el siguiente archivo `cobs.ino`:

Figura 3: Conflictos.

En este archivo tenemos el código de la rama `feature/cobs-encode` y el de la rama `feature/cobs-decode`. Ya he buscado un primer **ejemplo sencillo** para ver por primera vez la resolución de conflictos. En este caso, **queremos que las dos versiones prevalezcan puesto que son funciones distintas que queremos añadir en un mismo archivo**. Simplemente, **dejaremos que prevalezca todo el código** y **eliminaremos los marcadores de conflicto** de git. **Antes de hacerlo**, fijaros que al final del archivo no se indica que haya conflicto con la última línea, que contiene una llave.

Git no tiene inteligencia para reconocer conflictos. **Es tonto** y encuentra los conflictos por comparación simple. Por ello, como la última llave es común para ambas funciones, cree que eso está bien y con una sola llave es suficiente. Podéis comprobar como ha eliminado la llave final de la primera función. Es decir, git advierte de conflictos, **¡pero no os fiéis nunca!** Comete errores.

Vamos a dejar la versión final de nuestro código: **eliminamos los marcadores de conflicto** y **añadimos una llave al final de la primera función**. Hecho esto, marcamos el conflicto como resuelto **clicando sobre `Mark as resolved`**. Seguidamente, **clicamos a `Commit merge`**.

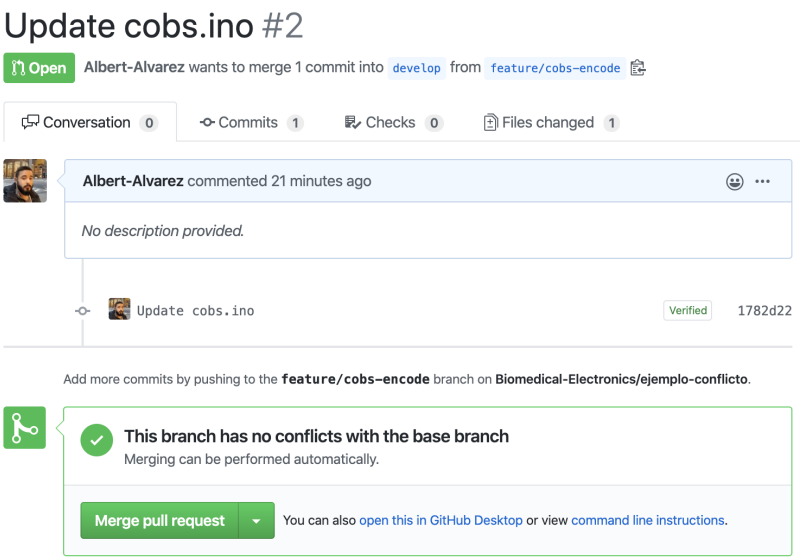

Figura 4: Pull Request sin conflictos

Los conflictos ya han desaparecido y podemos proceder al *merge* del *Pull Request* como siempre.

Easy peasy lemon squeezy.

> Ahora, provad que el código en la rama `develop` funciona descomentando las dos instrucciones del archivo `masb-p06.ino`. Si todo funciona, haced un *Pull Request* y *merge* a la `master`.

1.3. Reto

Leer. Ese es el reto. Leer el *paper* y la pequeña porción de código que hemos desarrollado para comprender el funcionamiento de COBS. **Debéis** de intentar **comprender todo el código** que hemos desarrollado en la práctica y tenerlo todo claro de cara a, en la siguiente práctica, poder seguir construyendo conocimiento sobre la comunicación serie sobre esta base. ¡Hacedlo! Y cualquier duda lo hablamos por el foro o en clase.

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. **Cread una nueva rama para el informe** para que este no entre en conflicto con el de vuestro compañero. Dejo a vuestra elección un **nombre apropiado** para la rama. Guardad el informe con el nombre `REPORT.md` en la misma carpeta que este documento.

El informe debe de contener:

- ☐ **Codificación:** Codifica en COBS la siguiente trama *raw*. **Indica los pasos realizados** para la codificación.

0x01	0x34	0xF2	0x00	0xA3	0x00	0x16	0x32	0x00	0x00	0x02

- ☐ **Descodificación:** Decodifica a *raw* la siguiente trama codificada en COBS. **Indica los pasos realizados** para la decodificación.

0x01	0x03	0xF3	0x23	0x02	0xE2	0x01	0x04	0x15	0xA5	0x11

- ☐ **Cuestionario:** Encontraréis un archivo `QUESTIONS.md` en la carpeta `arduino`. Copiad su contenido a vuestro `REPORT.md` y responded las preguntas indicando la/s respuesta/s con una X. Podéis ver cómo marcar las respuestas aquí.

1.4.2. Pull Request

Finalizado el informe, acordaos de hacer el *push* pertinente y cread un *Pull Request* (PR) de vuestra rama a la `master`. Acordaos de ponerme como *Reviewer*. El resto de la práctica la evaluaré directamente en la rama `master`. Si veo cualquier cosa, crearé un *issue* para solicitaros cambios.

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

Práctica diferente donde no hemos picado tanto código y hemos dedicado tiempo a **ver y comprender la (de)codificación en COBS**. De este modo, **logramos enviar paquetes de bytes con `0x00` como *term char* sin problemas**.

La **implementación está simplificada**. No hacemos gestión de errores. ¿Qué pasaría si se pierde un byte por el camino? Pero la implementación nos servirá para poder codificar las instrucciones que enviemos entre el microcontrolador y otro dispositivo.

También **hemos visto la existencia de tests unitarios**. Normalmente, se suelen utilizar *frameworks* ya existentes para realizar los tests. En este caso, hemos visto una implementación *full custom*.

Finalmente, hemos visto **cómo resolver conflictos que se pueden dar en git** al intentar editar un mismo archivo desde ramas distintas.

En la siguiente parte, implementaremos los **algoritmos de (de)codificación en STM32CubeIDE** y los guardaremos como oro en paño. ¿Porque? Porque tal cual creemos las funciones, **las podremos importar en el proyecto final** de la asignatura y ya las tendremos hechas.