MASB | STM32
Comunicación serie III

#stm32 #c #biomedical #microcontroladores #programacion

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

15 de abril de 2020

"MASB (STM32): Comunicación serie III" © 2020
por Albert Álvarez Carulla se distribuye bajo
una Licencia Creative Commons
Atribución-NoComercial-SinDerivadas 4.0
Internacional

Índice

1. Comunicación serie - Parte III

— "*El profe no ha subido la parte de Arduino… Se le habrá olvidado… Pues yo no digo nada y eso que me ahorro. Y que me diga algo; ha sido su culpa. ¡JA!*"

— "*Esta semana solo haremos la práctica en STM32CubeIDE ya que sería hacer lo mismo dos veces y no tiene sentido. Además, así se toman un respiro antes de la última práctica y el proyecto donde deberán darlo todo. ¡JAJA!*"

Pues sí, esta semana hacemos la **práctica solo en STM32CubeIDE** puesto que el código sería el mismo (o muy parecido) tanto en Arduino como STM32. Por ello, vamos a centrarnos en la parte de STM32. Esta será la última práctica que dedicaremos a la comunicación serie asíncrona. La **siguiente práctica** será la última y haremos también comunicación; pero síncrona y, además, lo haremos **de manera especial** ya que no la explicaré yo, sino que os la explicaréis vosotros. Pero no nos adelantemos y veamos qué vamos hacer en esta práctica.

Hasta ahora hemos visto cómo lograr que se comuniquen el microcontrolador y el PC (u otro microcontrolador) con la UART y cómo codificar los datos (COBS). Esto, en un ejemplo simplificado de comunicación, seria el equivalente a "hemos visto cómo usar la voz" y "hemos visto cómo codificar nuestra voz para enviar un mensaje (idioma)", respectivamente. Sabiendo hablar y sabiendo un idioma, ahora **nos falta saber qué decir** y cómo interpretar qué nos dicen. Es decir, el set de instrucciones.

El término "set de instrucciones" suena… ¿abstracto?, pero veréis que nada más lejos. Un **set de instrucciones** es básicamente un **conjunto de comandos que todos los interlocutores de una comunicación conocen de antemano** y que deben de utilizar para pedir que se lleve a cabo una acción o responder a una petición. Por ejemplo, ¿algún aficionado al *baseball*?… Nadie, obviamente. Pues resulta que en *baseball* se utilizan signos para comunicarse entre los miembros del equipo y, además, que los adversarios no sepan que se están diciendo. Es decir, cada equipo tiene sus propios signos que además van cambiando periódicamente para que no los decodifiquen.

Figura 1: Signo de bola curva.

Por ejemplo, el *catcher* de la imagen está indicando al lanzador que lance una bola curva. Luego el lanzador responde con un "sí" o "no" con la cabeza (esto todo el mundo sabe lo que significa…) y lanza. Es todo un mundo dentro del *baseball* y solo quería usarlo como ejemplo para mostraros que no hay un porqué un comando hace una cosa u otra. Simplemente, nos ponemos de acuerdo en que tal comando hace una cosa y tal comando hace otra y listos.

Entonces, antes de empezar, lo importante es: ¿qué comandos utilizaremos para comunicarnos entre el ordenador y el microcontrolador? Podría inventarme unos para la práctica, pero seamos eficientes y **vamos a implementar los comandos o instrucciones que utilizaremos en el proyecto**. Los podéis encontrar en el **siguiente documento**. Allí están detallados cada uno de los comandos, pero aquí os dejo un resumen.

Básicamente, tenemos **dos tipos** de instrucciones: de **comando** y de **datos**. El primer tipo hace referencia a los comandos que indican que se lleve a cabo una **acción**, mientras que el segundo tipo solo tienen como objetivo **enviar/comunicar datos**. En las instrucciones de comando tenemos:

- START_CV_MEAS
- START_CA_MEAS
- STOP_MEAS

Puesto que el **PC hace de *master***, todas las instrucciones de comando las enviará el PC al microcontrolador. El primer comando indica al microcontrolador que inicie una voltametría cíclica (una

medición electroquímica). En este comando también se incluyen los parámetros de configuración de esta medida.

El segundo comando hace lo mismo, pero en lugar de indicar que se inicie una voltametría cíclica, indica que se inicie una cronoamperometría junto con sus pertinentes parámetros de configuración.

Por último, está el comando para detener una medición en curso.

En el documento, veréis que cada comando tiene asignado un valor (0x01, 0x02 y 0x03, respectivamente). Esto es porque **el PC no nos enviará una cadena de texto con el nombre del comando**. **Enviará el número correspondiente de cada comando**. Si enviásemos el comando START_CV_MEAS en texto con codificación ASCII, necesitaríamos 13 bytes. Enviando un 0x01 solo necesitamos 1 byte. ¿Y porqué un 0x01 para START_CV_MEAS, un 0x02 para START_CA_MEAS y un 0x03 para STOP_MEAS? Porqué así lo hemos acordado o, en este caso, lo he decidido yo. **No hay un motivo más allá de que acordemos unos valores en común y los utilicemos.**

El otro tipo de instrucciones, el de datos, lo utiliza el microcontrolador para, simplemente, enviar los datos que va leyendo al PC. Podéis ver en el documento qué datos se envían.

Pues bien, con el documento abierto, **vamos a implementar la parte de comunicación serie** de nuestro set de instrucciones. **Importante**: solo la parte de comunicación. ¿Qué quiere decir esto? Que sabremos **interpretar los comandos que nos lleguen** desde el PC y **desgranar/separar cada uno de los parámetros** que acompañen este comando, pero hasta ahí. Luego **será trabajo vuestro durante el proyecto qué hacer con esos parámetros para iniciar una medición u otra**.

"*¡Allévamos!*"

> Aviso que puede **parecer** larga la práctica por la de código que aparece, pero éste está repetido múltiples veces. He creído conveniente repetir todo el código en los ejemplos e indicar explícitamente qué se añade o quita para hacerlo más claro (en lugar de solo poner como código lo que se añade). Por eso puede parecer más larga de lo que en realidad es.

2. Objetivos

- Implementar un set de instrucciones.
- Uso de estructuras.
- Encapsulación de funciones.
- Conversión entre vector de bytes y variables de diferente tamaño.

3. Procedimiento

3.1. Creación del proyecto y archivos básicos

Pues vamos empezar **creando nuestro proyecto** y **añadiendo los archivos básicos**. Ya sabéis como hacerlo así que pasaremos rápido:

- Crea un proyecto **que se llame `masb-p07` dentro de la carpeta `stm32cube`**.
- Asegúrate de que la configuración del microcontrolador en el archivo `.ioc` sea la correcta para poder realizar comunicaciones con la UART **mediante interrupciones** con una configuración 115200 8N1.
- Crea una carpeta `components` **tanto dentro de la carpeta `Core/Src` como de la carpeta `Core/Inc`**.
- **Crea el componente `stm32main`**(su correspondiente fichero `.c`en`Core/Src/components` y su fichero `.h`en`Core/Inc/components`).
- Añadir la llamada a la **funciones `setup()` y `loop()` en el archivo `main.c`**.
- **Crea el componente `cobs`**. (Acuérdate de hacer **siempre un `#include` del archivo `.h` del componente** que quieras utilizar allá donde toque.)

Y con esto ya tenemos lista la creación del proyecto y su configuración inicial. Vamos al meollo del asunto.

3.2. Definiendo el *header*

Hasta ahora nos hemos dedicado a desarrollar siguiendo una **aproximación *bottom-up***. Es decir, **definíamos las funciones en nuestros archivos `.c` y luego, las que quisiéramos utilizar en otros archivos, las declarábamos en un archivo `.h`**. Así pues, quien quisiera utilizar esas funciones solo debía hacer un `#include` del `.h` y ya está. Esta vez haremos una **aproximación *top-down***. Es decir, **definiremos en el archivo `.h` las funciones y variables de nuestro componente que estarán disponibles y luego ya las definiremos en nuestro `.c`**. Esta suele ser la aproximación más común en el desarrollo de proyectos grandes/profesionales. Básicamente, porque establece las reglas entre todos los desarrolladores y se puede desarrollar en paralelo funciones dependientes y funciones de test. Más adelante os comento un poco más sobre esto.

Bien, vamos a **crear un componente llamado `masb_comm_s`** que albergará toda la funcionalidad relativa a la comunicación de nuestro futuro proyecto. Nos vamos al archivo `masb_comm_s.h` y os voy a ir indicando qué vamos añadiendo (resaltado en verde) y su función.

Al crear el archivo `masb_comm_s.h` partimos del siguiente archivo:

```c
/*
 * masb_comm_s.h
 *
 *  Created on: Apr 14, 2020
 *      Author: albert
 */

#ifndef INC_COMPONENTS_MASB_COMM_S_H_
#define INC_COMPONENTS_MASB_COMM_S_H_

#endif /* INC_COMPONENTS_MASB_COMM_S_H_ */
```

Vamos añadir primeramente las macros que utilizaremos en los fichero `.c` que facilitarán tanto su legibilidad como su actualización cuando cambiemos algún parámetro.

```c
/*
 * masb_comm_s.h
 *
 *  Created on: Apr 14, 2020
 *      Author: albert
 */

#ifndef INC_COMPONENTS_MASB_COMM_S_H_
#define INC_COMPONENTS_MASB_COMM_S_H_

+ #define UART_BUFF_SIZE          50
+ #define UART_TERM_CHAR          0x00
+ #define TRUE                   1
+ #define FALSE                  0

+ // Commands
+ #define START_CV_MEAS 0x01
+ #define START_CA_MEAS 0x02
+ #define STOP_MEAS     0x03

#endif /* INC_COMPONENTS_MASB_COMM_S_H_ */
```

> Como os estoy resaltando lo que voy añadiendo, tengo que añadir un signo + en cada línea que quiero resaltar. Si copiáis y pegáis el código, puede que se os escape un símbolo + por ahí y os dé error. Avisados.

Hemos añadido macros para definir el tamaño del *buffer* de comunicación (acordaros, el tamaño (número de elementos) del vector donde guardaremos los bytes recibidos o los que queramos enviar), el carácter de final de mensaje que utilizaremos (en este caso, `0x00`) y las macros `TRUE`/`FALSE`. También hemos añadido las macros de los comandos que están definidos en el documento de especificaciones del set de instrucciones que utilizaremos. De este modo, en lugar de utilizar `0x01`, `0x02` o `0x03` que nos dan poca información o ninguna sobre lo que ello significa, podemos utilizar

las macros `START_CV_MEAS`, `START_CA_MEAS` y `STOP_MEAS`. Además, si las especificaciones del set de instrucciones cambia y los valores de cada comando cambian, con solo cambiar el número en la macro tendremos todo el código funcionando.

Ahora vamos añadir los `#include`.

```
1  /*
2   * masb_comm_s.h
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  #ifndef INC_COMPONENTS_MASB_COMM_S_H_
9  #define INC_COMPONENTS_MASB_COMM_S_H_
10
11 #define UART_BUFF_SIZE      50
12 #define UART_TERM_CHAR      0x00
13 #define TRUE                1
14 #define FALSE               0
15
16 // Commands
17 #define START_CV_MEAS   0x01
18 #define START_CA_MEAS   0x02
19 #define STOP_MEAS       0x03
20
21 + #include "stm32f4xx_hal.h"
22 + #include "components/cobs.h"
23 + #include "components/cyclic_voltammetry.h"
24
25 #endif /* INC_COMPONENTS_MASB_COMM_S_H_ */
```

El primer `#include` añade las referencias a las librerías HAL del microcontrolador. Las necesitamos para poder utilizar dentro de las funciones del archivo `masb_comm_s.c` las funciones HAL tipo `HAL_UART_Receive_IT()` o `HAL_UART_Transmit_IT()`, por ejemplo. Luego añadimos el `#include` al componente `cobs` puesto que dentro de `masb_comm_s.c` necesitaremos llamar las funciones `COBS_decode()` y `COBS_encode()` para poder decodificar y codificar los mensajes.

Y por último, añadimos un`#include` más. Este último incluye un archivo que aún no existe. Lo creamos. **Creamos un componente llamado `cyclic_voltammetry`** que incluirá toda la funcionalidad referente al control de la medición de voltametría cíclica. Este componente no lo definiremos. Eso será tarea vuestra durante el proyecto, pero lo creamos ya (en blanco) para que el componente `masb_comm_s` pueda llamarlo.

En el archivo `cyclic_voltammetry.h` añadimos la siguiente definición estructura:

```
1  /*
2   * cyclic_voltammetry.h
```

```
 3  *
 4  *  Created on: Apr 14, 2020
 5  *      Author: albert
 6  */
 7
 8 #ifndef INC_COMPONENTS_CYCLIC_VOLTAMMETRY_H_
 9 #define INC_COMPONENTS_CYCLIC_VOLTAMMETRY_H_
10
11 + struct CV_Configuration_S {
12 +
13 +   double eBegin;
14 +   double eVertex1;
15 +   double eVertex2;
16 +   uint8_t cycles;
17 +   double scanRate;
18 +   double eStep;
19 +
20 + };
21
22 #endif /* INC_COMPONENTS_CYCLIC_VOLTAMMETRY_H_ */
```

Hemos declarado un tipo de estructura llamada **`CV_Configuration_S` que contiene los parámetros de configuración de una voltametría cíclica**. Cuando el PC nos envíe el comando `START_CV_MEAS`, guardaremos los parámetros recibidos en una estructura de este tipo. Al incluir este archivo `cyclic_voltametry.h` en nuestro `masb_comm_s.h` ahora podemos hacer uso de esta estructura en el archivo `masb_comm_s.c`.

Volvemos a nuestro `masb_comm_s.h` y añadimos la declaración de u**na estructura llamada `Data_S`**. Esta estructura **servirá para contener los datos** que el microcontrolador enviará al PC.

```
 1 /*
 2  * masb_comm_s.h
 3  *
 4  *  Created on: Apr 14, 2020
 5  *      Author: albert
 6  */
 7
 8 #ifndef INC_COMPONENTS_MASB_COMM_S_H_
 9 #define INC_COMPONENTS_MASB_COMM_S_H_
10
11 #define UART_BUFF_SIZE      50
12 #define UART_TERM_CHAR      0x00
13 #define TRUE                1
14 #define FALSE               0
15
16 // Commands
17 #define START_CV_MEAS   0x01
18 #define START_CA_MEAS   0x02
19 #define STOP_MEAS       0x03
```

```
20
21 #include "stm32f4xx_hal.h"
22 #include "components/cobs.h"
23 #include "components/cyclic_voltammetry.h"
24
25 + // Structures
26 + struct Data_S {
27 +
28 +   uint32_t point;
29 +   uint32_t timeMs;
30 +   double voltage;
31 +   double current;
32 +
33 + };
34
35 #endif /* INC_COMPONENTS_MASB_COMM_S_H_ */
```

Ya por último, añadimos la funciones del componente `masb_comm_s` que **el resto de componentes de nuestro proyecto podrán utilizar**.

```
1  /*
2   * masb_comm_s.h
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  #ifndef INC_COMPONENTS_MASB_COMM_S_H_
9  #define INC_COMPONENTS_MASB_COMM_S_H_
10
11 #define UART_BUFF_SIZE      50
12 #define UART_TERM_CHAR      0x00
13 #define TRUE                1
14 #define FALSE               0
15
16 // Commands
17 #define START_CV_MEAS   0x01
18 #define START_CA_MEAS   0x02
19 #define STOP_MEAS       0x03
20
21 #include "stm32f4xx_hal.h"
22 #include "components/cobs.h"
23 #include "components/cyclic_voltammetry.h"
24
25 // Structures
26 struct Data_S {
27
28     uint32_t point;
29     uint32_t timeMs;
30     double voltage;
31     double current;
```

```
32
33  };
34
35  + // Prototypes.
36  + void MASB_COMM_S_setUart(UART_HandleTypeDef *newHuart);
37  + void MASB_COMM_S_waitForMessage(void);
38  + _Bool MASB_COMM_S_dataReceived(void);
39  + uint8_t MASB_COMM_S_command(void);
40  +
41  + struct CV_Configuration_S MASB_COMM_S_getCvConfiguration(void);
42  + void MASB_COMM_S_sendData(struct Data_S data);
43
44  #endif /* INC_COMPONENTS_MASB_COMM_S_H_ */
```

Un breve resumen de cada función sería:

- `MASB_COMM_S_setUart()`: acordémonos que la estructura con la configuración de la UART (`huart2`) está en el archivo `main.c`. Para obtener acceso a esa estructura, en ocasiones anteriores hemos utilizado la *keyword* `extern`. Esto ya comentamos que era hacer "trampa" y que hay otros modos de hacerlo. Ese otro modo es pasar el puntero a esa estructura y así poder tener acceso a ella. Esto es lo que hacemos con esta función. Le pasamos el puntero a la estructura de configuración de la UART y así poder tener acceso a ella desde el archivo `masb_comm_s.c`.
- `MASB_COMM_S_waitForMessage()`: activa la interrupción para la recepción de mensajes.
- `MASB_COMM_S_dataReceived()`: indica si se ha recibido un mensaje o no.
- `MASB_COMM_S_command()`: devuelve el comando recibido.
- `MASB_COMM_S_getCvConfiguration()`: extrae del mensaje recibido los parámetros de configuración de una voltametría cíclica y los devuelve.
- `MASB_COMM_S_sendData()`: envía los datos al PC.

Así quedaría nuestro `masb_comm_s.h`. Ahora alguien, aunque no hayamos definido las funciones (no las hemos escrito en el archivo `.c`), puede coger nuestro archivo `.h`, importarlo (`#include`) y hacer su parte del código sin que le salte error porque falten funciones. No están definidas y el programa no funcionará, pero no dará errores de compilación y el compañero podrá programar tranquilamente, ir compilando y viendo si ella/él ha cometido errores. Por otro lado, en un entorno profesional, el equipo de test ya tendría disponible la declaración de las funciones y podría ir desarrollando las funciones de test para comprobar, una vez definidas las funciones en el archivo `.c`, que todo funciona correctamente.

Saltemos al archivo `masb_comm_s.c`.

3.3. Definición de funciones en el *source*

En el archivo `masb_comm_s.c` vamos a definir (escribir código) todas las funciones indicadas en el archivo `.h` más aquellas funciones auxiliares que solo utilicemos dentro del archivo `masb_comm_s.c`. Todas **las funciones y variables que solo deban de ser accedidas desde el propio archivo `masb_comm_s.c` las declararemos con la *keyword* `static`** que, entre otras consecuencias, hará que solo estén disponibles en este archivo.

Empezamos añadiendo el pertinente `#include`:

```
1  /*
2   * masb_comm_s.c
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  + #include "components/masb_comm_s.h"
```

Luego, vamos a definir la función más básica que es `MASB_COMM_S_setUart` para obtener el puntero (dirección de memoria) a la configuración de la UART. Para ello, además de definir la función, declararemos una variable de tipo puntero donde almacenaremos esa dirección de memoria. Puesto que este puntero solo se utilizará en este archivo, lo declaramos como estático.

```
1  /*
2   * masb_comm_s.c
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  #include "components/masb_comm_s.h"
9
10 + static UART_HandleTypeDef *huart;
11
12 + void MASB_COMM_S_setUart(UART_HandleTypeDef *newHuart) {
13 +   huart = newHuart;
14 + }
```

Con esto tendremos acceso a la UART mediante la variable `huart`.

Ahora vamos a definir la función `MASB_COMM_S_waitForMessage`. Aquí no os despistéis. Simplemente, empaquetaremos lo que hicimos en la sesión anterior dentro de una función (podéis ir al guion de la sesión anterior y ver que no os engaño). Cuando esperemos un nuevo mensaje, pondremos en `FALSE`una variable de bandera, inicializaremos a 0 un índice de conteo de los bytes recibidos y habilitaremos las interrupciones de recepción de bytes. Quedaría del siguiente modo.

```
/*
 * masb_comm_s.c
 *
 *  Created on: Apr 14, 2020
 *      Author: albert
 */

#include "components/masb_comm_s.h"

static UART_HandleTypeDef *huart;
+ static _Bool dataReceived = FALSE;
+ uint8_t rxIndex = 0;
+ uint8_t rxBuffer[UART_BUFF_SIZE] = { 0 };

void MASB_COMM_S_setUart(UART_HandleTypeDef *newHuart) {
    huart = newHuart;
}

+ void MASB_COMM_S_waitForMessage(void) {
+
+   dataReceived = FALSE;
+   rxIndex = 0;
+   HAL_UART_Receive_IT(huart, &rxBuffer[rxIndex], 1);
+
+ }
```

Lo siguiente que haremos es definir la función que nos indicará si se ha recibido un mensaje (`MASB_COMM_S_dataReceived()`). Es decir, devolveremos el valor de la variable `dataReceived` y, aprovechando, si se ha recibido un mensaje lo decodificamos.

```
/*
 * masb_comm_s.c
 *
 *  Created on: Apr 14, 2020
 *      Author: albert
 */

#include "components/masb_comm_s.h"

static UART_HandleTypeDef *huart;
+ static _Bool dataReceived = FALSE;
+ uint8_t rxIndex = 0;
uint8_t rxBuffer[UART_BUFF_SIZE] = { 0 };
+ uint8_t rxBufferDecoded[UART_BUFF_SIZE] = { 0 };

void MASB_COMM_S_setUart(UART_HandleTypeDef *newHuart) {
    huart = newHuart;
}

void MASB_COMM_S_waitForMessage(void) {
```

```
21
22      dataReceived = FALSE;
23      rxIndex = 0;
24      HAL_UART_Receive_IT(huart, &rxBuffer[rxIndex], 1);
25
26  }
27
28  + _Bool MASB_COMM_S_dataReceived(void) {
29  +
30  +    if (dataReceived) {
31  +
32  +        COBS_decode(rxBuffer, rxIndex, rxBufferDecoded);
33  +
34  +    }
35  +
36  +    return dataReceived;
37  +
38  + }
```

Ahora toca la función `MASB_COMM_S_command()`. Este comando nos devuelve el comando que hayamos recibido. Podemos ver en el documento que el comando se indica en el primer byte que recibimos, así que simplemente es una función que devuelve el primer byte.

```
1  /*
2   * masb_comm_s.c
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  #include "components/masb_comm_s.h"
9
10 static UART_HandleTypeDef *huart;
11 static _Bool dataReceived = FALSE;
12 uint8_t rxIndex = 0;
13 uint8_t rxBuffer[UART_BUFF_SIZE] = { 0 };
14 uint8_t rxBufferDecoded[UART_BUFF_SIZE] = { 0 };
15
16 void MASB_COMM_S_setUart(UART_HandleTypeDef *newHuart) {
17     huart = newHuart;
18 }
19
20 void MASB_COMM_S_waitForMessage(void) {
21
22     dataReceived = FALSE;
23     rxIndex = 0;
24     HAL_UART_Receive_IT(huart, &rxBuffer[rxIndex], 1);
25
26 }
27
```

```
28 _Bool MASB_COMM_S_dataReceived(void) {
29
30     if (dataReceived) {
31
32         COBS_decode(rxBuffer, rxIndex, rxBufferDecoded);
33
34     }
35
36     return dataReceived;
37
38 }
39
40 + uint8_t MASB_COMM_S_command(void) {
41 +
42 +   return rxBufferDecoded[0];
43 +
44 + }
```

Nos quedan dos funciones: `MASB_COMM_S_getCvConfiguration()` y `MASB_COMM_S_sendData()`. Vamos, ¡ya falta nada! Empezamos con la primera. Debemos de **coger los bytes del mensaje y unirlos** para obtener los diferentes parámetros la voltametría cíclica. ¿Cómo pasamos de vector de bytes a otro tipos de variables y viceversa? Eeexacto. Uniones. Vamos a ello.

```
1  /*
2   * masb_comm_s.c
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  #include "components/masb_comm_s.h"
9
10 static UART_HandleTypeDef *huart;
11 static _Bool dataReceived = FALSE;
12 uint8_t rxIndex = 0;
13 uint8_t rxBuffer[UART_BUFF_SIZE] = { 0 };
14 uint8_t rxBufferDecoded[UART_BUFF_SIZE] = { 0 };
15
16 void MASB_COMM_S_setUart(UART_HandleTypeDef *newHuart) {
17     huart = newHuart;
18 }
19
20 void MASB_COMM_S_waitForMessage(void) {
21
22     dataReceived = FALSE;
23     rxIndex = 0;
24     HAL_UART_Receive_IT(huart, &rxBuffer[rxIndex], 1);
25
26 }
27
```

```
28  _Bool MASB_COMM_S_dataReceived(void) {
29
30      if (dataReceived) {
31
32          COBS_decode(rxBuffer, rxIndex, rxBufferDecoded);
33
34      }
35
36      return dataReceived;
37
38  }
39
40  uint8_t MASB_COMM_S_command(void) {
41
42      return rxBufferDecoded[0];
43
44  }
45
46  + struct CV_Configuration_S MASB_COMM_S_getCvConfiguration(void) {
47  +
48  +   struct CV_Configuration_S cvConfiguration;
49  +
50  +   union Double_Converter {
51  +       double d;
52  +       uint8_t b[8];
53  +   } doubleConverter;
54  +
55  +   b[0] = rxBufferDecoded[1];
56  +   b[1] = rxBufferDecoded[2];
57  +   b[2] = rxBufferDecoded[3];
58  +   b[3] = rxBufferDecoded[4];
59  +   b[4] = rxBufferDecoded[5];
60  +   b[5] = rxBufferDecoded[6];
61  +   b[6] = rxBufferDecoded[7];
62  +   b[7] = rxBufferDecoded[8];
63  +
64  +   cvConfiguration.eBegin = d;
65  +
66  +   b[0] = rxBufferDecoded[9];
67  +   b[1] = rxBufferDecoded[10];
68  +   b[2] = rxBufferDecoded[11];
69  +   b[3] = rxBufferDecoded[12];
70  +   b[4] = rxBufferDecoded[13];
71  +   b[5] = rxBufferDecoded[14];
72  +   b[6] = rxBufferDecoded[15];
73  +   b[7] = rxBufferDecoded[16];
74  +
75  +   cvConfiguration.eVertex1 = d;
76  +
77  - ...
78  - ...
```

```
79 -    ______  _____ _____  ______ _____
80 -   |  ____|/ ____|  __ \|  ____|  __ \     /\
81 -   | |__   | (___ | |__) | |__  | |__) |   /  \
82 -   |  __|  \___ \|  ___/|  __| |  _  /   / /\ \
83 -   | |____ ____) | |    | |____| | \ \  / ____ \
84 -   |______|_____/|_|    |______|_|  \_\/_/    \_\
85 - ...
86 - ...
87 +
88 + }
```

J***r, hay 6 parámetros a leer, llevo 2 y me ocupa 200 líneas, veo que estoy haciendo lo mismo para cada uno de los parámetros,… Todo esto **son indicios claros que hay que modular el código**. Esto así ni es fácil de leer, ni de mantener, ni de escalar. Vamos a borrar lo anterior y a crear una función que nos haga la conversión.

```
1  /*
2   * masb_comm_s.c
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  #include "components/masb_comm_s.h"
9
10 static UART_HandleTypeDef *huart;
11 static _Bool dataReceived = FALSE;
12 uint8_t rxIndex = 0;
13 uint8_t rxBuffer[UART_BUFF_SIZE] = { 0 };
14 uint8_t rxBufferDecoded[UART_BUFF_SIZE] = { 0 };
15
16 + static double saveByteArrayAsDoubleFromBuffer(uint8_t *buffer,
      uint8_t index);
17
18 void MASB_COMM_S_setUart(UART_HandleTypeDef *newHuart) {
19     huart = newHuart;
20 }
21
22 void MASB_COMM_S_waitForMessage(void) {
23
24     dataReceived = FALSE;
25     rxIndex = 0;
26     HAL_UART_Receive_IT(huart, &rxBuffer[rxIndex], 1);
27
28 }
29
30 _Bool MASB_COMM_S_dataReceived(void) {
31
32     if (dataReceived) {
33
```

```
34          COBS_decode(rxBuffer, rxIndex, rxBufferDecoded);
35
36      }
37
38      return dataReceived;
39
40  }
41
42  uint8_t MASB_COMM_S_command(void) {
43
44      return rxBufferDecoded[0];
45
46  }
47
48  + struct CV_Configuration_S MASB_COMM_S_getCvConfiguration(void) {
49  +
50  +    struct CV_Configuration_S cvConfiguration;
51  +
52  +    cvConfiguration.eBegin = saveByteArrayAsDoubleFromBuffer(
        rxBufferDecoded, 1);
53  +    cvConfiguration.eVertex1 = saveByteArrayAsDoubleFromBuffer(
        rxBufferDecoded, 9);
54  +    cvConfiguration.eVertex2 = saveByteArrayAsDoubleFromBuffer(
        rxBufferDecoded, 17);
55  +    cvConfiguration.cycles = rxBufferDecoded[25];
56  +    cvConfiguration.scanRate = saveByteArrayAsDoubleFromBuffer(
        rxBufferDecoded, 26);
57  +    cvConfiguration.eStep = saveByteArrayAsDoubleFromBuffer(
        rxBufferDecoded, 34);
58  +
59  +    return cvConfiguration;
60  +
61  + }
62
63  + static double saveByteArrayAsDoubleFromBuffer(uint8_t *buffer,
       uint8_t index) {
64  +
65  +    union Double_Converter {
66  +        double d;
67  +        uint8_t b[8];
68  +    } doubleConverter;
69  +
70  +    for (uint8_t i = 0; i < 8; i++) {
71  +
72  +        doubleConverter.b[i] = buffer[i + index];
73  +
74  +    }
75  +
76  +    return doubleConverter.d;
77  +
78  + }
```

Como mil veces mejor. Ahora tenemos una función que coge los 8 bytes de un `buffer` a partir de una determinada posición (`index`), los añade a una unión y los lee como un doble para devolvernos ese valor decimal. Fijaros que en el parámetros `cvConfiguration.cycles` no ha hecho falta la conversión puesto que solo ocupa un byte. También fijaros que la función `saveByteArrayAsDoubleFromBuffer()` la hemos declarado como estática ya que solo se utiliza en este archivo y que la hemos prototipado arriba del todo.

Hemos recibido los datos, ahora vamos a enviarlos. Toca la función `MASB_COMM_S_sendData()`.

```
1  /*
2   * masb_comm_s.c
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  #include "components/masb_comm_s.h"
9
10 static UART_HandleTypeDef *huart;
11 static _Bool dataReceived = FALSE;
12 uint8_t rxIndex = 0;
13 uint8_t rxBuffer[UART_BUFF_SIZE] = { 0 };
14 uint8_t rxBufferDecoded[UART_BUFF_SIZE] = { 0 };
15 + uint8_t txBuffer[UART_BUFF_SIZE] = { 0 };
16 + uint8_t txBufferDecoded[UART_BUFF_SIZE] = { 0 };
17
18 static double saveByteArrayAsDoubleFromBuffer(uint8_t *buffer, uint8_t
       index);
19 + static void saveLongAsByteArrayIntoBuffer(uint8_t *buffer, uint8_t
       index, uint32_t longVal);
20 + static void saveDoubleAsByteArrayIntoBuffer(uint8_t *buffer, uint8_t
       index, double doubleVal);
21
22 void MASB_COMM_S_setUart(UART_HandleTypeDef *newHuart) {
23     huart = newHuart;
24 }
25
26 void MASB_COMM_S_waitForMessage(void) {
27
28     dataReceived = FALSE;
29     rxIndex = 0;
30     HAL_UART_Receive_IT(huart, &rxBuffer[rxIndex], 1);
31
32 }
33
34 _Bool MASB_COMM_S_dataReceived(void) {
35
36     if (dataReceived) {
37
38         COBS_decode(rxBuffer, rxIndex, rxBufferDecoded);
```

```
39
40      }
41
42      return dataReceived;
43
44  }
45
46  uint8_t MASB_COMM_S_command(void) {
47
48      return rxBufferDecoded[0];
49
50  }
51
52  struct CV_Configuration_S MASB_COMM_S_getCvConfiguration(void) {
53
54      struct CV_Configuration_S cvConfiguration;
55
56      cvConfiguration.eBegin = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 1);
57      cvConfiguration.eVertex1 = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 9);
58      cvConfiguration.eVertex2 = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 17);
59      cvConfiguration.cycles = rxBufferDecoded[25];
60      cvConfiguration.scanRate = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 26);
61      cvConfiguration.eStep = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 34);
62
63      return cvConfiguration;
64
65  }
66
67  static double saveByteArrayAsDoubleFromBuffer(uint8_t *buffer, uint8_t
        index) {
68
69      union Double_Converter {
70          double d;
71          uint8_t b[8];
72      } doubleConverter;
73
74      for (uint8_t i = 0; i < 8; i++) {
75
76          doubleConverter.b[i] = buffer[i + index];
77
78      }
79
80      return doubleConverter.d;
81
82  }
83
```

```
84  + void MASB_COMM_S_sendData(struct Data_S data) {
85  +
86  +   saveLongAsByteArrayIntoBuffer(txBufferDecoded, 0, data.point);
87  +   saveLongAsByteArrayIntoBuffer(txBufferDecoded, 4, data.timeMs);
88  +   saveDoubleAsByteArrayIntoBuffer(txBufferDecoded, 8, data.voltage);
89  +   saveDoubleAsByteArrayIntoBuffer(txBufferDecoded, 16, data.current);
90  +
91  +   uint32_t txBufferLenght = COBS_encode(txBufferDecoded, 24, txBuffer
        );
92  +
93  +   txBuffer[txBufferLenght] = UART_TERM_CHAR;
94  +   txBufferLenght++;
95  +
96  +   HAL_UART_Transmit_IT(huart, txBuffer, txBufferLenght);
97  +
98  + }
99
100 + static void saveLongAsByteArrayIntoBuffer(uint8_t *buffer, uint8_t
      index, uint32_t longVal) {
101 +
102 +    union Long_Converter {
103 +       long l;
104 +       uint8_t b[4];
105 +    } longConverter;
106 +
107 +   longConverter.l = longVal;
108 +
109 +   for (uint8_t i = 0; i < 4; i++) {
110 +
111 +       buffer[i + index] = longConverter.b[i];
112 +
113 +   }
114 +
115 + }
116
117 + static void saveDoubleAsByteArrayIntoBuffer(uint8_t *buffer, uint8_t
      index, double doubleVal) {
118 +
119 +   union Double_Converter {
120 +       double d;
121 +       uint8_t b[8];
122 +   } doubleConverter;
123 +
124 +   doubleConverter.d = doubleVal;
125 +
126 +   for (uint8_t i = 0; i < 8; i++) {
127 +
128 +       buffer[i + index] = doubleConverter.b[i];
129 +
130 +   }
131 +
```

```
132 + }
```

Como en el caso anterior, hemos añadido las funciones `saveLongAsByteArrayIntoBuffer()` y `saveDoubleAsByteArrayIntoBuffer()` para realizar las conversiones de variables decimales a vectores de bytes.

Pues ya solo nos falta una última función que es el *callback* de la interrupción de recepción de un byte.

```
1  /*
2   * masb_comm_s.c
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  #include "components/masb_comm_s.h"
9
10 static UART_HandleTypeDef *huart;
11 static _Bool dataReceived = FALSE;
12 uint8_t rxIndex = 0;
13 uint8_t rxBuffer[UART_BUFF_SIZE] = { 0 };
14 uint8_t rxBufferDecoded[UART_BUFF_SIZE] = { 0 };
15 uint8_t txBuffer[UART_BUFF_SIZE] = { 0 };
16 uint8_t txBufferDecoded[UART_BUFF_SIZE] = { 0 };
17
18 static double saveByteArrayAsDoubleFromBuffer(uint8_t *buffer, uint8_t
     index);
19 static void saveLongAsByteArrayIntoBuffer(uint8_t *buffer, uint8_t
     index, uint32_t longVal);
20 static void saveDoubleAsByteArrayIntoBuffer(uint8_t *buffer, uint8_t
     index, double doubleVal);
21
22 void MASB_COMM_S_setUart(UART_HandleTypeDef *newHuart) {
23     huart = newHuart;
24 }
25
26 void MASB_COMM_S_waitForMessage(void) {
27
28     dataReceived = FALSE;
29     rxIndex = 0;
30     HAL_UART_Receive_IT(huart, &rxBuffer[rxIndex], 1);
31
32 }
33
34 _Bool MASB_COMM_S_dataReceived(void) {
35
36     if (dataReceived) {
37
38         COBS_decode(rxBuffer, rxIndex, rxBufferDecoded);
```

```
39
40      }
41
42      return dataReceived;
43
44  }
45
46  uint8_t MASB_COMM_S_command(void) {
47
48      return rxBufferDecoded[0];
49
50  }
51
52  struct CV_Configuration_S MASB_COMM_S_getCvConfiguration(void) {
53
54      struct CV_Configuration_S cvConfiguration;
55
56      cvConfiguration.eBegin = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 1);
57      cvConfiguration.eVertex1 = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 9);
58      cvConfiguration.eVertex2 = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 17);
59      cvConfiguration.cycles = rxBufferDecoded[25];
60      cvConfiguration.scanRate = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 26);
61      cvConfiguration.eStep = saveByteArrayAsDoubleFromBuffer(
            rxBufferDecoded, 34);
62
63      return cvConfiguration;
64
65  }
66
67  static double saveByteArrayAsDoubleFromBuffer(uint8_t *buffer, uint8_t
        index) {
68
69      union Double_Converter {
70          double d;
71          uint8_t b[8];
72      } doubleConverter;
73
74      for (uint8_t i = 0; i < 8; i++) {
75
76          doubleConverter.b[i] = buffer[i + index];
77
78      }
79
80      return doubleConverter.d;
81
82  }
83
```

```
84  void MASB_COMM_S_sendData(struct Data_S data) {
85
86      saveLongAsByteArrayIntoBuffer(txBufferDecoded, 0, data.point);
87      saveLongAsByteArrayIntoBuffer(txBufferDecoded, 4, data.timeMs);
88      saveDoubleAsByteArrayIntoBuffer(txBufferDecoded, 8, data.voltage);
89      saveDoubleAsByteArrayIntoBuffer(txBufferDecoded, 16, data.current);
90
91      uint32_t txBufferLenght = COBS_encode(txBufferDecoded, 24, txBuffer
            );
92
93      txBuffer[txBufferLenght] = UART_TERM_CHAR;
94      txBufferLenght++;
95
96      HAL_UART_Transmit_IT(huart, txBuffer, txBufferLenght);
97
98  }
99
100 static void saveLongAsByteArrayIntoBuffer(uint8_t *buffer, uint8_t
        index, uint32_t longVal) {
101
102     union Long_Converter {
103         long l;
104         uint8_t b[4];
105     } longConverter;
106
107     longConverter.l = longVal;
108
109     for (uint8_t i = 0; i < 4; i++) {
110
111         buffer[i + index] = longConverter.b[i];
112
113     }
114
115 }
116
117 static void saveDoubleAsByteArrayIntoBuffer(uint8_t *buffer, uint8_t
        index, double doubleVal) {
118
119     union Double_Converter {
120         double d;
121         uint8_t b[8];
122     } doubleConverter;
123
124     doubleConverter.d = doubleVal;
125
126     for (uint8_t i = 0; i < 8; i++) {
127
128         buffer[i + index] = doubleConverter.b[i];
129
130     }
131
```

```
132 }
133
134 + void HAL_UART_RxCpltCallback (UART_HandleTypeDef *huart) {
135 +
136 +   if (rxBuffer[rxIndex] == UART_TERM_CHAR) {
137 +       dataReceived = TRUE;
138 +   } else {
139 +       rxIndex++;
140 +       HAL_UART_Receive_IT(huart, &rxBuffer[rxIndex], 1);
141 +
142 +   }
143 + }
```

En este caso, no hace falta prototiparla ni podemos declararla como estática puesto que es el archivo `stm32f4xx_hal_uart.h` quien se encarga de ello.

Hasta aquí el archivo `masb_comm_s.c`.

3.4. Flujo de ejecución en el *stm32main*

Tenemos todas las funciones listas, pero no las estamos utilizando en ningún sitio. Vamos al componente `stm32main`.

Lo primero que debemos de hacer es hacer un `#include` de los componentes `masb_comm_s`.

```
1  /*
2   * stm32main.c
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
8  #include "components/stm32main.h"
9  + #include "components/masb_comm_s.h"
10
11 void setup(void) {
12
13 }
14
15 void loop(void) {
16
17 }
```

En la función `setup()` lo que haremos es pasarle el puntero con la configuración UART al componente `masb_comm_s`. Pero… uixxx… En el `stm32main` no tenemos ese puntero. Está en el `main.c`. Pues haremos que el `main.c` se lo pase a `stm32main` y éste a `masb_comm_s`. **Ahora no añadas lo que voy a mostrar hasta que te avise.**

En el `stm32main.c` hacemos:

```
/*
 * stm32main.c
 *
 *  Created on: Apr 14, 2020
 *      Author: albert
 */

#include "components/stm32main.h"
#include "components/masb_comm_s.h"

-void setup(void) {
+ void setup(UART_HandleTypeDef *huart) {
+     MASB_COMM_S_setUart(huart);
}

void loop(void) {

}
```

Y en el `stm32main.h`:

```
/*
 * stm32main.h
 *
 *  Created on: Apr 14, 2020
 *      Author: albert
 */

#ifndef INC_COMPONENTS_STM32MAIN_H_
#define INC_COMPONENTS_STM32MAIN_H_

// Prototypes.
- void setup(void);
+ void setup(UART_HandleTypeDef *huart);
void loop(void);

#endif /* INC_COMPONENTS_STM32MAIN_H_ */
```

Perfecto. ¿Funcionaría? Sí, pero ahora veamos que pasará cuando añadas durante el proyecto el ADC, luego el I2C, el SPI,…

```
/*
 * stm32main.c
 *
 *  Created on: Apr 14, 2020
 *      Author: albert
 */

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

```
#include "components/masb_comm_s.h"

- void setup(UART_HandleTypeDef *huart) {
+ void setup(UART_HandleTypeDef *huart, ADC_HandleTypeDef *hadc,
    I2C_HandleTypeDef *hi2c, SPI_HandleTypeDef *hspi) {
      MASB_COMM_S_setUart(huart);
}

void loop(void) {

}
```

```
/*
 * stm32main.h
 *
 *  Created on: Apr 14, 2020
 *      Author: albert
 */

#ifndef INC_COMPONENTS_STM32MAIN_H_
#define INC_COMPONENTS_STM32MAIN_H_

// Prototypes.
- void setup(UART_HandleTypeDef *huart);
+ void setup(UART_HandleTypeDef *huart, ADC_HandleTypeDef *hadc,
    I2C_HandleTypeDef *hi2c, SPI_HandleTypeDef *hspi);
void loop(void);

#endif /* INC_COMPONENTS_STM32MAIN_H_ */
```

No es un problemón en este caso porque utilizamos pocos periféricos, pero se puede ver que la lista va creciendo a medida que vas incrementando el número de periféricos y que, además, has de tocar más de un archivo. En su lugar, y **a partir de aquí ya puedes ir añadiendo lo que vaya mostrando**, vamos a crear una estructura que contenga todos punteros a las diferentes configuraciones de los diferentes periféricos y sea esa estructura lo que vayamos pasando entre funciones. Vamos allá.

En `stm32main.h` definimos la estructura y cambiamos el parámetro que acepta la función `setup()`.

```
/*
 * stm32main.h
 *
 *  Created on: Apr 14, 2020
 *      Author: albert
 */

#ifndef INC_COMPONENTS_STM32MAIN_H_
#define INC_COMPONENTS_STM32MAIN_H_

+ #include "stm32f4xx_hal.h"

```

```
13 + struct Handles_S {
14 +     UART_HandleTypeDef *huart;
15 +     // Aqui iriamos anadiendo los diferentes XXX_HandleTypeDef que
       necesitaramos anadir.
16 + };
17
18 // Prototypes.
19 - void setup(void);
20 + void setup(struct Handles_S *handles);
21 void loop(void);
22
23 #endif /* INC_COMPONENTS_STM32MAIN_H_ */
```

Y en el archivo `stm32main.c` haríamos:

```
 1 /*
 2  * stm32main.c
 3  *
 4  *  Created on: Apr 14, 2020
 5  *      Author: albert
 6  */
 7
 8 #include "components/stm32main.h"
 9 #include "components/masb_comm_s.h"
10
11 - void setup(void) {
12 + void setup(struct Handles_S *handles) {
13 +     MASB_COMM_S_setUart(handles->huart);
14 }
15
16 void loop(void) {
17
18 }
```

Y ya está. Ajora solo falta añadir a la función `setup()` que el componente `masb_comm_s` espere el primer byte.

```
 1 /*
 2  * stm32main.c
 3  *
 4  *  Created on: Apr 14, 2020
 5  *      Author: albert
 6  */
 7
 8 #include "components/stm32main.h"
 9 #include "components/masb_comm_s.h"
10
11 void setup(struct Handles_S *handles) {
12     MASB_COMM_S_setUart(handles->huart);
13 +   MASB_COMM_S_waitForMessage();
14 }
```

```
15
16 void loop(void) {
17
18 }
```

No os olvidéis de en el archivo `main.c`pasar el puntero de la UART a la función `setup()`.

```
1  ...
2
3  /* Private includes
      ----------------------------------------------------------*/
4  /* USER CODE BEGIN Includes */
5  #include "components/stm32main.h"
6  /* USER CODE END Includes */
7
8  ...
9
10   /* USER CODE BEGIN 2 */
11
12   struct Handles_S myHandles;
13   myHandles.huart = &huart2;
14
15   setup(&myHandles);
16
17   /* USER CODE END 2 */
18
19   /* Infinite loop */
20   /* USER CODE BEGIN WHILE */
21   while (1)
22   {
23       loop();
24     /* USER CODE END WHILE */
25
26     /* USER CODE BEGIN 3 */
27   }
28
29 ...
```

Ahora nos metemos con la función `loop()`, donde tendrá lugar el flujo principal de nuestra aplicación. **Haremos lo mismo que en la sesión anterior**. Comprobaremos si hemos recibido un mensaje. Si es así, leemos el comando para ver qué tenemos que hacer. En función del comando recibido hacemos una cosa o la otra. Repito, **igual que en la sesión anterior**.

```
1  /*
2   * stm32main.c
3   *
4   *  Created on: Apr 14, 2020
5   *      Author: albert
6   */
7
```

```
 8  #include "components/stm32main.h"
 9  #include "components/masb_comm_s.h"
10
11  + struct CV_Configuration_S cvConfiguration;
12  + struct Data_S data;
13
14  void setup(struct Handles_S *handles) {
15      MASB_COMM_S_setUart(handles->huart);
16      MASB_COMM_S_waitForMessage();
17  }
18
19  void loop(void) {
20  +     if (MASB_COMM_S_dataReceived()) { // Si se ha recibido un mensaje
        ...
21  +
22  +         switch(MASB_COMM_S_command()) { // Miramos que comando hemos
        recibido
23  +
24  +             case START_CV_MEAS: // Si hemos recibido START_CV_MEAS
25  +
26  +                 // Leemos la configuracion que se nos ha enviado en
        el mensaje y
27  +                 // la guardamos en la variable cvConfiguration
28  +                 cvConfiguration = MASB_COMM_S_getCvConfiguration();
29  +
30  +                 /* Mensaje a enviar desde CoolTerm para hacer
        comprobacion
31  +                  * eBegin = 0.25 V
32  +                  * eVertex1 = 0.5 V
33  +                  * eVertex2 = -0.5 V
34  +                  * cycles = 2
35  +                  * scanRate = 0.01 V/s
36  +                  * eStep = 0.005 V
37  +                  *
38  +                  * Mensaje previo a la codificacion (lo que teneis que
        poder obtener en el microcontrolador):
39  +                  * 01000000000000
        D03F000000000000E03F000000000000E0BF027B14AE47E17A843F7B14AE47E17A743F
        
40  +                  *
41  +                  * Mensaje codificado que enviamos desde CoolTerm (
        incluye ya el termchar):
42  +                  * 0201010101010103
        D03F010101010103E03F010101010114E0BF027B14AE47E17A843F7B14AE47E17A743F00
        
43  +                  */
44  +                 __NOP(); // Esta instruccion no hace nada y solo sirve
        para poder anadir un breakpoint
45  +
46  +                 // Aqui iria todo el codigo de gestion de la medicion
        que hareis en el proyecto
```

```
47 +                // si no quereis implementar el comando de stop.
48 +
49 +                break;
50 +
51 +            case STOP_MEAS: // Si hemos recibido STOP_MEAS
52 +
53 +                /*
54 +                 * Debemos de enviar esto desde CoolTerm:
55 +                 * 020300
56 +                 */
57 +                __NOP(); // Esta instruccion no hace nada y solo sirve
   para poder anadir un breakpoint
58 +
59 +                // Aqui iria el codigo para tener la medicion si
   implementais el comando stop.
60 +
61 +                break;
62 +
63 +            default: // En el caso que se envia un comando que no
   exista
64 +
65 +                // Este bloque de codigo solo lo tenemos para hacer
   el test de que podemos enviar los
66 +                // datos correctamene. EN EL PROYECTO FINAL TENEMOS
   QUE ELIMINAR ESTE CODIGO y pondremos
67 +                // lo que se nos indique en los requerimientos del
   proyecto. Repito, es solo para
68 +                // comprobar que podemos enviar datos del
   microcontrolador al PC.
69 +
70 +                data.point = 1;
71 +                data.timeMs = 100;
72 +                data.voltage = 0.23;
73 +                data.current = 12.3e-6;
74 +
75 +                /*
76 +                 * Debemos de enviar esto desde CoolTerm (un comando
   inventado):
77 +                 * 010100
78 +                 *
79 +                 * Datos en el microcontrolador antes de la
   codificacion:
80 +                 * 0100000064000000713D0AD7A370CD3F7050B12083CBE93E
81 +                 *
82 +                 * Datos codificados que debes de recibir en CoolTerm:
83 +                 * 020101010264010111713D0AD7A370CD3F7050B12083CBE93E00
84 +                 */
85 +
86 +                // Enviamos los datos
87 +                MASB_COMM_S_sendData(data);
88 +
```

```
89  +                break;
90  +
91  +        }
92  +
93  +        // Una vez procesado los comando, esperamos el siguiente
94  +        MASB_COMM_S_waitForMessage();
95  +
96  +    }
97  +
98  +    // Aqui es donde deberia de ir el codigo de control de las
        mediciones si se quiere implementar
99  +    // el comando de STOP.
100 +
101 }
```

Os he adjuntado como comentarios el mensaje que debéis de enviar desde CoolTerm para hacer las pruebas y los datos que debéis de esperar recibir desde el microcontrolador (y viceversa para el caso de enviar datos desde el microcontrolador al PC).

Con esto ya tenemos implementado **parcialmente** la comunicación serie asíncrona de nuestro proyecto…

4. Reto

Pero nos falta algo… Solo hemos hecho la parte de la voltametría cíclica. **¡Falta la parte de la cronoamperometría!** Y ese es el reto. Implementa una función `MASB_COMM_S_getCaConfiguration()` en el componente `masb_comm_s` que lea la configuración de la cronoamperometría y la devuelva en forma de estructura (como en la voltametría cíclica). Acuérdate de crear esa estructura en un archivo `chrono_amperometry.h`, incluirlo en el archivo `masb_comm_s.h`, implementar una función estática `saveByteArrayAsLongFromBuffer()` (un **`long`** son 4 bytes) en el componente `masb_comm_s` y añadir la gestión del comando `START_CA_MEAS` en el archivo `stm32main.c`.

Para que puedas hacer test, te dejo los datos que podéis enviar desde CoolTerm para una cronoamperometría:

```
1  /* Mensaje a enviar desde CoolTerm para hacer comprobacion
2   * eDC = 0.3 V
3   * samplingPeriodMs = 10 ms
4   * measurementTime = 120 s
5   *
6   * Mensaje previo a la codificacion (lo que teneis que poder obtener en
        el microcontrolador):
7   * 02333333333333D33F0A00000078000000
8   *
```

```
 9   * Mensaje codificado que enviamos desde CoolTerm (incluye ya el
       termchar):
10   * 0B02333333333333D33F0A0101027801010100
11   */
```

5. Evaluación

5.1. 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** Implementación de la parte de cronoamperometría.
- ☐ ~~**Informe**~~ No hay informe.

5.2. Pull Request

Cread un *Pull Request* (PR) de vuestra rama a la `master`. Acordaos de ponerme como *Reviewer*.

5.3. 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.

6. Conclusiones

Duro… **Ha sido duro**, pero no por la cantidad de código (que es relativamente poco), sino porque hemos visto cómo haríamos una implementación de manera medianamente profesional de un set de instrucciones. Aún hay alguna cosilla que podríamos hacer diferente para añadir una mayor abstracción o lograr una menor dependencia entre componentes, ¡pero **nada mal lo que hemos logrado**! **¡Ahora tenemos ya medio proyecto hecho!** Ya tenemos que nuestro microcontrolador pueda leer comandos desde el PC y podemos enviar datos al mismo mediante un set de instrucciones que hemos implementado con una comunicación serie asíncrona.

En la siguiente práctica, la última, haremos un comunicación serie síncrona, pero de una manera especial… Haremos un 2 por 1 (haremos I2C y SPI) y encima no os lo explicaré yo, sino que uno explicará I2C al otro y el otro SPI al primero. Pero ya os daré más detalles en la siguiente práctica…