MASB | STM32
Comunicación serie I

#stm32 #c #biomedical #microcontroladores #programacion

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

6 de marzo de 2020

Índice

1. Comunicación serie - Parte I

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

En esta parte nos ahorramos la teoría que ya vimos en la primera parte de esta práctica y vamos directamente al asunto. En esta parte de la práctica veremos **cómo utilizar la función `sprintf`**, que utilizamos en la práctica anterior sin saber muy bien qué hacía, para enviar datos en codificación ASCII. Seguidamente, también veremos **cómo enviar datos en *raw***.

Puesto que esto nos llevará poco rato, aprovecharemos para introducir un **flujo de trabajo o *workflow* en equipo** para el desarrollo de proyectos basados en STM32CubeIDE. Veremos que no es trivial el *workflow* a utilizar. ¡Así que estad muy atentos e intentad comprender todo lo que vamos haciendo puesto que **será el *workflow* que deberéis de utilizar en vuestro proyecto**!

1.1. Objetivos

- Introducción a la comunicación serie en STM32CubeIDE.
- Función sprintf
- Envío de datos *raw*.
- *Workflow* de trabajo en equipo en STM32CubeIDE.

1.2. Procedimiento

1.2.1. Enviemos un ASCII, va...

Empezamos con el envío de “Hola, soy STM32.” en ASCII del microcontrolador al ordenador.

1.2.1.1. Creación y configuración del proyecto

En nuestra rama de desarrollo, vamos a crear un proyecto con el nombre **`masb-p05` en la carpeta `stm32cube`** de nuestro repositorio local. Lo primero que vamos a hacer es ver cómo **configurar el periférico USART** de nuestro microcontrolador. Este periférico es el que se encargará de la comunicación serie asíncrona.

Voy a corregirme a mi mismo: vamos a ver cómo STM32CubeMX nos ha configurado la USART. Al inicializar los periféricos, **la USART se configura por defecto** a una configuración **115200 8N1**. Vamos a ver cómo puesto que en el reto deberemos de configurar un segundo periférico USART que no está configurado por defecto.

Lo primero que debemos fijarnos es en el `Pinout view`. Ahí podemos ver como los pines **PA2 y PA3 son los pines TX y RX**, respectivamente, de nuestra comunicación asíncrona con el ordenador. Los **GNDs** del microcontrolador y el ordenador están **conectados entre sí mediante el cable USB**.

Podemos ver cómo STM32CubeMX ha configurado la USART desde el **menú lateral de la izquierda `Connectivity` > `USART2`**.

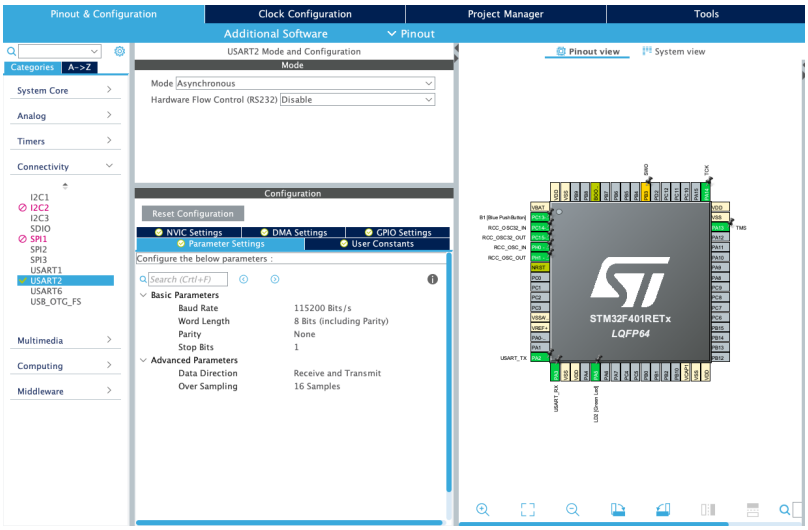

Figura 1: Configuración de la USART2.

Como vemos, es muy sencillo configurar la USART. Simplemente, seleccionamos **modo asíncrono** en el campo `Mode` y en la parte inferior seleccionamos los **parámetros de la comunicación**: *baud rate*, bits de datos, paridad y número de bits de stop. Podemos escoger si hacer una comunicación bidireccional o unidireccional (y dirección). Por último, tenemos el *over sampling*. Esto es una característica de cómo captura el microcontrolador los bits que recibe sin utilizar una señal de reloj. No nos meteremos en qué hace, pero lo dejaremos en 16 muestras.

Facilismo. Ahora **configurad el pulsador con una interrupción** de tal modo que **en el *callback* de esa interrupción haremos el envío de la frase "Hola, soy STM32."**. No. No os voy a decir cómo configurar los GPIOs y sus interrupciones.

Una vez lo tengamos, guardamos el archivo de configuración, generamos el código y nos vamos al `main.c` a enviar ASCII.

1.2.1.2. Datos codificados en ASCII

Ya en nuestro querido `main.c`, el periférico USART2 ya está inicializado y listo para ser usado. Implementamos el ***callback* de nuestra interrupción del GPIO** y **allí utilizamos** la función `HAL_USART_Transmit`. Esta función recibe como **parámetros** la **configuración de la USART**, un **puntero a un *array*** con que contendrá los datos a enviar, un número que indicará,

de ese *array*, **cuántos elementos queremos enviar**, y un **tiempo máximo**, en milisegundos, para realizar la acción. El *callback* quedaría así:

```
...

/* USER CODE BEGIN 4 */

void HAL_GPIO_EXTI_Callback (uint16_t GPIO_Pin) {

    // enviamos <txBufferElements> paquetes de los datos en txBuffer
    HAL_UART_Transmit(&huart2, &txBuffer, txBufferElements, 100);

}

/* USER CODE END 4 */

...
```

Como veis, hemos utilizado dos variables `txBuffer` y `txBufferElements`. La primera es un *array* donde guardaremos los elementos a enviar. La segunda es el número de elemento de ese *array* que vamos a enviar. Vamos a crear esas variables:

```
...

/* USER CODE BEGIN PV */
uint8_t txBuffer[TX_BUFFER_SIZE] = { 0 };
uint8_t txBufferElements = 0;
/* USER CODE END PV */

...
```

Podéis ver como para crear el *array*, hemos indicado el número de elementos de ese *array* (su tamaño) mediante una macro. **Definid esa macro** en vuestro `main.c` y poned un **valor de 32**.

> Debéis escoger un tamaño para el *buffer* acorde a vuestra aplicación. Ni demasiado grande porque ocuparéis espacio de memoria inútilmente, ni demasiado pequeño porque os faltará espacio para guardar todo aquello que queráis guardar. En este caso, el texto “Hola, soy STM32.” ocupará 16 bytes más 2 carácteres de *term char* (`\r+\n`). Un *buffer* con un tamaño de 32 elementos será más que suficiente.

Una vez definido el *buffer*, vamos a almacenar el texto a enviar en él. Lo haremos mediante la función `sprintf`.

1.2.1.3. La función sprintf

La **función** `sprintf` forma parte de la **librería estándar** `stdio.h`, por lo que, aunque el programa os deje compilar sin error, deberemos de hacer un ***include*** **de esa**

librería con `#include <stdio.h>`. A la función le pasaremos **mínimo dos parámetros**: un puntero al *buffer* donde queremos que se guarde el texto y el *pattern* del texto.

El texto lo queremos guardar en el *buffer* `txBuffer` que hemos creado. Cada elemento del *array* contendrá un carácter en codificación ASCII. Como debemos de pasarle el puntero, se lo haremos llegar utilizando el operador `&` que ya hemos visto: `&txBuffer`.

El segundo parámetro es un *pattern* de texto. Es la plantilla de texto que guardaremos en `txBuffer`. ¿Porqué decimos que *pattern*/plantilla de texto? Porque podemos utilizar *tags* (que se identifican por el carácter `%`) que pueden ser sustituidos por valores que pasemos como tercer, cuarto, quinto,..., parámetro de la función `sprintf`. Algunos ejemplos:

```
1  txBufferElements = sprintf(&txBuffer, "Hola, soy STM32.\r\n");
```

> En este ejemplo, no utilizamos ningún *tag* y tal cual el texto que hemos indicado se guardará en `txBuffer` en codificación ASCII.

```
1  txBufferElements = sprintf(&txBuffer, "Hola, soy STM%u.\r\n", 32);
```

> En este caso, hemos guardado exactamente el mismo texto, lo único que hemos utilizado un *tag* (`%u`) para que sea reemplazado por el siguiente parámetro de la función (32). Los parámetros después de la plantilla de texto podemos indicarlos con un número directamente o mediante una variable que contenga ese número.

```
1  txBufferElements = sprintf(&txBuffer, "Hola, soy STM%u (%u).\r\n", 32,
       year);
```

> Ahora enviamos el mismo texto, pero añadiendo entre paréntesis el año que hemos pasado como parámetro en la función mediante una variable.

En este enlace encontraréis otra explicación sobre la función `sprintf` y una lista con todos lo *tags* disponibles ya que, según el tipo de variable y el formato que queramos darle, tendremos que utilizar un *tag* u otro.

Además de aceptar parámetros, **la función `sprintf` devuelve un valor**. Ese valor es el **número de carácteres que tiene el texto guardado**. Lo utilizaremos para luego indicar a la USART el número de elementos que debe de enviar. Guardamos ese valor en la variable `txBufferElements`. Nuestra función `main` quedaría del siguiente modo:

```
1  ...
2
3    /* USER CODE BEGIN 2 */
4
```

```
  txBufferElements = sprintf(&txBuffer, "Hola, soy STM32.\r\n");

  /* USER CODE END 2 */

...
```

Compilamos, depuramos e iniciamos la ejecución del programa. **Abrimos CoolTerm** y en las opciones seleccionamos el **puerto pertinente** y configuramos un ***baud rate* de 115200 bps**. Clicamos en el icono `Connect` y si pulsamos el pulsador de nuestra EVB, y todo está correcto, nuestro microcontrolador nos saludará por pantalla.

1.2.2. Recibir datos *raw* con interrupciones

Me niego a que veamos cómo recibir datos en ASCII. Vamos a ver directamente cómo recibir datos en *raw*. Además, veremos cómo hacerlo con interrupciones. Así sabemos que el programa no se nos quedará colgado a la espera de recibir algo.

Vamos a apagar/encender el LED con las instrucciones `0x01` y `0x02`, respectivamente.

1.2.2.1. Configuración de interrupciones

Abrimos nuestro **archivo `.ioc`** y nos vamos al formulario de **configuración de USART2**. En la **pestaña `NVIC Settings` del formulario inferior**, **habilitamos `USART2 global interrupt`**. Hala. Interrupciones configuradas.

1.2.2.2. *Callback* y gestión de recepciones

En la documentación podemos encontrar fácilmente cómo habilitar la recepción mediante interrupciones y cuál es el *callback* a utilizar. Para habilitar la recepción mediante interrupciones utilizamos la **función `HAL_UART_Receive_IT`**. Debemos de indicar **dónde queremos guardar los bytes** recibidos y **cuántos esperamos recibir**. Crearemos un *buffer* para guardar los **bytes recibidos `rxBuffer`**. Puesto que solo esperamos recibir instrucciones de un solo byte, crearemos un *array* de un solo elemento. El `main.c` tomaría esta forma:

```
...

/* USER CODE BEGIN PV */
uint8_t rxBuffer[RX_BUFFER_SIZE] = { 0 };
/* USER CODE END PV */

...
```

```
...

  /* USER CODE BEGIN 2 */

```

```
  // decimos que estamos a la espera de recibir 1 byte
  HAL_UART_Receive_IT(&huart2, &rxBuffer, 1);

  /* USER CODE END 2 */

...
```

Cuando se realice la recepción de ese byte, saltará la interrupción de recepción: el *callback*. La función de *callback* es `HAL_UART_RxCpltCallback`. La implementamos del siguiente modo:

```
...

/* USER CODE BEGIN 4 */

void HAL_UART_RxCpltCallback (UART_HandleTypeDef *huart) {

    // hemos recibido un byte
    // ahora le decimos que volvemos a estar a la espera de recibir 1
       byte
    HAL_UART_Receive_IT(&huart2, &rxBuffer, 1);

}

/* USER CODE END 4 */

...
```

Con esto tendríamos la recepción de un byte lista. Falta hacer que el LED se encienda/apague en función del byte recibido. Puede ser tentador ponerlo dentro del *callback*, pero ¿que ocurriría si mientras estamos gestionando el LED recibimos otro byte? Pues que se perdería, básicamente. Hay que intentar que las ISR/*callbacks* sean lo más cortas posibles precisamente para evitar esto.

Por ello, **tendríamos que hacer la gestión del LED directamente en el *while loop* de la función `main`**. De momento, l**o haremos en el *callback***. En un par de prácticas veremos cómo hacerlo bien.

> Veréis que dedicaremos unas cuantas prácticas a comunicaciones porque, mientras que es muy "sencillo" operar de manera individual los periféricos de un microcontrolador, cuando tienes que comunicarte con alguien más, como en la vida misma, la cosa se complica. Por ello, repartiremos todo lo relacionado con comunicaciones en varias prácticas para no saturarnos al intentar verlo todo en una, que sería imposible…

Primeramente, configurad el pin del **LED como GPIO de salida con el *label* `LED`** en el archivo `.ioc` y re-generad el código. Una vez hecho, nuestro *callback* debería verse tal que así:

```
...

/* USER CODE BEGIN 4 */
```

```
4
5  void HAL_UART_RxCpltCallback (UART_HandleTypeDef *huart) {
6
7      switch(rxBuffer[0]) {
8      case 0x01:
9        HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET);
10       break;
11     case 0x02:
12       HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET);
13       break;
14     default:
15       break;
16     }
17
18     // hemos recibido un byte
19     // ahora le decimos que volvemos a estar a la espera de recibir 1
          byte
20     HAL_UART_Receive_IT(&huart2, &rxBuffer, 1);
21
22 }
23
24 /* USER CODE END 4 */
25
26 ...
```

Compilamos, depuramos e iniciamos el programa. Si todo está bien, **enviamos un 1 en hexadecimal desde el CoolTerm y se debería de encender el LED**. **Con un 2 deberíamos de apagarlo.** Pero… ~~J***r~~, con un simple Arduino devolvíamos un **byte de *status* para indicar si se nos había enviado una instrucción válida (`0x00`) o no (`0x01`)**. Vamos a implementarlo en un momento:

```
1  ...
2
3  /* USER CODE BEGIN 4 */
4
5  void HAL_UART_RxCpltCallback (UART_HandleTypeDef *huart) {
6
7      switch(rxBuffer[0]) {
8      case 0x01:
9        HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET);
10       txBuffer[0] = 0x00;
11       break;
12     case 0x02:
13       HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET);
14       txBuffer[0] = 0x00;
15       break;
16     default:
17       txBuffer[0] = 0x01;
18       break;
19     }
20
```

```
21      // enviamos el byte de status
22      HAL_UART_Transmit(&huart2, &txBuffer, 1, 100);
23
24      // hemos recibido un byte
25      // ahora le decimos que volvemos a estar a la espera de recibir 1
            byte
26      HAL_UART_Receive_IT(&huart2, &rxBuffer, 1);
27
28  }
29
30  /* USER CODE END 4 */
31
32  ...
```

Hemos utilizado un *buffer* de transmisión con un solo elemento `txBuffer`. Probamos el programa y deberíamos de poder encender y apagar el LED con `0x01` y `0x02` y recibir un *status* igual a `0x00`. Si enviamos una instrucción distinta, el microcontrolador nos debería de devolver un *status* `0x01`.

1.3. Reto: *workflow* con STM32CubeIDE

El reto consistirá en un **proyecto en equipo**. En este reto **conectaremos dos EVB mediante una comunicación serie asíncrona**. A su vez, cada EVB establecerá una **segunda comunicación serie asíncrona con el ordenador** de tal modo que todo lo que se reciba de la otra EVB se enviará al ordenador. De este modo, tendremos un **chat entre ordenadores**. Enviaremos los datos en ASCII para que podamos leerlos…

En este reto aprenderemos dos cosas nuevas: como **trabajar en proyectos con más de un archivo** y cómo **trabajar en equipo con proyectos basados STM32CubeIDE**.

1.3.1. *Workflow* en equipo

Vamos a **simplificar** el *workflow* al máximo. Veremos que al simplificar tanto el *workflow*, este tendrá algunas **limitaciones**. Este es el precio a pagar. Otra opción seria utilizar un flujo de trabajo avanzado que no haría más que hacernos la cabeza un lío ahora que estamos simplemente introduciéndonos en este mundo de los flujos de trabajo en equipo. Por ahora, leed la descripción del flujo de trabajo y en la siguiente sección lo llevaremos a cabo de manera guiada.

A continuación, tenéis el árbol git que seguiremos en el repositorio para realizar nuestro proyecto. Apoyaros en esta figura para seguir la explicación del flujo de trabajo.

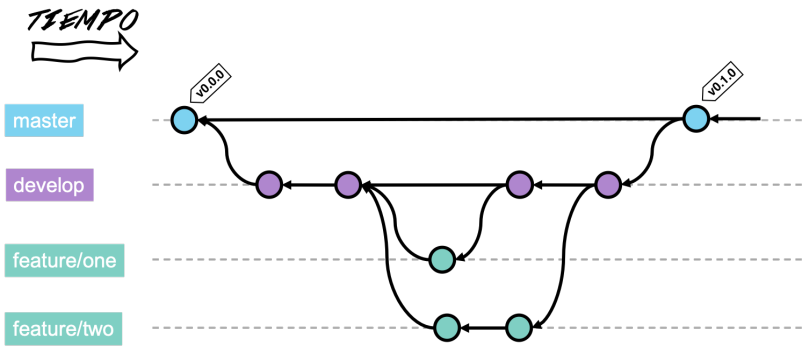

Figura 2: Git tree del repositorio.

Empezamos por ver para qué es y qué haremos en cada rama:

- **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.

Cada vez que queramos iniciar el desarrollo de una funcionalidad nueva, lo haremos creando una rama `feature/***` desde a rama `develop`. En esta rama `feature/***` **solo modificaremos archivos que hayamos creado nosotros y no generados automáticamente por STM32CubeMX**. **Esto último es muy importante**. Una vez finalizado el desarrollo y testeado, lo incorporamos a la rama `develop` con un *Pull Request*.

Si necesitamos modificar el archivo de configuración `.ioc`, lo haremos directamente en la rama `develop`. **En esta rama solo modificaremos el archivo `.ioc` y los archivos generados automáticamente por STM32CubeIDE, incluido `main.c`. Eso último es muy importante.**

Aquí viene la **limitación**: **solo una persona puede modificar la rama `develop` a la vez**. Si lo hacen dos a la vez, al unir las ramas habrán conflictos. ¿Cómo procedemos? Lo óptimo, y por lo tanto lo que recomendaré, es que **solo una persona se dedique a modificar la rama `develop`**. Cuando se tenga que modificar el archivo `.ioc` o un archivo generado por STM32CubeIDE, **deberéis de sentaros juntos delante de un mismo ordenador y editarlo conjuntamente** (que uno le diga al otro que es lo que necesita que se configure).

¿Es un rollo? Sí. Pero es el **precio a pagar por un *workflow* más sencillo.**

Si habéis creado la rama `feature` antes de realizar la configuración o si habéis cambiado la configuración en medio del desarrollo de una funcionalidad para corregir un error, ese cambio de configuración no estará en vuestra rama `feature`. Tendréis que hacer un `git rebase`. Chuleta de cómo deberéis de hacerlo:

```
git checkout develop
git pull
git checkout feature/***
git rebase develop
git push --force-with-lease
```

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).

Hasta aquí la descripción. Vamos a hacerlo todo paso por paso.

1.3.2. Repartición de tareas

La estructura de directorios/archivos de la carpeta `Core` del proyecto basado en STM32CubeIDE debe de ser la siguiente:

```
Core
- Inc
  - components
    - stm32main.h
    - usart1.h
    - usart2.h
  - main.h
  - stm32f4xx_hal_conf.h
  - stm32f4xx_it.h
- Src
  - components
    - stm32main.c
    - usart1.c
    - usart2.c
  - main.c
  - stm32f4xx_hal_msp.c
  - stm32f4xx_it.c
  - syscalls.c
  - sysmem.c
  - system_stm32f4xx.c
- Startup
  - startup_stm32f401retx.s
```

Crearemos dos carpetas, ambas con el nombre `components`, en `Core` > `Src` y en `Core` > `Inc`. En la carpeta `Core` > `Src` > `components` guardaremos nuestros propios archivos de código fuente.

En la carpeta `Core` > `Inc` > `components` guardaremos nuestros propios archivos de encabezado o *header*.

Los archivos que deben de trabajar cada miembro son:

- Miembro `A`
  - Creación del proyecto.
  - Configuración del archivo `.ioc`.
  - Generación del código.
  - `Core` > `Src` > `main.c`
  - `Core` > `Src` > `components` > `usart1.c`
  - `Core` > `Inc` > `components` > `usart1.h`
- Miembro `B`
  - `Core` > `Src` > `components` > `stm32main.c`
  - `Core` > `Inc` > `components` > `stm32main.h`
  - `Core` > `Src` > `components` > `usart2.c`
  - `Core` > `Inc` > `components` > `usart2.h`

Pese a que el miembro `A` tiene las tareas de crear el proyecto, configurarlo, generar el código y modificar un aspecto del código de `main.c` que ya veremos, **esto se realiza conjuntamente en un mismo ordenador ambos miembros**. Una vez estas tareas se hayan realizado, cada miembro podrá volver a sus tareas de manera individual.

Escoged quién hace las tareas del miembro `A` y quién las de miembro `B`. Antes de empezar tu tarea, asegúrate de leer las tareas que debe de hacer el otro miembro para que en todo momento sepas cómo ha de llevarse a cabo el flujo de trabajo.

1.3.3. *Workflow* del miembro A

1.3.3.1. Creación y configuración del proyecto Tienes la responsabilidad de crear el proyecto y **hasta que tú no acabes el otro miembro no puede empezar a desarrollar**. Vamos a empezar por **ir a la rama `master`** de vuestro repositorio y asegúrate de **importar cualquier posible cambio** que haya podido haber en el repositorio remoto.

```
1 git checkout master
2 git pull
```

A continuación, **crea una rama llamada `develop`**. Allí uniréis tu desarrollo y el de tu compañero para hacer pruebas antes de pasarlo a la `master`.

```
1 git checkout -b develop
```

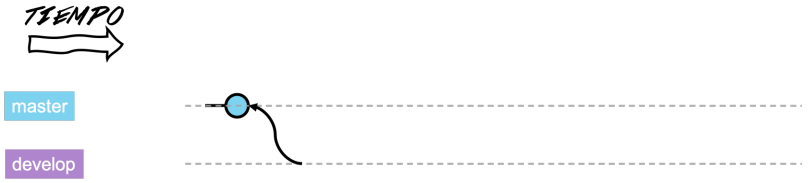

Figura 3: Git.

Ya en la rama, **crea un proyecto en STM32CubeIDE con el nombre `masb-p05-reto` en la carpeta `stm32cube`** del repositorio local. Una vez creado, **configura el microcontrolador** con STM32CubeMX.

> Recordad que esto lo hacéis juntos en un solo ordenador.

Solo tenéis que configurar un único periférico: la USART1. La USART2 se usa para la comunicación entre el ordenador y el microcontrolador. **En la USART2** simplemente **habilitad las interrupciones**. La USART1 la utilizaremos para comunicar nuestro microcontrolador con el de nuestro compañero. Configura la **USART1** con la misma configuración que la USART2: **115200 8N1** (no te olvides de activar también las interrupciones). Una vez configurado, guarda el archivo de configuración y genera el código. Haz un ***commit* de los archivos modificados**. No te olvides del ***push***.

```
1 git add -A
2 git commit -m "Proyecto creado y configurado."
3 git push
```

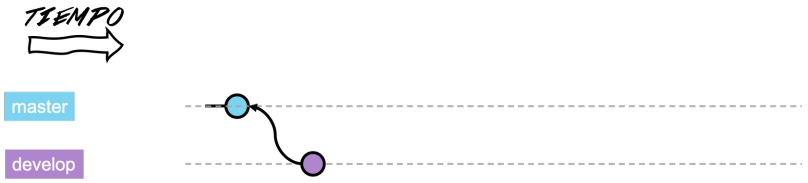

Figura 4: Git.

1.3.3.2. Modificación de main.c Vamos a **modificar el archivo `main.c`** para hacerlo lo suficientemente genérico para que no tengamos que estar modificándolo cada dos por tres. Seguro que lo que haremos os suena familiar…

Abre el archivo `main.c`. Modifícalo para **añadir un *include* a un archivo que creará el miembro `B`**.

```
...

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

...
```

Seguidamente, añade lo siguiente en la función `main`...

```
...

    /* USER CODE BEGIN 2 */
  setup();
  /* USER CODE END 2 */

...
```

```
...

  /* USER CODE BEGIN WHILE */
  while (1)
  {
    loop();
    /* USER CODE END WHILE */

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

...
```

¿Os suena? Hemos copiado la **filosofía de Arduino** y hemos utilizado dos funciones que el miembro `B` creará en el archivo `stm32main`. De este modo, **si queremos modificar/añadir cosas al `main`** podemos hacerlo añadiendo código **dentro de las funciones `setup` y `loop`** definidas en otro archivo distinto a `main.c`. ¡Ya no tendremos que modificar más el archivo `main.c`!

Guarda el proyecto y haz un *commit* y *push*.

```
git add -A
git commit -m "Incorporadas las funciones setup y loop a la función
    main."
git push
```

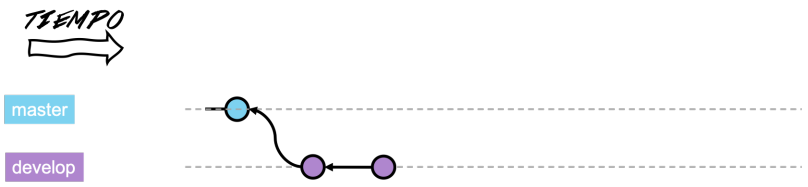

Figura 5: Git.

A partir de aquí, podéis trabajar cada uno por su lado.

1.3.3.3. Archivos usart1 Vamos a crear los archivos `usart1.c` y `usart1.h` que se encargarán de la gestión de la USART1, el periférico que se encargará de la comunicación entre microcontroladores.

Lo primero que haremos será, **desde la rama `develop`**, crear una **rama llamada `feature/usart1`**.

```
1 git checkout -b feature/usart1
```

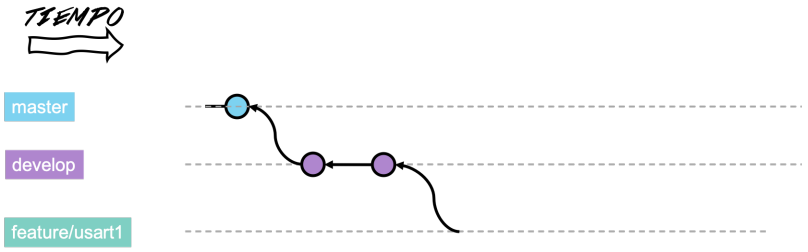

Figura 6: Git.

Vamos a **crear dos carpetas** llamadas, ambas, `components`. Una la creamos en la carpeta `Core` > `Src` y otra en `Core` > `Inc`. Para crear una carpeta, hacemos clic derecho sobre `Src` y `Inc` y seleccionamos `New` > `Folder`. Indicamos el nombre y clicamos `Finish`.

Ahora vamos a **crear el archivo `usart1.c`**. Lo hacemos haciendo clic derecho sobre la carpeta `Core` > `Src` > `components` y seleccionando `New` > `Source File`. Lo nombramos `usart1.c` y clicamos `Finish`. A este archivo le añadimos el siguiente código cuya explicación está directamente en los comentarios del mismo:

```
1 // Hacemos un include de components/usartx.
2 // Hacemos este include para tener disponibles las funciones definidas
     en
3 // los otros archivos.
4 #include "components/usart1.h"
5 #include "components/usart2.h"
6
7 // Creamos dos buffers de un elemento para
```

```
8  uint8_t rxU1Buffer[USART1_BUFF_SIZE] = { 0 }, // recibir
9          txU1Buffer[USART1_BUFF_SIZE] = { 0 }; // y enviar.
10
11 // La macro USART1_BUFF_SIZE la definiremos en usart1.h.
12
13
14 // Declaramos la variable huart1 sin definirla. Esto lo hacemos con la
       keyword extern.
15 // De este modo tenemos disponible la variable huart1 que esta definida
        en el archivo main.c.
16 extern UART_HandleTypeDef huart1;
17
18 // Funcion para inciar la espera de la recepcion de bytes mediante
       interrupciones.
19 void UART1_start(void) {
20
21     HAL_UART_Receive_IT(&huart1, &rxU1Buffer, 1);
22
23 }
24
25 // Funcion que utilizaremos para enviar un byte que pasaremos como
       parametro.
26 void UART1_transmit(uint8_t value) {
27
28     txU1Buffer[0] = value;
29
30     HAL_UART_Transmit_IT(&huart1, &txU1Buffer, 1);
31
32 }
33
34 // Funcion que sera llamada desde el callback al recibir un byte.
35 // El byte recibido de enviar por la otra USART
36 void UART1_receive(void) {
37
38     // Utilizamos esta funcion de la otra USART para enviar el byte.
39     // Por esto necesitabamos hacer el include del header del otro
           USART.
40     UART2_transmit(rxU1Buffer[0]);
41
42     // Volvemos a quedar a la espera de otro byte.
43     HAL_UART_Receive_IT(&huart1, &rxU1Buffer, 1);
44
45 }
```

Ahora vamos a **crear el fichero `usart1.h`** en la carpeta `Core` > `Inc` > `components`. Para ello, hacemos clic derecho sobre dicha carpeta y seleccionamos `New` > `Header File`. En la ventana emergente, introducimos el nombre `usart1.h` y clicamos `Finish`.

En este archivo indicaremos los prototipos de funciones, variables y macros que queremos que puedan estar disponibles para otros archivos de código fuente que hagan un *include* de este archivo. El

contenido del archivo sería el siguiente:

```
#ifndef INC_COMPONENTS_USART1_H_
#define INC_COMPONENTS_USART1_H_

// Incluimos este header de las HAL para poder usar algunas
// funcionalidades de las HAL.
#include "stm32f4xx_hal.h"

// Creamos una macro para el tamano de los buffer.
#define USART1_BUFF_SIZE 1

// Prototipos de las funciones para que esten disponibles en esos
   archivos de
// codigo fuentes donde se hage un include de este archivo.
void UART1_start(void); // Se utiliza en 'stm32main.c'
void UART1_transmit(uint8_t value); // Se utiliza en'usart2.c'
void UART1_receive(void); // Se utiliza en 'stm32main.'


#endif /* INC_COMPONENTS_USART1_H_ */
```

Con esto, habríamos finalizado todas nuestras tareas. Guardamos el proyecto y hacemos un *commit* y un *push*.

```
git add -A
git commit -m "UART1 implementado."
git push
```

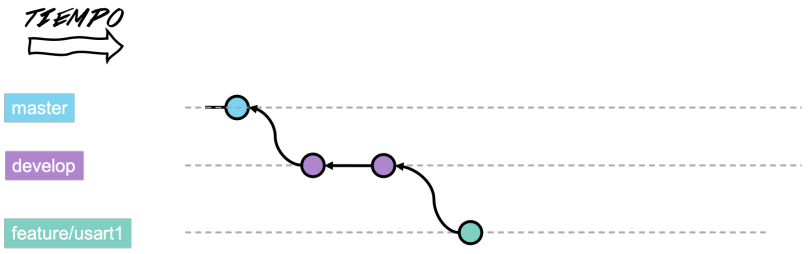

Figura 7: Git.

Ahora solo nos queda ir a la página web del repositorio remoto en GitHub y crear un ***Pull Request* de nuestra rama `feature` hacia la rama `develop`** poniendo como ***reviewer* a nuestro compañero**.

1.3.3.4. *Review* y *merge* No olvides, cuando tu compañero haga un *Pull Request* de sus ramas `feature` a la rama `develop`, de revisar sus desarrollos y aprobarlo o hacerle una petición de cambios antes de unir los cambios a la rama `develop`.

1.3.4. *Workflow* del miembro B

Antes de nada, **asegurate de hacer con tu compañero la creación del proyecto, su configuración y la pequeña modificación del archivo `main.c`**. Hecho esto, podemos empezar.

1.3.4.1. Archivos stm32main Vamos a crear el archivo que actuará de `main` a partir de ahora y que nos permitirá no tener que editar directamente el archivo `main.c` generado por STM32CubeMX.

Primeramente, **vamos a la rama `develop`** que ha creado nuestro compañero e incorporamos todos los posibles cambios que hayan podido haber en el repositorio remoto. Seguidamente, creamos nuestra rama de desarrollo `feature/stm32main`.

```
1 git checkout develop
2 git pull
3 git checkout -b feature/stm32main
```

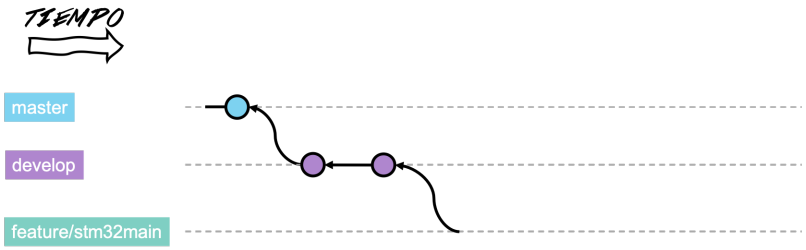

Figura 8: Git.

Vamos a **crear dos carpetas** llamadas, ambas, `components`. Una la creamos en la carpeta `Core > Src` y otra en `Core > Inc`. Para crear una carpeta, hacemos clic derecho sobre `Src` y `Inc` y seleccionamos `New > Folder`. Indicamos el nombre y clicamos `Finish`.

Ahora vamos a **crear el archivo `stm32main.c`**. Lo hacemos haciendo clic derecho sobre la carpeta `Core > Src > components` y seleccionando `New > Source File`. Lo nombramos `stm32main.c` y clicamos `Finish`. A este archivo le añadimos el siguiente código cuya explicación está directamente en los comentarios del mismo:

```
1 // Archivo header de stm32main
2 #include "components/stm32main.h"
3
4 // Funcion que solo se ejecuta una unica vez
5 void setup(void) {
6
7     UART1_start(); // Inicializamos USART1. Lo implementa nuestro
          companero.
8     UART2_start(); // Inicializamos USART2. Lo implementamos nosotros.
```

```
 9
10 }
11
12 // Funcion que se repite dentro del bucle while
13 void loop(void) {
14     // No hacemos nada. Solo lo ponemos de manera ilustrativa.
15 }
16
17 // Callback de la interrupcion de la recepcion de un byte
18 void HAL_UART_RxCpltCallback (UART_HandleTypeDef *huart) {
19
20     // Miramos quien ha recibido el byte, si la USART 1 o 2.
21     if (huart->Instance == USART1) {
22         UART1_receive(); // Lo implementa nuestro companero.
23     } else if (huart->Instance == USART2) {
24         UART2_receive(); // Lo implementamos nosotros.
25     }
26
27 }
```

Ahora vamos a **crear el fichero** `stm32main.h` en la carpeta `Core > Inc > components`. Para ello, hacemos clic derecho sobre dicha carpeta y seleccionamos `New > Header File`. En la ventana emergente, introducimos el nombre `stm32main.h` y clicamos `Finish`.

En este archivo indicaremos los prototipos de funciones, variables y macros que queremos que puedan estar disponibles para otros archivos de código fuente que hagan un *include* de este archivo. El contenido del archivo sería el siguiente:

```
1  #ifndef INC_COMPONENTS_STM32MAIN_H_
2  #define INC_COMPONENTS_STM32MAIN_H_
3
4  // Includes para poder usar funciones de
5  #include "stm32f4xx_hal.h"  // las HAL,
6  #include "components/usart1.h"  // la USART1 y
7  #include "components/usart2.h"  // la USART2.
8
9  // Prototipos que deben de estar disponibles en aquello archivos de
       codigo fuente
10 // que hagan un include de este archivo.
11 void setup(void);  // La utilizamos en el main.c.
12 void loop(void);  // La utilizamos en el main.c.
13
14 #endif /* INC_COMPONENTS_STM32MAIN_H_ */
```

Con esto, habríamos **finalizado esta funcionalidad**. Guardamos el proyecto y hacemos un *commit* y un *push*.

```
1 git add -A
2 git commit -m "STM32main implementado."
```

```
3 git push
```

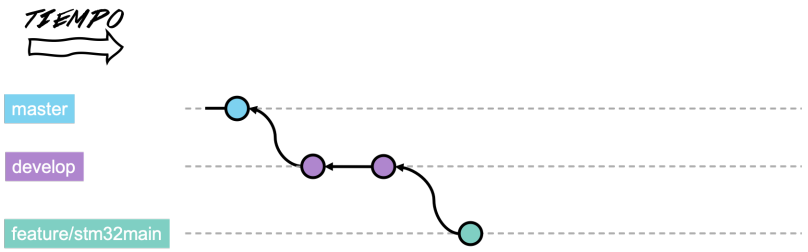

Figura 9: Git.

Ahora solo nos queda ir a la página web del repositorio remoto en GitHub y crear un *Pull Request* de nuestra rama `feature` hacia la rama `develop` poniendo como *reviewer* a nuestro compañero. Vamos a realizar la otra funcionalidad que nos queda: USART2.

1.3.4.2. Archivos usart2 Vamos a crear los archivos `usart2.c` y `usart2.h` que se encargarán de la gestión de la USART2, el periférico que se encargará de la comunicación entre el microcontrolador y el ordenador.

Lo primero que haremos será, **desde la rama `develop`**, crear una rama llamada `feature/usart2`.

```
1 git checkout develop
2 git pull
3 git checkout -b feature/usart2
```

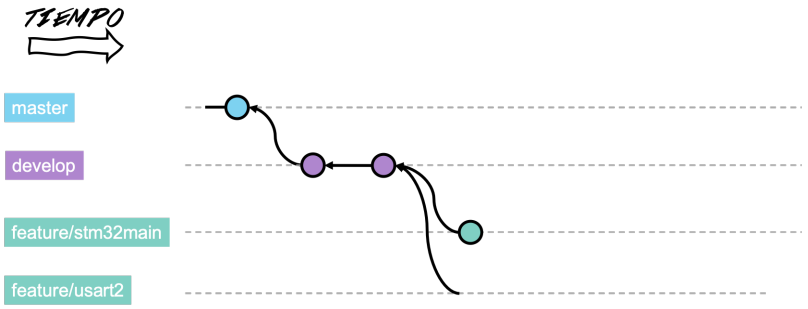

Figura 10: Git.

Vamos a **crear dos carpetas** llamadas, ambas, `components`. Una la creamos en la carpeta `Core` > `Src` y otra en `Core` > `Inc`. Para crear una carpeta, hacemos clic derecho sobre `Src` y `Inc` y seleccionamos `New` > `Folder`. Indicamos el nombre y clicamos `Finish`.

Ahora vamos a **crear el archivo `usart2.c`**. Lo hacemos haciendo clic derecho sobre la carpeta `Core > Src > components` y seleccionando `New > Source File`. Lo nombramos `usart2.c` y clicamos `Finish`. A este archivo le añadimos el siguiente código cuya explicación está directamente en los comentarios del mismo:

```
// Hacemos un include de components/usartx.
// Hacemos este include para tener disponibles las funciones definidas en
// los otros archivos.
#include "components/usart2.h"
#include "components/usart1.h"

// Creamos dos buffers de un elemento para
uint8_t rxU2Buffer[USART2_BUFF_SIZE] = { 0 }, // recibir
        txU2Buffer[USART2_BUFF_SIZE] = { 0 }; // y enviar.

// La macro USART2_BUFF_SIZE la definiremos en usart2.h.


// Declaramos la variable huart2 sin definirla. Esto lo hacemos con la keyword extern.
// De este modo tenemos disponible la variable huart2 que esta definida en el archivo main.c.
extern UART_HandleTypeDef huart2;

// Funcion para inciar la espera de la recepcion de bytes mediante interrupciones.
void UART2_start(void) {

    HAL_UART_Receive_IT(&huart2, &rxU2Buffer, 1);

}

// Funcion que utilizaremos para enviar un byte que pasaremos como parametro.
void UART2_transmit(uint8_t value) {

    txU2Buffer[0] = value;

    HAL_UART_Transmit_IT(&huart2, &txU2Buffer, 1);

}

// Funcion que sera llamada desde el callback al recibir un byte.
// El byte recibido de enviar por la otra USART
void UART2_receive(void) {

    // Utilizamos esta funcion de la otra USART para enviar el byte.
    // Por esto necesitabamos hacer el include del header del otro USART.
```

```
40      UART1_transmit(rxU2Buffer[0]);
41
42      // Volvemos a quedar a la espera de otro byte.
43      HAL_UART_Receive_IT(&huart2, &rxU2Buffer, 1);
44
45  }
```

Ahora vamos a **crear el fichero `usart2.h`** en la carpeta `Core` > `Inc` > `components`. Para ello, hacemos clic derecho sobre dicha carpeta y seleccionamos `New` > `Header File`. En la ventana emergente, introducimos el nombre `usart2.h` y clicamos `Finish`.

En este archivo indicaremos los prototipos de funciones, variables y macros que queremos que puedan estar disponibles para otros archivos de código fuente que hagan un *include* de este archivo. El contenido del archivo sería el siguiente:

```
1  #ifndef INC_COMPONENTS_USART2_H_
2  #define INC_COMPONENTS_USART2_H_
3
4  // Incluimos este header de las HAL para poder usar algunas
5  // funcionalidades de las HAL.
6  #include "stm32f4xx_hal.h"
7
8  // Creamos una macro para el tamano de los buffer.
9  #define USART2_BUFF_SIZE 1
10
11 // Prototipos de las funciones para que esten disponibles en esos
       archivos de
12 // codigo fuentes donde se hage un include de este archivo.
13 void UART2_start(void); // Se utiliza en 'stm32main.c'
14 void UART2_transmit(uint8_t value); // Se utiliza en'usart1.c'
15 void UART2_receive(void); // Se utiliza en 'stm32main.'
16
17 #endif /* INC_COMPONENTS_USART2_H_ */
```

Con esto, habríamos finalizado todas nuestras tareas. Guardamos el proyecto y hacemos un *commit* y un *push*.

```
1  git add -A
2  git commit -m "UART2 implementado."
3  git push
```

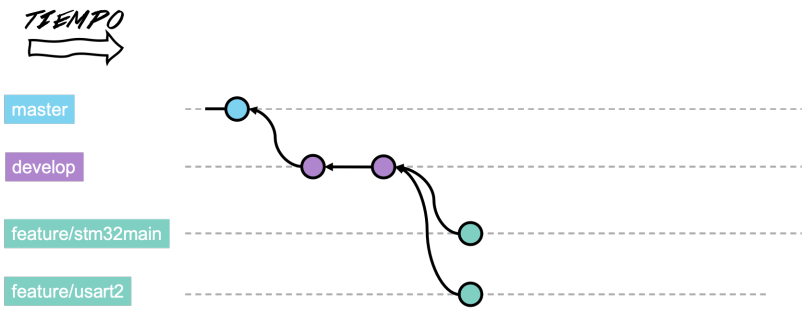

Figura 11: Git.

Ahora solo nos queda ir a la página web del repositorio remoto en GitHub y crear un *Pull Request* de nuestra rama `feature` hacia la rama `develop` poniendo como *reviewer* a nuestro compañero.

1.3.4.3. *Review* y *merge* No olvides, cuando tu compañero haga un *Pull Request* de sus ramas `feature` a la rama `develop`, de revisar sus desarrollos y aprobarlo o hacerle una petición de cambios antes de unir los cambios ala rama `develop`.

1.3.5. *Test y Pull Request* a la master

Una vez tengáis todo *merged* en la rama `develop`, cargad el mismo proyecto/código en las dos EVB.

> Id a la rama `develop` e importad todos los cambios del repositorio remoto.

```
1 git checkout develop
2 git pull
```

Seguidamente, **conectad las dos USART1 de ambas EVB entre sí**. Los pines a utilizar son el **`PA9` (TX) y `PA10` (RX)**. Acordaros de **cruzar los pines** tal y cómo se indicaba en la introducción de la primera parte de esta práctica. También acordaos de **unir los GNDs** de las dos EVB. Como siempre, es recomendable **desenchufar las EVB de cualquier alimentación antes de hacer cualquier conexión**.

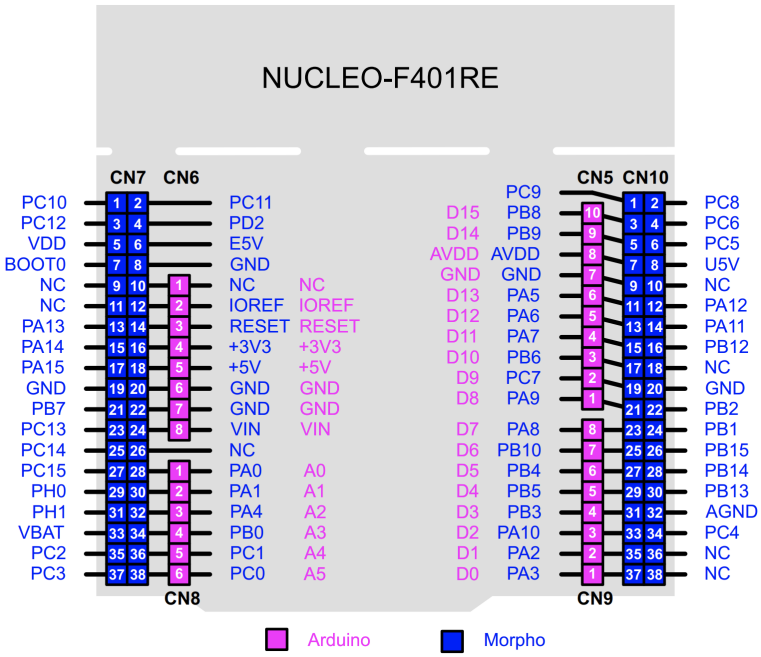

Figura 12: Pinout.

Seguidamente, abrimos cada uno la aplicación **CoolTerm** en su ordenador. Lo configuramos seleccionando el **puerto pertinente** y un ***baud rate* de 115200 bps**. También tocaremos una pequeña configuración para que la demostración funcione correctamente. Básicamente, es **añadir un *delay* entre transmisiones** y **añadir un retorno de carro y fin de línea al enviar un texto**. Esto lo hacemos desde el submenú `Transmit`.

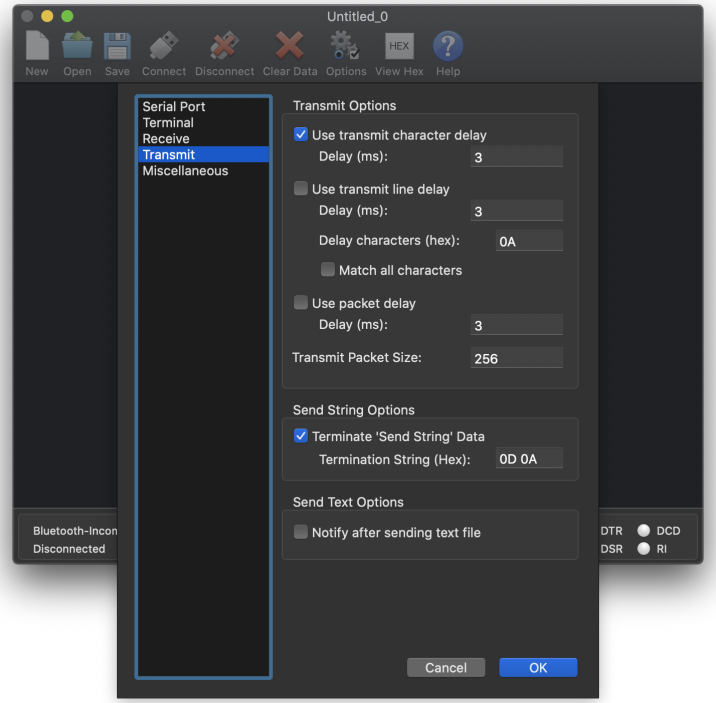

Figura 13: Configuración de Coolterm.

Probamos a enviar texto en ASCII y lo que enviamos desde un ordenador debería aparecer en el otro y viceversa. **Si no funciona**, deberéis de **buscar el error en el proyecto y solucionarlo**. **No lo arregléis directamente en la rama `develop`**. Creáis una **nueva rama** (normalmente, se les llama `hotfix/***` a las ramas utilizadas para reparar un *bug*) y una vez hechos los cambios necesarios los **incorporáis a la rama `develop` con un *Pull Request* y volvéis a testear**.

Si todo funciona correctamente, proceded a hacer un ***Pull Request* de la rama `develop` a la `master`**. Puesto que ya habéis probado el proyecto, podéis proceder a hacer el ***merge* directamente sin necesidad de hacer un *review***.

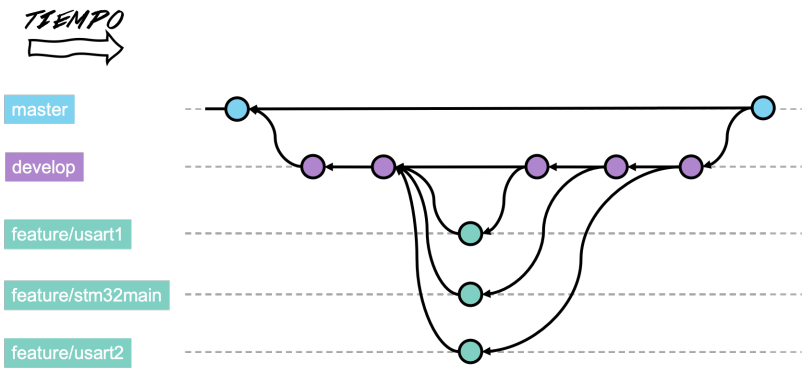

Figura 14: Git.

¡Ya habéis hecho vuestro primer proyecto basado en STM32CubeIDE en equipo!

1.4. Evaluación

1.4.1. Entregables

Estos son los elementos que deberán de estar disponibles para el profesorado de cara a vuestra evaluación.

- ☐ **Commits**

Se os deja a vuestro criterio cuándo realizar los *commits*. No hay número mínimo/máximo y se evaluará el uso adecuado del control de versiones (creación de ramas, *commits* en el momento adecuado, mensajes descriptivos de los cambios, …).

- ☐ **Reto**
- ☐ ~~**Informe**~~ No hay informe.

1.4.2. Pull Request

Finalizados todos los entregables, acordaos de hacer el *push* pertinente y cread un *Pull Request* (PR) de vuestra rama a la master. Acordaos de ponerme como *Reviewer* (menos en los del reto).

1.4.3. Rúbrica

La rúbrica que utilizaremos para la evaluación la podéis encontrar en el CampusVirtual. Os recomendamos que le echéis un vistazo para que sepáis exactamente qué se evaluará y qué se os pide.

1.5. Conclusiones

Bueno. Práctica muy completa donde hemos visto **cómo enviar datos codificados en ASCII con STM32CubeIDE** y utilizando la **función `sprintf`** para formatear el texto. También hemos visto cómo **recibir datos en *raw* mediante interrupciones**.

Finalmente, también hemos visto **cómo trabajar en equipo** en un proyecto basado en STM32CubeIDE y hemos hecho una **aplicación que comunica dos ordenadores utilizando los microcontroladores como puente**.

En la **siguiente práctica** abordaremos el ***framing* de bytes** en comunicaciones serie.