

UNIVERSIDAD TECNOLÓGICA NACIONAL



FACULTAD REGIONAL PARANÁ

MEDIDAS ELECTRÓNICAS II

TEMA: Analizador lógico y Analizador de protocolo SPI.
Entrega final

Grupo 4 :

*Alarcon, Ramiro (ramiroalarcon@alu.frp.utn.edu.ar)

*Muller, Mario (mario.daniel.muller@outlook.com)

*Sanchez, Joaquín (joaquinsanchez@alu.frp.utn.edu.ar)

*Vinzón, Alan (alanv3514@gmail.com)

Fecha de Entrega: 10/11/2021

Resumen

En el presente informe se describe lo referente a la entrega final del trabajo de investigación sobre analizadores lógicos y analizadores de protocolo.

Se comienza analizando sus esquemas, diagramas y funcionalidades. Dado que como grupo se decidió que tanto el analizador lógico como el analizador de protocolos utilicen un microcontrolador STM32 (más concretamente el modelo f103c8t6) también se realiza una introducción al mismo para comprender algunas particularidades al momento de utilizarlo.

En la implementación del analizador lógico, se llegó a resultados satisfactorios, logrando tener 8 puntas de prueba con un muestreo máximo de 1 MHz, siendo esta frecuencia de muestreo seteable por software. La forma de observar las señales tomadas por las puntas de prueba se realiza completamente mediante el software Matlab.

Con respecto al diseño e implementación en protoboard del equipo analizador de protocolo SPI, se comienza realizando un repaso de los conceptos teóricos del funcionamiento del protocolo SPI para introducir al lector en los detalles técnicos de cómo se realiza un intercambio de información entre un maestro y un esclavo. Luego se procede al diseño e implementación de los bloques fundamentales del equipo, siendo estos el bloque muestreador y el bloque de control. Finalmente se realizan pruebas prácticas del equipo sobre un banco de pruebas de comunicación SPI constituido por dos microcontroladores Arduino UNO, en donde uno ofrece de maestro y otro de esclavo. Los resultados obtenidos de las pruebas se presentan hacia el final del informe, determinando el correcto funcionamiento del analizador de protocolo SPI diseñado.

Índice de contenido

| | |
|--|----|
| Resumen..... | 2 |
| Índice de contenido | 3 |
| Índice de figuras..... | 5 |
| Introducción | 7 |
| Analizador Lógico..... | 9 |
| 1 – Criterios de diseño..... | 9 |
| 1.1 – Diagrama en bloques de la implementación..... | 10 |
| 2 - Diseño | 11 |
| 2.1 - Razones por la que se elige un STM32F103..... | 11 |
| 3 – Configuración del microcontrolador | 14 |
| 3.1 – Configuración grafica..... | 14 |
| 3.2 – Configuración del cristal | 15 |
| 3.3 – Interrupción por timer | 16 |
| 3.4 – UART | 17 |
| 3.5 – Selección de pines | 17 |
| 3.6 – Habilitación de interrupciones | 20 |
| 4 – Programación del microcontrolador..... | 20 |
| 4.1 – Variables globales | 22 |
| 4.2 – Iniciación de la interrupción TIM2 | 22 |
| 4.3 – Programa para envío y recepción UART | 23 |
| 4.4 – Configuración del timer | 23 |
| 4.5 – Programación de interrupciones externas y cíclicas..... | 24 |
| 5 – Programación en Matlab | 25 |
| 6 – Resultados prácticos..... | 27 |
| 7 – Presupuesto | 30 |
| 8 – Conclusiones | 30 |
| Analizador de Protocolo SPI..... | 31 |
| 1 - El protocolo SPI..... | 31 |
| 2 – Diseño e implementación del Analizador de protocolo SPI..... | 33 |
| 2.1 – Bloque muestreador..... | 34 |
| 2.2 – Bloque de control | 35 |
| 2.2.1 – Conexión del microcontrolador | 36 |

| | |
|---|----|
| 2.2.2 – Programación del microcontrolador..... | 37 |
| 2.2.3 – Conexión del microcontrolador al ordenador..... | 39 |
| 3 – Banco de pruebas SPI..... | 40 |
| 4 – Resultados prácticos..... | 42 |
| 5 – Presupuesto | 46 |
| 6 – Conclusión..... | 46 |
| 7 – Anexo | 47 |
| 7.1 – Código STM32..... | 47 |
| 7.2 – Código Maestro Arduino..... | 54 |
| 7.3 – Código Esclavo Arduino | 55 |
| Bibliografía | 57 |

Índice de figuras

| | |
|--|----|
| Figura 1: Diferencia entre analizador lógico y analizador de protocolos..... | 7 |
| Figura 2: Placa “bluepill” | 9 |
| Figura 3: Esquemático utilizando solo el microcontrolador | 10 |
| Figura 5: Disposición de pines STM32F103C8T..... | 12 |
| Figura 6: Diagrama conexión UART | 13 |
| Figura 7: Formato de trama en UART | 13 |
| Figura 8: Diagrama de conexión entre el uC y la PC..... | 13 |
| Figura 9: Configuración gráfica STM32 | 14 |
| Figura 10: Configuración cristal 1 | 15 |
| Figura 11: Configuración cristal 2 | 15 |
| Figura 12: Configuración interrupción por timer | 16 |
| Figura 13: Configuración UART | 17 |
| Figura 14: Vista grafica STM32f103c8t6 | 18 |
| Figura 15: Ventana pines GPIO. | 18 |
| Figura 16: Ventana pin para interrupción 1..... | 19 |
| Figura 17: Ventana pin para interrupción 2..... | 19 |
| Figura 18: Ventana pin para interrupción 3..... | 19 |
| Figura 19: Ventana habilitación de interrupciones..... | 20 |
| Figura 20: Variables globales..... | 22 |
| Figura 21: Interrupción TIM2 | 22 |
| Figura 22: Programa envío - recepción UART | 23 |
| Figura 23: Configuraciones del timer..... | 24 |
| Figura 24: Rutinas interrupción timer e interrupción externa | 25 |
| Figura 25: Código Matlab parte 1 | 25 |
| Figura 26: Código Matlab parte 2 | 26 |
| Figura 27: Conexionado realizado protoboard..... | 27 |
| Figura 28: Esquema de conexión divisores resistivos..... | 28 |
| Figura 29: Esquema conexionado del botón | 28 |
| Figura 30: Conversor USB – UART CP2102 | 28 |
| Figura 31: Conexionado CP2102 | 29 |
| Figura 32: Resultados de mediciones sobre placa Arduino. | 29 |

| | |
|---|----|
| Figura 33: Conexión típica maestro y esclavo SPI..... | 31 |
| Figura 34: Modo 1 transmisión SPI | 32 |
| Figura 35: Diagrama del sistema completo..... | 33 |
| Figura 36: Registro de desplazamiento Proteus | 34 |
| Figura 37: Registro de desplazamiento protoboard..... | 34 |
| Figura 38: Microcontrolador STM32f103c8t6..... | 35 |
| Figura 39: Pines del STM32 utilizados | 37 |
| Figura 40 : Modulo USB – UART CP2102 | 39 |
| Figura 41: ST-LINK V2..... | 40 |
| Figura 42: Banco de pruebas SPI | 41 |
| Figura 43: Mensaje enviado 1 | 43 |
| Figura 44: Puerto Serie STM32 | 43 |
| Figura 45: Mensaje capturado 1 | 44 |
| Figura 46: Mensaje enviado 2 | 44 |
| Figura 47: Mensaje capturado 2..... | 45 |
| Figura 48: Mensaje enviado 3 | 45 |
| Figura 49: Mensaje capturado 3..... | 46 |

Introducción

Un analizador lógico al igual que un analizador de protocolos posee el mismo principio de funcionamiento, el cual se basa en tomar señales que circulan por una red de comunicación. La diferencia radica en que para el caso del analizador lógico, estas señales tomadas se interpretan como variaciones de niveles lógicos (unos o ceros) y se presentan estas conmutaciones al usuario gráficamente a través de una pantalla/display, o se almacenan estas señales eléctricas en una memoria para posterior análisis. En un analizador de protocolos entra en juego la interpretación de estas variaciones y su significado según el tipo de protocolo con el que se maneja la información que viaja por la red de comunicación analizada. En base a esto dependerá la forma en que debe ser procesada la señal, de modo de poder presentarla en pantalla al usuario en un modo legible y fácil de analizar.

Lo dicho anteriormente se ve de mejor manera en la figura 1:

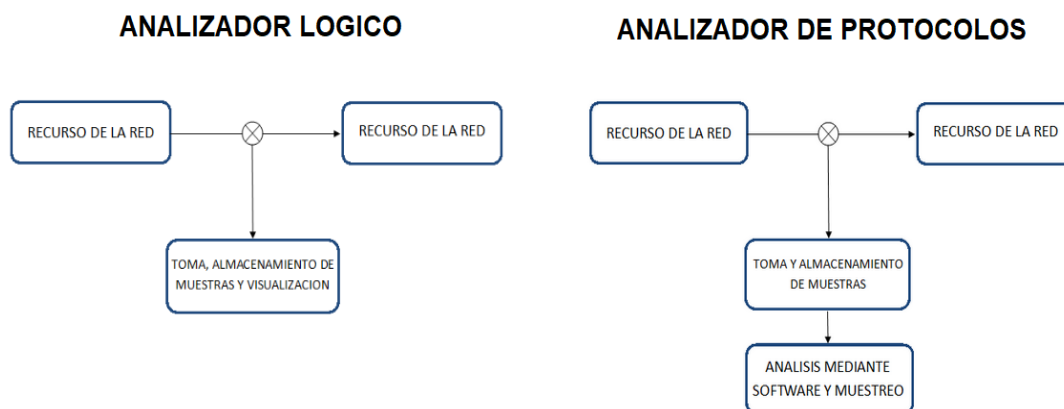


Figura 1: Diferencia entre analizador lógico y analizador de protocolos

Con respecto al analizador de protocolos, es necesario definir un tipo de protocolo específico, a modo de diseñar el equipo con las características necesarias para capturar este protocolo he interpretar esta información de la manera correcta. Con esto en mente, se decidió elegir el protocolo SPI.

El protocolo SPI, acrónimo para Serial Peripheral Interface, es un estándar entre los protocolos de comunicación para circuitos integrados entre ellos. Fue desarrollado en la década del 80 por la empresa Motorola para permitir la comunicación entre diferentes sistemas electrónicos digitales como lo eran sus microcontroladores de la época. SPI se trata de una comunicación tipo Master - Slave sincrónica, utilizando un bus de reloj para sincronizar la transmisión de datos entre los dispositivos. Estas transmisiones se realizan

mediante dos buses, siendo uno para los datos que el maestro envía al esclavo o recibe del mismo, y otro bus para los datos que el esclavo envía al maestro o recibe del mismo.

Con el paso del tiempo nuevos protocolos surgieron, pero el SPI se mantuvo activo desde sus inicios, siendo hoy una de las formas de comunicación básicas que se encuentra en la mayoría de los microcontroladores y placas de desarrollo comerciales ampliamente usadas, como por ejemplo las placas Arduino o las ST. Teniendo en cuenta esto, la posibilidad de contar con un dispositivo que permita analizar las comunicaciones entre un esclavo y maestro SPI resalta su importancia y nivel de aplicación real, convirtiéndose en una herramienta importante en ambientes en donde procesos importantes se lleven a cabo mediante SPI, como puede ser alguno de nivel industrial o menor escala, con el fin de detectar posibles errores en una comunicación entre dispositivos sin necesidad de desconectar los equipos de la red de trabajo, ya que un analizador de protocolo SPI funciona en paralelo con las líneas SPI, facilitando su utilización.

Analizador Lógico

1 - Criterios de diseño

Un analizador lógico es un instrumento para capturar señales de voltaje de un circuito digital y realizar un análisis posterior de las mismas, permitiendo visualizar las señales provenientes de varios canales. Los analizadores lógicos no muestran las señales en tiempo real, sino que presentan en pantalla los valores lógicos de señales que previamente han sido muestreadas y guardadas en la memoria, estos valores son almacenados a partir de un instante programado por el usuario.

Las características de un analizador lógico se simplifican en tres parámetros: velocidad, anchura y profundidad. En palabras simples, debe ser capaz de capturar eventos rápidos, permitir la entrada de los suficientes canales y disponer de una buena cantidad de memoria para guardar los datos en la etapa de adquisición.

Al momento de realizar nuestro proyecto primero definimos cuáles van a ser nuestros criterios de diseño. Por una cuestión de simplicidad y costos, decidimos implementar un analizador lógico con 8 puntas de pruebas o entradas, el instante a partir del cual se comienza a tomar datos estará dado por un pulsador que deberá ser accionado por el usuario. Como hardware principal utilizaremos la placa de desarrollo conocida popularmente como “bluepill”, la cual funciona con el microcontrolador STM32F103C8T6 el cual trabaja internamente con una estructura de 32 bits, con arquitectura ARM y un núcleo tipo cortex M3 a una frecuencia máxima de trabajo de 72 MHz, incluye 20 KB de memoria RAM y 64 KB de memoria Flash. Aunque en la implementación que realizamos utilizamos la placa de desarrollo nombrada anteriormente si se quisiese hacer un desarrollo más profesional o industrial bastaría con diseñar la parte del PCB y como núcleo usar solamente el microcontrolador STM32.

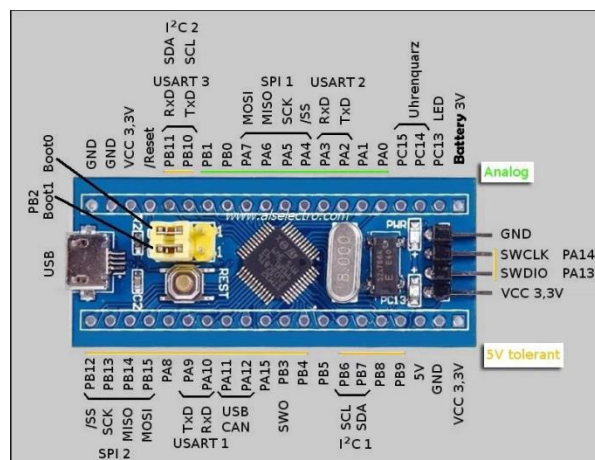


Figura 2: Placa “bluepill”

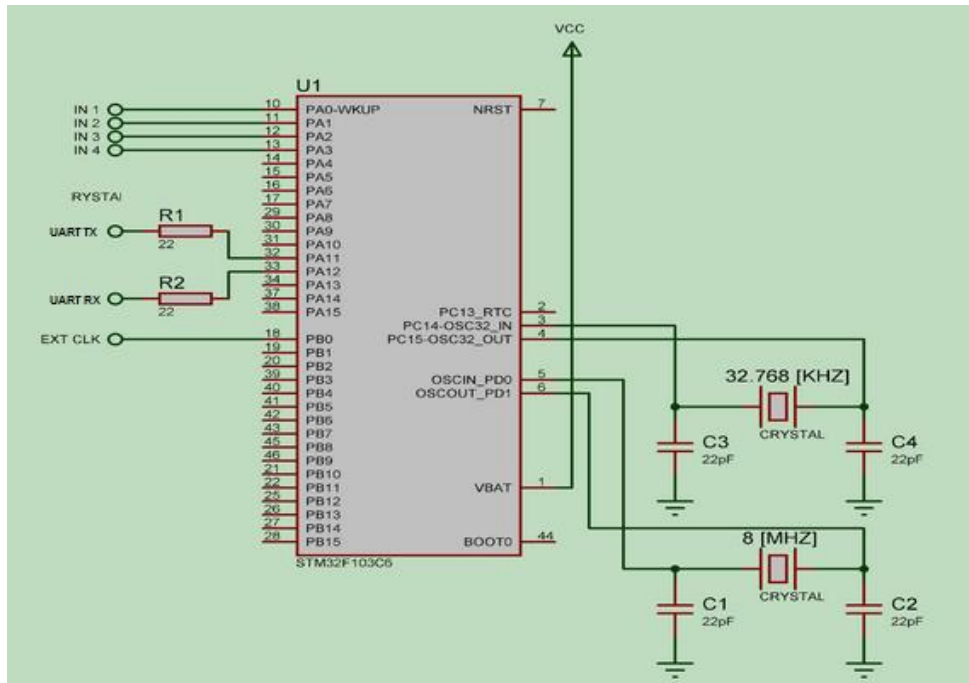


Figura 3: Esquemático utilizando solo el microcontrolador

Al momento de recabar información del analizador lógico encontramos que ya existen aplicaciones desarrolladas, estas utilizan un diagrama que puede variar en mayor o menor medida, pero siempre poseen un esquema similar al siguiente.

1.1 – Diagrama en bloques de la implementación

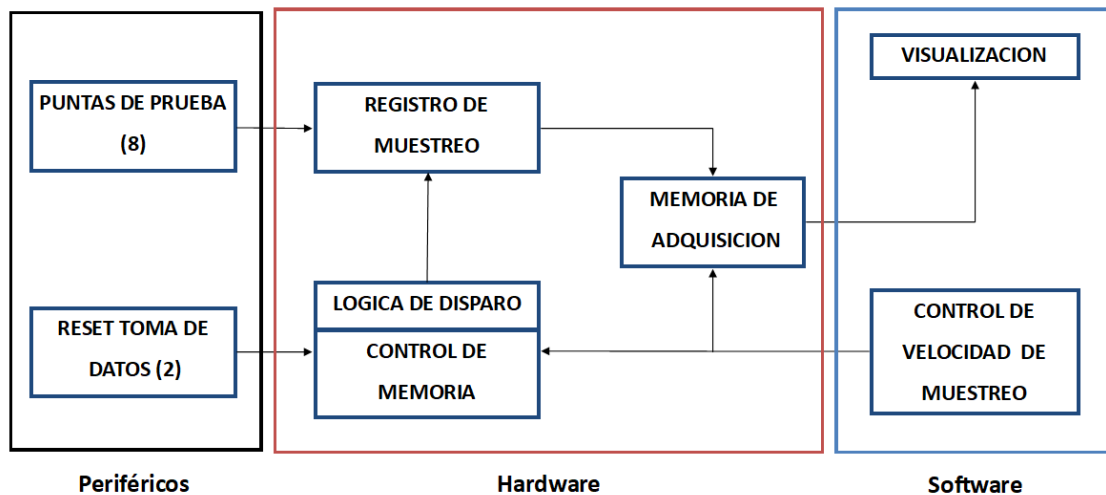


Figura 4: Diagrama en bloques Analizador Lógico implementado

Como periféricos tenemos las 8 puntas de prueba, a través de las cuales se van a tomar las señales digitales a analizar. También posee dos pulsadores a través de los cuales indicaremos al sistema que deberá comenzar a realizar la toma de muestras.

Dado que la mayoría de los subbloques se pueden agrupar en bloque del hardware, podríamos decir que el grueso del diseño estará en realizar una buena programación del microcontrolador, es por eso que haremos foco principalmente en el firmware implementado en el STM32.

Para realizar la visualización de las señales muestreadas se utilizó una comunicación USB entre el STM32 y la PC, en dicha PC se deberá estar corriendo el software Matlab con el script correctamente programado.

2 - Diseño

2.1 - Razones por la que se elige un STM32F103

Basado en que este es un trabajo práctico, y los recursos que se manejan son limitados, debemos de buscar un microcontrolador que sea alcanzable para nuestro presupuesto. Es ahí donde los procesadores cortex M de gama media se encuentran a un precio similar a los ya conocidos atmega328 utilizados en placas de Arduino, pero con características mucho más convenientes en capacidad de memoria tanto flash como ram y velocidad del microprocesador. El más económico entre los que ofrecía la empresa es el STM32F103.

Ventajas de los microcontroladores stm32:

- Todos tienen un procesador Cortex M como núcleo de procesamiento basado en la arquitectura ARM.
- STM32 tiene herramientas IDE excelentes simples de utilizar creadas por la empresa STM y varios desarrolladores externos, por lo que la variedad de herramientas es basta.
- La distribución de pines en los microcontroladores STM32 se respeta en compatibilidad siempre se encuentre en la misma familia de microcontroladores. Por lo que si cambio el microcontrolador por uno que tenga más flash o un procesador más potente, mientras se encuentren dentro de la misma subfamilia prácticamente no es necesario modificar el circuito.
- El precio, existen muchos microcontroladores baratos dentro de las familias de STM32, dentro de los más baratos es el que utilizamos nosotros STM32F103.
- Muchos microcontroladores vienen con la placa de desarrollo incluida lo que ahorra mucho tiempo de diseño para probar prototipos, por supuesto es más barato comprar los microcontroladores sin la placa. Lo cual para el periodo de producción de una empresa es muy importante.
- Bootloader: Nos permite conectar el micro a través de un cable USB para programar el micro sin la necesidad de un programador. Los Stm32 vienen con un Bootloader preinstalado de fábrica lo que nos asegura que va a funcionar bien.
- Un punto a considerar como ventaja solo aplicable a un grupo muy reducido de micros entre ellos el STM32F103C8T6 que estamos usando, es que la plataforma de arduino puede fácilmente a través de librerías adaptar el IDE para trabajar con la placa blue pill que contienen este micro. (Sin embargo el uso de esa ventaja

está a considerar debido a que se pierde control de algunas características finas del micro.)

- Permite trabajar con una frecuencia de lectura de puertos de hasta 40[KHz], lo que nos da una buena frecuencia de muestreo para el diseño del analizador lógico.

Desventajas:

- Documentación: Aprender a programar desde 0 los STM32 es complejo si no se sabe programar otros micros más simples.
- Hasta el momento no hay microcontroladores cortex m3 de STM32 con comunicación wifi. En este caso particular no nos afecta, pero es algo para mencionar.

Como mencionamos anteriormente, el foco del diseño estará en la programación del STM32 y en la configuración de sus pines, en nuestro diseño los pines tienen la siguiente disposición.

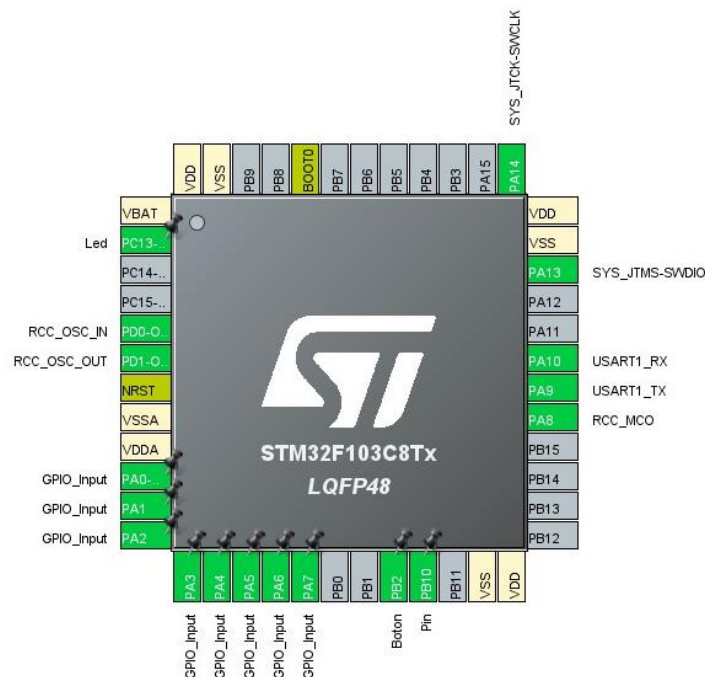


Figura 5: Disposición de pines STM32F103C8T

Dado que el STM32 no dispone directamente de un puerto USB sino que tiene comunicación UART, daremos un pequeño pantallazo de este protocolo de comunicación.

Las siglas de UART (universal asynchronous receiver / transmitter, por sus siglas en inglés), define un protocolo o un conjunto de normas para el intercambio de datos en serie entre dos dispositivos. UART es sumamente simple y utiliza solo dos hilos entre el transmisor y

el receptor para transmitir y recibir en ambas direcciones. Ambos extremos tienen una conexión a masa tal como se ve en la figura 6.

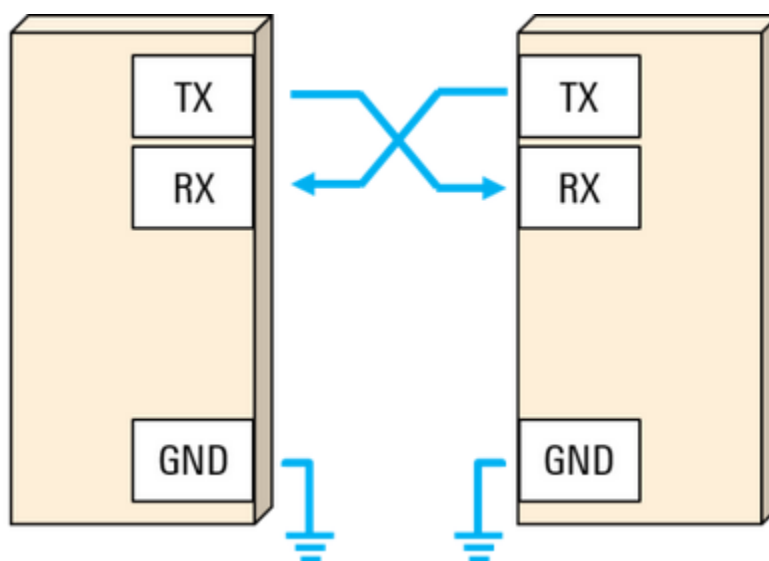


Figura 6: Diagrama conexión UART

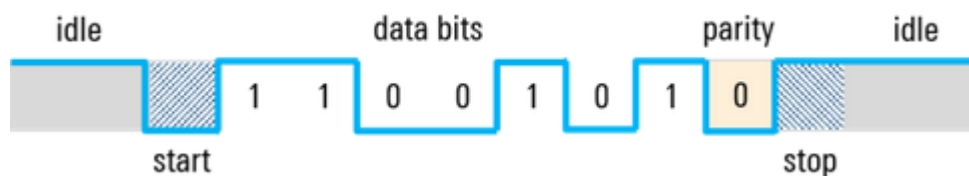


Figura 7: Formato de trama en UART

La comunicación en UART puede ser simplex (los datos se envían en una sola dirección), semidúplex (cada extremo se comunica, pero solo uno al mismo tiempo), o dúplex completo (ambos extremos pueden transmitir simultáneamente). En nuestro caso tenemos una comunicación del tipo dúplex completo. Esta comunicación se va a dar entre el microcontrolador y el conversor al protocolo USB desde el cual se va a conectar directamente a la PC para procesar la información y poder mostrarla a través de Matlab.



Figura 8: Diagrama de conexión entre el uC y la PC

3 – Configuración del microcontrolador

Para poder analizar la programación del software del Analizador lógico (AL) se dividirá el proyecto en tres partes:

1. Configuración gráfica del microcontrolador: Donde se utilizarán las herramientas propias del CubeIDE para el STM32f103 para configurar puertos e interrupciones.
2. Programación del microcontrolador: Aquí se hará uso de los puertos previamente configurados para generar un código que nos permita tomar las muestras en los tiempos requeridos y enviarlos al pc por el protocolo UART
3. Programación en Matlab: Se revisará el código que nos permite recibir los datos para poder visualizarlos.

3.1 – Configuración grafica

En este prototipo fue utilizado el STM32f103 acoplado a la placa de desarrollo BluePill. Sin embargo, la instalación del CubeIDE y la conexión del programador junto con la formación de un proyecto no será explicada dado que es información general muy simple de conseguir. Para este proyecto se partirá del hecho de que se tiene un proyecto generado.

Al comenzar un proyecto, una sugerencia: Antes de comenzar a programar es recomendable dirigirse a System Core (Dentro de la pestaña Pinout & Configuration) y dentro de SYS seleccionar Serial Wire, esto permitirá al programador no solo tener control del microcontrolador a la hora de debuggear, sino que facilitará su programación sin tener que hacer modificaciones en la placa.

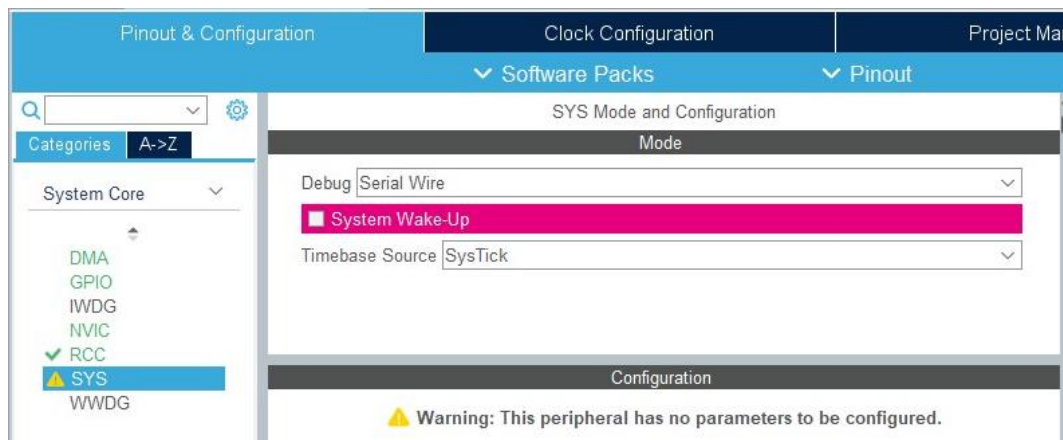


Figura 9: Configuración gráfica STM32

3.2 – Configuración del cristal

Para poder alcanzar la velocidad máxima de la placa BluePill se deberá hacer uso del cristal externo el cual nos dará una frecuencia que podremos amplificar hasta los 72[MHz]. Para ello habrá que dirigirse a RCC dentro de System Core para elegir la opción de Crystal/Ceramic Resonator entre las opciones de HSE.



Figura 10: Configuración cristal 1

Una vez habilitado el cristal externo podremos dirigirnos a la pestaña Clock Configuration y escribir en la casilla señalada la máxima velocidad permitida para el clock interno. (En este caso 72[MHz]). El software hará los demás cálculos.

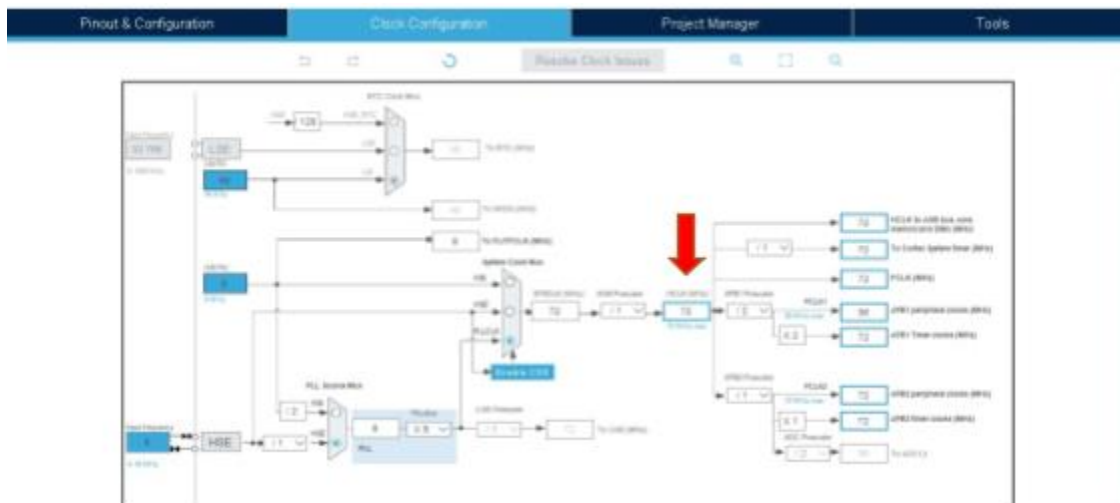


Figura 11: Configuración cristal 2

3.3 – Interrupción por timer

Para el Analizador Lógico el poder tomar muestras en una frecuencia precisa es importante por lo que el uso de la interrupción por tiempo se vuelve esencial para el proyecto. Para poder configurarlo basta con ir a la sección de Timer y seleccionar uno de los TIM, en este caso se seleccionó el TIM2 y se colocó el reloj interno como clock source.

En la configuración del mismo se modificó el valor de preescaler para que de los 72[MHz] solo se tomase 1[MHz] el cual podría ampliarse. Antes de finalizar es recomendable prestar atención al Counter Period el cual se modificará luego en el código.

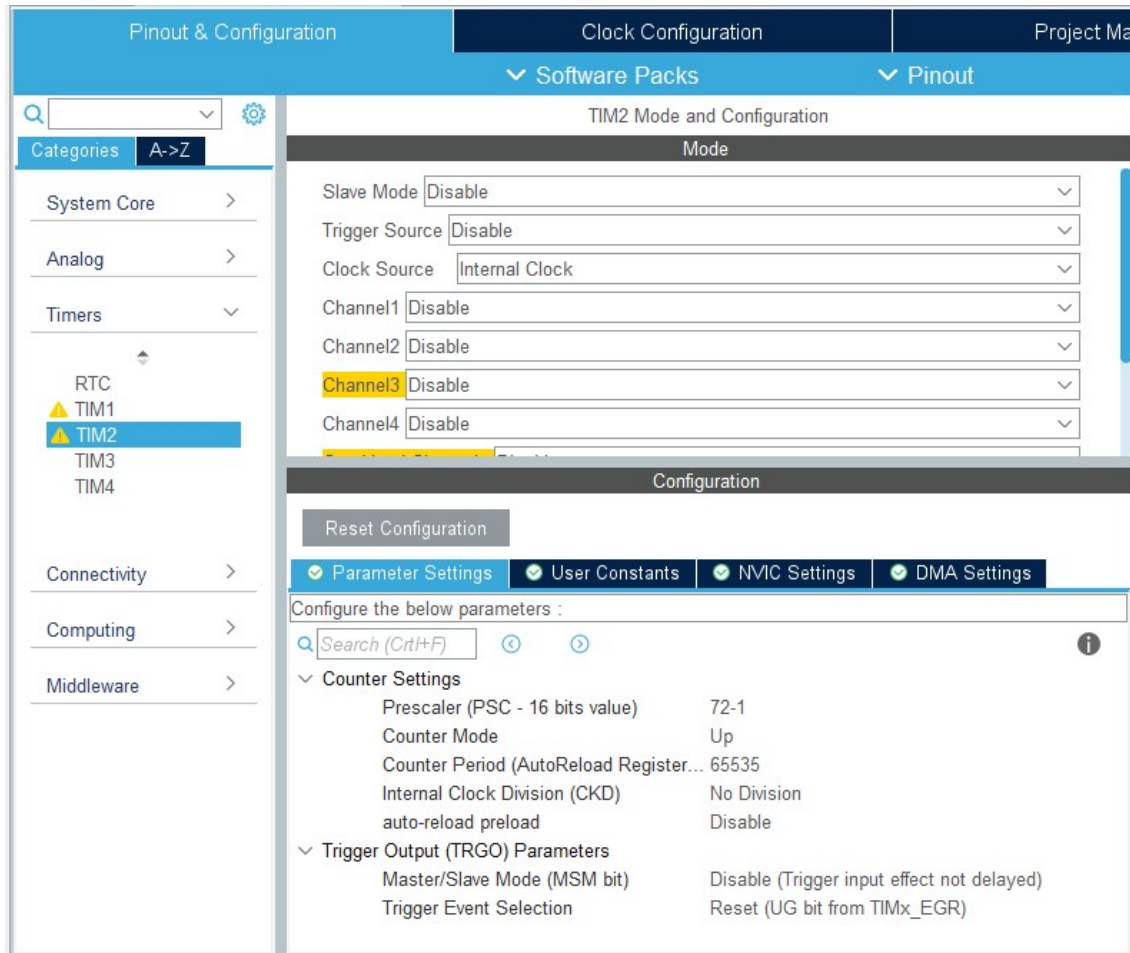


Figura 12: Configuración interrupción por timer

3.4 – UART

Para configurar la comunicación serial asíncrona UART hay que dirigirse a Connectivity y luego seleccionar una USART disponible. Para este caso ya se habían seleccionado los pines que se iban a usar en el AL por lo que solo quedo la opción USART1 el cual se configuro en modo asíncrono. En la configuración se seleccionó la velocidad en 115200Bits/s y el ancho de palabra en 8bits, en este caso no se trabajó con paridad. (De haber elegido paridad se seleccionaría un ancho de palabra de 9 bits.)

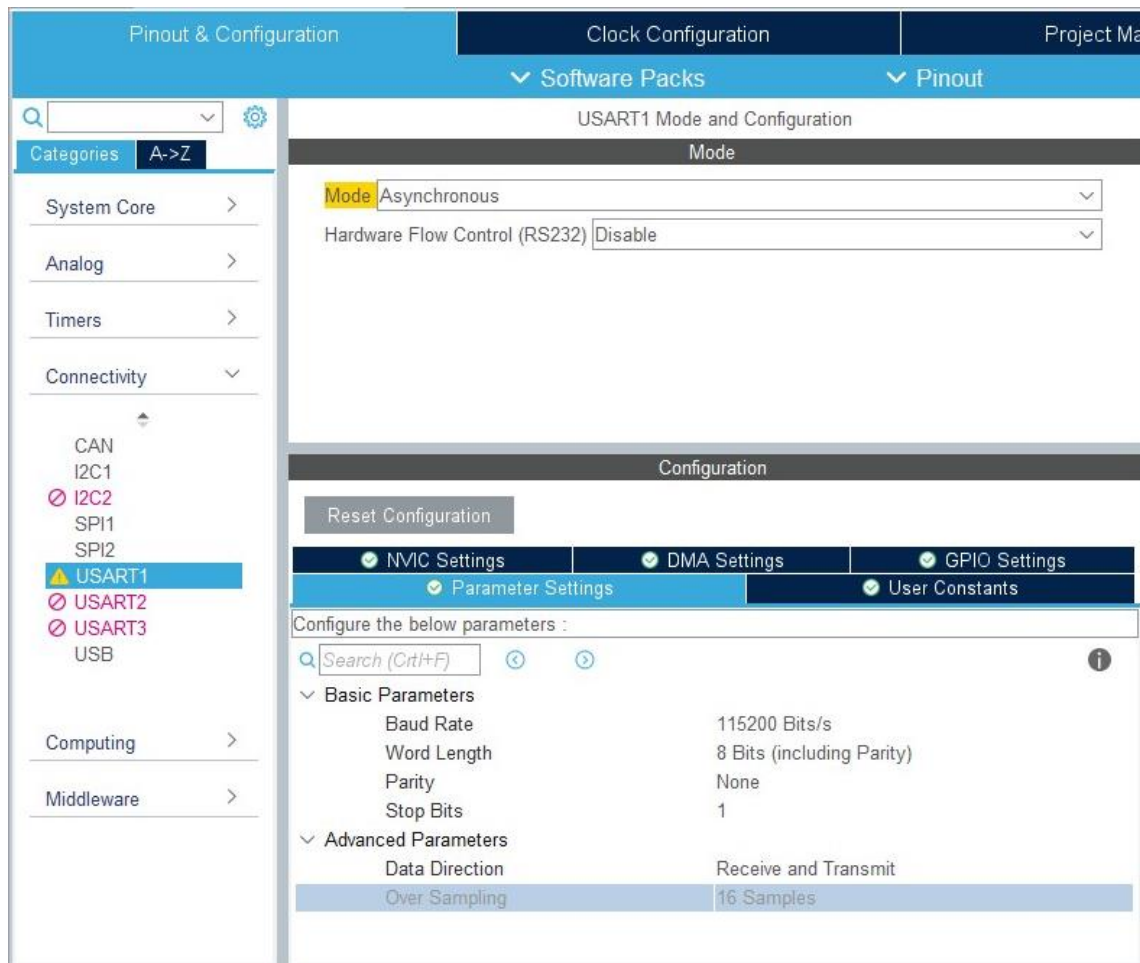


Figura 13: Configuración UART

3.5 – Selección de pines

Para este trabajo se seleccionaron los primeros 8 pines del puerto A (en configuración input) por estar espacialmente consecutivos y pertenecer todos al mismo puerto lo que más tarde permitirá leer todos los pines con una sola instrucción (IDR) ahorrando tiempo.

Así también, se seleccionó un pin Led en el pin 13 del puerto C y dos interrupciones externas en el pin 2 del puerto B (Boton) y el pin 10 del puerto B (Pin) cuyas configuraciones se explican más tarde.

La interrupción externa “Boton” se configura para activarse en el flanco ascendente de una señal producida por un botón en pull down.

| | | | | | | | |
|------------|-----|-----|---------------|---------------|-----|-------|-------------------------------------|
| PB2 | n/a | n/a | External I... | Pull-down | n/a | Boton | <input checked="" type="checkbox"/> |
| PB10 | n/a | n/a | External I... | No pull-up... | n/a | Pin | <input checked="" type="checkbox"/> |
| PC13-TA... | n/a | Low | Output P... | No pull-up... | Low | Led | <input checked="" type="checkbox"/> |

PB2 Configuration :

GPIO mode

External Interrupt Mode with Rising edge trigger detection

GPIO Pull-up/Pull-down

Pull-down

User Label

Boton

Figura 16: Ventana pin para interrupción 1

La interrupción externa “Pin” se configura para activarse en el flanco descendente de un pin conectado a un microcontrolador “No pull-up and no pull-down”

| | | | | | | | |
|------------|-----|-----|---------------|---------------|-----|-------|-------------------------------------|
| PB2 | n/a | n/a | External I... | Pull-down | n/a | Boton | <input checked="" type="checkbox"/> |
| PB10 | n/a | n/a | External I... | No pull-up... | n/a | Pin | <input checked="" type="checkbox"/> |
| PC13-TA... | n/a | Low | Output P... | No pull-up... | Low | Led | <input checked="" type="checkbox"/> |

PB10 Configuration :

GPIO mode

External Interrupt Mode with Falling edge trigger detection

GPIO Pull-up/Pull-down

No pull-up and no pull-down

User Label

Pin

Figura 17: Ventana pin para interrupción 2

El pin “Led” se mantiene sin cambios recordando que fue configurado como salida.

| | | | | | | | |
|------------|-----|-----|---------------|---------------|-----|-------|-------------------------------------|
| PB2 | n/a | n/a | External I... | Pull-down | n/a | Boton | <input checked="" type="checkbox"/> |
| PB10 | n/a | n/a | External I... | No pull-up... | n/a | Pin | <input checked="" type="checkbox"/> |
| PC13-TA... | n/a | Low | Output P... | No pull-up... | Low | Led | <input checked="" type="checkbox"/> |

GPIO output level

Low

GPIO mode

Output Push Pull

GPIO Pull-up/Pull-down

No pull-up and no pull-down

Maximum output speed

Low

User Label

Led

Figura 18: Ventana pin para interrupción 3

3.6 – Habilitación de interrupciones

No basta con configurar las interrupciones, estas deben ser habilitadas. Para ellos hay que dirigirse a NVIC donde se seleccionaran las interrupciones.

Nótese que la interrupción TIM2 fue modificada en prioridad para que no esté por encima de las demás interrupciones externas.

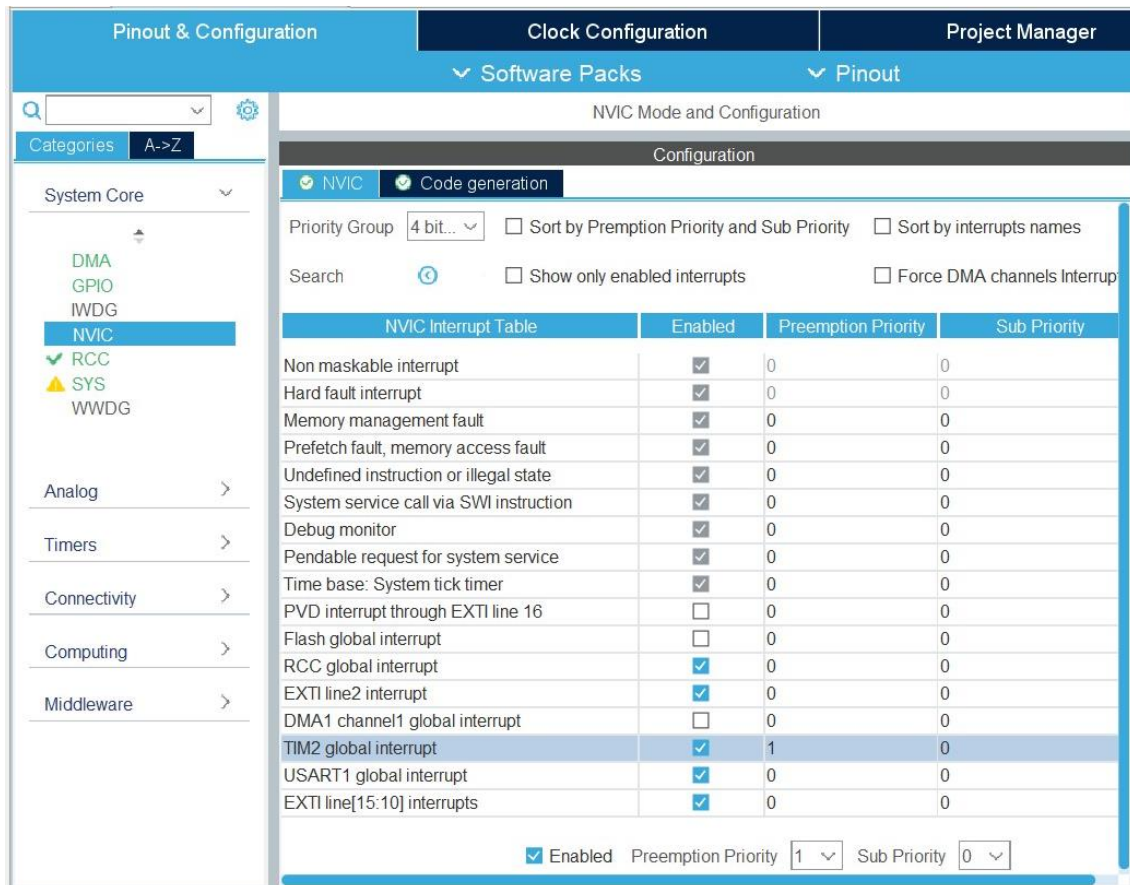
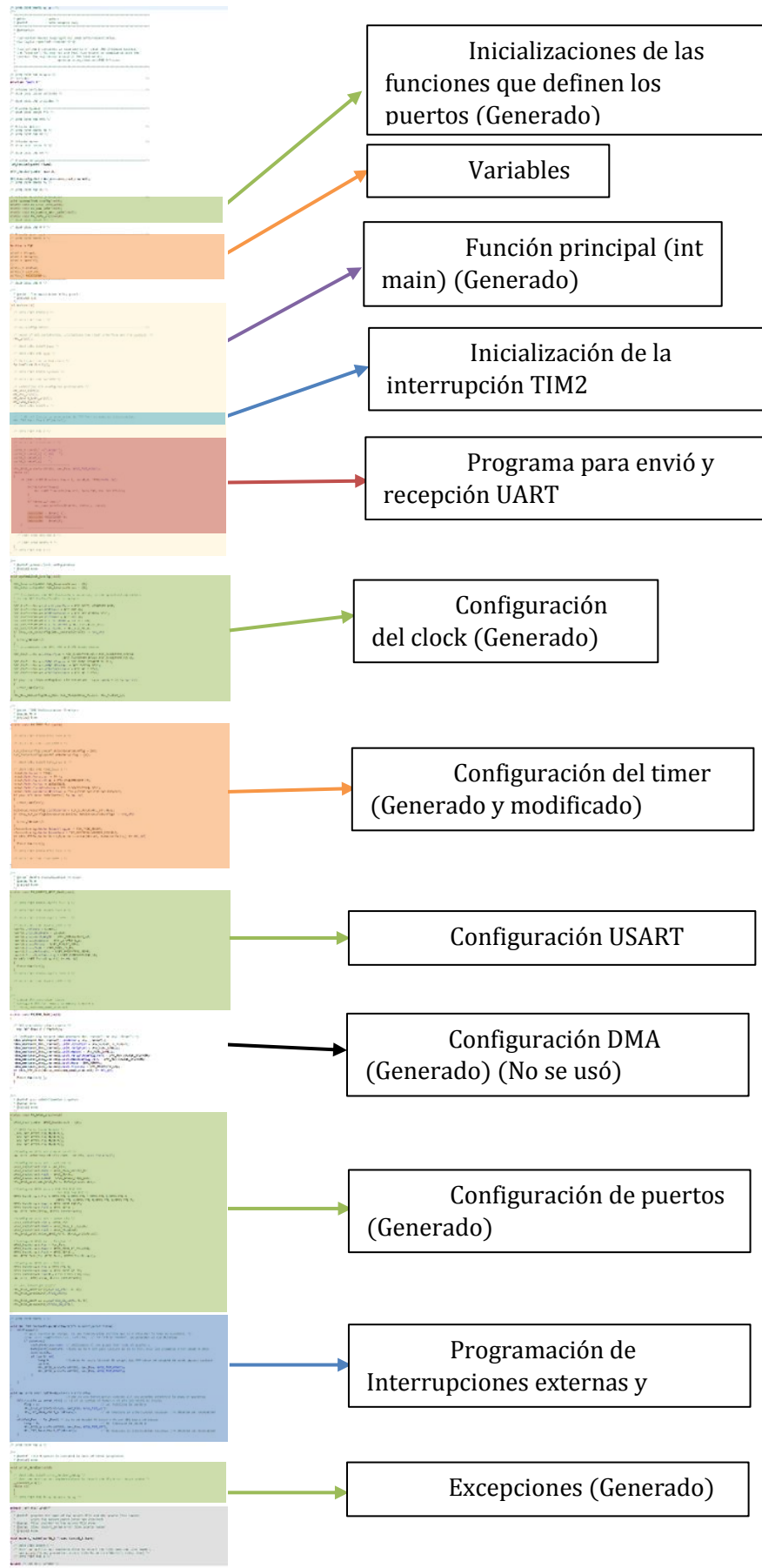


Figura 19: Ventana habilitación de interrupciones

4 – Programación del microcontrolador

Una de las grandes dificultades al analizar código es diferenciar sus partes, especialmente cuando se trata de códigos muy extensos, en la siguiente página se puede ver un análisis espacial del código. Las secciones de código en gris son las que fueron generadas a través de la configuración gráfica y no se les prestara mucha atención en esta etapa dado que ya fueron explicados.



4.1 – Variables globales

```
62Ⓜ /* Private user code ----- */
63 /* USER CODE BEGIN 0 */
64 //-----
65 #define n 500
66 //-----
67 uint8_t flag=1;
68 uint8_t Dato[n];
69 //-----
70 uint16_t cont=0;
71 uint16_t Lectura;
72 uint16_t Velocidad=1;
73Ⓜ //-----
74 /* USER CODE END 0 */
75
```

Figura 20: Variables globales

Dentro de la sección USER CODE se puede colocar código el cual no será afectado si se hace alguna modificación nueva en la configuración gráfica.

USER CODE 0 Se encuentra fuera de cualquier función por lo que lo que se inicializa aquí se inicializa en todo el programa.

En la línea:

65: se define un valor n que corresponde al número de muestras que quiero tomar.

67: La bandera de inicialización de toma de muestra (El cual puede ser reemplazado por una variable booleana)

68: Un array de 500 valores de 8 bit.

70: Un contador de 16 bit (8 bits son 256 y debe llegar a 500)

71: Variable de lectura de datos para leer los 16 pines del puerto A

72: Valor de 16 bit para reemplazar por el valor “Counter Period” antes mencionado.

4.2 – Iniciación de la interrupción TIM2

```
107 /* USER CODE BEGIN 2 */
108
109 //-----
110 // * @brief Inicia la generación de TIM Base en modo de interrupción.
111 HAL_TIM_Base_Start_IT(&htim2);
112 //-----
113
114 /* USER CODE END 2 */
```

Figura 21: Interrupción TIM2

No se encontró manera de configurarlo gráficamente por lo que es importante agregar esta línea en la etapa de programación. Esta línea fue agregada tanto en la línea 111 como en todas las interrupciones que reiniciaban la toma de muestra.

4.3 – Programa para envío y recepción UART

```

116  /* Infinite loop */
117  /* USER CODE BEGIN WHILE */
118  //-----
119  uint8_t Comp[6] ={"Cargar"};
120  uint8_t Comp2[6] ={"Vel  "};
121  uint8_t dataR[6] = "";
122  uint8_t dataV[2] = "";
123  //-----
124  HAL_GPIO_WritePin(GPIOC, Led_Pin, GPIO_PIN_RESET);
125  while (1)
126  {
127      if (!HAL_UART_Receive( &huart1, dataR,6, 1500)==HAL_OK){
128          //-----
129          if(*dataR==*Comp){
130              HAL_UART_Transmit(&huart1, Dato,500, HAL_MAX_DELAY);
131          }
132          //-----
133          if(*dataR==*Comp2){
134              HAL_UART_Receive(&huart1, dataV,2, 1000);
135
136              Velocidad = dataV[ 1];
137              Velocidad=Velocidad<<8;
138              Velocidad = dataV[0];
139          }
140          //-----
141      }
142  }
143  /* USER CODE END WHILE */
144  /* USER CODE BEGIN 3 */

```

Figura 22: Programa envío - recepción UART

En la línea 127 el microcontrolador revisa en cada loop si un mensaje de 6 bit fue enviado. De ser así, pasa a comprobar entre las dos palabras claves Comp ("Cargar") o Comp2 ("Vel ") En el primer caso se transmiten los 500 datos cargados, en el segundo se reciben dos variables de 8 bit las cuales se cargan una a una en la variable Velocidad haciendo uso de una técnica de corrimiento de bit para convertir dos datos de 8 bit en un dato de 16 bits.

4.4 – Configuración del timer

Como se puede ver en la línea 209 de la figura 23, se cambia el valor generado por la configuración grafica 65535 por una variable de 16 bits que cambiaremos a gusto a través de Matlab cambiando así la velocidad de muestreo.

Para tener en consideración, el valor por defecto es el máximo valor que puede tener y no puede tener comas por lo que si bien es amplio el espectro de frecuencias es limitado en detalle, esto se complica más si en vez de 1[MHz] de velocidad máxima se usan 70[MHz]

Otro aspecto importante, es que esta sección no se encuentra en una zona segura, por lo que cualquier modificación del código a través de la configuración grafica renovara los valores por defecto.

```
198  /* USER CODE END TIM2_Init 0 */
199
200  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
201  TIM_MasterConfigTypeDef sMasterConfig = {0};
202
203  /* USER CODE BEGIN TIM2_Init 1 */
204
205  /* USER CODE END TIM2_Init 1 */
206  htim2.Instance = TIM2;
207  htim2.Init.Prescaler = 72-1;
208  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
209  htim2.Init.Period = Velocidad;
210  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
211  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
212  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
213  {
214      Error_Handler();
215  }
216  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
217  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
218  {
219      Error_Handler();
220  }
221  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
222  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
223  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
224  {
225      Error_Handler();
226  }
227  /* USER CODE BEGIN TIM2_Init 2 */
```

Figura 23: Configuraciones del timer.

4.5 – Programación de interrupciones externas y cíclicas

En la figura 24 se muestra la rutina para la interrupción por timer y la rutina para la interrupción externa. La primera función es la interrupción timer que se configura para que tome muestras a la velocidad configurada a través de los 16 pines del puerto A en una variable de 16 bit. Pero como solo queremos los primeros 8 bit igualamos la variable Lectura con una de las variables del array Dato de 8 bit. Cuando el contador llega a 500 se deshabilita la bandera, se reinicia el clock y se apaga el led.

En ambas interrupciones externas se habilita la bandera, se enciende el Led y se reinicia la base de tiempo de la interrupción cíclica.


```

353= /* USER CODE BEGIN 4 */
354 //-----
355 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
356 { if(flag==1){
357     /* Aquí escribo mi código, es una interrupción cíclica que va a ejecutar la toma de muestras. */
358     //HAL_GPIO_TogglePin(GPIOC, Led_Pin); // Pin led de control, si parpadea es que funciona
359     if (cont<n){
360         Lectura=GPIOA->IDR; // Utilizando el IDR puedo leer todo el puerto A
361         Dato[cont]=Lectura; //Dato es de 8 bit pero Lectura es de 16 bit, solo los primeros 8 bit pasan a Dato
362         cont=cont+1;
363         if (cont>n){
364             flag=0; //Cuando el micro termina de cargar los 500 datos se bloquea la interrupción cíclica
365             cont=0;
366             HAL_GPIO_WritePin(GPIOC, Led_Pin, GPIO_PIN_RESET);
367             HAL_GPIO_WritePin(GPIOC, Led_Pin, GPIO_PIN_RESET);
368         }
369     }
370 }
371 }
372 //-----
373 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
374 { //Esta es una interrupción externa que nos permite reiniciar la toma de muestras
375     if(GPIO_Pin == Boton_Pin){ // Si el se actiba el boton o el pin del micri de inicio
376         flag = 1; // Se habilita la bandera
377         HAL_GPIO_WritePin(GPIOC, Led_Pin, GPIO_PIN_SET);
378         HAL_TIM_Base_Start_IT(&htim2); // Se reactiva la interrupción cíclica. (no debería ser necesaria)
379     }
380     if(GPIO_Pin == Pin_Pin){ // Si el se actiba el boton o el pin del micri de inicio
381         flag = 1; // Se habilita la bandera
382         HAL_GPIO_WritePin(GPIOC, Led_Pin, GPIO_PIN_SET);
383         HAL_TIM_Base_Start_IT(&htim2); // Se reactiva la interrupción cíclica. (no debería ser necesaria)
384     }
385 }
386= //-----
387 /* USER CODE END 4 */

```

Figura 24: Rutinas interrupción timer e interrupción externa

5 – Programación en Matlab

Se generaron dos códigos en Matlab para interactuar con el Analizador Logico, uno cuya función es variar la velocidad de muestreo y otro que nos permite llamar al microcontrolador para tomar las muestras que ha tomado para posteriormente poder graficarlas.

Variación de velocidad de muestreo:

```

%% Elegimos velocidad entre 15[Hz] y 1[MHz]
Fm=5625;
Div = round(1000000/Fm);
if(Div>65535)
    Div=65535;
elseif (Div<1)
    Div=1;
end
%% Comprobamos
Div
Fm=1000000/Div

%% Enviamos la configuracion de velocidad:
S = serialport("COM5",115200);
Veloc = uint16(Div);
Divisor = typecast(Veloc,'uint8');

write(S,'Vel ', "uint8");
write(S,Divisor, "uint8");

```

Figura 25: Código Matlab parte 1

Este se divide en tres simples secciones, la primera toma el dato de la velocidad deseada, lo divide por la velocidad máxima que es un mega Hertz y obtiene el entero más próximo y revisa que no exceda los límites de los 16 bits (0 a 65535).

La segunda sección simplemente muestra en pantalla la división real posible por el programa y la frecuencia de muestreo real obtenida.

Por último se habilita la comunicación serie, se toma el dato de 16 bits y se lo divide en dos datos de 8 bits. Se envía una palabra clave para que el micro reconozca que lo que se envía es la velocidad para concluir enviando los dos datos. Estos como ya se explicó en el código de stm32 en la sección de UART se reconstruyen dentro del micro usando corrimiento de bits.

Recepción de datos y gráficos:

```
4      %% Creamos variable con el numero de muestras.
5      M=500;
6      %% Cargamos el puerto serial.
7      S = serialport("COM5",115200);
8      %% Recibimos datos:
9          write(S,'Cargar','uint8');
10         pause(0.2)
11         Muestras = read(S,500,'uint8');
12     %% Convertimos dato a binario.
13     bin = zeros(M,8,'uint8');
14     for i=1:M
15         bin(i,:)= bitget(Muestras(i),8:-1:1);
16     end
17     %% Graficar
18     Min = 0;
19     Max = 50;
20     Ancho=1000/(Max-(Min-1));
21     figure(1)
22     for n=1:8
23         subplot(8,1,n)
24         stem(bin(:,n),'.','LineWidth',Ancho);
25
26         title(['Canal numero ' num2str(n)]);
27         ylim ([-0.1 1.1])
28         xlim ([Min Max])
29     end
```

Figura 26: Código Matlab parte 2

Luego de crear la variable del número de muestras y habilitar la comunicación serial se envía una palabra clave “Carga” que le hace entender al microcontrolador que debe enviarnos los 500 datos tomados.

Estos datos si bien son de 8 bits no están en formato binario, por lo que uno a uno vamos transformando a binario haciendo uso de la función `bitget` y depositándolo en la variable “bin” previamente creada de forma estática para ahorrar tiempo.

En la última parte del código se grafica de forma parcial o total. Para restringir la sección graficada hacemos uso de Min y Max a partir de los cuales se calculará un ancho de las barras con respecto a la cantidad de barras graficadas. Este cálculo fue generado basado en prueba y error, no se conoce la matemática detrás de ello pero funciona.

Por último y para no ser recursivos en el código hacemos uso de un `for` para graficar cada uno de los 8 canales con su respectivo nombre uno debajo del otro.

6 – Resultados prácticos

Para que todo esto funcione debemos hacer el conexionado correcto.

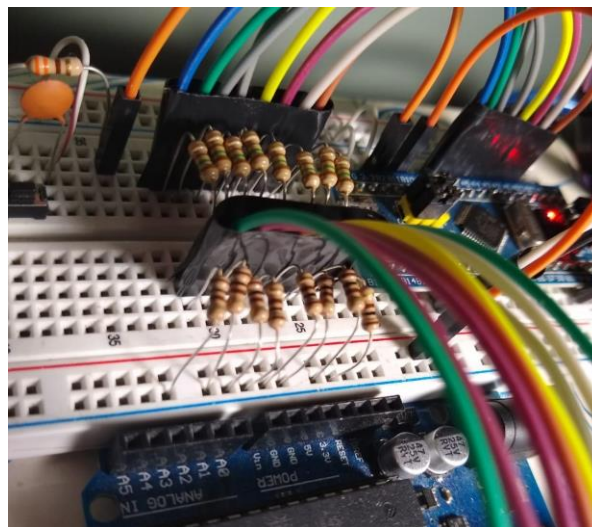


Figura 27: Conexionado realizado protoboard

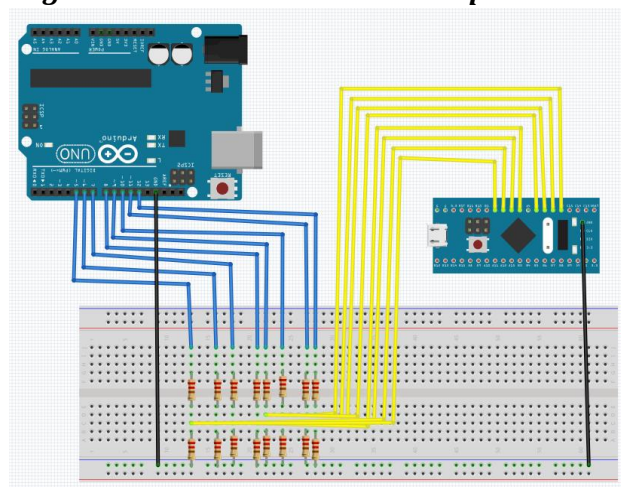


Figura 28: Esquema de conexión divisores resistivos

En este caso en concreto como el microcontrolador stm32 funciona a 3.3v y el Arduino uno al que está conectado envía señales de 5v. Debimos hacer un divisor resistivo con resistencias de 100y 150 ohm para convertir 5v en 3v lo cual funciona perfectamente. A su izquierda se puede apreciar el botón que va conectado al pin “Boton” como se muestra en el esquema.

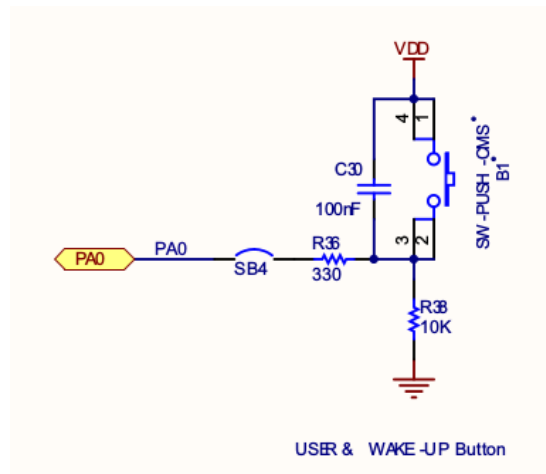


Figura 29: Esquema conexionado del botón

El microcontrolador se comunica con la Pc a través de un adaptador de comunicación serial a USB. Cp2102, mostrado en la figura 30



Figura 30: Conversor USB – UART CP2102

Conexión que puede apreciarse en la siguiente imagen:

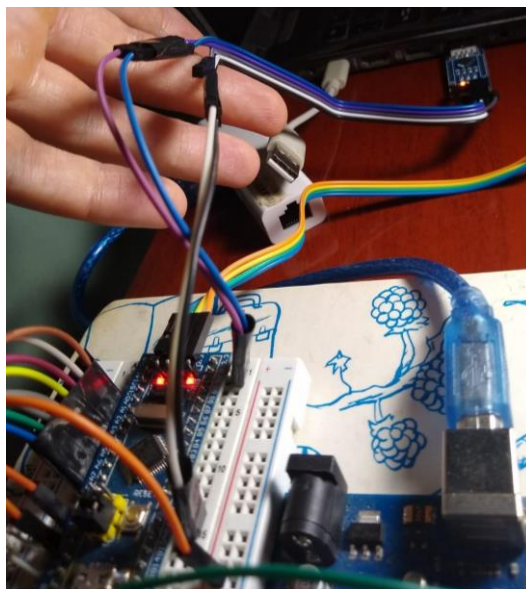


Figura 31: Conexionado CP2102

El programa parece cumplir con las expectativas. Las gráficas son comprensibles y las barras tienen el ancho óptimo. Para este caso se configuro el Arduino para que genere datos al azar así poder medirlos con el analizador lógico. Fueron medidos los primeros 50 valores de 500 para una mayor comodidad visual.



Figura 32: Resultados de mediciones sobre placa Arduino.

7 – Presupuesto

Teniendo en cuenta todos los componentes necesarios para implementar el analizador de lógico y la mano de obra requerida (incluida la carcasa plástica para el producto final), el presupuesto final para el proyecto da un total de \$14.720 pesos argentinos. El detalle del presupuesto se encuentra en un archivo Excel anexado junto con los demás archivos del proyecto.

8 – Conclusiones

Podemos resumir las conclusiones arribadas a lo largo del desarrollo del proyecto del analizador lógico en los siguientes puntos:

- El microcontrolador Blue Pill STM32f103 cumple con las expectativas para la fabricación del Analizador lógico.
- No se terminó utilizando DMA (Acceso Directo a Memoria) sin embargo podría implementarse.
- La comunicación UART funciona correctamente pero una comunicación síncrona podría ser más efectiva.
- El micro puede tomar muestras de 15Hz a 1Mhz pero es posible obtener mas velocidad.
- Podría utilizarse todo el puerto A obteniendo así un analizador lógico de 16 canales sin alterar su velocidad de muestreo.
- De poder comprarse los microcontroladores al por mayor y mejorando detalles de programación se obtendría un producto que podría competir en el mercado.

Analizador de Protocolo SPI

1 - El protocolo SPI

El protocolo SPI, acrónimo para Serial Peripheral Interface, es un estándar en las comunicaciones entre circuitos integrados. Fue desarrollado en la década del 80 por la empresa Motorola para permitir la comunicación entre diferentes sistemas electrónicos digitales como lo eran sus microcontroladores de la época.

El SPI es una comunicación tipo Master - Slave sincrónica, utilizando un bus de reloj para sincronizar la transmisión de datos entre los dispositivos. Estas transmisiones se realizan mediante dos buses, siendo uno para los datos que el maestro envía al esclavo o recibe del mismo, y otro bus para los datos que el esclavo envía al maestro o recibe del mismo. Esta configuración indica que el protocolo SPI es full duplex, es decir, que la comunicación entre los dispositivos que la utilizan puede suceder en simultáneo. La transmisión es de tipo serie y en cada ciclo de reloj se envía 1 bit de la cantidad total que componen el mensaje transmitido. Por último, existe un bus de selección de chip para identificar el Slave con el que el Master se quiere comunicar. En total, el SPI es un protocolo que solo requiere 4 líneas físicas de conexión para comunicar a los dispositivos que lo utilizan. Una conexión típica entre un maestro y un esclavo SPI luce como muestra la **figura 33**.

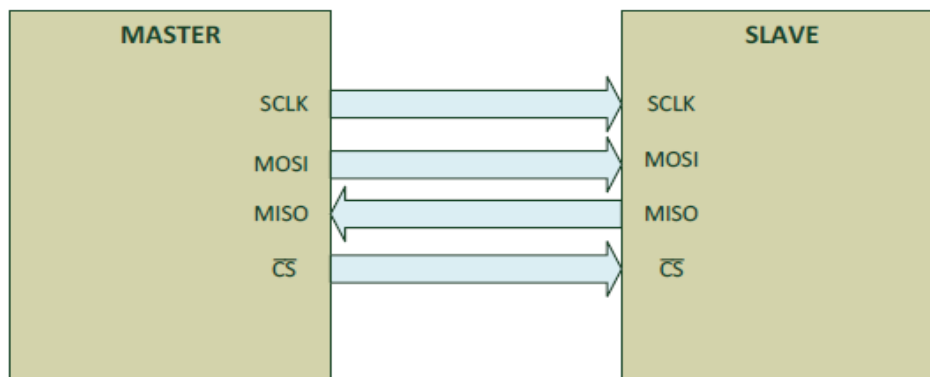


Figura 33: Conexión típica maestro y esclavo SPI

❖ SCLK

Señal de reloj generada por el maestro y utilizada para sincronizar la transferencia del mensaje. Cada bit del mensaje que se envía por las líneas de datos MOSI y MISO dura en dicha línea un ciclo de este reloj.

❖ MOSI

Línea de datos que utiliza el maestro para enviar datos al esclavo. De haber más de un esclavo, todos se conectan en paralelo a esta línea.

❖ MISO

Línea de datos que utiliza el esclavo para enviar datos al maestro. De haber más de un esclavo todos se comunican con el maestro a través de esta línea.

❖ CS

Línea de selección de esclavos gestionada por el maestro. Cada esclavo posee una entrada propia de selección y el maestro posee una salida para cada esclavo conectado a esta línea. Por medio de esta línea el maestro elige el esclavo con el que se quiere comunicar.

Existen cuatro formas en las que se puede configurar una transferencia de datos entre maestro y esclavo, variando entre ellas el momento exacto en el que se escriben los bits en las líneas MISO y MOSI y el estado inicial de la línea SCK (alto o bajo por defecto). De estas cuatro, nosotros nos atenderemos a la más sencilla: el MODO 1.

En el modo de transferencia 1, una transmisión de una secuencia de bits entre el maestro y el esclavo por las líneas MOSI y MISO se ve como indica la figura 34:

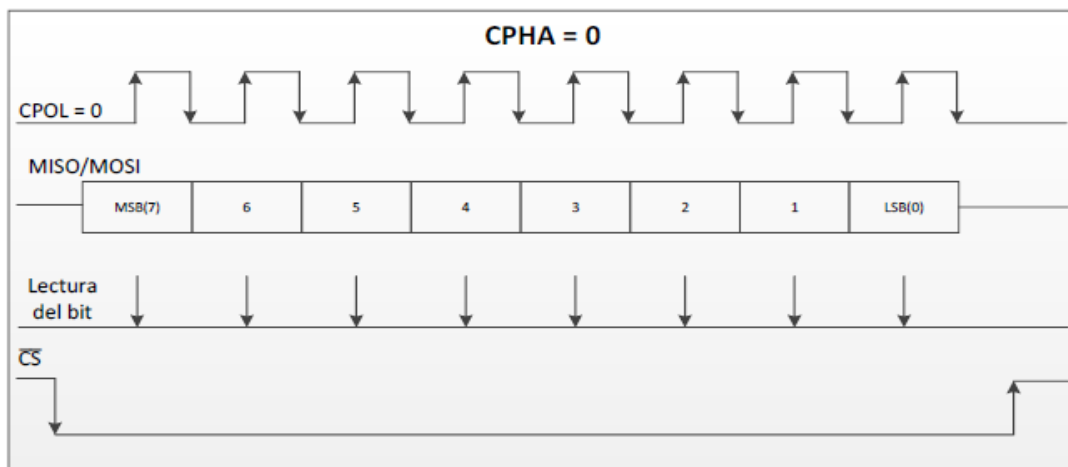


Figura 34: Modo 1 transmisión SPI

El procedimiento es el siguiente:

1. Se da comienzo a la transferencia de bits al poner en bajo la línea CS.
2. El maestro y el esclavo escriben el primer bit de la secuencia a transmitir antes de que llegue el primer flanco de subida de la señal SCK.
3. En el primer flanco de subida de SCK se lee en la línea correspondiente el bit de mayor peso (MSB) de la secuencia, y en los flancos de subida siguiente se leen los demás bits. En los flancos de bajada se escriben los bits en la línea

correspondiente. El maestro escribe en la línea MOSI y lee en la MISO, y el esclavo escribe en la MISO y lee en la MOSI.

4. Una vez transferidos todos los datos a las líneas, se detiene la señal SCK y se pone en alto la línea CS indicando el fin del proceso.

Si observamos la **figura 34**, veremos como en el modo 1 la cantidad de bits transmitidos será igual a la cantidad de flancos ascendentes de la señal SCK. Esto de gran importancia para determinar cuántos bits se están transfiriendo el maestro y el esclavo, ya que en el protocolo SPI la longitud del mensaje no está definida, pudiendo ir desde un mínimo de 1 bit a un máximo indefinido.

2 – Diseño e implementación del Analizador de protocolo SPI

El diseñar un analizador de protocolo SPI se puede dividir en dos partes fundamentales: Un bloque muestreador, encargado de capturar los mensajes de las líneas MISO y MOSI, y un bloque de control, encargado de procesar los mensajes capturados y enviarlos al PC de manera legible para un usuario. A modo de visualización, el esquema completo del analizador de protocolo SPI conectado en paralelo a una comunicación SPI entre dos dispositivos y enviando los datos capturados a un ordenador sería como muestra la **figura 35**.

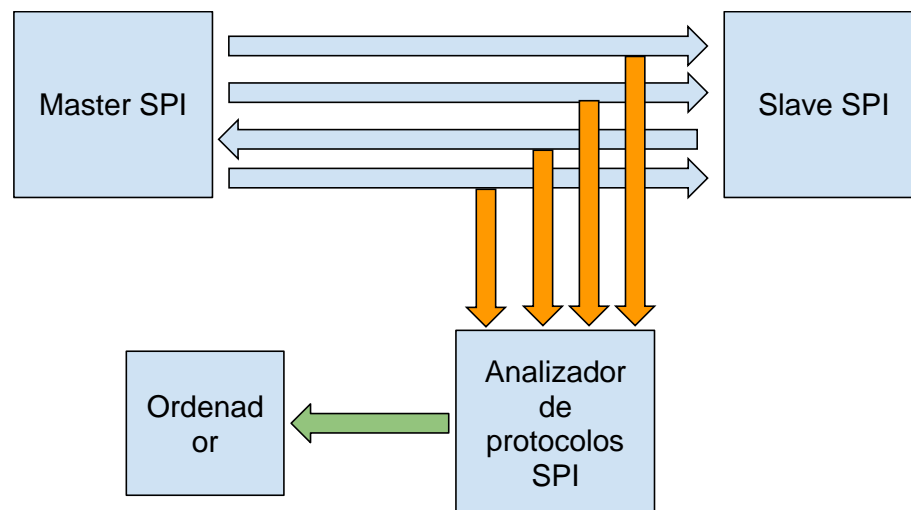


Figura 35: Diagrama del sistema completo

Antes de continuar, es importante destacar que el diseño realizado en el presente proyecto se pensó para utilizarse sobre una comunicación entre dos Arduino UNO. Esto es importante debido a que en Arduino, el protocolo SPI solo manda mensajes fijos de una longitud igual a 1 byte. Es decir que siempre se capturarán mensajes de la misma longitud, lo cual simplifica el diseño del analizador de protocolo SPI ya que no será necesario

implementar un circuito que determine la longitud del mensaje enviado entre el maestro y el esclavo o viceversa. Siendo estas las condiciones, comenzaremos con el diseño del bloque muestreador.

2.1 – Bloque muestreador

El bloque de muestreo o muestreador es el encargado de capturar los mensajes que viajan por las líneas MISO y MOSI. En dichas líneas, los dispositivos Maestro y Esclavo transmiten bits en forma serial que en conjunto componen el mensaje enviado. Por teoría de funcionamiento del protocolo SPI, estos bits comienzan a escribirse en su respectiva línea cuando la señal en la línea CS pasa de un estado alto a bajo, y terminan de escribirse cuando CS vuelve a estado alto, escribiéndose un bit en cada ciclo de la señal de clock SCK. En Arduino, cada mensaje está representado por 8 bits y por lo tanto ocurren 8 señales de clock. Como los bits solo duran una señal de clock, si no se los almacena en algún registro a medida que aparecen en las líneas MISO y MOSI, en el próximo ciclo de clock, desaparecerán. Es por esto que evitar perder información, es necesario implementar un buffer que almacene estos bits, para luego guardarlos en una memoria. Será necesario un buffer para la línea MISO y uno para la línea MOSI, ya que la comunicación es full dúplex y tanto esclavo como maestro pueden enviar mensajes al mismo tiempo.

En nuestro caso, decidimos implementar estos buffers mediante un registro de desplazamiento con Flip – Flops tipo D. Para que el buffer pueda almacenar 8 bits, necesitaremos 8 flip flops. La **figura 36** y **figura 37** muestran el diseño en Proteus del registro de desplazamiento y su implementación en protoboard, respectivamente.

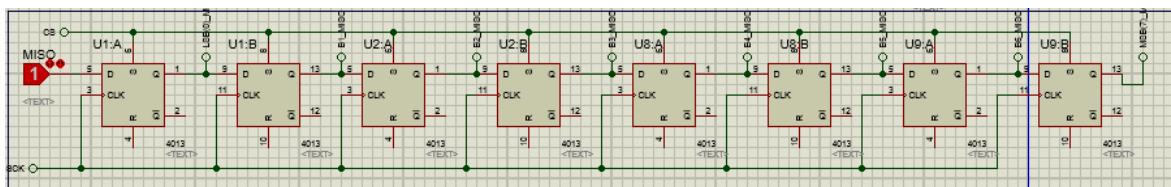


Figura 36: Registro de desplazamiento Proteus

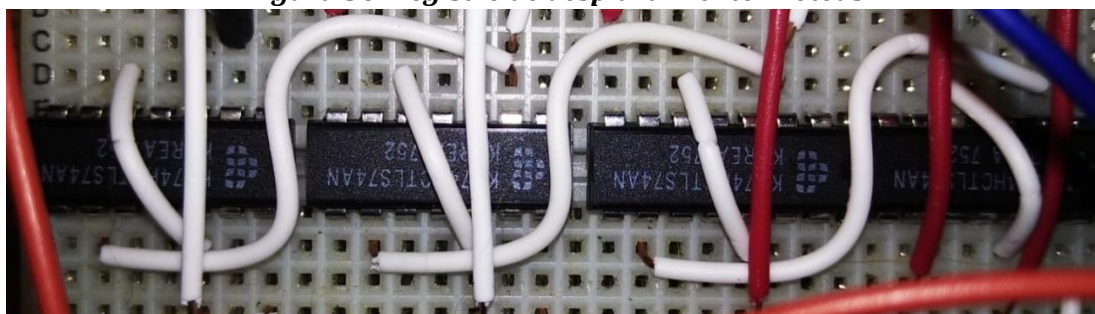


Figura 37: Registro de desplazamiento protoboard

Como dijimos anteriormente, debido a que en Arduino los mensajes siempre son de 8 bits, no será necesario implementar un circuito que permita detectar cuantos bits se enviaron.

Por lo tanto, el bloque muestreador consta solamente de dos registros de desplazamiento, uno para la línea MISO y otro para la MOSI. En ambos registros hacemos ingresar la señal de SCK y su respectiva señal de la línea MISO o MOSI. Las salidas Q de cada uno de los flip flops se conectan al bloque de control, el cual se encarga de guardarlas, procesarlas y enviarlas.

2.2 – Bloque de control

Luego de almacenados momentáneamente los bits que se enviaron por la línea MOSI y MISO en sus respectivos registros de desplazamiento, en necesario procesarlos lo antes posible, ya que de no hacerlo se perderán cuando lleguen nuevos bits por las líneas. El bloque que se encarga de este procesamiento de los datos se llama bloque de control. Las funciones principales de este bloque son: leer los bits almacenados en los registros de desplazamiento y a continuación darle un formato interpretable para un usuario que desee conocer que información está viajando por las líneas MOSI y MISO. Luego de darle este formato, inmediatamente se lo envía al ordenador para que pueda ser leído en pantalla. Todas estas funciones fueron implementadas por medio del microcontrolador STM32f103c8t6, o popularmente conocido como “Blue Pill”, el cual se muestra en la **figura 38**. Los detalles técnicos de este microcontrolador se obviarán en este informe, pero a modo de mencionar algunas características sobresalientes, podemos decir que el Blue Pill nos permite trabajar a una frecuencia de reloj máxima de 62 – 74 MHz (la velocidad puede depender según el uso que se le quiera dar al microcontrolador), lo cual es favorable para un rápido procesamiento de la información capturada. También, posee la cantidad casi justa de pines GPIO necesarios para leer todos los bits de información.

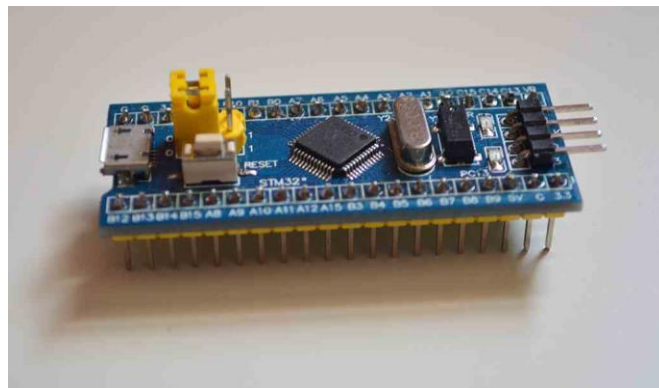


Figura 38: Microcontrolador STM32f103c8t6

Por medio de programación en lenguaje C y algunas funciones específicas del micro STM32, las funciones requeridas por el bloque de control pueden ser realizadas fácilmente.

2.2.1 – Conexionado del microcontrolador

El Blue Pill es un microcontrolador de la empresa STMicroelectronics, cuyo lenguaje de programación se basa en el lenguaje C. Al igual que por ejemplo Arduino, el Blue Pill cuenta con funciones propias de su entorno que sirven para acceder al hardware del dispositivo de una manera más sencilla y directa. Como dijimos anteriormente, el bloque de control, y por lo tanto el microcontrolador, se debe encargar de:

- Leer los bits en los registros de desplazamiento.
- Darle formato a la información.
- Enviarla al ordenador mediante UART.

Las funciones mencionadas arriba ocurren una detrás de la otra, por lo que el sistema trata de capturar un dato y enviarlo lo más rápido posible al computador, a modo de estar listo para recibir el siguiente dato.

Como los registros de desplazamiento cuentan con 8 bits cada uno, al STM32 deberán ingresar 16 bits, por lo que tendremos que destinar 16 GPIO pins para la lectura de estos bits. En adición, 1 GPIO pin deberá ser destinado a detectar mediante una interrupción externa el momento en el que se terminó de enviar un dato, tanto por la línea MOSI como la MISO. La finalización del envío de un dato/mensaje se puede detectar mediante el cambio de estado de la línea CS. Cuando CS pasa de bajo a alto, significa que una transferencia de mensajes entre el esclavo y el maestro ocurrió y terminó, por lo que por lógica encontraremos almacenados en los registros de desplazamiento esos bits que viajaron en las líneas MISO y MOSI. En el próximo mensaje, CS pasa nuevamente a estado bajo y así se repite el proceso para cada mensaje. Esto quiere decir que mediante el disparo de una interrupción cuando ocurre esta conmutación entre bajo y alto de la línea CS podremos detectar y capturar correctamente los mensajes.

También debemos considerar 2 pines más para utilizar una UART (un pin para Rx y otro para Tx), y enviar los mensajes capturados al ordenador.

En total, tal como muestra el conexionado de la **figura 39**, se requerirán utilizar 19 GPIO pins del micro.

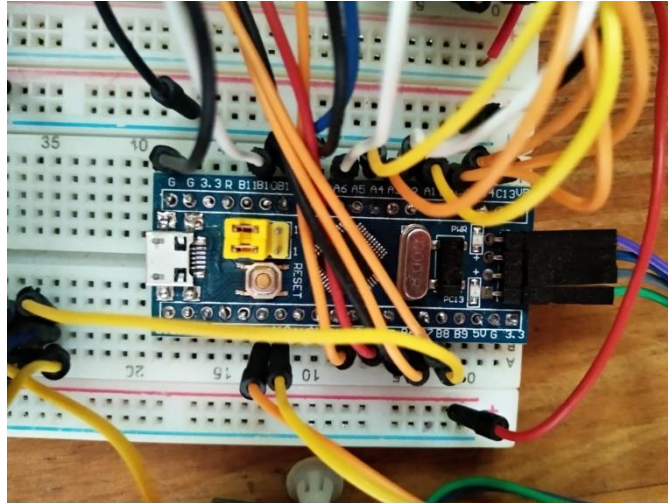


Figura 39: Pines del STM32 utilizados

Los cables que se ven salientes a la derecha en la **figura 39** son para programar y debugear el dispositivo.

2.2.2 – Programación del microcontrolador

Explicada la forma en que se conectan a los pines del micro los bites de información, señal de control CS y pines Rx y Tx de la UART, pasemos ahora a detallar el código realizado.

Comenzamos analizando la rutina más importante del código, la cual se muestra a continuación:

```

1 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
2 {
3     //Almacenamos los bits correspondientes a la linea MOSI
4     MOSI_Buffer[0] = HAL_GPIO_ReadPin(MOSI_MSB_GPIO_Port, MOSI_MSB_Pin);
5     MOSI_Buffer[1] = HAL_GPIO_ReadPin(MOSI_6_GPIO_Port, MOSI_6_Pin);
6     MOSI_Buffer[2] = HAL_GPIO_ReadPin(MOSI_5_GPIO_Port, MOSI_5_Pin);
7     MOSI_Buffer[3] = HAL_GPIO_ReadPin(MOSI_4_GPIO_Port, MOSI_4_Pin);
8     MOSI_Buffer[4] = HAL_GPIO_ReadPin(MOSI_3_GPIO_Port, MOSI_3_Pin);
9     MOSI_Buffer[5] = HAL_GPIO_ReadPin(MOSI_2_GPIO_Port, MOSI_2_Pin);
10    MOSI_Buffer[6] = HAL_GPIO_ReadPin(MOSI_1_GPIO_Port, MOSI_1_Pin);
11    MOSI_Buffer[7] = HAL_GPIO_ReadPin(MOSI_LSB_GPIO_Port, MOSI_LSB_Pin);
12    //Repetimos para la linea MISO
13    MISO_Buffer[0] = HAL_GPIO_ReadPin(MISO_MSB_GPIO_Port, MISO_MSB_Pin);
14    MISO_Buffer[1] = HAL_GPIO_ReadPin(MISO_6_GPIO_Port, MISO_6_Pin);
15    MISO_Buffer[2] = HAL_GPIO_ReadPin(MISO_5_GPIO_Port, MISO_5_Pin);
16    MISO_Buffer[3] = HAL_GPIO_ReadPin(MISO_4_GPIO_Port, MISO_4_Pin);
17    MISO_Buffer[4] = HAL_GPIO_ReadPin(MISO_3_GPIO_Port, MISO_3_Pin);
18    MISO_Buffer[5] = HAL_GPIO_ReadPin(MISO_2_GPIO_Port, MISO_2_Pin);
19    MISO_Buffer[6] = HAL_GPIO_ReadPin(MISO_1_GPIO_Port, MISO_1_Pin);
20    MISO_Buffer[7] = HAL_GPIO_ReadPin(MISO_LSB_GPIO_Port, MISO_LSB_Pin);
21    //Cambiamos el estado del flag buffers_listos
22    buffers_listos = true;

```

23 }

La función o más precisamente el callback **HAL_GPIO_EXTI** es una rutina que se ejecutara cada vez que se detecte el disparo de la interrupción externa. Es decir, cuando la línea CS indique que hay un mensaje para ser capturado en los registros de desplazamiento. La función es muy simple y se basa en leer todos los pines relacionados a los registros para la línea MISO y MOSI, guardando los bits leídos en vectores de 8 elementos llamados MISO_Buffer y MOSI_Buffer respectivamente. Antes de salir de la rutina, se cambia el estado de una bandera para habilitar la ejecución del código ubicado en el bucle principal del programa. Dicho código se muestra a continuación.

```
while (1)
1  {
2      if(buffers_listos){
3          //Borramos contenido anterior del Mensaje
4          memset(Mensaje,0,sizeof(Mensaje));
5          //Colocamos "M" (0x4d) como identificador de cabecera del
6  mensaje
7          Mensaje[0] = Mensaje[0] | 0x4d;
8          //Concatenamos en aux los bits capturados por el registro de
9  desplazamiento
10         for(i=0;i<8;i++){
11             aux = aux <<1 | MOSI_Buffer[i];
12         }
13         Mensaje[1] = Mensaje[1] | aux;
14         //Colocamos "S" (0x53) como identificador de cabecera
15         Mensaje[2] = Mensaje[2] | 0x53;
16         for(i=0;i<8;i++){
17             aux = aux <<1 | MISO_Buffer[i];
18         }
19         Mensaje[3] = Mensaje[3] | aux;
20         Mensaje[4] = Mensaje[4] | '\n';
21         Mensaje[5] = Mensaje[5] | '\r';
22         //Enviamos el mensaje por el puerto UART1
23         HAL_UART_Transmit(&huart1,Mensaje,6,HAL_MAX_DELAY);
24         buffers_listos = false;
25     }
26
27     /* USER CODE END WHILE */
28
29     /* USER CODE BEGIN 3 */
}
```

Si la bandera `buffer_listos` lo habilita, el loop principal del programa se encargara de procesar el mensaje de la línea MOSI y el mensaje de la línea MISO, dándoles a ambos un formato de fácil entendimiento para un usuario cualquiera. Para esto, se procedió a utilizar una variable llamada `Mensaje`, el cual es un vector de 6 elementos tipo `uint8_t`. La idea es que el usuario vea en una misma línea de la pantalla, el mensaje capturado en la línea MOSI y el mensaje capturado en la línea MISO, separando los mismos mediante un identificador de cabecera. Para el mensaje MOSI el identificador será la letra mayúscula

M, indicando que proviene del maestro, y en el caso del mensaje MISO el identificador será la letra mayúscula S, indicando que proviene del esclavo. Finalmente se agregan los identificadores \n y \r para indicar el salto de línea y el retorno de carretel. Para lograr todo esto, se utiliza el vector Mensaje[] para almacenar ordenadamente los caracteres a enviar por el puerto UART.

Los bits de las líneas MISO y MOSI se encuentran en los vectores MISO_Buffer[] y MOSI_Buffer[] respectivamente, pero aquí los ocho bites que conforman el mensaje se encuentran divididos en 8 elementos de vector. Es necesario entonces concatenar estos bites a modo de formar el mensaje. Esto se logra mediante un ciclo for y una variable auxiliar llamada aux, en la cual se ira concatenando el contenido de los vectores para luego guardar a aux en una posición del vector Mensaje[].

Con todo este proceso, finalmente en Mensaje[] tendremos los mensajes capturados en las líneas MOSI y MISO, con sus respectivos identificadores de cabecera e identificadores de control. Por último, se procede a enviar el vector Mensaje[] por el puerto UART y cambiamos el estado de la bandera.

Todo este código en conjunto con la rutina de la interrupción externa conforman el funcionamiento completo del bloque de control. En el programa que será anexado junto con los demás archivos del proyecto se encontrara más secciones de código que no están explicadas aquí, pero las mismas pertenecen puramente a configuraciones del microcontrolador. Estas configuraciones se generan automáticamente en base a lo que el usuario indico en la configuración gráfica del microcontrolador y por lo tanto no es fundamental explicarlas.

2.2.3 – Conexión del microcontrolador al ordenador

El envío de los mensajes capturados al ordenador, en este caso particular una PC, se realizó mediante un módulo conversor UART a USB CP2102, tal como muestra la **figura 40**



Figura 40 : Módulo USB – UART CP2102

La utilización de este módulo fue necesaria ya que, a diferencia de algunas placas Arduino, el STM32f103c8t6 no cuenta con la posibilidad de conectar su UART directamente al puerto USB de la PC.

Por otro lado, para cargar el código escrito en el entorno de programación CubeIDE al microcontrolador es necesario utilizar el dispositivo ST-LINK V2. El mismo, además de cargar el código, permite realizar un debugeo en tiempo real del microcontrolador, lo cual resulta muy provechoso al momento de realizar pruebas de funcionamiento. La **figura 41** muestra al ST-LINK V2 utilizado.



Figura 41: ST-LINK V2

3 – Banco de pruebas SPI

Diseñado e implementado el analizador de protocolo SPI, es necesario ahora determinar la comunicación SPI entre dos dispositivos que se utilizara para probar el analizador. Para esto se procedió a hacer uso de dos microcontroladores Arduino UNO, en donde uno ofrece de maestro y el otro de esclavo. Se decidió utilizar estos microcontroladores principalmente porque eran los que tenían disponibles los integrantes del grupo, y además porque Arduino provee funciones para el uso del protocolo SPI que optimizan un poco su manejo, aunque se comprobó que dichas funciones cuentan con algunos problemas por parte del esclavo y la escritura de datos sobre la línea MISO. Más allá de esto, resulta ser una buena opción debido a la disponibilidad y fácil manejo.

Los microcontroladores utilizados se muestran en la **figura 42**. En la misma, el Arduino inferior ofrece de Maestro y el superior de Esclavo. El maestro se encuentra conectado a la computadora mediante su cable USB y el esclavo es alimentado por los pines GND y 5 V del maestro.

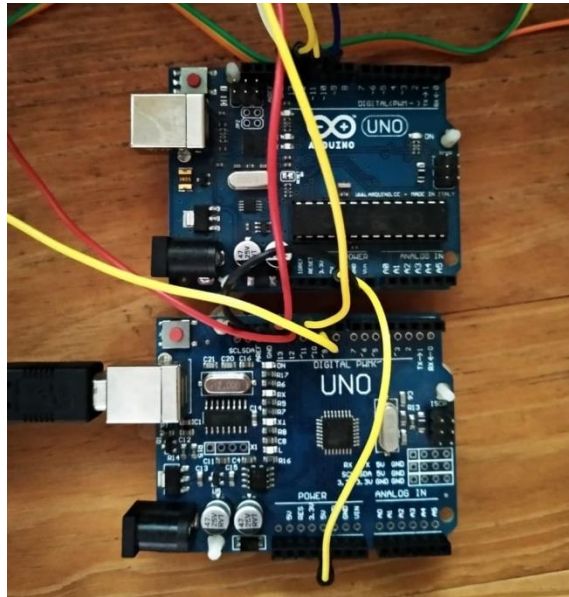


Figura 42: Banco de pruebas SPI

En las placas Arduino UNO, los pines del protocolo SPI son:

- Pin 10: Señal CS.
- Pin 11: Señal MOSI.
- Pin 12: Señal MISO.
- Pin 13: Señal SCK.

Para conectar maestro y esclavo, se conectan línea a línea las señales del protocolo SPI (CS del maestro con CS del esclavo por ejemplo).

Tanto para el maestro como para el esclavo, es necesario escribir un código. El código del maestro y del esclavo se anexaran al final del informe para su revisión si así se deseara. Aquí procederemos a explicar resumidamente que hace cada uno.

El objetivo de este banco de pruebas es realizar una comunicación SPI entre el maestro y el esclavo. El maestro le envía palabras al esclavo y el esclavo responde con otras palabras al maestro. El código del maestro nos permite escribir en la ventana del puerto serial una palabra o frase cualquiera, siempre y cuando sea menor a 100 caracteres (aunque podría ser mayor sin problema alguno), y al presionar la tecla enter dicha palabra o frase se envía carácter a carácter por la línea MOSI hacia el esclavo. El código del esclavo se encarga de escribir en la línea MISO continuamente un carácter al azar, el cual llegara al maestro si este decide leerlo. Como el maestro es el que controla la conversación, será él el que decida leer o no el carácter enviado por la línea MISO. La lectura del carácter enviado por el esclavo se realiza mediante el valor devuelto por la función **SPI.transfer**. El envío del carácter que el maestro quiere enviar al esclavo se realiza mediante el argumento de

SPI.transfer(). Por ejemplo, si hacemos `b = SPI.transfer(a)`, la variable "a" contiene el carácter que el maestro envía por la línea MOSI al esclavo, y la variable "b" almacena el carácter que el esclavo envió por la línea MISO al maestro. Sin embargo, como se había dicho anteriormente, la escritura de caracteres en la línea MISO en la placa Arduino UNO no está muy optimizada y resulta dificultoso que el esclavo envíe palabras o frases coherentes al maestro del mismo modo que él lo hace con el esclavo. Es por esto que los caracteres que envía el esclavo son aleatorios sin sentido alguno, ya que es lo que se logró implementar con eficacia.

Un punto muy importante en el código del maestro para el correcto funcionamiento del analizador de protocolo SPI es la inclusión de un delay de 600 microsegundos entre cada envío de carácter por la línea MISO y MOSI. De no estar este delay, se comprobó que el analizador de protocolo implementado solo logra capturar el último carácter que se envía por la línea, perdiendo a todos los demás. Esto resulta una gran desventaja, ya que utilizar un recurso como un delay dentro del código del maestro para que el analizador pueda capturar todos los datos no es viable en una aplicación real. Esto no sería necesario si dentro del bloque de muestreo contáramos con unas memorias FIFO físicas para el almacenado instantáneo de los bites que se encuentran en los registros de desplazamiento, pero esto no se pudo lograr debido a la dificultad para conseguir este tipo de memorias y también debido a su alto precio. Por lo tanto, el uso de un delay fue la única solución que pudimos encontrar a este problema de pérdida de datos.

4 – Resultados prácticos

Realizadas todas las conexiones necesarias, procedimos a realizar unas pruebas enviando mensajes entre el maestro y el esclavo y verificando si el analizador de protocolo logra capturar correctamente los caracteres enviados. A modo de comprobación del funcionamiento del dispositivo, los mensajes que viajan por la línea MISO y MOSI se imprimen en el puerto serial del IDE de Arduino. Si estos son los mismos que se ven en el puerto serial del IDE del STM32, efectivamente podemos decir que los mensajes se capturaron correctamente. La **figura 43** nos muestra una captura de pantalla del puerto serial en el IDE de Arduino. En la misma podemos ver que el maestro envió por la línea MOSI la frase "Medidas Electronicas II*", y el esclavo envió la palabra aleatoria "bdcdbcdcbdbcbcdcdcdcbdb". Nótese que al finalizar la frase del maestro, el mismo se envía por la línea MOSI un último carácter "*", a modo de indicar visualmente la finalización del mensaje. Esto sirve para poder ver con mayor claridad en el IDE del STM32 el mensaje capturado.



Figura 43: Mensaje enviado 1

Ahora viendo la **figura 44** , la cual corresponde a una captura de pantalla del puerto serial del IDE del STM32, vemos que se imprimieron en pantalla línea a línea los mensaje capturados tanto del maestro como del esclavo. Para leer correctamente el mensaje entero, tenemos que ver desde arriba hacia abajo los mensajes que se imprimieron. El identificador de cabecera M indica que el siguiente carácter corresponde a lo que se capturo por la línea MOSI y el identificador S indica que el siguiente carácter corresponde a lo que se capturo por la línea MISO. La **figura 45** muestra remarcado el mensaje capturado en ambas líneas.

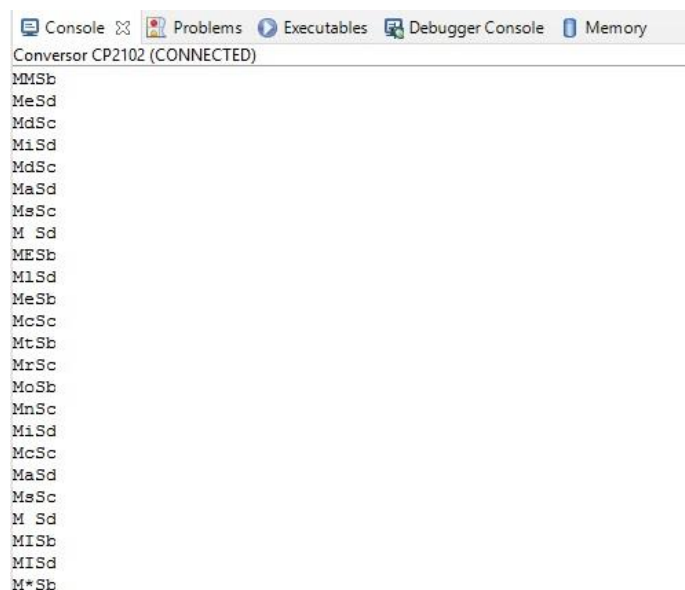


Figura 44: Puerto Serie STM32

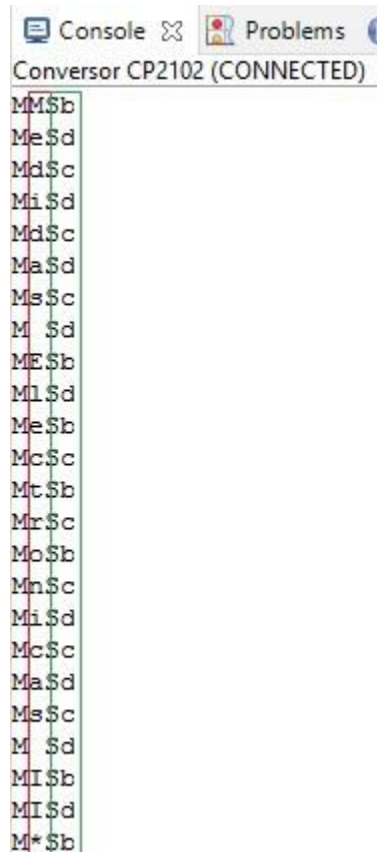


Figura 45: Mensaje capturado 1

Comparando las **figuras 45 y 43** , vemos efectivamente que los mensajes que viajaron por las líneas MISO y MOSI fueron capturados por el analizador de protocolo y enviados al ordenador correctamente. En las **figuras 46,47,48 y 49** procedemos a realizar algunas pruebas más con otras palabras/frases y vemos que el resultado es el esperado.

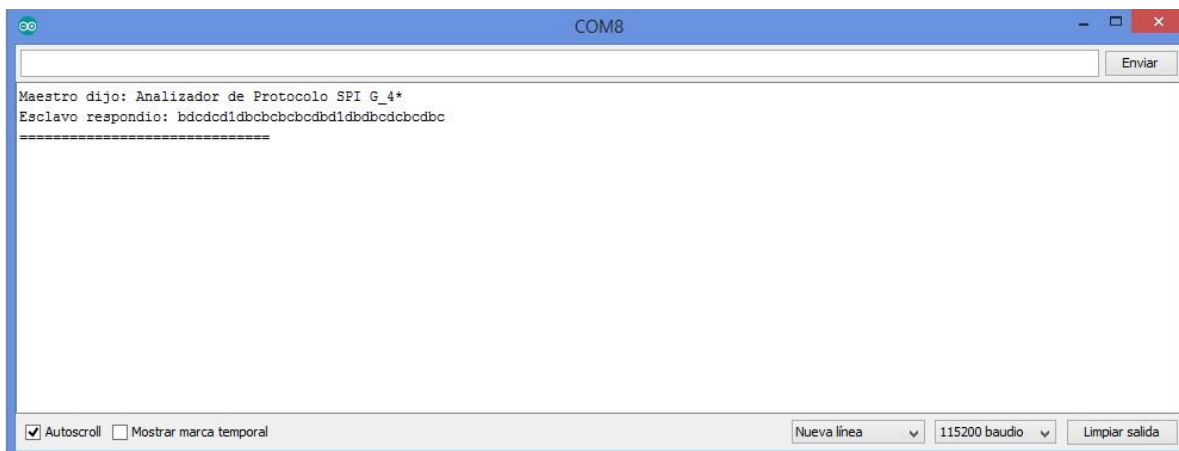


Figura 46: Mensaje enviado 2

CONVERSOR CP2102 (CONNECTED)

MASb
MnSd
MaSc
MlSd
MiSc
MzSd
MaS1
MdSd
MoSb
MrSc
M Sb
MdSc
MeSb
M Sc
MPSb
MrSc
MoSd
MtSb
MoSd
McS1
MoSd
MlSb
MoSd
M Sb
MSSc
MPSd
MISc
M Sb
MGSc
M_Sd
M4Sb
M*Sc

Figura 47: Mensaje capturado 2

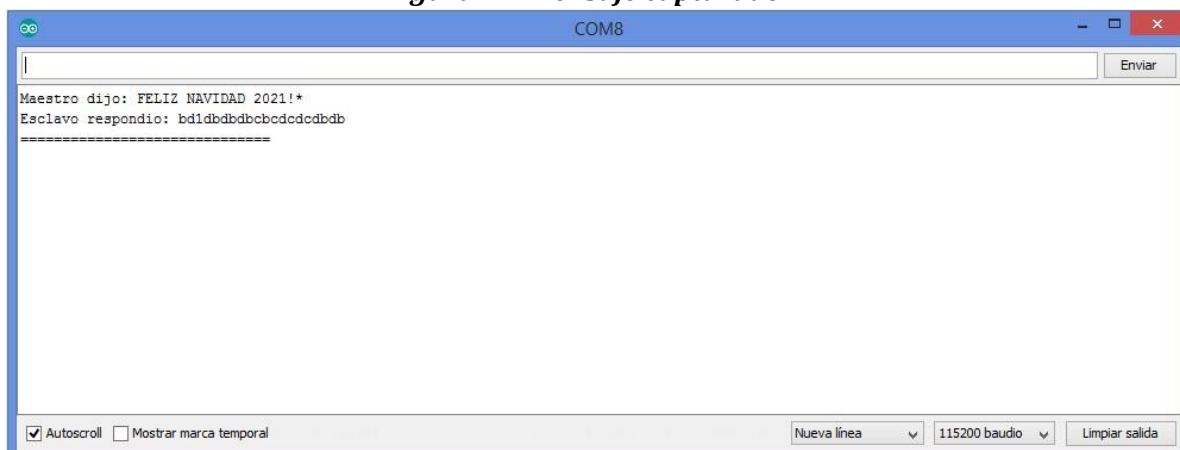
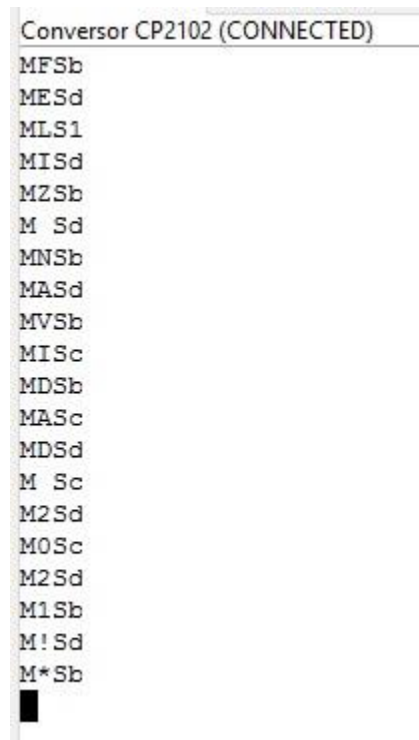


Figura 48: Mensaje enviado 3



```
Conversor CP2102 (CONNECTED)
MFSb
MESd
MLS1
MISd
MZSb
M Sd
MNSb
MASd
MVSb
MISC
MDSb
MASc
MDSd
M Sc
M2Sd
MOSc
M2Sd
M1Sb
M!Sd
M*Sb
█
```

Figura 49: Mensaje capturado 3

Con estas pruebas queda comprobado que el analizador de protocolo captura y envía al ordenador correctamente los mensajes de las líneas MISO y MOSI.

5 – Presupuesto

Teniendo en cuenta todos los componentes necesarios para implementar el analizador de protocolo y la mano de obra requerida, el presupuesto final para el proyecto da un total de \$15.780 pesos argentinos. El detalle del presupuesto se encuentra en un archivo Excel anexo junto con los demás archivos del proyecto.

6 – Conclusión

A lo largo del desarrollo del proyecto, distintas dificultades se fueron presentando para un correcto funcionamiento del analizador de protocolo SPI, tales como la obtención de los componentes necesarios o incluso el coordinar con los integrantes del grupo para poder realizar avances. Gracias a la constancia en la investigación, se pudo sortear los desafíos hasta llegar a un producto que si bien no está finalizado, ya que faltaría el diseño de una placa y también una carcasa, cumple con el objetivo para el cual fue diseñado.

Como conclusión final, podemos resaltar los siguientes puntos. Los mismos sintetizan nuestra experiencia a lo largo del desarrollo de todo el proyecto, considerando los puntos

fuertes y también aquellos que representan cuestiones a considerar al momento de pensar en utilizar este dispositivo en una situación real. Los puntos son:

- ❖ La utilización del protocolo SPI en Arduino cuenta con dificultades importantes por parte del esclavo.
- ❖ El analizador de protocolo SPI funcionará solamente con dispositivos que envíen mensajes de 8 bits de longitud.
- ❖ La velocidad de captura del dispositivo se ve limitada debido a la falta de asincronismo entre captura y envío de mensajes (ausencia de memorias FIFO).
- ❖ El dispositivo captura sin errores los mensajes enviados tanto en la línea MISO como la MOSI.
- ❖ El dispositivo es de utilidad en sistemas de comunicación SPI donde la verificación de los mensajes enviados por parte del maestro y el esclavo por las líneas MISO y MOSI es difícil o imposible de realizar.

7 – Anexo

7.1 – Código STM32

```
1 #include "main.h"
2
3 /* Private includes -----
4 */
5 /* USER CODE BEGIN Includes */
6 #include "stdbool.h"
7 /* USER CODE END Includes */
8
9 /* Private typedef -----
10 */
11 /* USER CODE BEGIN PTD */
12
13 /* USER CODE END PTD */
14
15 /* Private define -----
16 */
17 /* USER CODE BEGIN PD */
18 /* USER CODE END PD */
19
20 /* Private macro -----
21 */
22 /* USER CODE BEGIN PM */
23
24 /* USER CODE END PM */
25
26 /* Private variables -----
```

```

27 */
28 UART_HandleTypeDef huart1;
29 DMA_HandleTypeDef hdma_usart1_tx;
30
31 /* USER CODE BEGIN PV */
32
33 /* USER CODE END PV */
34
35 /* Private function prototypes -----
36 */
37 void SystemClock_Config(void);
38 static void MX_GPIO_Init(void);
39 static void MX_DMA_Init(void);
40 static void MX_USART1_UART_Init(void);
41 /* USER CODE BEGIN PFP */
42
43 /* USER CODE END PFP */
44
45 /* Private user code -----
46 */
47 /* USER CODE BEGIN 0 */
48 bool buffers_listos = false;
49 uint8_t MOSI_Buffer[8];
50 uint8_t MISO_Buffer[8];
51 uint8_t aux=0xFF;
52 uint8_t Mensaje[6];
53 int i;
54
55 /* USER CODE END 0 */
56
57 /**
58  * @brief The application entry point.
59  * @retval int
60  */
61 int main(void)
62 {
63     /* USER CODE BEGIN 1 */
64
65     /* USER CODE END 1 */
66
67     /* MCU Configuration-----
68     */
69
70     /* Reset of all peripherals, Initializes the Flash interface and the
71     SysTick. */
72     HAL_Init();
73
74     /* USER CODE BEGIN Init */
75
76     /* USER CODE END Init */
77
78     /* Configure the system clock */

```



```

79  SystemClock_Config();
80
81  /* USER CODE BEGIN SysInit */
82
83  /* USER CODE END SysInit */
84
85  /* Initialize all configured peripherals */
86  MX_GPIO_Init();
87  MX_DMA_Init();
88  MX_USART1_UART_Init();
89  /* USER CODE BEGIN 2 */
90
91  /* USER CODE END 2 */
92
93  /* Infinite loop */
94  /* USER CODE BEGIN WHILE */
95  while (1)
96  {
97      if(buffers_listos){
98          //Borramos contenido anterior del Mensaje
99          memset(Mensaje,0,sizeof(Mensaje));
100          //Colocamos "M" (0x4d) como identificador de cabecera del
101 mensaje
102          Mensaje[0] = Mensaje[0] | 0x4d;
103          //Concatenamos en aux los bits capturados por el registro
104 de desplazamiento
105          for(i=0;i<8;i++){
106              aux = aux <<1 | MOSI_Buffer[i];
107          }
108          Mensaje[1] = Mensaje[1] | aux;
109          //Colocamos "S" (0x53) como identificador de cabecera
110 Mensaje[2] = Mensaje[2] | 0x53;
111          for(i=0;i<8;i++){
112              aux = aux <<1 | MISO_Buffer[i];
113          }
114          Mensaje[3] = Mensaje[3] | aux;
115          Mensaje[4] = Mensaje[4] | '\n';
116          Mensaje[5] = Mensaje[5] | '\r';
117          //Enviamos el mensaje por el puerto UART1
118          HAL_UART_Transmit(&huart1,Mensaje,6,HAL_MAX_DELAY);
119          buffers_listos = false;
120      }
121
122      /* USER CODE END WHILE */
123
124      /* USER CODE BEGIN 3 */
125  }
126  /* USER CODE END 3 */
127 }
128
129 /**
130  * @brief System Clock Configuration

```

```

131  * @retval None
132  */
133 void SystemClock_Config(void)
134 {
135     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
136     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
137
138     /** Initializes the RCC Oscillators according to the specified parameters
139     * in the RCC_OscInitTypeDef structure.
140     */
141     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
142     RCC_OscInitStruct.HSISTate = RCC_HSI_ON;
143     RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
144     RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
145     RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI_DIV2;
146     RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL16;
147     if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
148     {
149         Error_Handler();
150     }
151     /** Initializes the CPU, AHB and APB buses clocks
152     */
153     RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
154                                   |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
155     RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
156     RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
157     RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
158     RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
159
160     if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
161     {
162         Error_Handler();
163     }
164 }
165
166 /**
167  * @brief USART1 Initialization Function
168  * @param None
169  * @retval None
170  */
171 static void MX_USART1_UART_Init(void)
172 {
173
174     /* USER CODE BEGIN USART1_Init 0 */
175
176     /* USER CODE END USART1_Init 0 */
177
178     /* USER CODE BEGIN USART1_Init 1 */
179
180     /* USER CODE END USART1_Init 1 */
181     huart1.Instance = USART1;
182     huart1.Init.BaudRate = 115200;

```

```

183  huart1.Init.WordLength = UART_WORDLENGTH_8B;
184  huart1.Init.StopBits = UART_STOPBITS_1;
185  huart1.Init.Parity = UART_PARITY_NONE;
186  huart1.Init.Mode = UART_MODE_TX;
187  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
188  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
189  if (HAL_UART_Init(&huart1) != HAL_OK)
190  {
191      Error_Handler();
192  }
193  /* USER CODE BEGIN USART1_Init 2 */
194
195  /* USER CODE END USART1_Init 2 */
196
197 }
198
199 /**
200  * Enable DMA controller clock
201  */
202 static void MX_DMA_Init(void)
203 {
204
205     /* DMA controller clock enable */
206     __HAL_RCC_DMA1_CLK_ENABLE();
207
208     /* DMA interrupt init */
209     /* DMA1_Channel4_IRQn interrupt configuration */
210     HAL_NVIC_SetPriority(DMA1_Channel4_IRQn, 0, 0);
211     HAL_NVIC_EnableIRQ(DMA1_Channel4_IRQn);
212
213 }
214
215 /**
216  * @brief GPIO Initialization Function
217  * @param None
218  * @retval None
219  */
220 static void MX_GPIO_Init(void)
221 {
222     GPIO_InitTypeDef GPIO_InitStruct = {0};
223
224     /* GPIO Ports Clock Enable */
225     __HAL_RCC_GPIOC_CLK_ENABLE();
226     __HAL_RCC_GPIOA_CLK_ENABLE();
227     __HAL_RCC_GPIOB_CLK_ENABLE();
228
229     /*Configure GPIO pin : MOSI_2_Pin */
230     GPIO_InitStruct.Pin = MOSI_2_Pin;
231     GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
232     GPIO_InitStruct.Pull = GPIO_PULLUP;
233     HAL_GPIO_Init(MOSI_2_GPIO_Port, &GPIO_InitStruct);
234

```

```

235  /*Configure GPIO pin : MOSI_LSB_Pin */
236  GPIO_InitStruct.Pin = MOSI_LSB_Pin;
237  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
238  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
239  HAL_GPIO_Init(MOSI_LSB_GPIO_Port, &GPIO_InitStruct);
240
241  /*Configure GPIO pins : MOSI_MSB_Pin MOSI_6_Pin MOSI_5_Pin MOSI_4_Pin
242                        MOSI_3_Pin MOSI_1_Pin */
243  GPIO_InitStruct.Pin = MOSI_MSB_Pin|MOSI_6_Pin|MOSI_5_Pin|MOSI_4_Pin
244                        |MOSI_3_Pin|MOSI_1_Pin;
245  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
246  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
247  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
248
249  /*Configure GPIO pins : MISO_MSB_Pin MISO_6_Pin MISO_LSB_Pin MISO_5_Pin
250                        MISO_4_Pin MISO_3_Pin MISO_2_Pin MISO_1_Pin */
251  GPIO_InitStruct.Pin = MISO_MSB_Pin|MISO_6_Pin|MISO_LSB_Pin|MISO_5_Pin
252                        |MISO_4_Pin|MISO_3_Pin|MISO_2_Pin|MISO_1_Pin;
253  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
254  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
255  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
256
257  /*Configure GPIO pin : CS_Pin */
258  GPIO_InitStruct.Pin = CS_Pin;
259  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
260  GPIO_InitStruct.Pull = GPIO_PULLUP;
261  HAL_GPIO_Init(CS_GPIO_Port, &GPIO_InitStruct);
262
263  /* EXTI interrupt init*/
264  HAL_NVIC_SetPriority(EXTI9_5_IRQn, 0, 0);
265  HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
266
267 }
268
269 /* USER CODE BEGIN 4 */
270 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
271 {
272     //Almacenamos los bits correspondientes a la linea MOSI
273     MOSI_Buffer[0] = HAL_GPIO_ReadPin(MOSI_MSB_GPIO_Port, MOSI_MSB_Pin);
274     MOSI_Buffer[1] = HAL_GPIO_ReadPin(MOSI_6_GPIO_Port, MOSI_6_Pin);
275     MOSI_Buffer[2] = HAL_GPIO_ReadPin(MOSI_5_GPIO_Port, MOSI_5_Pin);
276     MOSI_Buffer[3] = HAL_GPIO_ReadPin(MOSI_4_GPIO_Port, MOSI_4_Pin);
277     MOSI_Buffer[4] = HAL_GPIO_ReadPin(MOSI_3_GPIO_Port, MOSI_3_Pin);
278     MOSI_Buffer[5] = HAL_GPIO_ReadPin(MOSI_2_GPIO_Port, MOSI_2_Pin);
279     MOSI_Buffer[6] = HAL_GPIO_ReadPin(MOSI_1_GPIO_Port, MOSI_1_Pin);
280     MOSI_Buffer[7] = HAL_GPIO_ReadPin(MOSI_LSB_GPIO_Port, MOSI_LSB_Pin);
281     //Repetimos para la linea MISO
282     MISO_Buffer[0] = HAL_GPIO_ReadPin(MISO_MSB_GPIO_Port, MISO_MSB_Pin);
283     MISO_Buffer[1] = HAL_GPIO_ReadPin(MISO_6_GPIO_Port, MISO_6_Pin);
284     MISO_Buffer[2] = HAL_GPIO_ReadPin(MISO_5_GPIO_Port, MISO_5_Pin);
285     MISO_Buffer[3] = HAL_GPIO_ReadPin(MISO_4_GPIO_Port, MISO_4_Pin);
286     MISO_Buffer[4] = HAL_GPIO_ReadPin(MISO_3_GPIO_Port, MISO_3_Pin);

```

```

287     MISO_Buffer[5] = HAL_GPIO_ReadPin(MISO_2_GPIO_Port, MISO_2_Pin);
288     MISO_Buffer[6] = HAL_GPIO_ReadPin(MISO_1_GPIO_Port, MISO_1_Pin);
289     MISO_Buffer[7] = HAL_GPIO_ReadPin(MISO_LSB_GPIO_Port, MISO_LSB_Pin);
290     //Cambiamos el estado del flag buffers_listos
291     buffers_listos = true;
292 }
293 /* USER CODE END 4 */
294
295 /**
296  * @brief This function is executed in case of error occurrence.
297  * @retval None
298  */
299 void Error_Handler(void)
300 {
301     /* USER CODE BEGIN Error_Handler_Debug */
302     /* User can add his own implementation to report the HAL error return state
303     */
304     __disable_irq();
305     while (1)
306     {
307     }
308     /* USER CODE END Error_Handler_Debug */
309 }
310
311 #ifndef USE_FULL_ASSERT
312 /**
313  * @brief Reports the name of the source file and the source line number
314  * where the assert_param error has occurred.
315  * @param file: pointer to the source file name
316  * @param line: assert_param error line source number
317  * @retval None
318  */
319 void assert_failed(uint8_t *file, uint32_t line)
320 {
321     /* USER CODE BEGIN 6 */
322     /* User can add his own implementation to report the file name and line
323     number,
324     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line)
325     */
326     /* USER CODE END 6 */
327 }
328 #endif /* USE_FULL_ASSERT */
329
330 /***** (C) COPYRIGHT STMicroelectronics *****/
331 FILE****/

```

7.2 – Código Maestro Arduino

```
1  #include <SPI.h>
2
3  bool flag=false;
4  char C;
5  char buff[100];
6  char buff2[100];
7  int pos=0;
8  int i=0;
9
10 void setup (void)
11 {
12
13   digitalWrite(SS, HIGH);
14   SPI.begin ();
15   Serial.begin(115200);
16   SPI.setClockDivider(SPI_CLOCK_DIV32);
17
18 }
19
20
21 void loop (void)
22 {
23   String readString;
24   String Q;
25   //Bucle while para leer los mensajes escritos en el puerto serial del Arduino
26   while(Serial.available()){
27     delay(1);
28     if(Serial.available()>0){
29       C = Serial.read();
30       if(isControl(C)){
31         readString += C;
32         Q = readString;
33         flag = true;
34         break;
35       }
36       readString += C;
37     }
38   }
39   char c;
40   byte p = 0;
41   if(flag){
42     Q.toCharArray(buff,100);
43     Serial.print("Maestro dijo: ");
44     //Bucle for para enviar por la linea MOSI el mensaje escrito en el puerto
45     Serial
46     for ( const char *p = buff; c = *p; p++){
47       if(!isControl(c)){
48         digitalWrite(SS, LOW);
49         //La variable c es el byte enviado al esclavo por la linea MOSI y buff[i]
50         almacena lo que el esclavo envia al maestro por la linea MISO
51         buff2[i] = SPI.transfer(c);
```

```

52     digitalWrite(SS, HIGH);
53     //Delay importante para darle tiempo al analizador de protocolos de
54 procesar el mensaje capturado
55     delayMicroseconds(600);
56     //Imprimimos en el puerto serie el caracter enviado por la linea MOSI.
57     Serial.print(c);
58 }
59 else{
60     digitalWrite(SS, LOW);
61     //El ultimo byte que envia el maestro es el caracter *
62     buff2[i] = SPI.transfer('*');
63     digitalWrite(SS, HIGH);
64     delayMicroseconds(600);
65     Serial.print("*");
66     Serial.print(c);
67     Serial.print("Esclavo respondio: ");
68     //Imprimimos todos los caracteres enviados por la linea MISO.
69     Serial.println (buff2);
70     Serial.println("=====");
71 }
72 i++;
73 }
74 pos = 0;
75 i=0;
76 memset(buff,0,sizeof(buff));
77 memset(buff2,0,sizeof(buff2));
78 flag = false;
79 }
80 }

```

7.3 – Código Esclavo Arduino

```

1  #include <SPI.h>
2
3  uint8_t i=0;
4  bool flag;
5  void setup (void)
6  {
7      Serial.begin (115200);
8      //Configuramos el arduino como esclavo
9      SPI.setClockDivider(SPI_CLOCK_DIV32);
10     pinMode(SS, INPUT_PULLUP);
11     pinMode(MISO, OUTPUT);
12     SPCR |= _BV(SPE);
13     SPCR |= !(_BV(MSTR));
14     SPI.attachInterrupt();
15 }
16
17 ISR (SPI_STC_vect)
18 {
19
20 }

```



```
21
22 //Colocamos en el registro SPDR un byte que cambia en base al valor de i
23 void loop (void)
24 {
25     SPDR = 'b'+i;
26     i++;
27     if(i >2){
28         i=0;
29     }
30 }
```

Bibliografía

- [1] Salvador, C. G. *Implementación de analizadores de protocolos de comunicaciones SPI, I2C*. Universidad Carlos III de Madrid. Trabajo de fin de grado. Octubre del 2014.
- [2] Microchip. *ATmega328p datasheet*. Microchip Technology Inc. 2018.
- [3] Delgado, A. M. M. *Contador ascendente 4 bits activado por flanco de subida*. Universidad de Valladolid.
- [4] Renesas. *CMOS ASYNCHRONOUS FIFO IDT7203*. Renesas Electronics Corporation. 2019.
- [5] Moreno Navas, J. C.; Toapanta Silvero, P. G. *Diseño e implementación del sistema de adquisición de datos para el analizador lógico Hewlett packard 1662CS, que permita el análisis de interfaces de pórtricos seriales y paralelos de computadoras personales*. Escuela Politécnica Nacional. Tesis de fin de grado. Quito. Enero de 2003.