PRÁCTICA 1

1. OBJETIVOS

1.1. Objetivos Académicos

El objetivo principal de esta práctica es familiarizarnos con los sistemas de desarrollo de microcontroladores actuales, entendiendo sus conceptos básicos.

1.2. Objetivos Prácticos

Los objetivos prácticos incluyen la instalación del entorno de desarrollo STM32CubeIDE y del simulador QEMU, encender un LED al presionar un botón, usar la interrupción del botón para cambiar el estado del LED y, finalmente, hacer que el LED parpadee a un ritmo determinado.

2. INTRODUCCIÓN

En este proyecto, nos adentraremos en el entorno de desarrollo STM32CubeIDE de STMicroelectronics. Este entorno, basado en Eclipse, está diseñado para facilitar la creación de aplicaciones para los microcontroladores STM32 y sus placas de desarrollo. Su robustez y versatilidad abarcan desde la configuración inicial del dispositivo hasta la depuración avanzada del código. La familiaridad con Eclipse hace que sea accesible y eficiente para muchos desarrolladores.

Además, utilizaremos el simulador QEMU (Quick Emulator), una herramienta de código abierto que será fundamental en nuestro desarrollo. QEMU puede emular diversas plataformas de hardware, incluyendo CPUs, buses, periféricos y dispositivos de almacenamiento. Esta flexibilidad nos permite ejecutar sistemas operativos, aplicaciones y firmware en un entorno virtual, lo que es valioso para el desarrollo de software, pruebas de hardware y propósitos educativos.

La combinación de STM32CubeIDE y QEMU nos permitirá desarrollar de manera eficiente y efectiva, aprovechando las características avanzadas del entorno de desarrollo y la capacidad de emulación del simulador. Este enfoque integral nos proporcionará un entorno robusto y versátil para explorar y perfeccionar nuestras aplicaciones con los microcontroladores STM32.

3. DESARROLLO DE LA PRÁCTICA

Siguiendo las instrucciones, instalamos el entorno de desarrollo STM32CubeIDE, versión 1.3.0, a través del enlace proporcionado. Es necesario registrarse para obtener el software. Durante la instalación, aceptamos todas las opciones predeterminadas. También instalamos el plugin para usar el simulador QEMU, buscando "GNU MCU" en el marketplace del programa. Finalmente, instalamos el paquete de simulación ARM Cortex siguiendo las instrucciones del enlace correspondiente.

Primero, crearemos un programa que encienda un LED al presionar un botón. Para ello, creamos un nuevo proyecto (File -> New -> STM32 Project). La primera vez puede ser lenta. Seleccionamos el dispositivo con el que trabajaremos; podemos usar una placa de evaluación de STMicroelectronics o un microcontrolador en una placa hecha por nosotros o por un fabricante. En este caso, simularemos usando NUCLEO-F103RB. Nombramos el proyecto y finalizamos la creación.

El generador de código de configuración, CubeMx, facilita la generación de código para configurar y utilizar todos los elementos del microcontrolador, desde el reloj hasta los GPIOs. Los cambios en la configuración, como modificar parámetros, requieren regenerar el código para que los cambios se reflejen en el programa. El generador

visualiza el chip con pines coloreados, donde el color indica su función. Por ejemplo, el pin PC13 se configura como entrada para un botón, y PA5 como salida para un LED. Pulsaremos en Project -> Generate Code para generar el código. El programa generado se guarda con extensión .ioc, y al abrirlo, accedemos al generador CubeMX. Para el primer programa, no modificaremos la configuración predeterminada y generaremos el código directamente. Luego, podemos editar el programa principal en el archivo main.c del explorador del proyecto.

En el main.c, observaremos muchos comentarios del tipo "/* USER CODE BEGIN... */ ... /* USER CODE END ... */". Si escribimos nuestro código fuera de estos comentarios, en si el proyecto funcionará, pero cuando le volviésemos a dar a generar código por alguna modificación (añadir pines, leds...) nuestro código se borraría.

La estructura del archivo main.c es sencilla:

-SystemClock_Config: Sirve para inicializar el reloj.

-MX_GPIO_Init: Sirve para iniciarlizar los GPIO.

-MX_USART2_UART_Init: Sirve para inicializar un puerto serie.

Para lograr la tarea propuesta, se escribirá el siguiente código dentro del while(1):

```
  while (1) {
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin,
      HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin));
  }
/* USER CODE END 3 */
```

Compilamos el proyecto y configuramos el modo debug como se indica en la práctica. Al hacer debug, pulsamos "Resume" para ejecutar el código. En la ventana abierta con la imagen de la placa, interactuamos con el botón para encender y apagar el LED, comprobando su correcto funcionamiento.

A continuación, modificamos el programa para que el botón encienda y apague el LED con un solo toque. Cambiamos a la perspectiva MX para visualizar el chip y verificar la configuración de los pines en System View y GPIO. Revisamos la configuración del controlador de interrupciones en NVIC. Buscamos las rutinas de interrupción en el archivo STM32F1xx_it.c, donde la última llama a una subrutina en STM32F1xx_hal_gpio.c.

Para implementar la tarea, usamos la función de Callback en el archivo main.c, comentando el código anterior y compilando nuevamente. La variable "estado" se declara como static para que mantenga su valor entre ejecuciones. Con estas modificaciones, al compilar y depurar, confirmamos que el LED responde correctamente.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    static char estado = 0;
    if (estado == 1) {
        HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
        estado = 0;
    } else {
        HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
        estado = 1;
    }
}
```

Finalmente, controlamos la intermitencia del LED mediante timer e interrupciones, utilizando el timer del sistema Systick. Accedemos a la función de interrupción en el archivo stm32f1xx_it.c, antes de la

función previamente mencionada. Pegamos el código entre los dos primeros comentarios, compilamos y depuramos para observar su funcionamiento.

```
static char estado=0;

if (estado == 1) {

        HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);

        estado = 0;

} else {

        HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin,

        GPIO_PIN_RESET);

        estado = 1;

}
```

Dado que esta función no usa el botón, no es necesario comentar el código anterior. Para reducir la velocidad de parpadeo, disminuimos la frecuencia del reloj a /16 según las indicaciones prácticas. Tras esta modificación, compilamos y depuramos nuevamente para confirmar la reducción efectiva en la velocidad de intermitencia.

4. PREGUNTAS

Pregunta 1: Buscar en la Wikipedia que es el Qemu y hacer un resumen en cinco líneas reflexionando sobre qué es lo que hemos montado en esta práctica.

QEMU es un emulador de procesadores basado en la traducción dinámica de binarios, que permite la virtualización en sistemas operativos como GNU/Linux y Windows. Se utiliza para ejecutar máquinas virtuales en diversas arquitecturas y microprocesadores. Licenciado bajo LGPL y GPL de GNU, QEMU emula hardware para ejecutar un sistema operativo dentro de otro sin necesidad de reparticionar el disco duro, utilizando cualquier directorio disponible. En esta práctica, hemos montado un entorno de desarrollo virtual que nos permite simular y probar nuestro código para microcontroladores STM32 sin necesidad de hardware físico, facilitando el desarrollo y la depuración.

Pregunta 2: ¿Qué podríamos haber hecho en nuestro código para que el diodo parpadeara 16 veces más lento que al principio de la fase 4 sin tocar para nada la configuración del reloj del sistema?

Para lograr eso, hemos introducido los siguientes cambios en el código:

```
static char estado = 0;


void gestionar_SysTick(void)
{
   switch (led_estado) {
      case 0:
         HAL_GPIO_WritePin(puerto_LED, pin_LED, GPIO_PIN_RESET);
         estado++;
         break;
      case 16:
         HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
         estado++;
         break;
      case 32:
         estado = 0;
         break;
      default:
```

```
        estado++;
        break;
    }
    HAL_IncTick();
}
```

Pregunta 3: La intermitencia del LED se realiza con precisión, pero con valores un poco al azar. Calcule con los parámetros que se han usado cuánto tiempo está encendido el LED y cuánto tiempo está apagado. Y ahora, al contrario, busca una combinación de la configuración del reloj HCLK (con el CubeMX) y el valor de recarga del SYSTICK que haga que el Led esté con una intermitencia de 0,5 segundos encendido y 0,5 segundos apagado. En la figura del esquema del timer Systick hay una expresión en la que se muestra el cálculo del periodo de interrupción

Consideramos un registro de 24 bits con 2^24 posibilidades que se decrementa con cada ciclo de reloj hasta llegar a 0, se activa la interrupción y enciende el LED. Teniendo una frecuencia de 64MHz, el tiempo para esta activación se calcula como (1 / (64*10^6)) * 2^24 = 0.262144. Para ajustar la interrupción cada 0.5 segundos, examinamos los valores de frecuencia para HCLK en el archivo de configuración ".ioc". Al cambiar el valor del AHB Prescaler, que divide la frecuencia del reloj HCLK, se prueba cada valor hasta encontrar el adecuado para que la interrupción ocurra cada 0.5 segundos, representado por la ecuación 0.5 = (x+1) * (1 /

(32*10^6)), donde una frecuencia de 32MHz y una recarga de 15999999 son válidos en el rango de 2^24.

5.CONCLUSIONES

Con esta práctica, he aprendido conocimientos fundamentales sobre el microcontrolador que emplearemos, así como conceptos básicos del entorno de desarrollo asociado. El entorno me parece bastante cómodo a la hora de realizar tareas en él . El simulador me parece fácil de comprender, utilizar e intuitivo al ahora de navegar por él, aunque de cara a futuras prácticas, creo que la interacción con la placa física será mucho más interesante, compleja y “divertida”.