Escuela Técnica Superior de Ingeniería Informática

Ingeniería Informática. Ingeniería de Computadores.

MEMORIA PRÁCTICA I

Asignatura: Sistemas Empotrados y de Tiempo Real I
Profesor: Gabriel Jiménez Moreno

Alumno: Álvaro José Gullón Vega

Índice

1. Objetivos

En esta primera práctica distinguimos dos objetivos:

1.1. Académico

Como objetivo académico, la práctica nos introduce al sector de la programación de microcontroladores a través de la instalación y posterior uso de el entorno de desarrollo STM32CubeIDE, que se puede complementar con el simulador QEMU para realizar proyectos en los que no haya posibilidad de adquirir el hardware a usar.

1.2. Práctico

En cuanto al objetivo práctico, la práctica se fundamenta en dar una explicación sobre el entorno para ir familiarizándonos con él junto a unas instrucciones detalladas para la correcta instalación de este junto al simulador a usar, incluyendo nociones básicas sobre configuraciones del proyecto y cómo debemos ejecutarlo.

2. Introducción

En esta práctica por motivos de calendario, se ha hecho de forma no presencial sin ninguna explicación de profesor (quitando el documento proporcionado). Como hemos mencionado anteriormente, esta práctica trata sobre la introducción al sistema de desarrollo STM32CubeIDE, que será el entorno de desarrollo que usaremos durante todo el curso, junto a la instalación de un simulador llamado QEMU para la implementación de aplicaciones cuando no tenemos recursos hardware disponibles.

La placa con la que vamos a probar el sistema es la STM32 Cortex M0, pese a que en el resto de las prácticas vamos a usar una STM32 Cortex M4 con gran cantidad de periféricos. Como parte esencial de esta y de todas las prácticas, usaremos STM32CubeIDE, que está basado en Eclipse, para programar nuestros microcontroladores.

Vemos diferenciadas varias fases o etapas a lo largo de la práctica que nos indican el desarrollo que debemos llevar. En la primera fase tenemos que llevar a cabo la instalación del entorno de desarrollo STM32CubeIDE. Es una mezcla de varias herramientas de desarrollo que han ido evolucionando e integrándose en un solo entorno, SW4STM32 + Atollic TrueStudio + STM32CubeMX, los primeros eran entornos de desarrollo IDE y el último un generador automático de código de configuración de los microcontroladores STM. Particularmente, he usado la última versión disponible en la página web del fabricante, la versión 1.17.0.

Tras esto, nos disponemos a comenzar con la creación del proyecto y posteriormente las fases contenidas en la práctica recogidas en el documento a descargar de la Enseñanza Virtual.

3. Desarrollo de la práctica

3.1. Fase 1: Instalación del sistema de desarrollo STM32CubeIDE y del simulador QEMU

En esta primera fase, el objetivo es la correcta instalación del entorno de desarollo. En primer lugar, debemos visitar la página web de STMicroelectronics y descargarnos el software, en mi caso he descargado la versión 1.17.0. La instalación de este es muy sencillo, pues basta con abrir el ejecutable descargado y darle a siguiente hasta acabar el proceso.

Una vez instalado, lo abrimos y procedemos a instalar el simulador QEMU para, valga la redundancia, simular el hardware de esta práctica. Para instalarlo, nos vamos al menú superior y pulsamos en: *Help* → *Eclipse Marketplace* y dentro de esta venta buscamos *GNU MCU Eclipse* y el resultado es lo que debemos instalar. Por último, debemos ir al enlace de GitHub del enunciado de la práctica donde están los paquete de simulación para los ARM Cortex e instalarlos en nuestro equipo. Con estos pasos ya estaría todo preparado para programar.

3.2. Fase 2: Encender un LED

Tras proceder con la correcta instalación del entorno de desarrollo y los ajustes necesarios para el funcionamiento de este, se nos propone probar su funcionamiento desarrollando un simple programa que nos permite encender y apagar un LED de la placa que estamos usando para comprobar que todo funciona correctamente. En la guía nos indica cómo crear un proyecto para este caso y la placa que debemos usar, que se trata de la **NUCLEO-F103RB**. Es importante que cuando al crear el proyecto nos pregunte sobre si queremos inicializar los periféricos debemos decir que sí, de esta forma existe un determinado pin, como el del LED que queremos encender, se coloca por defecto como salida.

Una vez creado el proyecto, disponemos de una funcionalidad que nos permite generar el código de configuración de nuestro microcontrolador en base a si están los valores por defecto o si hemos modificado elementos como el reloj o los GPIO. Es importante que si tenemos una configuración y por cualquier motivo hemos tenido que modificar por ejemplo la frecuencia del reloj, debemos volver a generar el código para que las modificaciones se vean efectuadas. Para esta fase, no vamos a modificar nada de la configuración inicial, por lo que pulsamos en *Project* → *Generate Code* para generar nuestro fichero de configuración.

Procedemos a escribir el código en el fichero *main.c*. Debemos escribir el código entre los comentarios de **/*USER CODE BEGIN... *//* USER CODE END ... *** para evitar que, en caso de que se vuelva a hacer, cuando generemos el código de configuración de nuestro proyecto se pierda.

La estructura es sencilla, primero los includes y luego el código. Entonces, arriba del main, en la sección de los includes debemos escribir lo siguiente:

```
1    void SystemClock_Config(void); // sirve para inicializar el reloj
2    static void MX_GPIO_Init(void); // sirve para inicializar los GPIO
3    static void MX_USART2_UART_Init(void); // sirve para inicializar un
        puerto serie
```

Posterior a esto debemos ir a nuestra función main, donde se encuentra el while(1) tan importante y fundamental de los microcontroladores en el que debemos escribir nuestro código, pero antes debemos saber como se escribe y se lee un pin. Nosotros usaremos la librería HAL que nos proporcionan estas dos funciones:

```
1    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, 0); // pone a cero el LED
2    HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13); // devuelve el valor del boton
```

En nuestro caso, si queremos que se encienda el LED en base a si pulsamos o no el botón debemos poner la siguiente línea dentro del while(1):

```
1    HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin,
        HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin));
```

Tras esto, podemos terminar de configurar el simulador QEMU para que funcione correctamente. Para ello, vamos a *Window* → *Preferences* y aquí dentro indicamos la ruta de donde se encuentra la carpeta *bin* de los archivos xPack que nos habíamos descargado previamente. Después de esto, el último paso es dirigirnos a *Run* → *Debug Configuration* y configurar el depurador de nuestro proyecto. Una vez configurado correctamente, procedemos a darle *Debug* y nos debería salir la siguiente imagen simulando a nuestra placa:

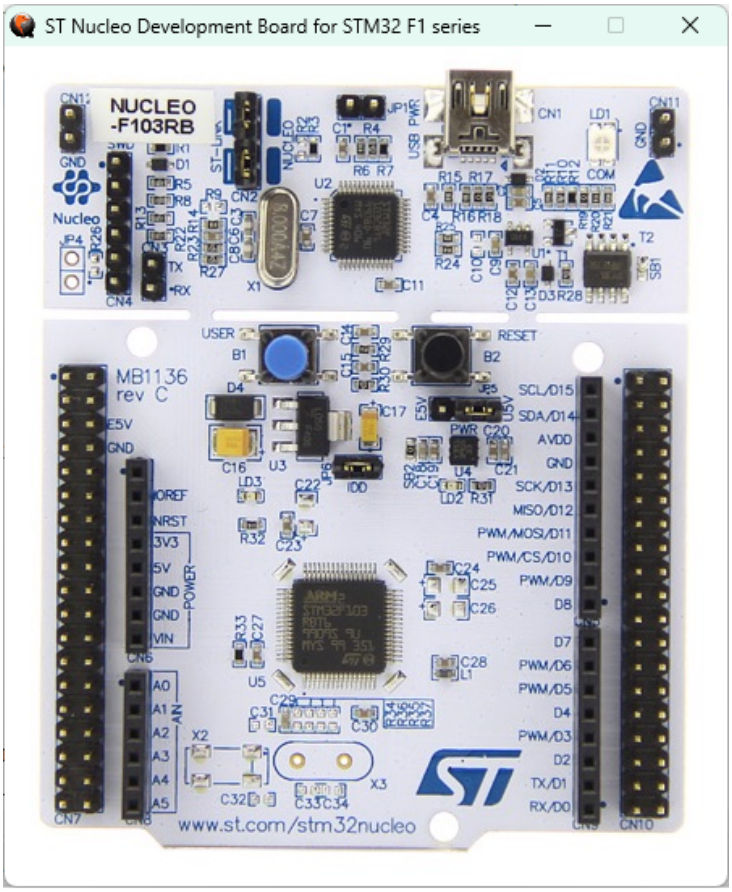

Figura 1: Inicialización de la placa seleccionada.

Si todo funciona como debe ser, si pulsamos el botón azul se debería encender el LED, tal y como lo hemos programado. El resultado sería el siguiente:

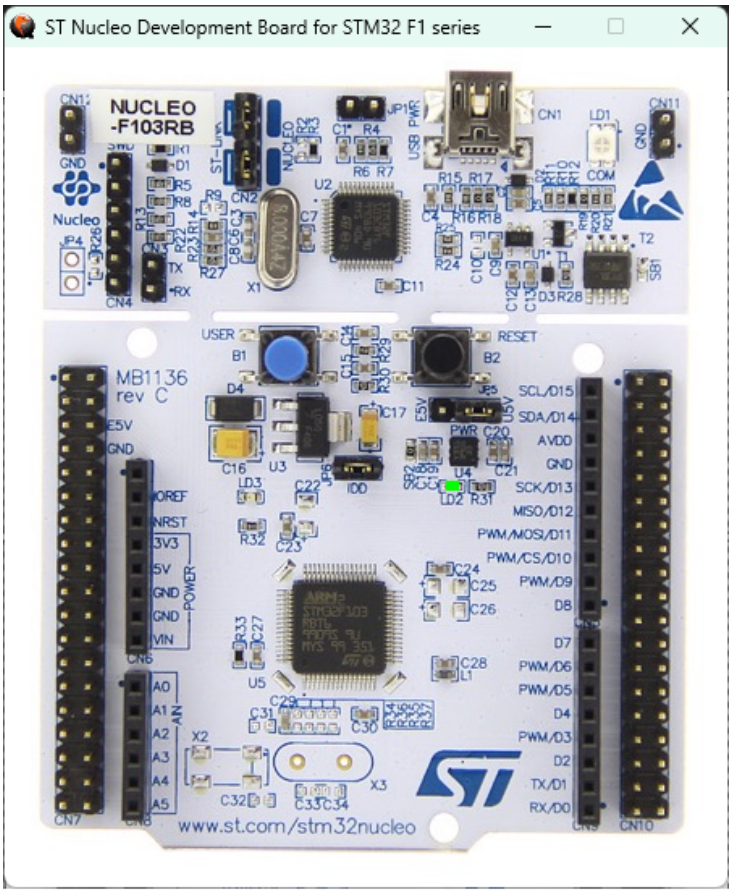

Figura 2: Resultado Fase 2.

3.3. Fase 3: Utilizar la interrupción de pulsar botón para cambiar el estado del LED

En esta fase lo que vamos a realizar es que el botón conmute el estado del LED, de forma que si este está apagado y pulsamos el botón se encienda y el caso contrario, que si está encendido y pulsamos el botón este se apague. Se podría hacer sin el uso de interrupciones, pero en este ocasión vamos a utilizarlas.

Abrimos el fichero .io de nuestro proyecto para ver cómo están configurados los pines de nuestra placa y hacer modificaciones varias. Observamos que el pin PC13 está configurado como GPIO_EXTI13. Seleccionamos la pestañana *System view* y posteriormente seleccionamos GPIO. Si seleccionamos PA5 podemos ver las caracterícticas del pin:

- **GPIO Output Level:** valor inicial del pin, si es salida
- **GPIO Mode:** en caso de salida, si es Push Pull u Open Drain
- **GPIO Pull Up/Pull Down:** si se activa o no algunas de las resistencias internas del microcontrolador correspondientes a ese pin y conectadas a VCC o a GND.
- **Maximum output speed:** si la conmutación del pin en caso de que fuera de salida es rápida o lenta.
- **User label:** es una etiqueta que se le asigna al pin y con la que podemos referirnos a él en nuestro código, se podría dejar en blanco y utilizar la denominación de los pines como vimos antes, o colocar cualquier nombre en la etiqueta.

Si seleccionamos PC13 podemos elegir además el modo de disparo de la interrupción (flanco de bajada, flanco de subida o ambos de la señal de entrada). Ajustado esto, procedemos a comprobar la configuración del controlador de interrupciones accediendo a las siguientes pestañas: *System Core* → *NVIC* → *Code Generation*. No hace falta que ajustemos nada ya que tras haber generado el código está todo correctamente, por tanto, nos dirigimos al fichero ***STM32F1xx_it.c*** en el que tenemos todas las rutinas de interrupción principales. La interrupción por un pin externo se maneja en la función *HAL_GPIO_EXTI_IRQHandler()*, que accediendo a su declaración nos encontramos con el archivo en el que debemos colocar nuestro código en la función *HAL_GPIO_EXTI_Callback*. Dentro de esta función nos indica que no hay que modificarla ni tocar nada en ella, que debemos crear una igual en el fichero *main.c*, pero tendríamos dos funciones que se llamen igual, ¿cómo sabemos a cuál salta cuando se llame? Muy sencillo. Existe una forma de declarar la función como *__weak*, que significa que si hay dos funciones que se denominan igual, el linker se decidirá por la que no es *__weak*.

Entonces, debemos colocar en el fichero *main.c* el nombre de esa función de callback:

```
1        void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
```

Debemos colocarlo entre algún BEGIN - END para evitar que se borre en la posibilidad de que se tenga que volver a usar la función de Generate Code. En nuestro caso lo hemos colocado entre estos BEGIN - END:

```
1    /* USER CODE BEGIN 4 */
2    void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
3    {
4        static char estado=0;
5        if (estado ==1)
6        {
7            HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
8            estado=0;
9        }
10       else
11       {
12           HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
13           estado=1;
14       }
15   }
16   /* USER CODE END 4 */
```

Junto a esto, debemos comentar la línea que pusimos en la fase pasada dentro del bucle while(1). Una vez completado esto podemos comprobar que funciona correctamente, por tanto, compilamos y depuramos como hemos hecho en la fase anterior.

Nota informativa

Hay una pregunta sobre por qué la variable estado se declara como *static* que la he respondido en la Sección 4.

3.4. Fase 4: Hacer que le LED se encienda y apague a un determinado ritmo

Para hacer intermitencia con el LED con un periodo bastante exacto vamos a utilizar el timer del sistema Systick, que es común a todos los Cortex M independientemente del fabricante del chip. El Systick es un temporizador de 24 bits de cuenta descendente que produce una interrupción y lo recarga del contador cuando el registro interno llega a cero desde el valor de recarga inicial.

Para realizar esta fase, debemos volver al generador de código y seleccionamos la pestaña *Clock Configuration* para configurar y ver al estructura de los diferentes relojes. El Systick, con una entrada de 64 MHz y un contador de 24 bits, permite una interrupción periódica máxima de 0,262 segundos (con valor de recarga 0xFFFFFF, es decir, poniendo todos sus 24 bits a 1). Este valor se configura con el siguiente código que debemos poner antes del while(1) tratándolo como inicializamos un periférico:

```
1         HAL_SYSTICK_Config(0xffffff);
```

Para que el LED conmute, debemos escribir lo siguiente en el fichero *stm32f1xx_it.c* en la función de interrupción:

```
1     /* USER CODE BEGIN SysTick_IRQn 0 */
2     static char estado=0;
3     if (estado ==1)
4     {
5         HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
6         estado=0;
7     }
8     else
9     {
10        HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
11        estado=1;
12    }
13    /* USER CODE END SysTick_IRQn 0 */
```

Y con esto estaría todo funcionando correctamente.

> **Nota informativa**
>
> Preguntas sobre si debemos comentar el código anterior y cómo podemos hacer que el LED parpadee más lento resueltas en la Sección 4.

4. Contestación de preguntas

Fase 3 - ¿Por qué la variable `estado` está declarada como `static`?

La palabra clave `static` indica que la variable conserva su valor entre ejecuciones del bucle, como si se inicializara una sola vez. Esto es necesario porque, si se declarara como una variable local normal, se reinicializaría en cada iteración y no podría mantener el estado anterior.

Fase 4 - ¿Es necesario que comentemos el código anterior que controla el Led con el botón azul?

No, no es necesario comentar el código anterior, puesto que el uso del botón en este caso es nulo y no afecta a la intermitencia del LED.

Fase 4 - ¿Y si quisiéramos que parpadeara más lento el LED, por ejemplo 16 veces más lento, sin cambiar nada del código?
Tendríamos que cambiar la frecuencia del reloj HCL. Cambiaríamos en el generador de código el pre-escalador AHB a /16, eso pone el HCLK a 4Mhz, generar de nuevo el código, compilar y simular.

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.
No, no es necesario comentar el código anterior, puesto que el uso del botón en este caso es nulo y no afecta a la intermitencia del LED.

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?
Debemos modificar el código escrito en el archivo *stm32f1xx.it.c* al siguiente:

```
1   /* USER CODE BEGIN SysTick_IRQn 0*/
2   static char estado = 0;
3   if (estado == 0) {
4       HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin,
            GPIO_PIN_SET);
5       estado = estado + 1;
6   } else if (estado == 16) {
7       HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_PIN,
            GPIO_PIN_RESET);
8   } else if (estado == 32) {
9       estado = 0;
10  } else {
11      estado = estado + 1;
12  }
13  /* USER CODE END SysTick_IRQn 0*/
```

El LED cambia de estado solamente cuando la variable *estado* es igual a 0 o 16. Luego de alcanzar *estado == 32*, este vuelve a 0 reiniciando el ciclo. Esto significa que el LED cambia de estado cada 16 interrupciones de SysTick.

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
Como el registro del contador posee 24 bits, tenemos 2^{24} posibles valores. La frecuencia más baja que podemos generar será $\frac{2^{24}}{6400000}$, ya que la frecuencia del reloj es de 64 MHz. Calculando:

$$\frac{2^{24}}{6400000} = 0{,}262144 \,\text{segundos estará el LED encendido/apagado}$$

Para el caso contrario, accedemos al *Clock Configuration* y cambiamos el valor del AHB Prescaler para dividir la frecuencia del reloj HCLK. Probando valores, obtenemos que con una frecuencia de 32MHz podemos tener los 0,5 segundos de encendido y apagado.

5. Conclusiones

Tras haber realizado la práctica he aprendido a como simular una placa y he aprendido a instalar, tener conocimientes básicos sobre cómo usar y configurar el IDE y una introducción a la programación de los sistemas empotrados. En mi caso, sí he conseguido los objetivos tras la realización de la práctica y las fases propuestas por el profesorado. He tenido que indagar más en la parte de las preguntas puesto que al ser la primera práctica y no haber tenido contacto con este entorno ni profundizar tanto en aspectos de frecuencia de reloj, pines, interrupciones, etc... me ha costado más, pero he acabado con más conocimientos sobre ello que al fin y al cabo es lo importante.