MASB | Arduino Interrupciones

#stm32 #c #biomedical #microcontroladores #programacion

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

10 de febrero de 2020

Índice

1. Interrupciones

En la práctica anterior hemos trabajado con los GPIOs y hemos visto cómo trabajar con ellos: leer y escribir. También vimos algunas de sus limitaciones si se utiliza directamente la lectura de estos. En el ejemplo de encender/apagar el LED con el pulsador B1, al leer el pulsador directamente (*level triggered*) el microcontrolador leía ~~infinidad~~ múltiples veces haciendo que el LED se apagara o encendiera de manera aleatoria. Para solucionarlo, utilizamos una variable booleana para realizar la detección por flanco o *edge triggered*. Pese a que lográbamos la funcionalidad deseada, **esta aproximación presenta dos mayores desventajas**:

1. El microcontrolador destina recursos (muy limitados en un microcontrolador) para leer continuamente el valor de la entrada digital. **Esta técnica se conoce como *polling***. **Mientras sondea el valor de la entrada digital, el microcontrolador no puede hacer otras tareas**.
2. **Si la pulsación tiene lugar mientras el microcontrolador hace otra cosa** que no sea leer el valor de la entrada digital, **la pulsación se pierde y es como si no se hubiera pulsado el botón**. Esto sucedía con el reto de la práctica anterior.

¿Solución? Las interrupciones.

Una **interrupción** es una **señal de entrada a la CPU** (*Control Process Unit*), a veces llamada ***Interrupt Request* (IRQ)**, que indica al microcontrolador que **un evento ha tenido lugar**. Cuando el microcontrolador recibe una interrupción, deja de ejecutar el código que esté procesando en ese momento y pasa a ejecutar el código, o **ISR (*Interrupt Service Routine*)**, que hayamos asociado con la interrupción que ha tenido lugar. Una vez finalizada la ejecución de la ISR, el microcontrolador vuelve a retomar el código que estaba ejecutando allí donde lo dejó.

Así que podemos pensar en las interrupciones como **eventos que detienen la ejecución del código del microcontrolador momentáneamente para ejecutar un código distinto que hayamos indicado**. ¿Qué puede generar una interrupción? Pues, por ejemplo, una transición/flanco en un pin de entrada digital, la finalización de una cuenta atrás de un contador, el final de una conversión de analógico a digital o la recepción de un mensaje en serie, entre otros.

En esta práctica vamos a realizar exactamente lo mismo que en la anterior, pero mediante el uso de interrupciones.

También veremos una implementación distinta de cómo hacer parpadear el LED, pero **sin utilizar la función `delay`**, la cual **empezaremos a evitar su uso siempre que se pueda**. La función `delay` es una función que *secuestra* el microcontrolador mientras se realiza la espera, evitando que el microcontrolador haga otras cosas. La función `delay` era también la causante de que no se leyera a veces el pulsador en el reto de la práctica anterior.

1.1. Objetivos

- Primera toma de contacto con las interrupciones.
- Implementación de interrupciones en pines digitales en Arduino
- Alternativas a implementaciones basadas en la función `delay`.

1.2. Procedimiento

Como siempre, antes de empezar, vamos a clonar este repositorio en nuestro ordenador y a crear/saltar a una nueva rama para nuestro desarrollo llamada `develop/A-<tu-nombre-sin-espacios-ni-caracteres-raros>`. Podéis ver los pasos en la práctica anterior.

1.2.1. *Blink the LED like a Pro*

La función que utilizaremos para reemplazar la función `delay` será la función `millis`. Esta función **devuelve los milisegundos que han pasado desde que se inicia la ejecución del programa en el microcontrolador**. Según la documentación, el valor que devuelve la función `millis` **se reinicia a 0 después de aproximadamente 50 días**. Es algo fácilmente gestionable, pero que debéis de tener en cuenta en aplicaciones donde el microcontrolador tenga que estar operando durante un periodo superior a esos 50 días. Lo que haremos con esta función es, en lugar de utilizar una espera, realizar una **programación** (en el sentido de programar que ocurra algo en un momento dado, no programación de escribir código…). Por poner otro tipo de ejemplo, en lugar de poner un cronómetro y esperar a que pasen 10 minutos mirando el cronómetro sin hacer nada, miraremos que hora es e iremos mirando el reloj de vez en cuando para ir haciendo otras cosas de mientras hasta que hayan pasado los 10 minutos.

Lo primero que vamos a hacer es crear nuestro *sketch*, que llamaremos `masb-p02`, en la carpeta `arduino` de nuestro repositorio local. Una vez creado, y asegurándonos que tenemos configurada nuestra EVB y el método de programación de esta, inicializamos el pin del LED `D13`.

```
#define LED       13

void setup() {
  // put your setup code here, to run once:
  pinMode(LED, OUTPUT); // configuramos el LED como GPIO de salida
  digitalWrite(LED, LOW); // configuramos el LED apagado por defecto
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

¿Os habéis quedado con el detalle? Para hacer el código más legible, hemos creado la macro `LED` para indicar el pin del LED.

Ahora utilizaremos hasta 3 variables para realizar la programación del cambio de estado del LED. Estas variables son:

- `periodMillis`: periodo deseado de conmutación del LED en milisegundos.
- `currentMillis`: valor leído de la función `millis`.
- `previousMillis`: valor leído anteriormente de la función `millis`.

El código quedaría del siguiente modo:

```
1  #define LED       13
2
3  const unsigned long periodMillis = 1000; // ms deseados entre cambio
       del estado del LED
4  unsigned long currentMillis = 0; // ms transcurridos obtenidos
5  unsigned long previousMillis = 0; // ms transcurridos obtenidos la
       ultima vez
6
7  void setup() {
8    // put your setup code here, to run once:
9    pinMode(LED, OUTPUT); // configuramos el LED como GPIO de salida
10   digitalWrite(LED, LOW); // configuramos el LED apagado por defecto
11 }
12
13 void loop() {
14   // put your main code here, to run repeatedly:
15
16 }
```

Estas variables las hemos declarado como del tipo `unsigned long`. Además, puesto que la variable `periodMillis` no cambiará su valor durante la ejecución del programa, la hemos declarado como **`const`**. Esta *keyword* nos permite realizar optimizaciones de memoria pudiendo seleccionar en qué memoria se guarda esta constante.

Veamos directamente en código cómo emplear estas variables en nuestra aplicación.

```
1  #define LED       13
2
3  const unsigned long periodMillis = 1000; // ms deseados entre cambio
       del estado del LED
4  unsigned long currentMillis = 0; // ms transcurridos obtenidos
5  unsigned long previousMillis = 0; // ms transcurridos obtenidos la
       ultima vez
6
7  void setup() {
8    // put your setup code here, to run once:
9    pinMode(LED, OUTPUT); // configuramos el LED como GPIO de salida
```

```
10    digitalWrite(LED, LOW); // configuramos el LED apagado por defecto
11  }
12
13  void loop() {
14    // put your main code here, to run repeatedly:
15
16    currentMillis = millis(); // capturamos los ms transcurridos desde el
          inicio del programa
17
18    if (currentMillis - previousMillis >= periodMillis) { // si ha pasado
          el periodo deseado
19
20      previousMillis = currentMillis; // guardamos los ms transcurridos
            desde el inicio del programa
21
22      // y conmutamos el LED
23      if (digitalRead(LED) == LOW) { // si esta apagado
24        digitalWrite(LED, HIGH); // lo encendemos
25      } else { // si no
26        digitalWrite(LED, HIGH); // lo apagamos
27      }
28    }
29
30  }
```

Lo probamos y… *et voilà*. Tenemos nuestro LED parpadeando sin usar la función *delay*. A diferencia que en el ejemplo de la práctica anterior donde hacíamos parpadear el LED, la función `loop` no tarda en ejecutarse 2 segundos.

Vamos a refactorizar un poco la parte del código que se encarga de alternar el estado del LED utilizando una variable booleana adicional.

```
1   #define LED       13
2
3   const unsigned long periodMillis = 500; // ms deseados entre cambio del
        estado del LED
4   unsigned long currentMillis = 0; // ms transcurridos obtenidos
5   unsigned long previousMillis = 0; // ms transcurridos obtenidos la
        ultima vez
6
7   bool estadoLED = LOW;
8
9   void setup() {
10    // put your setup code here, to run once:
11    pinMode(LED, OUTPUT); // configuramos el LED como GPIO de salida
12    digitalWrite(LED, estadoLED); // configuramos el LED apagado por
          defecto
13  }
14
15  void loop() {
```

```
16    // put your main code here, to run repeatedly:
17
18    currentMillis = millis(); // capturamos los ms transcurridos desde el
          inicio del programa
19
20    if (currentMillis - previousMillis >= periodMillis) { // si ha pasado
          el periodo deseado
21
22      previousMillis = currentMillis; // guardamos los ms transcurridos
            desde el inicio del programa
23
24      // y conmutamos el LED
25      estadoLED = !estadoLED; // utilizamos el operador de negacion para
            alternar el estado
26    }
27
28    digitalWrite(LED, estadoLED); // fijamos el estado correspondiente
29
30  }
```

Lo volvemos a probar, comprobamos que todo está correcto y funciona, y, si es así, hacemos el pertinente *commit* y *push* del archivo.

1.2.2. Interrupciones de los GPIO

Ahora vamos a hacer que el pulsador funcione mediante interrupciones. Así nos ahorramos recursos y tiempo de procesado durante su lectura, y no nos perdemos ninguna pulsación indiferentemente de cuándo y cuán larga se haga la pulsación.

Arduino Vanilla solo deja hacer interrupciones para GPIOs y no todos los pines. Así que en esta práctica será muy fácil implementar una interrupción, pero cuando tengamos que hacer interrupciones para contadores (*timers*), ADCs, comunicación serie,..., la cosa se complica.

Para implementar una interrupción necesitamos un **vector de interrupción**, una **ISR** y un **tipo de evento**.

Un **vector de interrupción** es una posición de memoria. En esa posición de memoria se indica al microcontrolador dónde puede encontrar la ISR que debe de ejecutar para esa interrupción. Cada periférico tiene su propio vector de interrupción.

Una **ISR o *Interrupt Service Routine*** es la función que debe de ejecutar el microcontrolador cuándo recibe la interrupción pertinente.

El **tipo de evento** es, entre todos los eventos que puede gestionar un periférico, cuál es el que escogemos gestionar. Por ejemplo, en los GPIOs hay hasta 3 eventos básicos: 1) que la señal vaya de `HIGH` a

`LOW`, 2) que la señal vaya de `LOW` a `HIGH`, y, una combinación de ambos, 3) que el evento se dé para cualquier tipo de transición de nivel.

En Arduino, la función a utilizar es `attachInterrupt`. Y su uso, implementando una interrupción en el pin del pulsador para una transición de `HIGH` a `LOW`, sería el siguiente:

```
#define LED       13
#define PULSADOR  23

void setup() {
  // put your setup code here, to run once:
  pinMode(LED, OUTPUT); // configuramos el LED como GPIO de salida
  digitalWrite(LED, estadoLED); // configuramos el LED apagado por defecto

  pinMode(PULSADOR, INPUT); // configuramos el PULSADOR como GPIO de entrada
  // anadimos una interrupcion al pin del PULSADOR para una transicion de HIGH to LOW (falling)
  attachInterrupt(digitalPinToInterrupt(PULSADOR), cambiarEstadoLED, FALLING);
}

void loop() {
  // put your main code here, to run repeatedly:

}

void cambiarEstadoLED() {
  // ISR del PULSADOR
}
```

En el código hemos indicado el vector de interrupción a la función `attachInterrupt` mediante la función `digitalPinToInterrupt(PULSADOR)`. Esta segunda función nos facilita la vida a la hora de indicar el vector de interrupción. Como ya sabemos, Arduino funciona para diferentes placas y cada una tiene su propia tabla de vectores de interrupción. La función `digitalPinToInterrupt` nos devuelve automáticamente el vector de interrupción del pin que le indiquemos, de tal modo que no tenemos que ir a la documentación del microcontrolador a consultar cuál es ese vector de interrupción (cuyo valor, nada tiene que ver con el número de pin).

Seguidamente, indicamos el nombre de la función que hará de ISR. En este caso, hemos implementado una función (fuera de `setup` y `loop`) llamada `cambiarEstadoLED` la cual se ejecutará siempre que haya una interrupción.

De entre los 3 tipos distintos de interrupción, configuramos que la interrupción que se debe de atender es la que tiene lugar para una transición de nivel alto a bajo, que es lo que ocurre cuando se pulsa el botón.

A continuación, vamos a implementar el cambio del estado del LED mediante la variable booleana `estadoLED`.

```
#define LED       13
#define PULSADOR  23

volatile bool estadoLED = LOW;

void setup() {
  // put your setup code here, to run once:
  pinMode(LED, OUTPUT); // configuramos el LED como GPIO de salida
  digitalWrite(LED, estadoLED); // configuramos el LED apagado por defecto

  pinMode(PULSADOR, INPUT); // configuramos el PULSADOR como GPIO de entrada
  // anadimos una interrupcion al pin del PULSADOR para una transicion de HIGH to LOW (falling)
  attachInterrupt(digitalPinToInterrupt(PULSADOR), cambiarEstadoLED, FALLING);
}

void loop() {
  // put your main code here, to run repeatedly:

  digitalWrite(LED, estadoLED); // fijamos el estado correspondiente

}

void cambiarEstadoLED() {
  // ISR del PULSADOR

  estadoLED = !estadoLED; // cambiamos estado del led
}
```

A un ojo experto como el nuestro no se le habrá pasado por alto el calificador **`volatile`** añadido enfrente de la declaración de la variable `estadoLED`. Este calificador es para el compilador. Este calificador indica al compilador que, cuando compile el programa, haga que **el programa coja el valor de la variable de la memoria RAM y no de un registro**.

El microcontrolador guarda el **valor de las variables en la memoria RAM**, pero, **para realizar operaciones, mueve las variables a registros**. Estos registros (¿habéis oído hablar alguna vez de la memoria cache?) son elementos de memoria que permite realizar operaciones con una velocidad notablemente más veloz. El microcontrolador sigue este proceso para ir lo más rápido posible. Sin embargo, ¿qué puede ocurrir si más de una función intenta acceder a esa variable? Veamos un ejemplo.

Imaginemos que la función `loop` quiere editar una variable `varNumerica` inicializada a `0` y hacer que incremente `+1`. La función `loop` se llevará la variable `varNumerica` de la memoria RAM a un

registro para luego editar su valor y volverlo a guardar a la RAM. Si antes de que lo guarde en la RAM, salta una interrupción y se ejecuta la pertinente ISR donde también se incrementa el valor de `varNumerica`, la ISR irá a la RAM (cuyo valor es aún `0`), moverá la variable a un registro, incrementará `+1` y volverá a guardar la variable en la RAM. Una vez finalizada la ISR, la función `loop` seguirá su ejecución donde estaba. Cogerá el valor de la `varNumerica` que tenía en un registro, cambiará su valor a `1` y lo almacenará en la RAM. Resumen: la variable `varNumerica` debería de haberse incrementado 2 veces, una por la función `loop` y otra por la ISR, por lo que debería de tener un valor igual `2`. En cambio, por no haber puesto el calificador **`volatile`**, el valor de `varNumerica` es `1`. Esta casuística se conoce como **condición de carrera**.

Moraleja: si una variable puede ser editada desde dos funciones de manera *simultánea*, debemos de añadir el calificador `volatile` a esa variable para evitar condiciones de carrera.

Probamos el código y… nos ponemos una medalla en el pecho. El código funciona. ¿Hace falta decir qué hacer si tenemos una versión de código que funciona? Pues ya sabéis. Haced lo que hay que hacer.

1.3. Reto

Ahora, sí. Haz que el LED esté apagado inicialmente y se pueda alternar su estado entre parpadeando (500 ms) y apagado mediante el pulsador B1 sin que tenga ningún tipo de influencia cuándo se pulsa el pulsador o durante cuánto tiempo. Debe de obtenerse el reto sin ningún tipo de restricción y sin usar la función `delay`.

1.4. Evaluación

1.4.1. Entregables

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

- ☐ **Commits**

En el repositorio remoto en GitHub, debe de haber vuestra rama con, como mínimo, los 3 *commits* realizados durante la práctica: LED parpadeando, LED encendido y apagado con el pulsador, y el reto.

- ☐ **Reto**

Cómo mínimo, un *commit* que contenga vuestro código que solucione el reto propuesto.

- ☐ **Informe**

Informe con el mismo formato/indicaciones que el anterior.

El informe debe de contener:

- ☐ Tabla con las funciones de Arduino vistas en la práctica así como una explicación de qué hacen y cómo se utilizan.

1.4.2. Pull Request

Finalizados todos los entregables, acordaos de hacer el *push* pertinente y cread un *Pull Request* (PR) de vuestra rama a la master, como en la práctica anterior (¡pero no hagáis el *merge*!). Acordaros de ponerme como *Reviewer*.

1.4.3. Rúbrica

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

1.5. Conclusiones

Hemos finalizado esta primera parte de la segunda práctica basada en Arduino. En esta hemos visto por primera vez el uso de interrupciones. En particular, el uso de interrupciones con los GPIOs, pero en prácticas posteriores, cuando veamos nuevos periféricos, veremos siempre su uso y como configurar sus interrupciones.

También hemos visto cómo hacer parpadear el LED utilizando la función `millis` en lugar de la función `delay` para *programar* su alternancia de estado. Más adelante, veremos como hacer lo mismo mediante el periférico *timer* o contador y sus interrupciones.

En la siguiente parte de esta práctica, veremos cómo implementar interrupciones de los GPIOs con STM32CubeIDE.