Eventos de temporización

Supongamos que deseamos realizar cierta tarea periódicamente, en este caso, cada una cantidad X de segundos

```
#define SECONDS X
delay(SECONDS*1000);
```

Demora software

La manera más simple de lograr este objetivo, es mediante una demora software. La demora software no es más que un 'tiempo muerto' durante el cual microcontrolador no ejecuta ninguna otra tarea. En Arduino, estas demoras se ejecutan mediante las instrucciones *delay()* y *delay_us()*, las cuales esperan una cantidad de milisegundos o microsegundos respectivamente.
Este método tiene una gran desventaja, mientras el microcontrolador espera a que ocurra este 'tiempo muerto', no puede ejecutar ninguna otra instrucción. Esto lo hace un método poco práctico debido a que en la mayoría de los casos, los microcontroladores ejecutan numerosas tareas.

```
void loop() {
  // put your main code here, to run repeatedly:
  delay(SECONDS*1000);
  Serial.println("Do Something");
}
```

Por ejemplo, supongamos ahora, que además de realizar esta tarea cada X segundos, se debe enviar por el puerto serie los datos recibidos del mismo. Si afrontamos este problema con una demora software, sería ineficiente debido a que solo podremos leer los datos recibidos por puerto serie, una vez hayan transcurrido los X segundos.

Encuesta de temporizadores

Para realizar esta tarea de una manera más eficiente, se usan los **temporizadores**.
Los temporizadores son periféricos internos al microcontrolador sobre los cuales este delega eventos de temporización, de tal manera que el microcontrolador no tiene que encargarse continuamente de saber cuando transcurre un determinado tiempo. Los timers disponibles varían para cada placa Arduino.

Placa	Timers disponibles
Uno, Nano (Basados en AtMega 328)	Timer 0 (8 bits), Timer 1 (16 bits), Timer 2 (8 bits)
Mega, Mega ADK, Mega 2560	Timer 0 (8 bits), Timer 1 (16 bits), Timer 2 (8 bits), Timer 3 (16 bits), Timer 4 (16 bits), Timer 5 (16 bits)

Para encuestar los temporizadores, tenemos disponibles dos funciones en Arduino, estas son *millis()* y *micros()*. Estas funciones devuelven el tiempo en milisegundos y microsegundos respectivamente desde que el microcontrolador fue energizado. Este método nos da la ventaja que el microcontrolador no tiene que esperar el 'tiempo muerto' como en la demora software. Simplemente almacenamos el valor anterior devuelto y se lo restamos al valor actual. Este método se ilustra en el siguiente código:

```
void loop() {
  // put your main code here, to run repeatedly:
  unsigned long difference = millis() - lastMillisValue;
  if (difference >= SECONDS*1000)
  {
    Serial.println(String(difference) + " milliseconds ellapsed");
    lastMillisValue = millis();
  }
  while (Serial.available())
  {
    auto str = Serial.readString();
    Serial.println(str);
  }
}
```

Es necesario conocer que estos métodos devuelven un *unsigned long*, que en Arduino es una variable de 32 bits. El valor máximo alcanzado por una variable de 32 bits es 4,294,967,295, lo que significa que la función *millis()* reinicia el valor devuelto luego de 49.7 días, mientras que *micros()* lo hace luego de 1.2 horas aproximadamente.
Este método es más eficiente que el anterior por lo explicado, sin embargo, tiene sus deficiencias. Supongamos que el microcontrolador empieza a recibir datos por puerto serie, pero son tantos que leyendo la cadena de caracteres demora más de X segundos. Si esto sucede, el mensaje enviado hacia el monitor serie reflejará que transcurrió un tiempo mayor al deseado para ejecutar esta tarea. Para solucionar este problema, se hará uso de los temporizadores de la manera más eficiente.

Interrupciones de los temporizadores

Es necesario entender varios conceptos clave para adentrarnos en el funcionamiento de las interrupciones de los temporizadores.
Los temporizadores de las placas Arduino son de 8 y 16 bits, esto significa que pueden contar hasta 256 y 65536 estados respectivamente. Como la mayoría de las placas Arduino trabajan con un reloj de 16 MHz, **un estado equivale a 63 nanosegundos**, esto significa que mientras el temporizador esté habilitado, incrementará su registro de conteo cada 63 nanosegundos. De esta manera, un temporizador de 8 bits solo es capaz de contar 16 microsegundos (256 * 63 nanosegundos), mientras que uno de 16 bits 4 milisegundos (65536 * 63 nanosegundos). Como es evidente, estos tiempos no son muy útiles en aplicaciones reales, por esto, existe el **preescaler**.
El preescaler modifica la velocidad del timer. Un preescaler de 8 (16 MHz / 8), provocará que un estado equivalga a 500 nanosegundos (2 MHz), haciendo que un temporizador de 8 bits pueda contar hasta 128 microsegundos, mientras que uno de 16 bits puede contar hasta 32.768 milisegundos. El valor del preescaler puede establecerse a 1, 8, 64, 256, 1024.
Las interrupciones CTC (*Clear timer on compare match*) se disparan cuando el valor del contador alcanza el

valor del registro *compare match*.
Existen numerosas librerías que se encargan de realizar estos cálculos y de configurar los registros del temporizador para que este funcione como deseamos. Para resolver el problema planteado se hará uso de la librería *TimerOne* como se muestra a continuación:

```
void setup()
{
  // put your setup code here, to run once:
  Serial.begin(BAUDRATE);
  Timer1.initialize(SECONDS*1000000);
  Timer1.attachInterrupt(timeEllapsed);
}
```

Primeramente, se llama al métdo *initialize()*, el cual recibe la cantidad de microsegundos cada los cuales se desea ejecutar la acción deseada. Luego, se indica que método debe ejecutarse luego de que transcurra el tiempo, en este caso *timeEllapsed*, mostrado a continuación:

```
void timeEllapsed()
{
  Serial.println("Time ellapsed");
}
```

Finalmente el *loop()* que solo se encarga de enviar por puerto serie los datos recibidos del mismo:

```
void loop() {
  // put your main code here, to run repeatedly:
  while (Serial.available())
  {
    auto str = Serial.readString();
    Serial.println(str);
  }
}
```

Este acercamiento es el más eficiente a la hora de darle solución al problema planteado, ya que cada vez que transcurra el tiempo deseado, se disparará la interrupción del temporizador y como consecuencia se ejecutará el método *timeEllapsed()*.

Aspectos a tener en cuenta a la hora de usar los temporizadores

Muchas de las funciones y librerías usadas en Arduino, hacen uso interno de los temporizadores. Es necesario tener en cuenta esto para evitar posibles conflictos, a continuación los ejemplos más comunes:

- Las funciones *delay()*, *millis()*, *micros()* usan el **Timer0**
- La librería *Servo* usa el **Timer1** excepto en Arduino Mega, que usa el **Timer5**
- La función *tone()* usa el **Timer2**