Timing events

Let's suppose that we want to do some task periodically, each X seconds.

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

Delay

The simplest way to achieve this goal is by using a delay. Delay is 'a dead time' during which the microcontroller can't do any other task. On Arduino, these delays are done by using functions *delay()* and *delay_us()*, which wait for some milliseconds or microseconds respectively. This is the worst approach to solve this problem, because while the microcontroller is waiting for this 'dead time', it isn't able to do any other task.

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

If now, in addition to doing this task, data received through serial port must be sent, we shouldn't use a delay, because we will receive data through serial port after the 'dead time' has elapsed.

Timers reading

In order to do this task in a more efficient way, **timers** are used.
Timers are internal peripherals to microcontroller. By using them, microcontroller doesn't have to be worried about timing events, because this is their job. Each Arduino board has different timers as shown above:

Board	Available timers
Uno, Nano (Based on 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)

In order to read timers, we have available two functions in Arduino, these are *millis()* and *micros()*. These functions return time in milliseconds and microseconds since microcontroller was turned on. This method is better than the previous one because microcontroller doesn't have to wait for the 'dead time'. In the code, the previous returned value is substracted form actual value. This approach is shown below:

```
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);
  }
}
```

Is good to know that these functions return an unsigned long variable, which length in Arduino is 32 bits. Maximum value reached by this variable is 4,294,967,295, which means that function *millis()* restarts it returned value after 49.7 days, while *micros()* does it after 1.2 hours approximately.
This approach is more efficient than the previous one, however, it isn't the best solution. If the microcontroller starts receiving data through serial port, but there are lot of bytes to receive, it will last more than X seconds doing this task. If this happens, printed message will show that this task wasn't executed at the desired time. In order to solve this problem, timers will be used in the most efficient way.

Timers interrupts

Timers available at Arduino boards are of 8 and 16 bits. This means that they can count until 256 y 65536 states respectively. Due to almost Arduino boards work with a 16 MHz clock, **a state is equivalent to 63 nanoseconds**. Because of this, while timer is enable, it will increment its count register every 63 nanoseconds. An 8 bits timer is able to count 16 microseconds (256 * 63 nanoseconds), while a 16 bits timer can count 4 milliseconds (65536 * 63 nanoseconds). Evidently, these times aren't useful in real applications, this is why exists the **preescaler**.
Preescaler modifies the speed of the timer. A preescaler of 8 (16 MHz / 8), will provoke that a state is equivalent to 500 nanoseconds (2 MHz). By doing this, an 8 bits timer is able to count until 128 microseconds, while a 16 bits timer does it until 32.768 milliseconds. Preescaler value can be 1, 8, 64, 256, 1024.
CTC interrupts (*Clear timer on compare match*) are triggered when the counter value reaches the *compare match* register value.
There are several libraries which are responsible of configure the timer registers and do the calculations for us. To solve previous problem in the most efficient way, *TimerOne* library will be used as shown below:

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

First of all, *initialize()* function is called. This function receives the time in microseconds in order to do the desired task after this time passes. After this, is indicated which method must be executed when time passes. In this case this method is *timeEllapsed()*:

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

Finally, the only task to do in *loop()* function is to send through serial port data received by itself:

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

This is the most efficient solution to solve the raised problem because each time that passes the desired time, timer interrupt will be triggered and as a consequence method *timeEllapsed()* will be executed.

Things to consider when using timers

Lot of functions and libraries used in Arduino use internally timers. Is good to know this in order to avoid possible bad behaviors, these are the most common examples:

- Functions *delay()*, *millis()*, *micros()* use **Timer0**
- Library *Servo* use **Timer1** except on Arduino Mega, which use **Timer5**
- Function *tone()* use **Timer2**