Laboratorio RPM

Laura Camila Blanco Gómez
Escuela de Ingeniería
Universidad Sergio Arboleda
Bogotá, Colombia
laura.blanco01@usa.edu.co

Santiago Cáceres Linares
Escuela de Ingeniería
Universidad Sergio Arboleda
Bogotá, Colombia
santiago.caceres01@usa.edu.co

I. Introducción

El presente informe detalla el procedimiento llevado a cabo en el laboratorio para analizar y controlar un motor mediante un microcontrolador. Se realizaron una serie de pasos para configurar el sistema y llevar a cabo mediciones de R.P.M. (revoluciones por minuto) utilizando un sensor de herradura. Asimismo, se estableció una comunicación entre el microcontrolador y un PC mediante una interfaz USB, implementando un protocolo de comunicaciones bidireccional.

El objetivo principal de este experimento fue adquirir datos precisos sobre el comportamiento del motor en términos de velocidad y tiempo de respuesta en diferentes condiciones de operación.

II. Marco teórico

Como recapitulación de ciertos de los tópicos tratados durante el laboratorio se parte para el correcto entendimiento de algunos conceptos mencionados en la sección anterior.

RPM(revolución por minuto): Las revoluciones por minuto son una medida de frecuencia. Nos muestra la cantidad de vueltas completadas en un tiempo determinado en la máquina. Entre más revoluciones por minuto tenga un motor, podrá trabajar a más velocidad aprovechando al máximo la potencia del motor.

Sensores de herradura: Es un dispositivo que responde al cambio en la intensidad de la luz. Se componen de un emisor que genera la luz y un receptor que percibe la luz generada. Se suelen usar para detectar, clasificar y posicionar objetos en condiciones ambientales extremas.

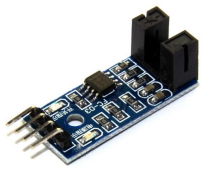

Figure 1: Sensor herradura

Comunicación USB: La comunicación USB (Universal Serial Bus) es un estándar de comunicación serial que define los protocolos y conectores para la transmisión de datos entre dispositivos electrónicos. Utiliza un bus jerárquico y se divide en varios tipos de conectores (como USB-A, USB-B, micro USB, USB-C, entre otros). Los dispositivos USB pueden funcionar como hosts o periféricos, y pueden soportar diferentes velocidades de transferencia de datos.

Protocolos de Comunicación: Un protocolo de comunicaciones en informática y telecomunicación consiste en un conjunto de reglas que facilitan la interacción entre entidades dentro de un sistema de comunicación, como computadoras o teléfonos celulares, para la transmisión de información a través de variaciones en una magnitud física. Este protocolo establece las pautas para la sintaxis, semántica, sincronización y métodos de recuperación de errores durante la comunicación, pudiendo ser implementado tanto a través de hardware, software o una combinación de ambos. Además, define cómo los ordenadores se identifican en una red, cómo se transmiten los datos y cómo se debe procesar la información. Cada mensaje intercambiado sigue un formato específico con un significado preciso, esperando obtener una respuesta entre una serie de posibles respuestas predeterminadas. Para que un protocolo funcione, todas las partes involucradas deben estar de acuerdo y, para esto, se puede desarrollar como un estándar técnico. Se establece una analogía cercana entre los protocolos y los lenguajes de programación, siendo esenciales para las comunicaciones como lo son los lenguajes de programación para los cálculos. En el contexto de las redes de computadoras, un protocolo de comunicación, también llamado protocolo de red, determina cómo los diferentes mensajes o tramas de bits circulan en la red.

III. Procedimiento

Para este apartado se divide en cuatro partes, las cuales son.

- **Montaje** Para este apartado del procedimiento se hizo la conexión entre la STM, el sensor de herradura y el programa en QT como se ve en la figura 3, adicionalmente se conecta un cable USB - C para la trasmisión de datos por el puerto serial.

- **Protocolo** para este apartado se definió un protocolo el cual es enviado por medio del puerto serial del computa-

dor en donde se ejecuta el código generado en el QT, el protocolo fue formado de la siguiente manera.

- head
- tamaño
- data
- runStop
- byte de validación
- end

Algunos de estos bytes de este protocolo fueron quemados debido a que estos nunca cambian, estos bytes fuero head y end porque cada uno indica el inicio y el fin de la transmisión del protocolo respectivamente, otro dato que no tiende a cambiar su valor a la hora hacer el envío de datos es el tamaño debido a que este se compone de la suma del tamaño de la data y el **overhead**, este último tiene un valor de 5 debido a que quitando los bytes de la data los cuales en este caso fueron 8 repartidos de la siguiente manera

- 4 bytes para el tiempo
- 4 bytes para el tiempo entre pulso y pulso

los demás componente del protocolo ocupan 1 byte y al sumar estos nos dan 5 el cual es el valor del overhead finalmente se suman el tamaño de la data y el overhead y así nos da un total de 13 por lo cual es valor para este caso siempre será 13. Por otro lado, los bytes runStop y byte de validación tiene el siguiente propósito en el protocolo, el byte runStop sirve para saber si se desea encender el motor o apagar desde el QT esto lo logra por medio de los comandos (0x11 y 0x12) los cuales son para encender y apagar el motor respectivamente. Finalmente, el byte de verificación nos sirve para saber si el paquete llego de manera correcta, esto lo logra por medio de una operación xor desde la el byte head hasta el byte runStop, en caso de que el protocolo llegue bien se retorna un ACK en caso contrario se retorna un NACK.

- **QT** Este apartado se divide en dos partes, las cuales son:
  - Interfaz gráfica: Este apartado consta de unos labels los cuales nos sirven para mostrar los siguientes datos.
    - rpm actual
    - rpm máximo
    - Tiempo de parada real

    además de estos labels se tiene dos botones loas cuáles nos sirven para encender y apagar el motor y un campo de texto que nos irá mostrando las rpm actuales de sistema las cuales se usaran posteriormente para la realización de unas gráficas que mostraran el comportamiento de encendido y apagado del motor. Finalmente, la interfaz final se ve de la siguiente manera

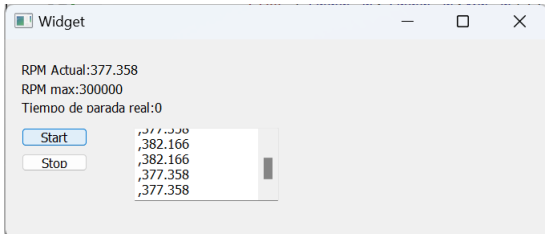

Figure 2: Interfaz QT

  - Código: El código consta de 3 funciones principales les cuáles son
    - readSerial()
      Esta función se encarga de recibir los datos que llegan por medio del puerto serial, esto lo hace por medio de una variable global de tipo **qByteArray** la cual funciona como un arreglo de datos que llegan a QT, la función principal de esta función es validad que el inicio y el fin del protocolo y el byte de verificación, eso lo hace en el caso del inicio y el fin del protocolo buscando en la primera y última posición del arreglo los valores de estos bytes los cuales fueron definidos al momento de definir el protocolo.

```
1  if(serialData.at(0) == 0x02 && ...
       serialData.at(serialData.at(1)-1)==0x3A){
2          qDebug()<<"Paquete con ...
               cabeza y cola iguales";
3          // calculate(serialData);
4          if(verify_xor(serialData,
5          serialData.length(),
6          serialData.at(serialData.length()-2)) ...
               == ACK) {
7              qDebug()<<"xor igual";
8              calculate(buffer);
9              serialData.clear();
10         }
11     }
12 }
```

    - calculate()
      Esta función se encarga de tomar el tiempo entre pulso y pulso y a partir de esto calcular el valor de rpm esto lo logra usando la siguiente fórmula

$$rpm = (1000.0 * 1000.0 * 60.0)/(2.0 * pulse * 100.0);$$

      esta fórmula se logra pasando los datos de u segundos a segundos lo cual es la parte de arriba de la fracción y la parte inferior de la fracción se toma el tiempo y se multiplica por 100 para tener el resultado y adicionalmente se multiplica por dos por las dos ranuras que se tiene en el disco, esto se puede ver en el siguiente código.

```
1  if (pulse != 0) {
2      rpm = (1000.0*1000.0*60.0)
3      /(2.0*pulse*100.0);
4          if(rpm > 0){
5              ui->textEdit->
```

```
6           append(QString::number(rpm));
7           }
8
9   } else {
10      rpm = 0;
11  }
```

para finalizar, cuando se detiene esta función se realiza el cálculo en minutos de tiempo real de parada, adicionalmente también actualiza el rpm máximo como se ve en el siguiente código.

```
1       if (rpm>rpmMax){
2           rpmMax=rpm;
3       }
4       if(RunStop == 0){
5           timeStop += ...
                (1000.0*1000.0*pulse)/60.0;
6           ui->tiempoParadaLabel->setText(
7           "Tiempo de parada real:" + ...
                QString::number(timeStop));
8       }
```

* create_protocol(uint8_t operation)
  Esta función se encarga de crear el protocolo que enviara los comando de inicio o parada del motor por medio del byte runStop, adicionalmente este método envía siempre como data un paquete de un byte teniendo este byte el valor de 1, esto se ve en el siguiente código.

```
1           uint8_t total_len = ...
                overhead + 1;
2       protocol.append(0x02);
3       protocol.append(total_len);
4       protocol.append(0x01);
5       protocol.append(operation); // ...
            OPERATION
6       protocol.append(
7       xor_operation(protocol,total_len));
8       protocol.append(0x3A);
```

- **Código STM** Para el apartado de la tarjeta, la cual va a servir como el dispositivo, el cual se va a encargar de recibir comandos y enviar datos para el cálculo de las rpm, para que esto se pueda realizar, se debe configurar el **Milddleware** de la tarjeta como **USB_DEVICE**, adicionalmente en el apartado de conectividad de la tarjeta se le debe escoger la opción **devide_only** una vez se hacen esta configuración se debe configurar el reloj el cual tendrá como frecuencia máxima 96MHz esto debido a la conexión USB.

  Una vez finalizadas estas configuraciones previas se debe realizar la configuración del **TIM1** el cual nos permitirá avanzar contar el tiempo que existe entre un pulso y otro aumentando su valor cada 10us
  - Preescaler: 960 - 1
  - CP: 65535

  Por otro lado, se configura el **TIM2** el cual generará una interrupción cada 20ms en esta interrupción hará el envío de los datos del protocolo, la configuración de este TIM se realizó de la siguiente manera
  - Preescaler: 960 - 1
  - CP: 2000 - 1

  Después de realizar estas configuraciones se procede a generar las funciones para el código las cuales son:
  - onOffMotor(uint8_t stopRun): Esta función se encarga de encender GPIO del pin B2, esta función realiza esta operación de encendido o apagado por medio de un comando que se envía por el parámetro stopRun de esta función, si llega un 0x11 esto significa que se pulsó el botón start en QT y se pidió encender el motor, de lo contrario si llega un 0x12 significa que se oprimió el botón stop y se desea apagar el motor esto se ve en el siguiente código.

```
1           if(stopRun==0x11){
2               HAL_GPIO_WritePin(GPIOB, ...
                    GPIO_PIN_2, ...
                    GPIO_PIN_SET);
3           }else if(stopRun==0x12){
4               HAL_GPIO_WritePin(GPIOB, ...
                    GPIO_PIN_2, ...
                    GPIO_PIN_RESET);
5           stoping();
6           }
```

  - createBuffer(): Esta función será de tomar el tiempo entre pulsos y los pulsos leídos por medio del puerto GPIO B1 que está conectado al sensor de herradura, este método envía estos datos en dos paquetes de 4 bytes cada uno, posterior a esto se hace el llamado de la función **createPakage()**, lo anteriormente mencionado se ve en el siguiente código.

```
1               uint8_t packageData[size];
2               uint32_t data = ...
                    *(uint32_t*) ...
                    &measuredTime;
3
4               packageData[0] = data & ...
                    0x000000FF;
5               packageData[1] = (data & ...
                    0x0000FF00) >> 8;
6               packageData[2] = (data & ...
                    0x00FF0000) >> 16;
7               packageData[3] = (data & ...
                    0xFF000000) >> 24;
8
9               data = *(uint32_t*) ...
                    &pulseNumber;
10
11              packageData[4] = data & ...
                    0x000000FF;
12              packageData[5] = (data & ...
                    0x0000FF00) >> 8;
13              packageData[6] = (data & ...
                    0x00FF0000) >> 16;
14              packageData[7] = (data & ...
                    0xFF000000) >> 24;
15
16              createPakage(packageData, ...
                    size);
```

– createPakage(): Esta función se encarga de la construcción del protocolo de comunicación y el envío de este último por medio del puerto serial hacia QT, esto lo logra tomando los datos de la función **createBuffer()** y anexando estos datos desde la tercera posición del protocolo porque los dos primeros bytes están ocupados por el inicio y el tamaño del protocolo, una vez es agreda la información del protocolo se agregan los bytes de runStop, Byte de verificación y final del protocolo, para dar valor al byte de verificación se hace por medio de la función **valueXorValidation()** el cual realizara un cálculo de xor desde el inicio hasta el byte runStop, una vez el protocolo es creado se llama la función **CDC_Transmit_FS()** la cual es la función que envía los datos a QT, esto se pueden ver en el siguiente código.

```
uint32_t totaLen = len + overHead;
uint8_t *dataEnvida = malloc(sizeof(uint8_t)*totaLen);
dataEnvida[0] = 0x02;
dataEnvida[1] = totaLen;
for (int i = 0; i < len; i++) {
    dataEnvida[i + 2] = buffer[i];
}
dataEnvida[totaLen - 3] = RunStop;
dataEnvida[totaLen - 2] = valueXorValidation(dataEnvida, totaLen - 2);
dataEnvida[totaLen - 1] = 0x3A;
CDC_Transmit_FS(dataEnvida, totaLen);
```

IV. Resultados y análisis de resultados

una vez realizado varias pruebas en el sistema con distintos valores de voltaje usados sobre el motor se llegaron a los siguientes resultados

– **QT**
Gracias a que el envío de datos se realiza cada 20 ms desde la stm al empezar a llegar los datos el label que muestra las rpm actuales no se pueden percibir con mucha claridad a valores pequeños esto se da debida a que el valor de rpm cambia bastante rápido por lo cual en un inicio no se pueden percibir estos valores de una manera agradable. Por otro lado, el label de rmp máximo tiene un incremento más amigable a la hora de verse, pero este se queda pausado en el valor de 300000, a pesar de que el valor de las rpm siga aumentando

– **Protocolo**
Con respecto al protocolo se decidió tomar la decisión de no enviar las rpm para que la stm guardara la información esto pasó debido a que como se enviaba la información tan rápido el sistema no tenía tiempo de poder limpiar el buffer en el cual iba la data de la RPM lo que hacía que el paquete de 4 bytes con cada envío se volviera más grande por lo cual al momento de llegar a la stm el valor del byte de verificación nunca daba por lo cual la stm siempre devolvía un **NACK**.

Por otra parte, al momento de realizar las primeras pruebas con el envío de datos desde la stm a QT se presentó un error en el cual siempre hacía que el valor de llegada del byte de validación no considera con el calculado por QT esto se presentaba porque no se usaba el tamaño que se envía el segundo byte del protocolo, sino la función **length()** de un arreglo del tipo **QByteArray**, esto se solucionó al usar el dato que se encuentra el segundo byte del protocolo.

Siguiendo con el punto anterior otra medida que se tomó para qué la llegada de los datos no se viera afectada por el envío tan rápido que se hacía de la información fue crear un arreglo global en el cual se copiaban los datos para posteriormente hacer el cálculo de la rpm se limpiaba lo cual hacía que un dato nuevo no sobrescribiera los datos actuales con los cuales se estaba haciendo tanto el cálculo del byte de verificación como el cálculo de la rpm.

– **Código STM**
debido a que la función que recibe los comando enviados desde QT en cualquier momento esto permite que el sistema pueda tomar inmediatamente la llegada del comando lo cual permite que el sistema no tenga lentitudes a la hora de realizar la acción que se desea hacer. Con respecto al valor que se toma desde el GPIO este es inversamente proporcional a la velocidad a la que vaya el motor esto se da debido a que a mayor velocidad menor es el tiempo en el que se da una vuelta completa y así mismo a menos velocidad mayor es el tiempo en que se da una vuelta entera por parte del disco

– **Gráfica RPM vs tiempo**
Se puede observar en la gráfica que se encuentra en la figura 4 el comportamiento de aceleración y desaceleración del motor desde el momento que se dio el comando **start** y el comando **stop**. A pesar de que el valor de las rpm máximas en la interfaz se queda quieto después del valor 300000 en la gráfica se puede observar como esta llega a los 400000 en su valor máximo.

Con respecto a otros valores de voltajes se puede observar en los valores que toma el RPM la relación entre el voltaje del sistema y el tiempo

en que se llega al valor máximo son inversamente proporcionales, ya que a mayor voltaje se hará girar el motor más rápido por lo cual se llegara más rápido al valor máximo de rpm que da el motor, esto pasara al revés si el voltaje es menor.

Finalmente, con un voltaje de 3,7 v se obtuvo que el tiempo dé respuesta del motor es de 19780ms y el valor que se alcanza en este tiempo es de 415.512 RPM.

V. CONCLUSIONES

- Como podemos ver la figura 4 el gráfico de carga y descarga cuenta con ciertos valores muy pequeños que no tendrían sentido en un sistema de este tipo, para evitar esto se pudo haber creado un sistema para errores donde se verificara que el valor resultante no fuera tan diferente del anterior y así mantener los resultados de las rpm más consistentes.

- Al momento de pasar datos desde el QT al STM32 se creaba un error al enviar datos variables, en este caso se optó por mandar las instrucciones de start y stop como comandos en vez de enviarlos como un número decimal que debía ser transformado y enviado.

- Al momento de hacer la estructura se deben tener en cuenta las medidas exactas y los materiales usados, al no quedar el motor estable empieza a chocar con el sensor lo que causa detenimiento en el proceso o una mala lectura en los datos, en el caso de los materiales se pudo ver que al crear la pieza en plástico por culpa de la fricción se derrite un poco, pero esto genera que no quede justo el disco por lo que cuando se aumenta el voltaje se sale el disco.

- A la hora de pasar la data del buffer al buffer que está de manera global se realizó con la función **append**, ya que al realizarlo con la asignación **=** se presentaba errores a la hora de tomar los datos de buffer.

- A la hora de escoger el material del cual va a ser el disco que va conectado al motor no sé muy liviano, ya que con este tipo de materiales se obtiene una medición menos precia por eso para este caso se escogió un disco.

REFERENCES

[1] U. technology. Usb-if technologies. [Online]. Available: https://www.usb.org/technologies

[2] Motorysa. Qué son las revoluciones por minuto en un motor. [Online]. Available: https://mitsubishi-motors.com.co/blog/revoluciones-por-minuto-que-son/

[3] MovilTronics. Sensor de herradura klh512. [Online]. Available: https://moviltronics.com/tienda/sensor-klh512/

[4] Wikipedia. Protocolo de comunicaciones. [Online]. Available: https://es.wikipedia.org/wiki/Protocolo_de_comunicaciones

VI. Anexos

1

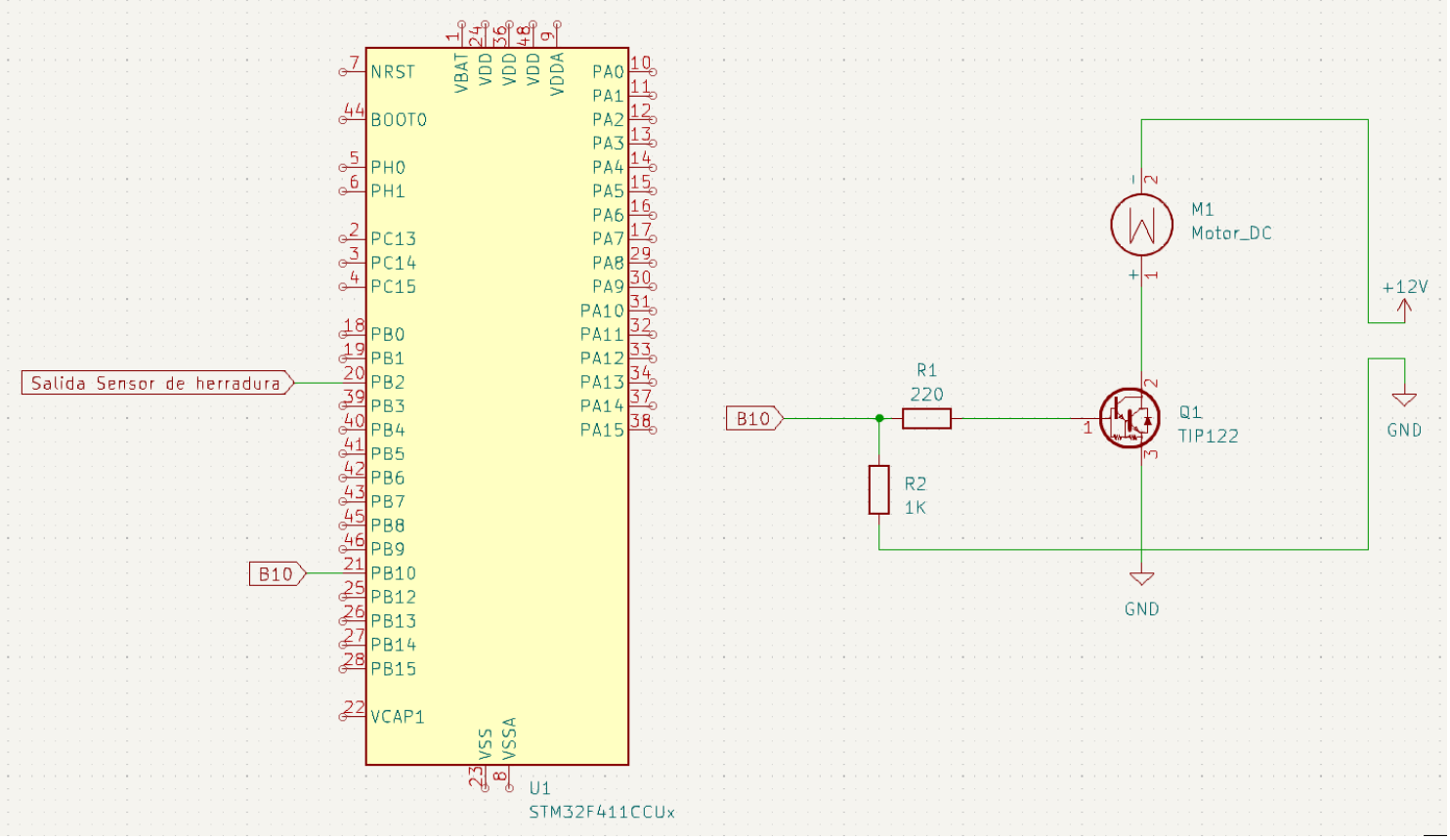

Figure 3: Esquemático

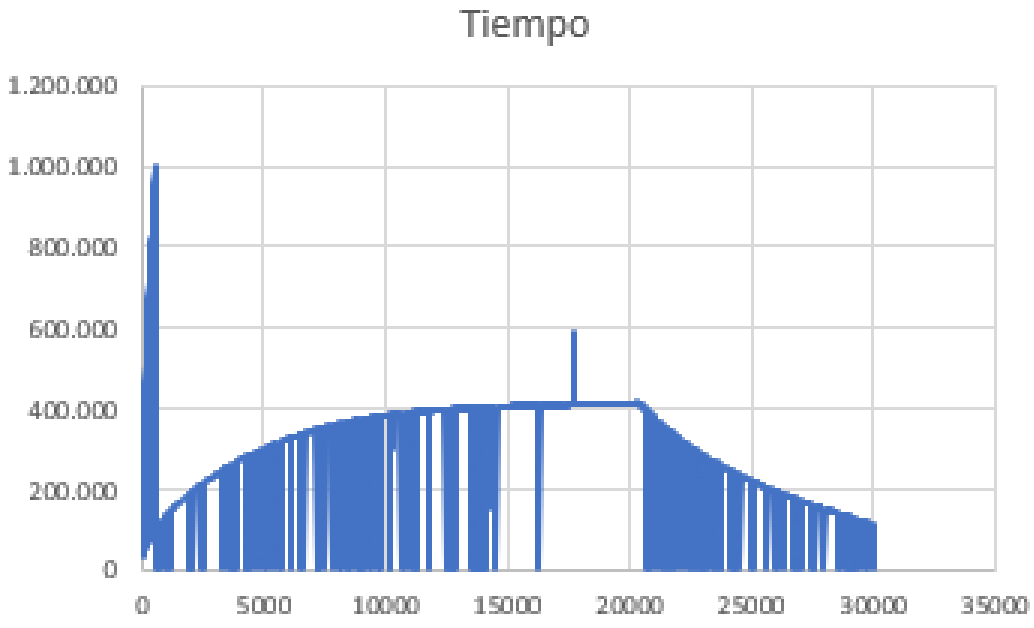

Figure 4: Gráfico rpm vs tiempo