Universidad Nacional Autónoma de México
Facultad de Ingeniería
División de Ingeniería Eléctrica
Ingeniería en Computación

Proyecto 2
"Comunicación de Sistemas Embebidos"

Integrantes:
- Castelan Ramos Carlos
- Corona Nava Pedro Jair
- Mendoza de los Santos Lirio Aketzalli
- Ortiz Camacho Jessica Elizabeth

Materia: Fundamentos de Sistemas Embebidos
Grupo: 04
Semestre: 2024-1

Fecha de entrega: 23 de noviembre 2023

Proyecto 2 “Comunicación de Sistemas Embebidos”

Desarrollo.

Interfaz Física.

Para la construcción de la interfaz física utilizamos elementos del proyecto anterior, pero conectados de forma diferente y agregando una tarjeta de ARDUINO. En este caso el ARDUINO es el que se encarga de realizar la lectura de la señal recibida, posteriormente se comunica con la Raspberry Pi y es esta última la que realiza el procesamiento de los datos y guardar los tiempos en un archivo.

Conexiones.

- I2C: Para esta comunicación se necesitan 4 cables uno para conectar el pin SCL (reloj), otro para el pin SDA (datos), uno para encender los componentes del bus y un cuarto para GND.

- SPI: Para conectar dispositivos con pines SPI es un poco más complejo, ya que requiere de 6 cables mínimo. Uno para el SCK (reloj), el Pin MISO (entrada maestra, salida esclava), el Pin MOSI (salida maestra, entrada esclava), uno para cada CS/SS (Selección de chip/Selección de esclavo), uno para alimentar los componentes y uno último para GND.

- UART: Se requiere de 3 conexiones GND que conecta al GND global de tu circuito, RX para recepción (conectado al pin TX del otro componente) y TX para transmisión (conectado al RX del otro componente). O bien se puede utilizar un convertidor UART a USB, al resultar este método más sencillo decidimos utilizar el convertidor USB.

Cada uno de los programas toma la lectura de 10,000 iteraciones de los pines definidos tomadas en un tiempo n, es entonces que a partir de estos resultados generamos códigos en Python para el análisis de los datos.

La primera parte de gráficas representa el cambio del crecimiento de tiempos en contra del número de iteraciones tomadas, en la cual se espera como resultado una gráfica cuadrática, ya que los tiempos deben crecer de manera constante.

Para esto únicamente se hizo uso del archivo generados en los programas del eje x, llamados SHx.txt, PYx.txt, Cx.txt y Cppx.txt los cuales son tratados en el programa itera.py.

La segunda parte de gráficas representa un histograma de frecuencias sobre el cálculo de Δt_n:

Amplitud	Tiempo	Δt_n
0	t_0	..
0	t_1	$\Delta t_0 = t_1 - t_0$
0	....	...
0	t_n	$\Delta t_n = t_{n+1} - t_n$

Tabla del cálculo de Δt_n

Para esto se hizo uso de un programa llamado resta.py que obtiene el cálculo de Δt a partir de los tiempos de eje x y los resultados se almacenan en otro archivo, llamados SHxRes.txt, PYxRes.txt, CxRes.txt y CppxRes.txt, en base a estos resultados generamos el histograma de frecuencias de las Δt haciendo uso de histo4.py

Para este proyecto se realizaron códigos que implementan distintos protocolos y canales de comunicación (I2C, SPI, UART, Sockets), a continuación, se explican dichos códigos y los resultados obtenidos tras su ejecución.

- I2C

  Código:

  Para comenzar se incluyen las bibliotecas necesarias para la entrada/salida estándar, manipulación de archivos y la interfaz I2C de WiringPi.

```
// Inclusión de bibliotecas necesarias para el programa.
#include <iostream>
#include <fstream>
#include <wiringPiI2C.h>
```

  Posteriormente define la constante DEVICE_ID con el valor hexadecimal 0x08, que representa la dirección del dispositivo I2C con el que se va a comunicar el programa. Despues de esto se utiliza la función wiringPiI2CSetup de la biblioteca WiringPi para inicializar la comunicación I2C con el dispositivo con la dirección previamente definida

```
// Definición de la dirección del dispositivo I2C.
#define DEVICE_ID 0x08

// Función principal del programa.
int main(int argc, char **argv) {

  // Configuración de la comunicación I2C utilizando la biblioteca WiringPi.
  int fd_comm = wiringPiI2CSetup(DEVICE_ID);
```

  Para cuestiones de análisis posteriores se abren dos archivos, "y.txt" y "timeFile.txt", para escribir datos y marcas de tiempo, respectivamente.

```
// Creación de archivos de salida para almacenar datos y marcas de tiempo.
ofstream yfile;
yfile.open("y.txt");

ofstream timeFile;
timeFile.open("timeFile.txt");
```

Con la finalidad de revisar la conexión con el dispositivo se implementó un bloque de verificación. Si la inicialización de la comunicación I2C falla, el programa imprime un mensaje de error y termina con un código de retorno -1. Por el contrario, si la conexión es exitosa se muestra un mensaje al usuario.

```
// Verificación de la inicialización de la comunicación I2C.
if (fd_comm == -1) {
  cout << "Failed to initialize I2C communication \n";
  return -1;
}

// Mensaje de éxito en la configuración de la comunicación I2C.
cout << "I2C communication successfully set up \n";
```

Una vez que la conexión fue exitosa se procede a entrar al bucle principal. En este bucle, se lee un dato del dispositivo I2C en cada iteración. Se obtiene la marca de tiempo actual utilizando la función clock gettime, y se registran tanto la marca de tiempo como el dato en los archivos correspondientes.

```
// Lectura de datos del dispositivo I2C en un bucle.
int received_data;
int N = 1000; // Número de iteraciones

for (int i = 0; i < N; i++) {
  // Lectura del dato desde el dispositivo I2C.
  received_data = wiringPiI2CRead(fd_comm);

  // Obtención de la marca de tiempo actual.
  timespec ts;
  clock_gettime(CLOCK_REALTIME, &ts);
  double timestamp_seconds = static_cast<double>(ts.tv_sec) + static_cast<double>(ts.tv_nsec) / 1000000000.0;

  // Registro de la marca de tiempo en el archivo "timeFile.txt".
  timeFile << std::fixed << timestamp_seconds << std::endl;

  // Registro del dato en el archivo "y.txt".
  yfile << received_data << "\n";
}
```

Para finalizar los archivos se cierran y el programa retorna 0, indicando una terminación exitosa.

```
// Cierre de los archivos de salida.
yfile.close();
timeFile.close();

// Retorno exitoso del programa.
return 0;
```

- SPI

Código:

De igual forma se incluyen las bibliotecas necesarias para la entrada/salida estándar, manipulación de archivos y la interfaz I2C de WiringPi.

```
//Inclusion de bibliotecas necesarias para el programa
#include <iostream>
#include <fstream>
#include <wiringPiSPI.h>
```

Se define la constante CHANNEL que representa el canal SPI a utilizar.

```
#define CHANNEL 0 // Canal SPI
```

La función principal comienza inicializando la comunicación SPI mediante wiringPiSPISetup, estableciendo el canal y la velocidad de transferencia. Que en este caso es el canal 0 con una velocidad de 500000 Hz. Si la inicialización falla, se muestra un mensaje de error y el programa se termina.

```
int main(int argc, char **argv) {
  // Inicialización de la comunicación SPI con un límite de velocidad de 500,000 Hz
  if (wiringPiSPISetup(CHANNEL, 500000) == -1) {
    cerr << "Failed to initialize SPI communication." << endl;
    return -1;
  }
  cout << "SPI communication successfully set up." << endl;
```

En este programa también se crean los archivos: "y.txt" y "timeFile.txt". Para recabar los datos y tiempos respectivamente.

```
    // Creación y apertura de archivos para almacenar datos y valores de tiempo
    ofstream yfile;
    yfile.open("y.txt");

    ofstream timeFile;
    timeFile.open("timeFile.txt");
```

En esta ocasión se establece el número de iteraciones en 1000. Posteriormente se preparan los datos a enviar (txData) y se realiza la transferencia SPI usando la función wiringPiSPIDataRW. Una vez realizado el envío se escriben las marcas de tiempo y los datos recibidos en los archivos correspondientes para cada una de las N iteraciones.

```
int N = 1000; // Número de iteraciones

// Bucle para realizar N iteraciones
for (int i = 0; i < N; i++) {
  unsigned char txData[1] = {0x00}; // Datos a enviar (1 byte)
  unsigned char rxData[1] = {0x00}; // Datos recibidos (1 byte)

  // Realización de la transferencia SPI
  wiringPiSPIDataRW(CHANNEL, txData, 1);

  // Obtención de la marca de tiempo actual en segundos
  timespec ts;
  clock_gettime(CLOCK_REALTIME, &ts);
  double timestamp_seconds = static_cast<double>(ts.tv_sec) + static_cast<double>(ts.tv_nsec) / 1000000000.0;

  // Almacenamiento de la marca de tiempo en el archivo timeFile
  timeFile << std::fixed << timestamp_seconds << std::endl;

  // Almacenamiento de los datos recibidos en el archivo yfile
  yfile << static_cast<int>(rxData[0]) << "\n";
}
```

Finalmente, los archivos se cierran para asegurar que los datos se guarden correctamente, y la función main devuelve 0 para indicar una finalización exitosa del programa.

```
// Cierre de los archivos después de completar las iteraciones
yfile.close();
timeFile.close();

return 0;
```

- UART

Código:

Además, utilizar las bibliotecas anteriores para este programa se añade la biblioteca wiringSerial a para las funciones relacionadas con la comunicación UART.

```
//Inclusion de bibliotecas necesarias para el programa
#include <iostream>
#include <fstream>
#include <wiringPi.h>
#include <wiringSerial.h> // Para la comunicación UART
```

Para lograr este tipo de comunicación es necesario especificar el puerto por el cual se dará la comunicación y la velocidad a la que se llevará a cabo. Por este motivo Se define el puerto UART a utilizar (DEVICE_PORT), en este caso es el puerto predeterminado para la Raspberry, y la velocidad de baudios (BAUD_RATE) para la comunicación UART.

```
#define DEVICE_PORT "/dev/ttyACM0" // Puerto UART predeterminado para Raspberry
#define BAUD_RATE 9600 // Velocidad de baudios para la comunicación UART
```

La función principal comienza inicializando la biblioteca WiringPi con wiringPiSetup. Si la inicialización falla, se muestra un mensaje de error y el programa se termina.

```
int main (int argc, char **argv)
{
  // Inicialización de la biblioteca WiringPi y del puerto UART
  if (wiringPiSetup() == -1) {
    cerr << "Failed to initialize WiringPi library" << endl;
    return 1;
  }
```

Para la apertura del puerto UART Se utiliza serialOpen con el puerto especificado y la velocidad de baudios especificada. Para corroborar la correcta apertura se realiza una verificación. Si la apertura falla, se muestra un mensaje de error y el programa se termina. Si la apertura es exitosa se le indica al usuario mediante un mensaje en la pantalla.

```
// Apertura del puerto serie UART con el baud rate especificado
int fd_comm = serialOpen(DEVICE_PORT, BAUD_RATE);

// Verificación de la apertura exitosa del puerto UART
if (fd_comm == -1) {
  cerr << "Failed to open UART communication" << endl;
  return 1;
}
cout << "UART communication successfully setup" << endl;
```

En este programa también se crean los archivos: "y.txt" y "timeFile.txt". Para recabar los datos y tiempos respectivamente.

```
// Creación y apertura de archivos para almacenar datos y valores de tiempo
ofstream yfile;
yfile.open("y.txt");

ofstream timeFile;
timeFile.open("timeFile.txt");
```

Nuevamente se establece el número de iteraciones en 1000. Con este número de iteraciones se puede generar el bucle for, en el cual se utiliza la función serialGetchar para leer un carácter desde el puerto UART.

Se verifica que se haya leído un carácter válido antes de proceder a obtener la marca de tiempo actual y almacenar tanto la marca de tiempo como el dato recibido en los archivos correspondientes dentro del bucle.

```
int N = 1000; // Número de iteraciones

// Bucle para realizar N iteraciones
for (int i = 0; i < N; i++) {
  // Lectura de un carácter del UART
  int received_data = serialGetchar(fd_comm);

  // Verificación de que se haya leído un carácter válido
  if (received_data != -1) {
    // Obtención de la marca de tiempo actual en segundos
    timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    double timestamp_seconds = static_cast<double>(ts.tv_sec) + static_cast<double>(ts.tv_nsec) / 1000000000.0;

    // Almacenamiento de la marca de tiempo en el archivo timeFile
    timeFile << std::fixed << timestamp_seconds << endl;

    // Almacenamiento del dato recibido en el archivo yfile
    yfile << received_data << "\n";
  }
}
```

Finalmente, los archivos y el puerto UART se cierran para asegurar que los datos se guarden correctamente y se asegure que la comunicación terminó, y la función main devuelve 0 para indicar una finalización exitosa del programa.

```
// Cierre de los archivos y del puerto serie después de completar las iteraciones
yfile.close();
timeFile.close();
serialClose(fd_comm);

return 0;
```

- Sockets

Código:

Es importante especificar que para este tipo de comunicación se tenía planeado utilizar cuatro programas diferentes. Uno que emplea arduino para configurar los pines de la Raspeberry, un programa que emplee UART para escribir los valores seriales, un cliente y un servidor. Pero debido a las complicaciones que se nos presentaron durante la realización de los mismo optamos por tener solamente dos programas en los cuales se integran los cuatro programas inicialmente planteados. Fueron distribuidos de la siguiente forma: En un programa se combinaron el cliente con el lector de seriales, es decir, una vez que se reciben valores serieales mediante el arduino estos serán enviados de forma inmediata al servidor como sockets. Mientras que en el servidor se registran los tiempos de recepción de dichos datos.

Servidor:

Se incluyen las bibliotecas necesarias para la programación de sockets y para la entrada/salida estándar. Además, se define el puerto 8888 mediante el cual se hará la comunicación.

```
// Inclusión de bibliotecas necesarias para el programa
#include <iostream>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 8888 //Puerto a utilizar
```

Dentro de la función principal se crea un socket usando la función **socket**. Si la creación del socket falla, se muestra un mensaje de error y el programa se termina.

```
int main() {
    int socket_desc, new_socket, c;
    struct sockaddr_in server, client;
    char client_message;

    // Creación de un socket
    socket_desc = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_desc == -1) {
        printf("Could not create socket");
        return 1;
    }
```

Una vez creado el socket se configura la estructura del servidor con la dirección IP (INADDR_ANY), esto indica que el servidor aceptará conexiones en todas las interfaces de red, y el número de puerto.

```
// Configuración de la estructura del servidor
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(PORT);
```

Ya con la configuración realizada se utiliza la función **bind** para enlazar el socket al puerto y dirección especificados en la estructura del servidor. Si el enlace falla, se muestra un mensaje de error y el programa se termina.

```
// Enlace del socket al puerto y dirección especificados
if (bind(socket_desc, (struct sockaddr *)&server, sizeof(server)) < 0) {
    puts("bind failed");
    return 1;
}
puts("bind done");
```

Utilizamos la función **listen** para configurar el socket para aceptar conexiones entrantes con máximo 3 conexiones pendientes que se permitirán en la cola. También se utiliza la función accept para aceptar una conexión entrante. Si la aceptación falla, se muestra un mensaje de error y el programa se termina. Si tiene éxito, se imprime un mensaje indicando que la conexión ha sido aceptada.

```
// Configuración del socket para escuchar conexiones entrantes
listen(socket_desc, 3);

puts("Waiting for incoming connections...");
c = sizeof(struct sockaddr_in);

// Aceptación de una conexión entrante
new_socket = accept(socket_desc, (struct sockaddr *)&client, (socklen_t *)&c);
if (new_socket < 0) {
    perror("accept failed");
    return 1;
}
puts("Connection accepted");
```

Se utiliza un bucle while para recibir datos del cliente. Los datos recibidos se almacenan en la variable client_message y se imprime un mensaje indicando la recepción de datos del cliente.

```
// Bucle para recibir datos del cliente
while (recv(new_socket, &client_message, sizeof(client_message), 0) > 0) {
    std::cout << "Received data from client: " << client_message << std::endl;
}
```

Finalmente, después de salir del bucle, se verifica el estado de la conexión. Si new_socket es 0, significa que el cliente se ha desconectado. Si new_socket es -1, indica un fallo en la recepción de datos. Posterior a esta comprobación se indica si el programa terminó con éxito.

```
// Comprobación del estado de la conexión
if (new_socket == 0) {
    puts("Client disconnected");
    fflush(stdout);
} else if (new_socket == -1) {
    perror("recv failed");
}

return 0;
```

ClienteUART.

Para este programa se incluyen varias bibliotecas. iostream se utiliza para entrada/salida estándar, wiringPi y wiringSerial para la comunicación UART con WiringPi, y las bibliotecas de sockets para la comunicación de red.

```
// Inclusión de bibliotecas necesarias para el programa
#include <iostream>
#include <wiringPi.h>
#include <wiringSerial.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
```

Se utiliza la misma configuración para el puerto UART y la velocidad de comunicación. Además se específica la dirección IP y el número de puerto del servidor.

```
#define DEVICE_PORT "/dev/ttyACM0"
#define BAUD_RATE 9600
#define SERVER_IP "192.168.1.77" // IP del servidor
#define PORT 8888
```

En la función principal se inicializa la biblioteca WiringPi y se abre el puerto serie UART. Y se verifica si la inicialización y la apertura del puerto son exitosas.

```
int main(int argc, char **argv) {
    // Inicialización de la biblioteca WiringPi
    if (wiringPiSetup() == -1) {
        cerr << "Failed to initialize WiringPi library" << endl;
        return 1;
    }

    // Apertura del puerto serie UART
    int fd_comm = serialOpen(DEVICE_PORT, BAUD_RATE);
    if (fd_comm == -1) {
        cerr << "Failed to open UART communication" << endl;
        return 1;
    }
    cout << "UART communication successfully setup" << endl;
```

Después de esto se crea un socket y se configura con la dirección IP y el número de puerto del servidor.

```
    // Creación del socket
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        printf("Could not create socket");
        return 1;
    }

    // Configuración de la estructura del servidor
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr(SERVER_IP);
    server.sin_port = htons(PORT);
```

Una vez creado el socket se establece una conexión al servidor. Si la conexión falla, se muestra un mensaje de error y el programa se termina. Si tiene éxito, se imprime un mensaje indicando que la conexión al servidor se ha establecido.

```
// Conexión al servidor
if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
    perror("connect failed. Error");
    return 1;
}
cout << "Connected to server" << endl;
```

Establecemos un número de iteraciones N=1000 y utilizamos un bucle for para leer datos del puerto serie y enviarlos al servidor.

```
int N = 1000; // Número de iteraciones
char received_data;

// Bucle para leer datos del puerto serie y enviarlos al servidor
for (int i = 0; i < N; i++) {
    received_data = serialGetchar(fd_comm);
    if (received_data != -1) {
        // Envío de datos al servidor a través del socket
        send(sock, &received_data, sizeof(received_data), 0);
    }
}
```

Para finalizar se cierra el puerto serie y el socket. Y la función main devuelve 0 para indicar una finalización exitosa del programa.

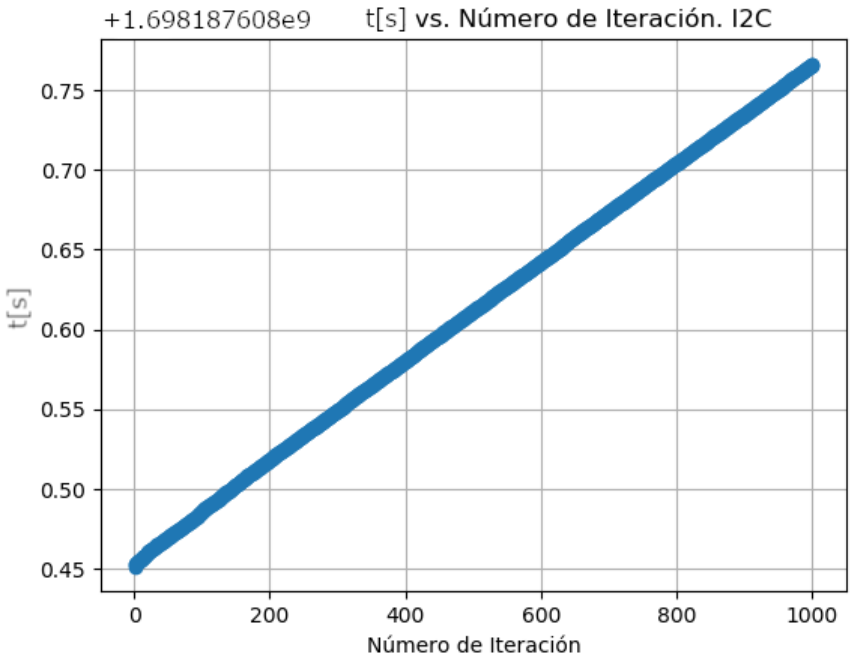
```
// Cierre del puerto serie y del socket después de completar las iteraciones
serialClose(fd_comm);
close(sock);

return 0;
```

Resultados

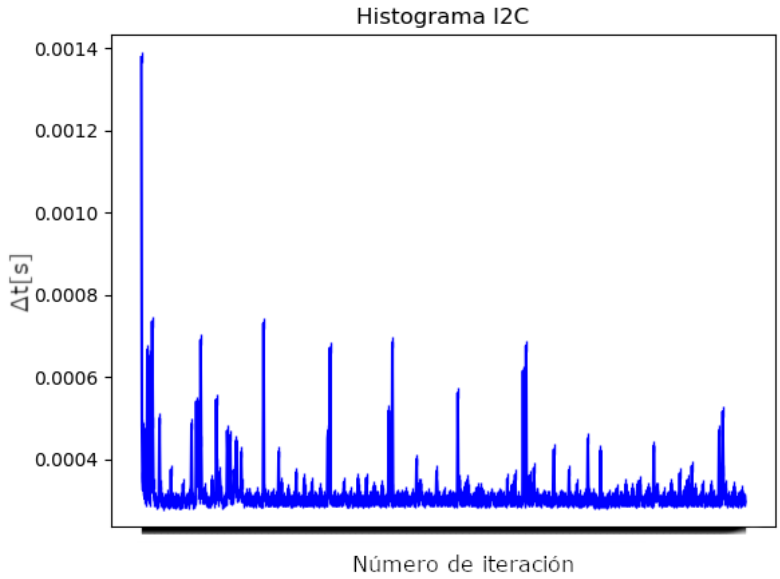
- Para I2C

Gráfica de Tiempo[s] vs Número de Iteraciónes:



En esta gráfica se muestra una relación lineal entre el número de iteraciones y el tiempo; la línea recta ascendente sugiere un incremento constante en el tiempo con cada iteración, un comportamiento esperado en una comunicación I2C adecuada.

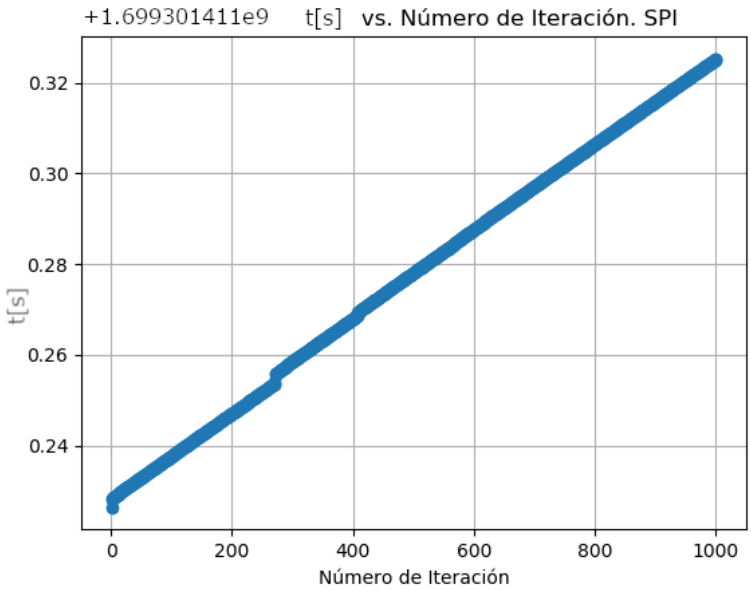
Histograma de frecuencias de Δt:



Este histograma muestra una distribución variable con múltiples picos, esto puede indicar que hay una serie de eventos que ocurren con diferente frecuencia, por lo que la comunicación no está siendo tan estable o que existen problemas de temporización. Como se presenta una mayor variabilidad en los eventos que se están registrando, queda en evidencia la naturaleza de la comunicación I2C, que a menudo maneja múltiples dispositivos en el mismo bus y podría tener una variabilidad en las respuestas de los dispositivos o en la longitud de los datos transmitidos. La naturaleza de las transacciones I2C, que pueden incluir la comunicación con dispositivos y la lectura de sensores, podría hacer que los datos registrados variasen.

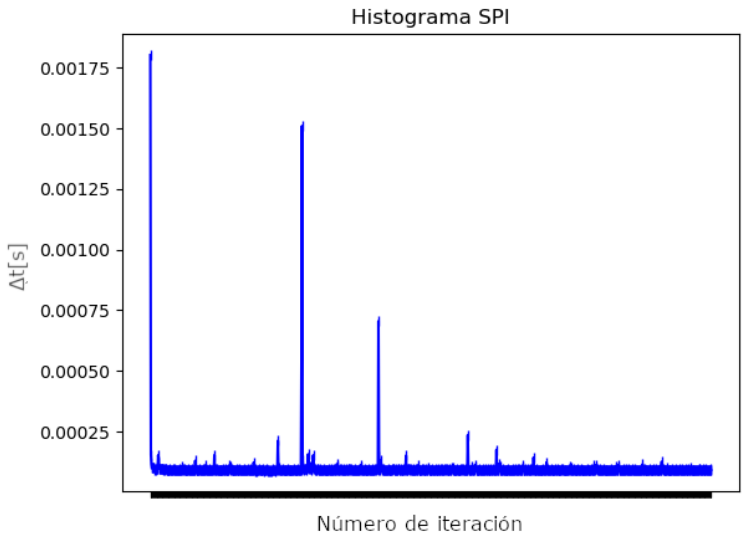
- Para SPI

Gráfica de Tiempos vs Iteraciones:



Al igual que en la gráfica anterior, se muestra una relación proporcional y constante entre el tiempo y las iteraciones. Además, tomando en cuenta que la comunicación SPI es conocida por ser más rápida que la UART, podemos explicar por qué la pendiente puede parecer más pronunciada mostrando valores mucho más pequeños en Y (indicando que la comunicación se realiza más rápidamente).

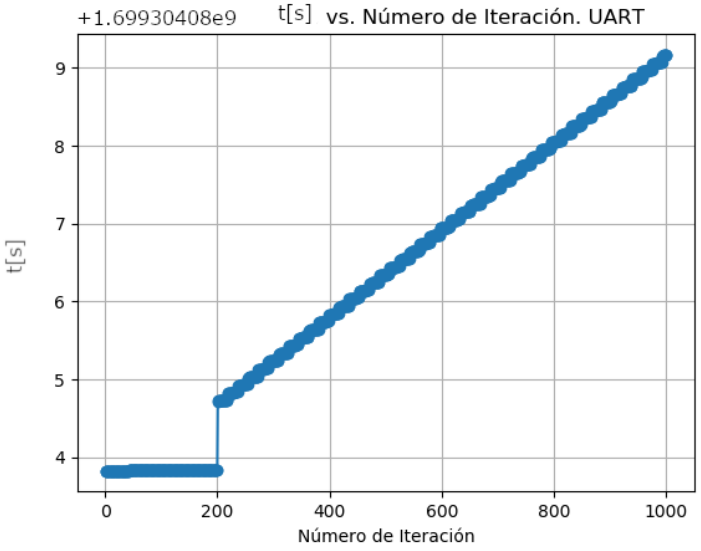
Histograma de frecuencias de Δt:



En esta ocasión, el histograma muestra una distribución más estable con unos cuantos picos, lo que puede significar que existe una consistencia en los tiempos entre las operaciones SPI. Esto nos da una idea de cómo se distribuyen los intervalos de tiempo entre las transacciones SPI. Al tener una distribución con picos agudos y estrechos podemos decir que tenemos una comunicación eficiente y uniforme.

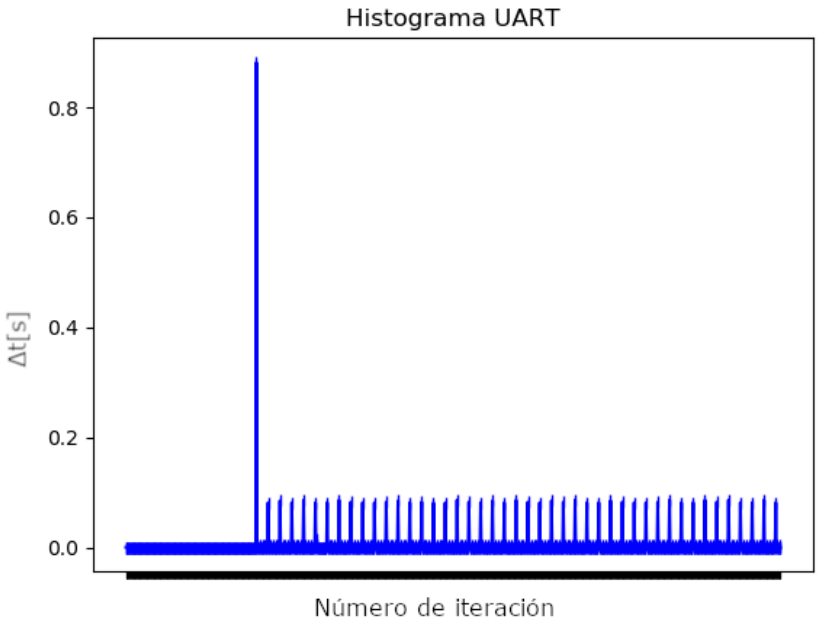
- Para UART

Gráfica de Tiempos vs Iteraciones:



Al igual que en los dos casos anteriores, la gráfica muestra una relación lineal entre el número de iteraciones y el tiempo transcurrido en segundos. Una particularidad de esta gráfica es el valor constante al principio antes de comenzar la tendencia lineal, lo que podría indicar una fase de inicialización o un periodo donde la comunicación o la recolección de datos no ha comenzado a cambiar.

Histograma de frecuencias de Δt:

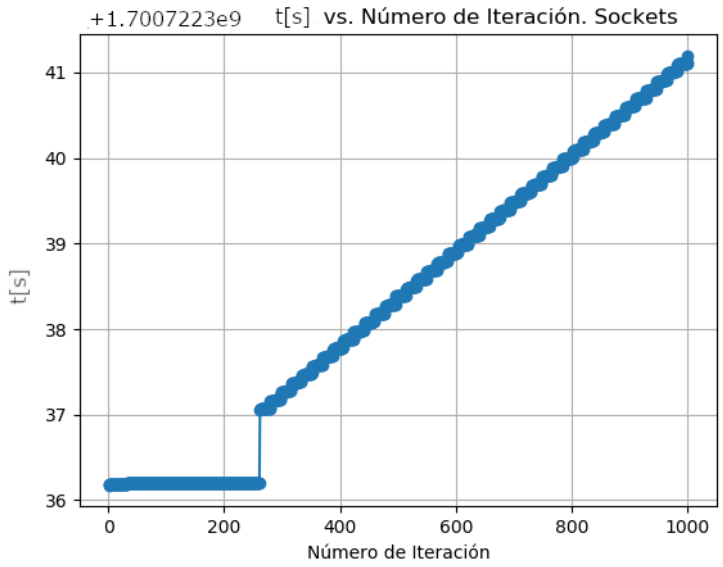


En este histograma hay un pico muy alto al principio, lo que sugiere que hay una gran cantidad de tiempo acumulado en este punto inicial. Esto podría indicar que hay una condición o un evento que ocurre con mucha más frecuencia que cualquier otro. En el contexto de la comunicación UART, esto podría representar una acumulación de intentos de comunicación o errores que ocurren al inicio del proceso. Después de este pico inicial, la distribución cae abruptamente y luego muestra una serie de valores muy bajos y constantes para los índices subsiguientes, lo que podría indicar que después del evento inicial, el comportamiento se vuelve uniforme.

Al observar el histograma junto con la gráfica anterior, se observa que ambas muestran que al iniciar la comunicación UART, hay un evento significativo (tal vez un tiempo de inicialización o un gran volumen de datos procesados inicialmente) seguido de un proceso de comunicación más estable y predecible. Esto es común en sistemas de comunicación donde la configuración inicial toma más tiempo o recursos antes de entrar en un estado de operación continua y regular.

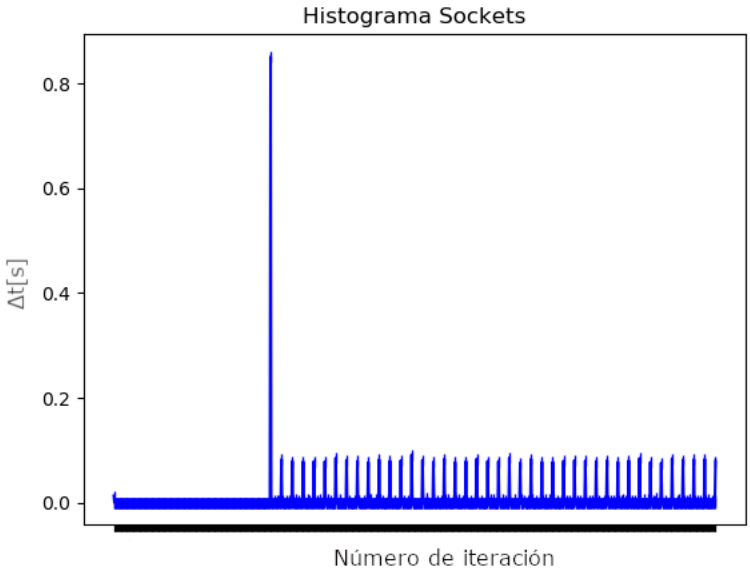
- Para Sockets:

Gráfica de Tiempos vs Iteraciones:



Una vez más observamos un comportamiento lineal de hecho esta gráfica es muy similar a la de comunicación UART. La principal diferencia es que se presenta un mayor número de iteraciones con un valor de tiempo constante al inicio, esto puede significar que al utilizar sockets el tiempo de inicialización resulta ser un poco más lento. El hecho de que esta gráfica sea muy parecida a la de UART no es ninguna sorpresa, ya que para implementar los sockets utilizamos comunicación UART.

Histograma de frecuencias de Δt:



Al igual que en el histograma de comunicación UART, al utilizar sockets se sigue presentando un pico pronunciado al inicio pero una seria de picos que se repiten de manera constante, aunque un poco más altos que los del histograma anterior. Por lo que, al emplear sockets sigue existiendo una medida que domina la comunicación, siendo esta bastante estable a lo largo del tiempo.

NOTA:

Las conclusiones son presentadas de manera individual al profesor.

Referencias

- WiringPi. GPIO Interface library for the Raspberry Pi. Recuperado de http://wiringpi.com/
- Free Software Foundation, Inc. *Getting the Time.* En *The GNU C Library.* Recuperado de **https://www.gnu.org/software/libc/manual/html_node/Getting-the-Time.html**
- Strickland, J. R., & Strickland, J. R. (2018). Meet WiringPi. *Raspberry Pi for Arduino Users: Building IoT and Network Applications and Devices*, 179-211.
- The Robotics Back End (s.f.) *Raspberry Pi 4 Pins – Complete Practical Guide.* Recuperado de: https://roboticsbackend.com/raspberry-pi-3-pins/
- The Robotics Back End (s.f.) *Arduino Uno Pins – A Complete Practical Guide.* Recuperado de: https://roboticsbackend.com/arduino-uno-pins-a-complete-practical-guide/
- The Robotics Back End (s.f.) *Raspberry Pi (master) Arduino (slave) I2C communication with WiringPi.* Recuperado de: https://roboticsbackend.com/raspberry-pi-master-arduino-slave-i2c-communication-with-wiringpi/
- The Robotics Back End (s.f.) *Raspberry Pi (master) Arduino Uno (slave) SPI communication with WiringPi.* Recuperado de: https://roboticsbackend.com/raspberry-pi-master-arduino-uno-slave-spi-communication-with-wiringpi/
- The Robotics Back End (s.f.) *Raspberry Pi Arduino Serial Communication – Everything You Need To Know.* Recuperado de: https://roboticsbackend.com/raspberry-pi-arduino-serial-communication/
- The Robotics Back End (s.f.) *Sockets in C/C++ for Raspberry Pi. Recuperado de:* https://github.com/Mad-Scientist-Monkey/sockets-ccpp-rpi
- Bagur, J., Chung, T., Söderby, K. (November 15, 2023) *Powering Alternatives for Arduino Boards.* Recuperado de: https://docs.arduino.cc/learn/electronics/power-pins#choosing-a-power-input
- Geek Factory (September 28, 2017) *Alimentar el Arduino: La guía definitiva.* Recuperado de: https://www.geekfactory.mx/tutoriales-arduino/alimentar-el-arduino-la-guia-definitiva/
- Next Hack (February 15, 2020) *How to interface a 3.3V output to a 5V input.* Recuperado en: https://next-hack.com/index.php/2020/02/15/how-to-interface-a-3-3v-output-to-a-5v-input/
- Next Hack (September 15, 2017) *5V to 3.3V logic level translation/conversion/shifting: how to interface a 5V output to a 3.3V input*. Recuperado en: https://next-hack.com/index.php/2017/09/15/how-to-interface-a-5v-output-to-a-3-3v-input/