_ TELECOM ESCUELA
TÉCNICA VLC SUPERIOR
DE INGENIERÍA DE
TELECOMUNICACIÓN

Diseño e implementación de un guante interactivo para la interpretación de lenguaje de signos.

Malory Gallardo Abad

Tutor: Germán Ramos Peinado

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2020-21

Valencia, 30 de Noviembre de 2020

Resumen

El presente trabajo final de grado tiene como finalidad el diseño y la implementación de un guante interactivo preparado para la interpretación de lenguaje de signos. El dispositivo contará con diversos sensores como acelerómetros, giroscopios y de flexión en los dedos y en la palma de la mano. Estos datos serán adquiridos en tiempo real y preprocesados por un sistema ESP32 que se encargará de enviarlos por WiFi a un dispositivo de cómputo (Raspberry Pi), que será capaz de ejecutar los posteriores algoritmos de clasificación para el reconocimiento de los signos. Estos signos serán mostrados a través de un servicio web, transmitidos a cualquier dispositivo móvil.

Resum

El present treball final de grau te com a finalitat el disseny i la implementació d'un guant interactiu preparat per a l'interpretació de llenguatge de signes. El dispositiu comptarà amb diversos sensors com acceleròmetres, giroscopis i de flexió en els dits i a la palma de la mà. Aquestes dades seran adquirides en temps real i pre-processats per un sistema ESP32 que s'encarregarà de enviar-los per WiFi a un dispositiu de còmput (Raspberry Pi), que serà capaç d'executar els posteriors algoritmes de classificació per al reconeiximent dels signes. Aquests signes es mostraran mitjançant un servei web, transmesos a qualsevol dispositiu mòbil.

Abstract

This Bachelor's thesis aims to design and implement an interactive glove prepared for the interpretation of sign language. The device will have several sensors such as accelerometers, gyroscopes and bending in the fingers and palm. These data will be acquired in real time and pre-processed by an ESP32 system that will send them via WiFi to a computer device (Raspberry Pi), which will be able to execute the subsequent classification algorithms for the recognition of the signs. These signs will be displayed through a web service, transmitted to any mobile device.

Índice general

I Memoria

1. Introducción y objetivos **1**

2. Metodología **3**
- 2.1. Gestión del proyecto . . . 3
- 2.2. Distribución en tareas . . . 4
- 2.3. Diagrama temporal . . . 4

3. Planteamiento técnico **5**
- 3.1. ESP32 . . . 5
  - 3.1.1. Módulo ESP32-WROOM-32 . . . 7
  - 3.1.2. Placa de desarrollo . . . 8
- 3.2. Sensores . . . 9
  - 3.2.1. MPU9250 . . . 9
    - 3.2.1.1. Protocolo I2C . . . 10
  - 3.2.2. Sensor flexible . . . 10
- 3.3. Raspberry Pi . . . 11

4. Desarollo y resultados **13**
- 4.1. Herramientas de desarrollo . . . 13
  - 4.1.1. PlatformIO . . . 13
  - 4.1.2. Code::Blocks/Eclipse . . . 15
- 4.2. Adquisición de datos . . . 15
  - 4.2.1. MPU-9250 . . . 15
  - 4.2.2. Sensor flexible . . . 18
    - 4.2.2.1. Calibración . . . 19
- 4.3. Comunicación ESP32-RPI . . . 21
  - 4.3.1. Configuración de la Raspberry como punto de acceso (AP) . . . 21
  - 4.3.2. Envío de datos . . . 23
    - 4.3.2.1. Conexión WiFi . . . 23
    - 4.3.2.2. Estructura de datos . . . 24
    - 4.3.2.3. Cliente TCP . . . 26
  - 4.3.3. Recepción de datos . . . 27
- 4.4. Servidor Web . . . 30
  - 4.4.1. Demo . . . 32
- 4.5. Diseño del guante interactivo . . . 33

4.5.1. Esquemático protoboard 33
4.5.2. Guante interactivo 34

5. Presupuesto **37**

6. Conclusión y líneas futuras **39**

Bibliografía **40**

II Anexos

A. Código principal **42**

B. Servidor TCP y Servidor Web **45**

Índice de figuras

1.1. Diagrama global 2

2.1. Tablero de Trello 3
2.2. Diagrama de Gantt 4

3.1. Diagrama de bloques ESP32 6
3.2. Módulos ESP32 7
3.3. ESP32-WROOM-32 7
3.4. AZDelivery DevKit V4 8
3.5. Pinout del módulo ESP32-WROOM-32 8
3.6. MPU-9250 9
3.7. Sensor flexible 11
3.8. Esquemático sensor flexible 11
3.9. Raspberry Pi 3 Modelo B 12

4.1. Logo PlatformIO 14
4.2. Archivo platformio.ini 14
4.3. Logo Code::Blocks 15
4.4. Logo Eclipse 15
4.5. Conexión MPU-9250 - ESP32 16
4.6. Conexión Sensor flexible - ESP32 18
4.7. Disposición de la red 22
4.8. Comprobación hostapd 23
4.9. Datos recibidos en el servidor TCP 29
4.10. Systemctl status 30
4.11. Gráficas servidor web 31
4.12. Gŕaficas servidor web 31
4.13. Demo página web 32
4.14. Esquemático protoboard 33
4.15. Guante interactivo (1) 34
4.16. Guante interactivo (2) 35
4.17. Guante interactivo (ESP32) 35
4.18. Guante interactivo (Sensor MPU-9250) 36
4.19. Guante interactivo (Sensor de flexión) 36

Parte I

Memoria

Capítulo 1

Introducción y objetivos

Este proyecto partió de la idea inicial de crear un guante que fuera capaz de interpretar el lenguaje de signos utilizando dispositivos y elementos de bajo coste. En el momento de analizar las partes en las que se dividiría el proyecto para llevarlo a cabo, resultó ser un trabajo muy extenso y complejo si se quería conseguir una interpretación completa y en tiempo real del lenguaje de signos en base a los datos capturados desde el guante. Tras este análisis, se decidió finalmente abordar el diseño del guante interactivo y la implementación de un sistema preparado para llevarlo a cabo, sin pasar a abordar la compleja interpretación del lenguaje de signos.

El guante interactivo estará compuesto por una serie de sensores, concretamente un sensor de flexión en cada dedo y un acelerómetro y giroscopio en la parte central de la mano. Además del diseño del guante, el sistema a realizar consistirá en dos partes fundamentales: la adquisición de datos desde un microcontrolador y la comunicación para el envío de esos datos a un dispositivo de cómputo externo que trabajará como servidor de los datos a terceros.

En primer lugar, los datos que ofrezca cada sensor serán adquiridos en tiempo real por un dispositivo ESP32, para después ser enviados a una Raspberry Pi. Para llevar a cabo la comunicación entre estas dos partes, primero será necesario crear la red entre los dos dispositivos, de forma que la Raspberry actuará como punto de acceso y el ESP32 se conectará a este punto mediante conexión WiFi. El siguiente paso será establecer la comunicación, y por tanto el envío de los datos. Para ello, se implementará un cliente TCP en el ESP32 y un servidor TCP en la Raspberry. Una vez recibidos los datos en la Raspberry, esta se encargará de mostrarlos en un servicio web en forma de gráficas de manera que vayan mostrando la variación de los valores en tiempo real. Además, se hará una pequeña demostración mostrando algunos gestos sencillos.

En la siguiente imagen se muestra un diagrama global del sistema completo que se va a llevar a cabo, en la que se indica además los lenguajes de programación y tecnologías empleadas en cada parte:

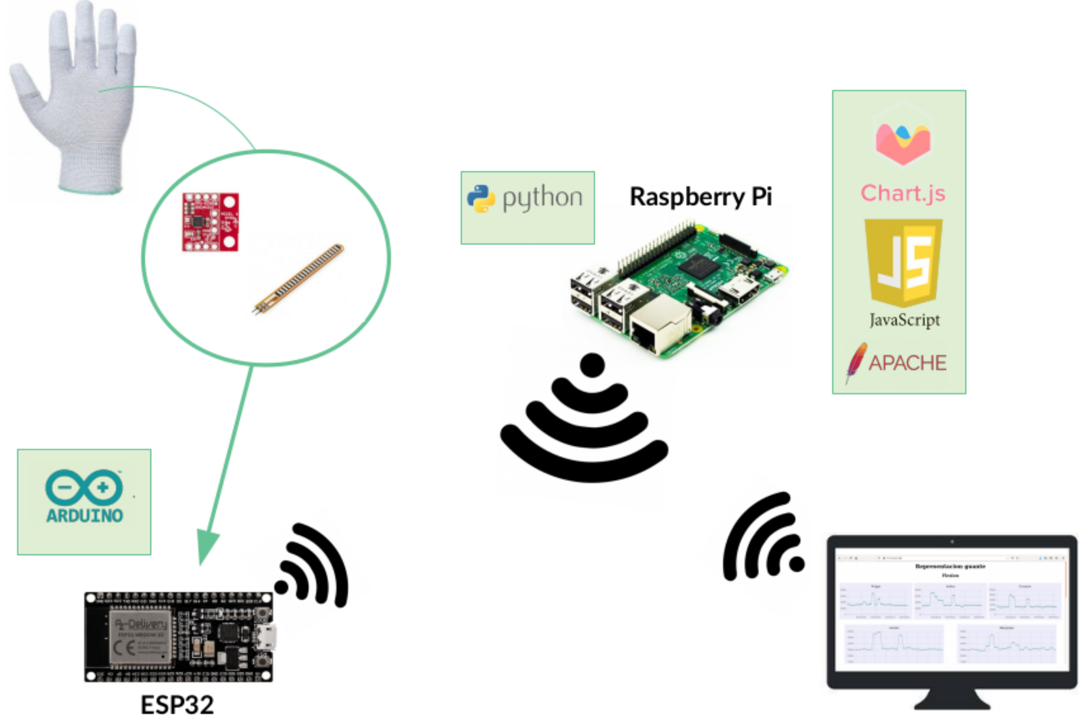

Figura 1.1: Diagrama global

Una vez visto el procedimiento que se va a llevar a cabo para la realización del sistema, se pueden plantear los siguientes objetivos:

- Diseño del guante interactivo
- Gestión de los datos procedentes de los sensores
- Comunicación para el envío de datos entre el microcontrolador ESP32 y la Raspberry Pi
- Presentación de los datos en un Servidor Web

A través del desarrollo de los objetivos concretos del proyecto, se profundizará en la programación de microcontroladores y en el uso de linux, así como en la adquisición de datos y el conocimiento de sensores. A la vez se abordará el desarrollo web y el lenguaje de programación JavaScript y las posibilidades que ofrece.

Capítulo 2

Metodología

2.1. Gestión del proyecto

Una de las herramientas utilizadas para la gestión del proyecto ha sido **Trello** [1]. Esta herramienta ha sido de gran ayuda para organizar las tareas que iban surgiendo y se iban resolviendo e ir organizando las páginas de donde se ha ido recopilando información, ya sea información técnica de los elementos que componen el proyecto como tutoriales para llevar a cabo algunas partes.

Figura 2.1: Tablero de Trello

Además de Trello, también **Git** [2] ha formado parte de la gestión del proyecto. Esta herramienta de control de versiones es muy útil para cualquier proyecto, ya que permite volver atrás si se han añadido cosas y algo falla, o para hacer comprobaciones de ciertas partes del proyecto. Aunque sobretodo está pensada para proyectos más complejos, con más archivos e incluso más personas trabajando a la vez en el mismo proyecto, ha sido útil en momentos de modificar partes y que surjan fallos, para poder volver atrás y solucionarlo.

2.2. Distribución en tareas

Para llevar a cabo el trabajo se ha dividido en las siguientes tareas:

- Tarea 1: Investigación sobre el microcontrolador ESP32
- Tarea 2: Investigación sobre los sensores flexibles y el sensor MPU9250
- Tarea 3: Configuración de la Raspberry como punto de acceso
- Tarea 4: Búsqueda del entorno de desarrollo
- Tarea 5: Puesta a punto del entorno de trabajo
- Tarea 6: Conexión WiFi entre los dispositivos ESP32 y Raspberry
- Tarea 7: Lectura de un único sensor MPU9250
- Tarea 8: Lectura de un único sensor flexible
- Tarea 9: Lectura de múltiples sensores flexibles y un MPU9250
- Tarea 10: Envío y recepción de los datos desde el ESP32 a la Raspberry
- Tarea 11: Presentación de datos en Servidor Web
- Tarea 12: Calibración de los sensores flexibles
- Tarea 13: Diseño del guante interactivo
- Tarea 14: Memoria del proyecto

2.3. Diagrama temporal

La siguiente imagen muestra la planificación temporal llevada a cabo mediante un diagrama de Gantt:

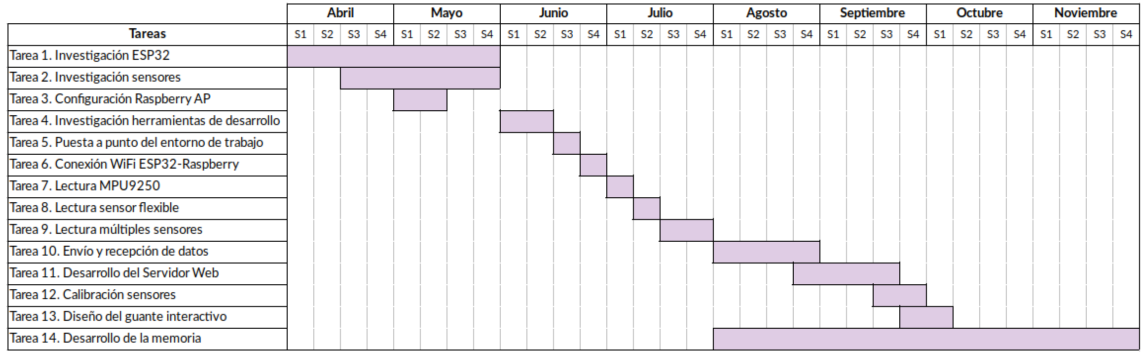

Figura 2.2: Diagrama de Gantt

Capítulo 3

Planteamiento técnico

En este capítulo se van a detallar los dispositivos y sensores que han formado parte del proyecto. En primer lugar, se eligió utilizar un ESP32, que como se verá a continuación, ofrece multitud de características a un precio realmente bajo. También se hará uso de una placa Raspberry Pi ya que se disponía de una y, además, cumple de sobra con las necesidades del proyecto.

En cuanto a los sensores, ya que el objetivo del proyecto es el diseño de un guante interactivo, se pensó que sería muy buena idea utilizar sensores flexibles ya que pueden dar mucha información acerca del movimiento que hacen los dedos de forma individual y de qué forma están flexionados. También, como se ha comentado anteriormente, se hará uso de un acelerómetro con giroscopio que irá en el centro de la mano. Este sensor ayudará a conocer la posición de la mano, por lo que también será importante conocer esos valores.

3.1. ESP32

El **ESP32** [3] es un SoC (System on Chip) de bajo coste y consumo de energía diseñado por Espressif Systems y fabricado por TSMC(Taiwan Semiconductor Manufacturing Company). Estos chips disponen de la tecnología Wi-Fi y Bluetooth integrada y emplean un microprocesador Tensilica Xtensa LX6 de simple o doble núcleo, según el módulo.

Este SoC es el sucesor del modelo ESP8266 [4], que añade muchas funciones y mejoras respecto a este último, como una mayor potencia, Bluetooth 4.0, cifrado por hardware, sensores, reloj de tiempo real, más buses, más puertos... En resumen, tiene un gran potencial para desarrollar proyectos, sobretodo en el sector de IoT. Aunque su lanzamiento no es muy lejano (año 2016), cada vez hay más artículos y productos comerciales que emplean el ESP32 como núcleo, sin embargo actualmente hay más artículos y proyectos sobre el ESP8266, ya que su precio es más bajo y lleva más tiempo en el mercado.

A continuación, se verá de forma más detallada las características principales de este SoC y su diagrama de bloques (3.1):

- Procesadores
    - Procesador principal: Tensilica Xtensa LX6 32-bit. Puede tener uno o dos núcleos según el módulo, con una frecuencia de reloj de hasta 240 MHz.
    - Coprocesador ULP (Ultra Low Power), que permite conversiones ADC, conexiones I2C y cálculos en el modo *deep sleep*.
- Conectividad inalámbrica
    - Wi-Fi: 802.11
    - Bluetooth: v4.2 y BLE (Bluetooth Low Energy).
- Memoria interna
    - ROM de 448 KB, con las instrucciones para el arranque.
    - SRAM de 520 KB, para datos e instrucciones.

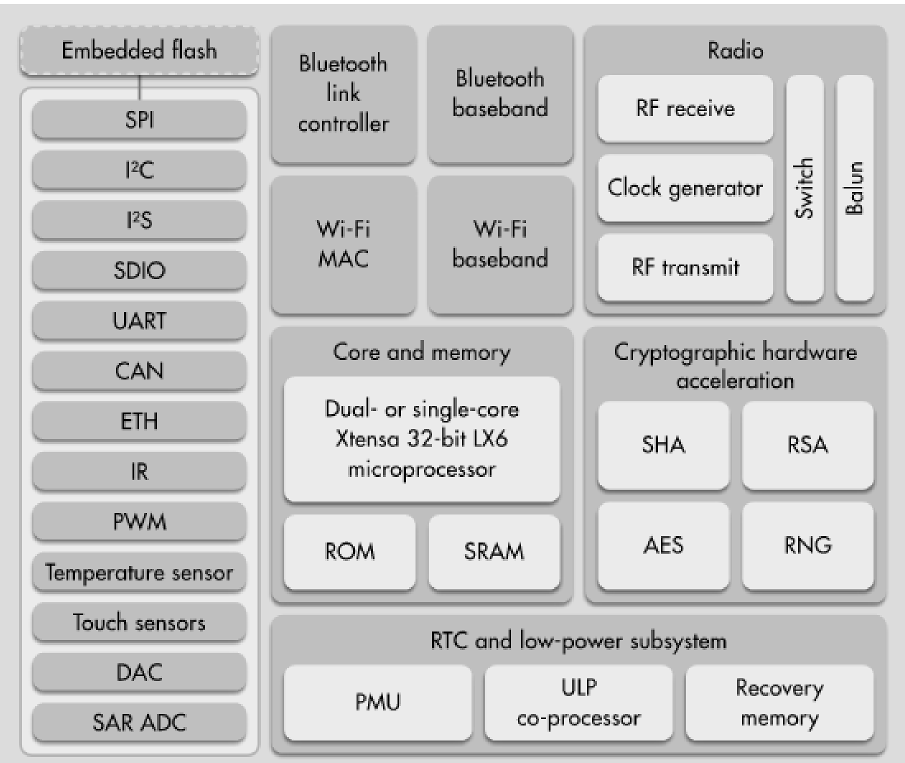

Figura 3.1: Diagrama de bloques ESP32

Hay diferentes módulos desarrollados por Espressif Systems, y para este trabajo se utilizará el módulo **ESP32-WROOM-32** [5]. Los principales módulos ESP32 se pueden ver en la siguiente imagen (3.2):

Module	Chip	Flash, MB	PSRAM, MB	Ant.	Dimensions, mm
ESP32-WROOM-32	ESP32-D0WDQ6	4	–	MIFA	18 × 25.5 × 3.1
ESP32-WROOM-32D	ESP32-D0WD	4, 8, or 16	–	MIFA	18 × 25.5 × 3.1
ESP32-WROOM-32U	ESP32-D0WD	4, 8, or 16	–	U.FL	18 × 19.2 × 3.1
ESP32-SOLO-1	ESP32-S0WD	4	–	MIFA	18 × 25.5 × 3.1
ESP32-WROVER (PCB)	ESP32-D0WDQ6	4	8	MIFA	18 × 31.4 × 3.3
ESP32-WROVER (IPEX)	ESP32-D0WDQ6	4	8	U.FL	18 × 31.4 × 3.3
ESP32-WROVER-B	ESP32-D0WD	4, 8, or 16	8	MIFA	18 × 31.4 × 3.3
ESP32-WROVER-IB	ESP32-D0WD	4, 8, or 16	8	U.FL	18 × 31.4 × 3.3

Figura 3.2: Módulos ESP32

3.1.1. Módulo ESP32-WROOM-32

Se trata del primer módulo que salió al mercado, con 4 MB de memoria flash. Incluye el chip ESP32-DOWDQ6 que indica que tiene doble núcleo (*) y antena MIFA.

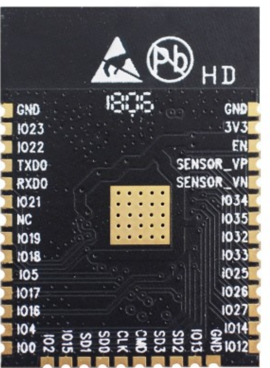

Figura 3.3: ESP32-WROOM-32

(*) ESP32-**D** indica que es de doble núcleo, ESP32-**S** indica que es un único núcleo.

3.1.2. Placa de desarrollo

Hay mucha variedad de placas de desarrollo que integran el SoC ESP32. Este trabajo ha sido desarrollado con la placa DevKit V4 de AzDelivery por sus amplias capacidades y bajo precio. En la imagen 3.5 se muestra el *pinout* del módulo.

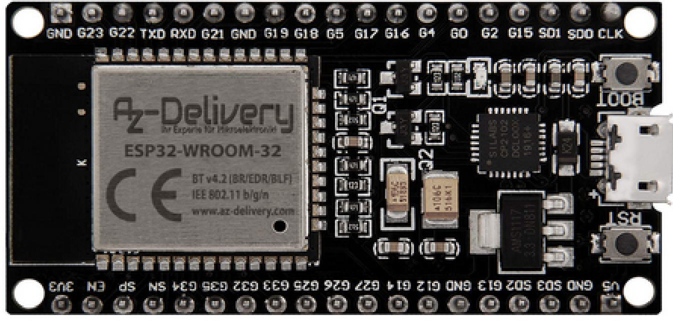

Figura 3.4: AZDelivery DevKit V4

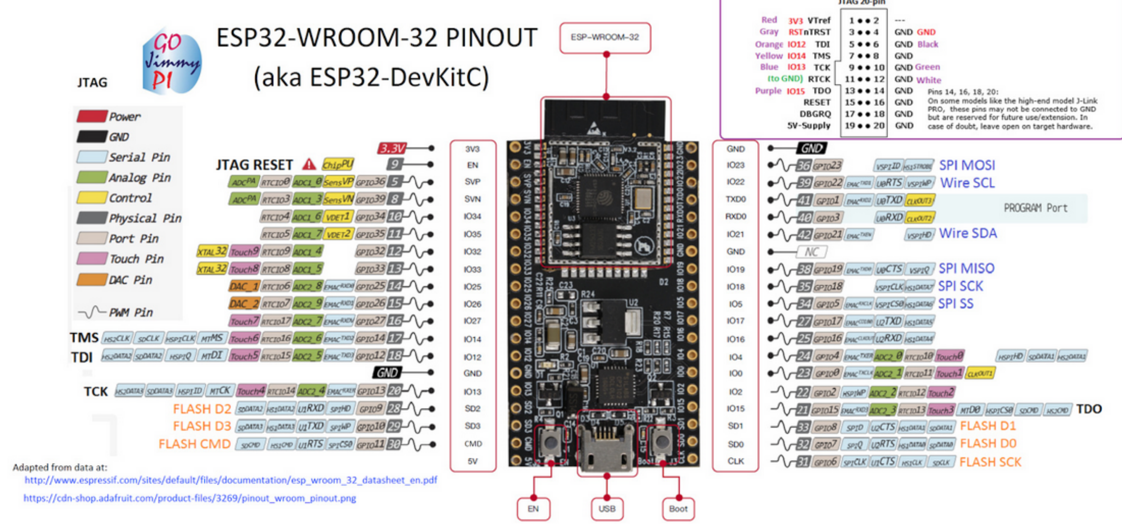

Figura 3.5: Pinout del módulo ESP32-WROOM-32

3.2. Sensores

Antes de estudiar las características de los sensores elegidos para llevar a cabo el diseño del guante, se hará una breve comparación de tres acelerómetros diferentes y la justificación de cuál se ha decidido elegir.

En un primer momento, habían tres posibles opciones: el sensor MPU-6050, el MPU-9150 y el MPU-9250. El primero integra acelerómetro y giroscopio, mientras que los otros dos integran además magnetómetro. Aunque son sensores muy similares, cuanta más información proporcionen mejor se podrán clasificar los gestos, por lo que el sensor MPU6050 se descartó. El sensor MPU-9250 es el sucesor del MPU-9150, ofreciendo una reducción de consumo del 44 % respecto a éste. Por lo que finalmente se decidió utilizar el sensor MPU-9250.

3.2.1. MPU9250

El sensor MPU-9250 combina en un solo chip un sensor MPU-6500 que contiene un acelerómetro junto con un giroscopio de 3 ejes cada uno y un sensor AK8963 que contiene un magnetómetro de 3 ejes. El sensor que se usará en este proyecto es de Sparkfun, mostrado en la imagen 3.6.

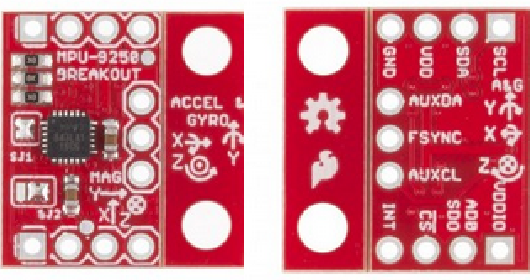

Figura 3.6: MPU-9250

A continuación se describen brevemente cada uno de los pines del módulo MPU9250:

- SCL/SCLK: (*Serial Clock Line*). Línea de la señal de reloj del protocolo I2C y SPI, respectivamente.
- SDA/SDI: (*Serial Data Line/Serial Data Input*). Línea de datos del protocolo I2C; Línea de datos de entrada del protocolo SPI.
- VDD: Tensión de alimentación. (en nuestro caso 3,3 V)
- GND: Tensión de referencia.
- AUXDA: Línea maestra de datos para conexiones a sensores externos.
- FSYNC: Entrada digital de sincronización.
- AUXCL: Línea maestra de la señal de reloj para conexiones a sensores externos.
- VDDIO: Tensión de suministro de I/O digitales.

- AD0/SDO: AD0 para el protocolo I2C es utilizado para cambiar la dirección del módulo si se utilizaran varios acelerómetros. En el caso de que el pin se encuentre a nivel bajo la dirección será 0x68, mientras que si está a nivel alto la dirección será 0x69. SDO (*Serial Data Output*) sería la línea de datos de salida del protocolo SPI.
- $\overline{CS}$: Chip Select. (protocolo SPI)
- INT: Salida digital de interrupción.

Para llevar a cabo la comunicación entre el sensor MPU-9250 y el ESP32 podrían usarse dos tipos de protocolos diferentes: I2C y SPI. Para este proyecto el elegido ha sido I2C, como se explica en el siguiente apartado.

3.2.1.1. Protocolo I2C

El protocolo I2C (Inter-Integrated Circuit) es un bus serie de datos desarrollado en 1982 por Phillips Semiconductors utilizado principalmente para la comunicación de baja velocidad entre diferentes partes de un circuito. Una de las características más importantes de I2C es su simplicidad, ya que utiliza únicamente dos líneas de datos bidireccionales, SDA (Serial Data) y SCL (Serial Clock), además de conexión a VDD y GND, mientras que el protocolo SPI utiliza cuatro (SCLK, SDI, SDO, $\overline{CS}$, además de VDD y GND). Otra característica clave es el mecanismo de la señal de reloj, ya que una única señal de reloj mantiene sincronizados a todos los dispositivos del bus (bus síncrono). De esta forma, se elimina la necesidad de que cada dispositivo tenga su propio reloj y de tener que acordar una velocidad de transmisión. La máxima velocidad de transmisión del bus I2C en el sensor MPU-9250 es de 400 kHz, en contraste con el protocolo SPI es menor, ya que este transmite a 1 MHz.

Por lo tanto, por su simplicidad y ya que una elevada velocidad de transmisión no es un requisito en este proyecto, el protocolo I2C ha sido el elegido para realizar la comunicación entre el sensor MPU-9250 y el ESP32.

3.2.2. Sensor flexible

Los sensores flexibles, también llamados sensores resistivos o de flexión, miden la cantidad de desviación causada por la flexión del sensor. Este sensor incrementa su resistencia al ser flexionado, por lo que está pensado sobretodo para entornos donde se utilicen guantes inteligentes, aunque también puede ser usado como sensor en puertas, accesorios para robots... Es una tecnología patentada por Spectra Symbol que fue utilizada en el *Power Glove* [6] de Nintendo [7].

Este sensor mide aproximadamente 7 cm de largo, y está diseñado para flexionarlo únicamente por un lado. Este lado está compuesto por una tinta polimérica con partículas conductoras incrustadas en él. Cuando el sensor está recto, tiene aproximadamente 30 kΩ, y conforme se dobla estas partículas se van alejando de forma que esta resistencia aumenta hasta llegar aproximadamente a 80 kΩ a 90 º.

Figura 3.7: Sensor flexible

Combinando este sensor con una resistencia fija para crear un divisor de tensión se puede obtener un valor de tensión variable que pueda ser leído por el convertidor analógico-digital (ADC) de un microcontrolador:

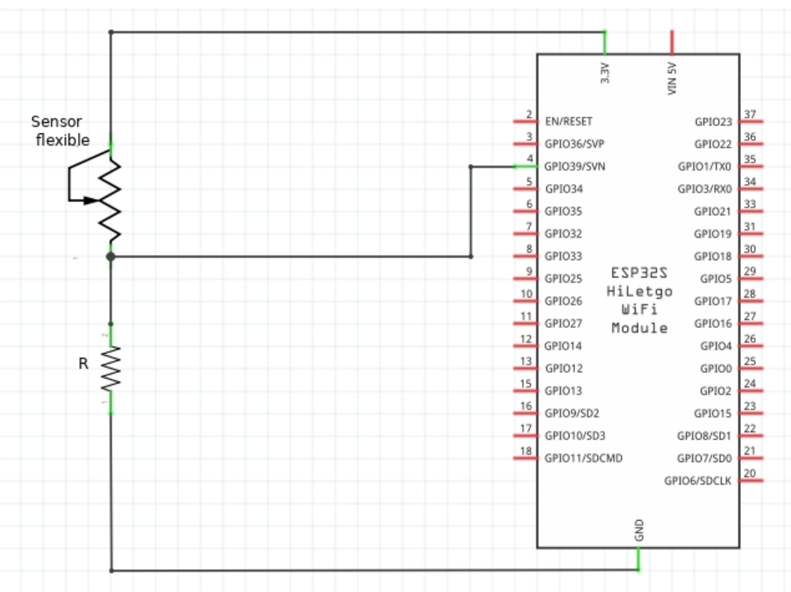

Figura 3.8: Esquemático sensor flexible

3.3. Raspberry Pi

Raspberry Pi [8] es un pequeño ordenador de bajo coste desarrollado en el Reino Unido por la Fundación Raspberry Pi, con el objetivo de estimular la enseñanza de informática y programación en las escuelas. Sin embargo, la versatilidad que ofrece y el bajo coste ha dado lugar a que se utilicen en multitud de proyectos, especialmente en el ámbito del IoT, que requiere dispositivos con capacidad de cómputo no elevada.

Aunque este dispositivo no es esencial en el proyecto, ya que podría haberse empleado un ordenador, se ha decidido utilizarlo ya que, como se ha mencionado anteriormente, se disponía de una y además se quería alcanzar uno de los objetivos comentados en la introducción: profundizar en el uso de linux.

El modelo que va a ser utilizado en este proyecto es Raspberry Pi 3 Modelo B, que cuenta con las siguientes características:

- Procesador: Broadcom BCM2837, Cortex-A53 (ARMv8) 64-bit SoC
- Soporta sistemas operativos GNU/Linux
- Frecuencia de reloj 1,2 GHz
- GPU VideoCore IV 400 MHz
- Memoria 1GB LPDDR2 SDRAM
- Conectividad inalámbrica 2.4GHz IEEE 802.11.b/g/n y Bluetooth 4.1
- Conectividad de red Fast Ethernet 10/100 Gbps
- 40 GPIO pines
- HDMI, 4xUSB 2.0 y más puertos

Figura 3.9: Raspberry Pi 3 Modelo B

Capítulo 4

Desarollo y resultados

4.1. Herramientas de desarrollo

Antes de comenzar a explicar cómo se han ido desarrollando cada una de las partes del proyecto, es conveniente conocer los entornos de desarrollo utilizados para llevarlo a cabo:

4.1.1. PlatformIO

PlatformIO [9] es un entorno de desarrollo de código abierto especialmente para sistemas embebidos y entornos IoT. Internamente está escrito en Python, y aunque abstrae de toda la configuración detrás del *framework* permite configurar prácticamente todos los elementos de compilación y de escritura en *flash*. Actualmente soporta:

- 40 plataformas
- 23 *frameworks*
- 913 placas de desarrollo
- 222 ejemplos de código
- 10.548 bibliotecas

En sí mismo es una aplicación de consola, pero puede ser usada junto a uno de los muchos IDEs compatibles que ofrece (Eclipse, VSCode, Vim, CodeBlocks, Visual Studio, Atom...).

Figura 4.1: Logo PlatformIO

Para inicializar un proyecto, bastará con el siguiente comando, donde se especifica el IDE desde dónde se importará el proyecto y la placa de desarrollo utilizada:

```
1 pio project init --ide eclipse --board az-delivery-devkit-v4
```

Este comando creará los directorios **include**, **lib**, **src** y **test**. Estos directorios almacenarán los archivos de cabecera, bibliotecas específicas, código fuente y testeo del proyecto, respectivamente. Además, creará el archivo de configuración **platformio.ini**, donde especifica la plataforma, la placa y el *framework*, que puede ser modificado editando el archivo de configuración.

Por defecto, el *framework* elegido es **Arduino** [10]. Aunque podría ser otro de los muchos compatibles, este será el elegido para desarrollar el proyecto, ya que es un entorno cómodo que además cuenta con bibliotecas y funciones que facilitarán el desarrollo del proyecto.

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:az-delivery-devkit-v4]
platform = espressif32
board = az-delivery-devkit-v4
framework = arduino
~
"platformio.ini" 14L, 466C                    1,2           All
```

Figura 4.2: Archivo platformio.ini

Por último, bastará con importar el proyecto en el IDE que se desee.

4.1.2. Code::Blocks/Eclipse

En un principio, el editor elegido para llevar a cabo el proyecto fue **Code::Blocks**. Este editor es de código abierto, siendo compatible en diversos sistemas operativos para el desarrollo de programas en C/C++ y proporcionando una interfaz que permite trabajar con facilidad. Es un IDE que permite a los usuarios extenderlo y mejorarlo mediante *plugins*, sin necesidad de modificar el código fuente de **Code::Blocks**.

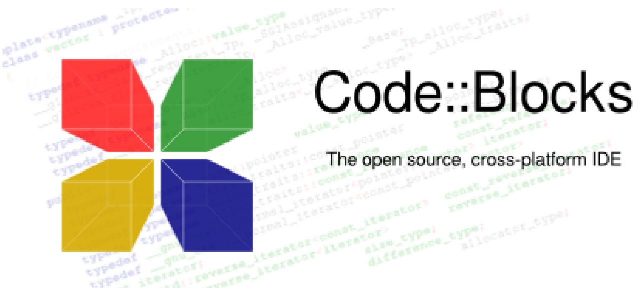

Figura 4.3: Logo Code::Blocks

Después de un tiempo utilizándolo, **Eclipse** pasó a ser el editor elegido para seguir con el proyecto ya que es un entorno más maduro y más profesional, facilitando el trabajo por sus características y buen funcionamiento.

Figura 4.4: Logo Eclipse

4.2. Adquisición de datos

4.2.1. MPU-9250

Para llevar a cabo la lectura de datos del sensor MPU-9250 se hará uso del protocolo I2C comentado anteriormente. Se puede observar el esquemático entre el sensor y el ESP32 en la imagen 4.5, siendo el conexionado el siguiente:

MPU-9250	**ESP32**
SCL	GPIO 22
SDA	GPIO 21
VDD	3V3
GND	GND

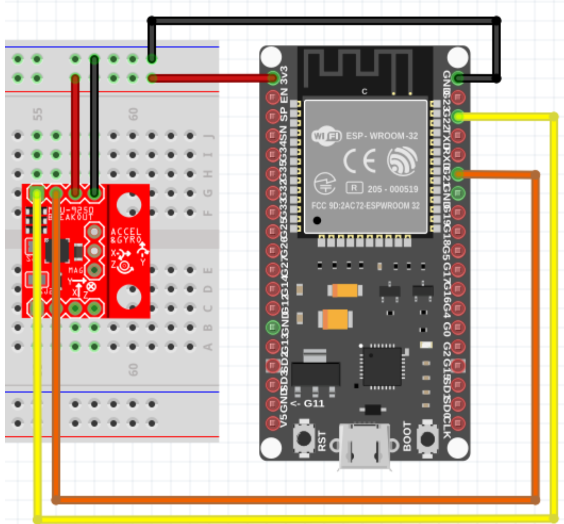

Figura 4.5: Conexión MPU-9250 - ESP32

Además, se hará uso de la biblioteca MPU-9250 [11], perteneciente a un repositorio de **GitHub** [12]. Esta biblioteca permite declarar un objeto MPU-9250 y hacer uso de diferentes funciones que permitan inicializarlo, leer los valores que ofrece, configurarlo y calibrarlo. Por lo tanto, el primer paso será declarar el objeto:

```
MPU9250 IMU(Wire,0x68);
```

Una vez declarado, se implementará la función **initMPU**, que indicará si ha sido inicializado correctamente el MPU-9250:

```
bool initMPU (MPU9250 *IMU)
{
    int status = IMU->begin();

    for(int i = 0; (i<5) && (status < 0); i++) {
        delay(500);
        status = IMU->begin();
    }
    if (status < 0) {
        return false;
    }
    Serial.println("IMU initialization successful");
    return true;

}
```

El método **begin** de IMU devuelve un número positivo si se ha inicializado correctamente y uno negativo en caso contrario, por lo que se harán 5 intentos de inicialización del sensor en el caso de que de un valor negativo. Esta función será llamada desde la función **setup** del código principal:

```
1  statusMPU = initMPU(&IMU);
2  if (!statusMPU) {
3      Serial.println("IMU initialization unsuccessful");
4      exit(1);
5  }
```

Una vez inicializado el sensor el siguiente paso será leer los datos que ofrece. Para ello se ha implementado la función **dataMPU**, que utilizará el método **readSensor** de IMU. Este método lee el sensor y almacena los datos más recientes, por lo que será llamado cada vez que se quieran volver a leer los datos. Para obtener cada valor del acelerómetro, giroscopio y magnetómetro para cada eje x,y, z habrá un método destinado a leer cada dato, como se puede observar en el siguiente código de la función implementada **dataMPU**:

```
1  void dataMPU (MPU9250 *IMU)
2  {
3      IMU->readSensor();
4
5      Serial.println("Valores accel: ");
6      Serial.println(IMU->getAccelX_mss(),6);
7      Serial.println(IMU->getAccelY_mss(),6);
8      Serial.println(IMU->getAccelZ_mss(),6);
9      Serial.println("");
10
11     Serial.println("Valores gyro: ");
12     Serial.println(IMU->getGyroX_rads(),6);
13     Serial.println(IMU->getGyroY_rads(),6);
14     Serial.println(IMU->getGyroZ_rads(),6);
15     Serial.println("");
16
17     Serial.println("Valores magn: ");
18     Serial.println(IMU->getMagX_uT(),6);
19     Serial.println(IMU->getMagY_uT(),6);
20     Serial.println(IMU->getMagZ_uT(),6);
21     Serial.println("");
22 }
```

4.2.2. Sensor flexible

Para obtener los datos del sensor resistivo se hará uso de las señales de entrada ADC del ESP32, siguiendo el esquemático del divisor de tensión de la imagen 3.8 explicado anteriormente. Cada sensor irá conectado a una de las diferentes entradas ADC del ESP32:

Sensores flexibles	ESP32
Sensor 1	GPIO 39
Sensor 2	GPIO 34
Sensor 3	GPIO 35
Sensor 4	GPIO 32
Sensor 5	GPIO 33

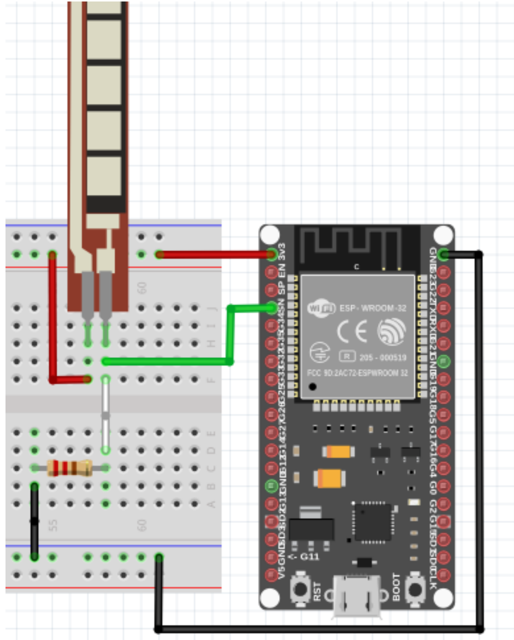

Figura 4.6: Conexión Sensor flexible - ESP32

El valor de la resistencia elegida es de $39\,k\Omega$.

Para la obtención de los valores se ha implementado la función **readSingleDataFlex**:

```
float readSingleDataFlex(int FLEX_PIN)
{
    analogReadResolution(12); //cambiar resolucion de analogRead a 12 bits
    int flexADC = analogRead(FLEX_PIN);
    float flexV = flexADC * (VCC/4095); //12 bits ADC
    float flexR = R * (VCC/flexV - 1.0);
    return flexR;
}
```

Mediante la función **analogRead** se lee el valor de tensión en la entrada ADC. Esta función tiene por defecto leer con 10 bits de resolución, y la entrada ADC del ESP32 tiene 12 por lo que antes de llamarla es necesario cambiar la resolución a 12 bits mediante la función **analogReadResolution** enviándole el valor de 12 como argumento. Una vez hecho esto, simplemente se obtendrá el valor de la resistencia variable (sensor flexible) mediante la ecuación del divisor de tensión:

$$flexV = VCC \cdot \frac{R}{R + flexR} \tag{4.1}$$

que despejando flexR, queda de la siguiente manera:

$$flexR = \frac{VCC \cdot R - flexV \cdot R}{flexV} = R \cdot \left(\frac{VCC}{flexV} - 1 \right) \tag{4.2}$$

4.2.2.1. Calibración

Una vez definido el procedimiento para leer los valores de los sensores flexibles, el siguiente paso será calibrarlos, ya que estando en la misma posición cada uno muestra un valor diferente de resistencia.

En primer lugar, se implementará la función **calibrationFlex**. El objetivo de esta función será adquirir 10 valores distintos por cada sensor en la posición de la mano completamente abierta, y otros 10 valores por cada sensor con la mano cerrada. De este modo, se acumularán los 10 valores de cada sensor en dos *arrays* diferentes, uno por cada posición de la mano, para luego hacer la media y obtener el rango en el que se encuentra cada sensor.

```
1  void calibrationFlex (float *sflexCal)
2  {
3      int cont1 = 10;
4      for (int i=0; i < 10; i++) {
5          setCalibrationFlex(sflexCal);
6          Serial.println(cont1--);
7          delay(1000);
8      }
9      for (int i = 0; i < 5; i++) {
10         sflexCalLow[i] = sflexCalLow[i] /10;
11     }
12 }
```

Siendo la función **setCalibrationFlex** la encargada de leer los valores sin calibrar de la función comentada en el apartado 4.2.2 y acumularlos en cada *array*:

```
void setCalibrationFlex (float *sflexCal)
{
    for (int i = 0; i < (sizeof(fpin)/sizeof(fpin[0])); i++){
        sflexCal[i] += readSingleDataFlex(fpin[i]);
    }
}
```

La función **calibrationFlex** será llamada desde la función **setup** del código principal, en dos ocasiones diferentes, una para calibrar con la mano completamente abierta, y otra para calibrar con la mano cerrada:

```
    //MANO ABIERTA
    Serial.println("Modo calibracion mano abierta");
    delay(2000);
    calibrationFlex(sflexCalLow);

    delay(2000);

    //MANO CERRADA
    Serial.println("Modo calibracion mano cerrada");
    delay(2000);
    calibrationFlex(sflexCalHigh);
```

Una vez determinado el rango de valores que ofrece cada sensor, el siguiente paso será obtener el valor calibrado. Para ello, se ha implementado la función **readSingleFlexCalibrated**. Esta función lee el valor de resistencia sin calibrar de la función comentada anteriormente **readSingleDataFlex**, para después junto a la función **map** obtener el valor correspondiente, pero en un mismo rango para todos los sensores. La función **map** permite hacer equivalencias entre diferentes rangos. En este caso, el primer valor será el que lee del sensor, los dos siguientes serán el rango real que ofrece el sensor, es decir, el calculado anteriormente mediante la función **calibrationFlex**. Los dos últimos valores serán el rango definido entre 30 kΩ (SFLEX_LOW) y 80 kΩ (SFLEX_HIGH), que será el rango en el que estarán todos los sensores una vez calibrados.

```
float readSingleDataFlexCalibrated(int FLEX_PIN, float sflexCalL, float sflexCalH)
{
    float flexR = readSingleDataFlex(FLEX_PIN);
    int flexRCal = map(flexR, sflexCalL, sflexCalH, SFLEX_LOW, SFLEX_HIGH);
    return flexRCal;
}
```

4.3. Comunicación ESP32-RPI

El primer paso para llevar a cabo la comunicación entre los dos dispositivos será configurar la Raspberry como un punto de acceso, creando la red WiFi a la que el ESP32 se conectará. Una vez realizada la configuración del punto de acceso y de la conexión WiFi, el siguiente paso consistirá en implementar en el ESP32 un cliente TCP que sea capaz de enviar los datos de los sensores a la Raspberry, que actuará como servidor TCP.

4.3.1. Configuración de la Raspberry como punto de acceso (AP)

Para configurar la Raspberry como un punto de acceso al que se conectará el ESP32 se hará uso del programa **Hostapd** [13]. Este programa permite que una tarjeta de interfaz de red actúe como un punto de acceso y servidor de autenticación. En este proyecto solo se utilizará para crear el punto de acceso.

En primer lugar se tendrá que instalar el paquete correspondiente: (antes de instalarlo sería conveniente comprobar actualizaciones y actualizarlas, mediante *update* y *upgrade*, respectivamente).

```
sudo apt install hostapd
```

Una vez descargado el paquete, se abrirá el archivo de configuración de **Hostapd**:

```
vim /etc/hostapd/hostapd.conf
```

Y se añadirán las siguientes líneas al archivo:

```
interface = wlan0 #nombre de la interfaz usada por el AP
ssid = PiNetMal #nombre red WiFi
hw_mode = g #g hace referencia a 2,4 GHz
channel = 1
macaddr_acl = 0
auth_algs = 1 #1 hace referencia a wpa
```

Ahora que está configurado el hostapd, se configurará la interfaz de red. Primero se abrirá el archivo de configuración:

```
vim /etc/network/interfaces
```

Y se añadirán las siguientes líneas, que definirán la dirección IP estática de la Raspberry (172.16.0.100):

```
auto wlan0
iface wlan0 inet static
    address 172.16.0.100
    netmask 255.255.255.0
```

La siguiente imagen muestra la disposición de la red creada. La Raspberry actuando como un punto de acceso, con una IP definida, y el ESP32 actuando como una estación que se conectará a este punto de acceso. La IP del dispositivo ESP32 se autoasignará al conectarse al AP, que se verá en el siguiente apartado, ya que no se ha implementado el protocolo DHCP al configurar la Raspberry como punto de acceso.

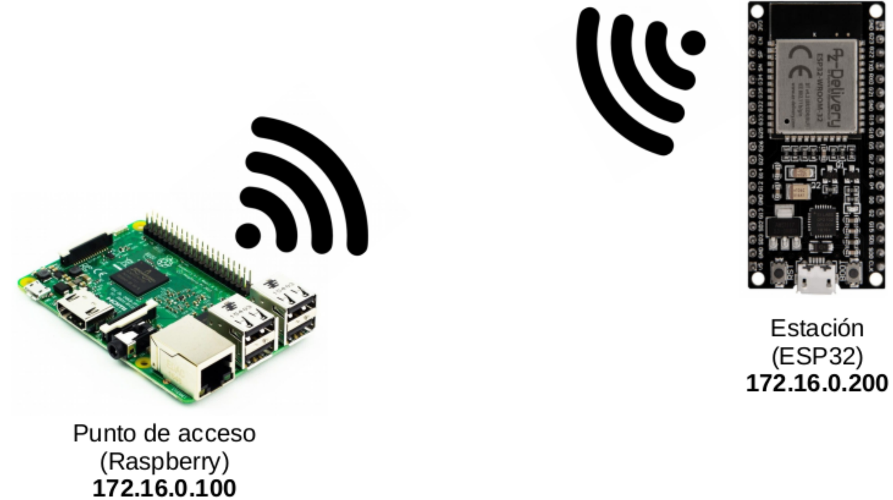

Figura 4.7: Disposición de la red

Por último, para que el punto de acceso se inicie cada vez que la Raspberry se encienda, hará falta activarlo con los siguientes comandos:

```
sudo systemctl unmask hostapd
sudo systemctl enable hostapd
sudo systemctl start hostapd
```

Para comprobar que efectivamente se ha activado, bastará con el siguiente comando:

```
sudo systemctl status hostapd
```

```
● hostapd.service - Advanced IEEE 802.11 AP and IEEE 802.1X/WPA/WPA2/EAP Authenticator
   Loaded: loaded (/lib/systemd/system/hostapd.service; enabled; vendor preset: enabled)
   Active: active (running) since Fri 2020-09-25 05:18:07 CEST; 59min ago
 Main PID: 598 (hostapd)
    Tasks: 1 (limit: 2065)
   CGroup: /system.slice/hostapd.service
           └─598 /usr/sbin/hostapd -B -P /run/hostapd.pid -B /etc/hostapd/hostapd.conf

sep 25 05:18:07 raspberrypi hostapd[575]: Using interface wlan0 with hwaddr b8:27:eb:c4:85:76 and ssid "PiNetMal"
sep 25 05:18:07 raspberrypi hostapd[575]: wlan0: interface state UNINITIALIZED->ENABLED
sep 25 05:18:07 raspberrypi hostapd[575]: wlan0: AP-ENABLED
sep 25 05:18:07 raspberrypi systemd[1]: Started Advanced IEEE 802.11 AP and IEEE 802.1X/WPA/WPA2/EAP Authenticator.
sep 25 06:11:38 raspberrypi hostapd[598]: wlan0: STA 24:77:03:83:af:6c IEEE 802.11: associated
sep 25 06:11:38 raspberrypi hostapd[598]: wlan0: STA 24:77:03:83:af:6c RADIUS: starting accounting session A1DC15CD3A5A10C2
sep 25 06:13:31 raspberrypi hostapd[598]: wlan0: STA 24:77:03:83:af:6c IEEE 802.11: disassociated
sep 25 06:13:31 raspberrypi hostapd[598]: wlan0: STA 24:77:03:83:af:6c IEEE 802.11: disassociated
sep 25 06:15:59 raspberrypi hostapd[598]: wlan0: STA 24:77:03:83:af:6c IEEE 802.11: associated
sep 25 06:15:59 raspberrypi hostapd[598]: wlan0: STA 24:77:03:83:af:6c RADIUS: starting accounting session 0926B4E5709142FF
```

Figura 4.8: Comprobación hostapd

4.3.2. Envío de datos

4.3.2.1. Conexión WiFi

Para establecer la conexión WiFi entre el ESP32 y la Raspberry se hará uso de la biblioteca **WiFi.h** [14] de Arduino. Esta biblioteca permite a la placa conectarse a internet, ser utilizada como servidor que acepta conexiones entrantes o como cliente que realiza esas conexiones.
Se implementará la siguiente función **wifiConnect** que será llamada desde la función **setup()** del código principal:

```
1  #include "WiFi.h"
2
3  #include "wifi.h"
4
5  bool wifiConnect(IPAddress esp, IPAddress rpi, IPAddress subnet)
6  {
7      WiFi.config(esp, rpi, subnet); //esp: ip  rpi: gateway
8
9      for(int i = 0; (i < 5) && (WiFi.status() != WL_CONNECTED); i++) {
10         delay(500);
11         Serial.println("Connecting to WiFi...");
12         WiFi.begin(SSID);
13     }
14     if (WiFi.status() != WL_CONNECTED) {
15         return false;
16     }
17     Serial.println("Connected to the WiFi network");
18     return true;
19 }
```

Esta función consiste en la comprobación de la conexión a la red WiFi, y llevar a cabo la propia conexión, por lo que devolverá *true* o *false* según se haya conseguido conectar o no.

En la línea 7, el método **config** permite configurar la IP del dispositivo, el *gateway*, la subred y opcionalmente el DNS. Una vez hecho esto se harán varios intentos de conexión por si fallara, hasta un máximo de 5 intentos. La conexión se realizará con el método **begin**, pasándole como argumento SSID, que será la red WiFi configurada en la Raspberry (AP).

4.3.2.2. Estructura de datos

Para agrupar los datos pertenecientes a los sensores se hará uso de estructuras. Este tipo de datos permite agrupar varios datos que mantienen algún tipo de relación pudiendo ser de distinto tipo.

En primer lugar se definirá la estructura **flexibles**, que agrupará los datos de los cinco sensores de flexión. También se definirá la estructura **acelerometro**, donde estarán almacenados los datos que ofrece el sensor MPU-9250 del acelerómetro, giroscopio y magnetómetro por cada eje. Esta última también almacenará el objeto MPU9250, desde el cual se obtienen los datos pertenecientes al sensor. Estas dos estructuras serán almacenadas en otra estructura llamada **guante**, para recoger todos los datos en una misma estructura que será la que se enviará a la Raspberry.

A continuación se muestra el archivo de cabecera **datos.h** donde se han definido:

```
1  #ifndef DATOS_H_INCLUDED
2  #define DATOS_H_INCLUDED
3  #include "MPU9250.h"
4  struct flexibles
5  {
6      float dedos[5];
7  };
8  typedef flexibles flexibles_t;
9
10 struct acelerometro
11 {
12     float accX;
13     float accY;
14     float accZ;
15
16     float gyroX;
17     float gyroY;
18     float gyroZ;
19
20     float magX;
21     float magY;
22     float magZ;
23
24     MPU9250 *IMU;
25 };
26 typedef acelerometro acelerometro_t;
27
```

```
struct guante
{
    flexibles_t sflex;
    acelerometro_t acel;
};
typedef guante guante_t;

#endif // DATOS_H_INCLUDED
```

Una vez definidas las estructuras, hará falta una función encargada de rellenarlas con los valores correspondientes: **getDataFlex** y **getDataMPU**.

Para rellenar la estructura **flexibles** se ha implementado la siguiente función:

```
void getDataFlex (flexibles_t *sflex)
{
    for (int i = 0; i < (sizeof(fpin)/sizeof(fpin[0])); i++)
    sflex->dedos[i] = readSingleDataFlexCalibrated(fpin[i], sflexCalLow[i],
    ↪  sflexCalHigh[i]);
}
```

fpin ha sido definido como un *array* que contiene los 5 pines referentes a los sensores flexibles:

```
float fpin[5] = {FLEX_PIN1,FLEX_PIN2,FLEX_PIN3,FLEX_PIN4,FLEX_PIN5};
```

Siendo la función **readSingleDataFlexCalibrated** la implementada anteriormente para leer los datos de cada sensor flexible (apartado 4.2.2).

Para rellenar la estructura **acelerometro** se ha implementado la siguiente función:

```
void getDataMPU (acelerometro_t *acel)
{
    MPU9250 *IMU = acel->IMU;
    IMU->readSensor();
    //acel
    acel->accX = IMU->getAccelX_mss();
    acel->accY = IMU->getAccelY_mss();
    acel->accZ = IMU->getAccelZ_mss();
    //giro
    acel->gyroX = IMU->getGyroX_rads();
    acel->gyroY = IMU->getGyroY_rads();
    acel->gyroZ = IMU->getGyroZ_rads();

```

```
14      //magn
15      acel->magX = IMU->getMagX_uT();
16      acel->magY = IMU->getMagY_uT();
17      acel->magZ = IMU->getMagZ_uT();
18
19  }
```

4.3.2.3. Cliente TCP

Para implementar el cliente TCP se hará uso de la biblioteca **WiFi.h** y su clase **Client**. En primer lugar se inicializará el cliente, y después se comprobará si se ha podido realizar la conexión al servidor. El método **connect** devuelve "1" si se ha realizado correctamente la conexión, y un "0" en el caso contrario, siendo **rpi** la dirección IP de la Raspberry y **port** el puerto 2320 elegido aleatoriamente.

```
1  WiFiClient client;
2  if(!client.connect(rpi, port)) {
3      Serial.println("Connection to server failed");
4      delay(1000);
5      return;
6  Serial.println("Connected to server");
```

Una vez establecida la conexión, el siguiente paso será enviar los datos de los sensores mediante el método **write** y cerrar la conexión con el servidor mediante el método **stop**, siendo **txbuffer** un buffer creado para almacenar la estuctura **guante** (*)

```
1  client.write(txbuffer,sizeof(txbuffer));
2  Serial.println("Disconnecting...");
3  client.stop();
4  delay(2000);
```

(*) el buffer **txbuffer** ha sido copiado de la siguiente manera:

```
1  memcpy(txbuffer, &guante1, sizeof(txbuffer));
```

4.3.3. Recepción de datos

Para la recepción de los datos en la Raspberry se creará un servidor TCP en python [15] que recogerá los datos enviados por el ESP32. Para realizar el servidor de red, se pueden usar dos bibliotecas: **socketserver** [16] o **socket** [17]. La biblioteca socket es de bajo nivel, siendo necesario implementar la mayoría de las cosas, mientras que socketserver es de más alto nivel, y utiliza la biblioteca socket internamente, por lo que proporciona una interfaz más sencilla dejando a un lado el bajo nivel que implica el uso de sockets. Por lo tanto, se usará **socketserver**. Además, también se hará uso de la biblioteca **struct** [18] para gestionar los datos de los sensores enviados desde el ESP32.

- Socketserver: Permite crear servidores de red, definiendo clases para el manejo de solicitudes de red sobre TCP, UDP, flujos Unix y datagramas Unix. Hay cinco clases de servidores diferentes, en nuestro caso, se utilizará la clase TCPServer, que utiliza conectores TCP/IP para comunicarse.

- Struct: Realiza conversiones entre los valores de Python y las estructuras C representadas como objetos de bytes de Python. Este módulo será clave para manejar las estructuras enviadas desde el ESP32 que almacenan los datos de los sensores.

En primer lugar se creará la subclase **MyTCPHandler** que heredará de la superclase **BaseRequestHandler**, que contiene los métodos para el manejo de peticiones. Después, será definido el método **handler** que será el encargado de analizar la solicitud y procesar los datos.El método **unpack** de **struct** permitirá pasar la estructura recibida en bytes al formato adecuado, en nuestro caso, 14 valores de tipo float (14f) y 4 bytes que se ignorarán (4x) que representan al objeto MPU9250 que está incluido en la estructura pero que no aporta ninguna información, ya que ha sido añadido para facilitar la lectura de los valores en el momento de rellenar la estructura:

```
1  import socketserver
2  import struct
3  import os
4  import json
5
6  try:
7      os.remove("/var/www/html/archivo.json")
8  except:
9      pass
10
11 class MyTCPHandler(socketserver.BaseRequestHandler):
12     def handle(self):
13         self.data = self.request.recv(1024)
14         print("{} wrote:" .format(self.client_address[0]))
15         self.unpacked_data = struct.unpack('14f4x', self.data)
16         print(self.unpacked_data)
```

Una vez visto esto, lo siguiente será introducir esos datos que recibe desde el ESP32 en un tipo de datos de tipo diccionario, que es una estructura de datos que será muy útil para almacenarlos y trabajar con ella. El primer paso será declarar el diccionario **d** y definirlo con los datos de los sensores recibidos.

Una vez definido el diccionario, el siguiente paso será pasarlo a un objeto tipo **JSON** (JavaScript Object Notation) e ir almacenándolo en un archivo según lleguen los datos al servidor. En este archivo se almacenarán los datos conforme lleguen, borrando los anteriores, de tal forma que solo quede en el archivo un listado con un único valor de cada sensor. Si nos fijamos en el código anterior en las líneas 6-9, estas líneas simplemente eliminan el archivo json creado antes de comenzar a recibir datos, ya que al usar el método **open**, el código generaría un error al encontrar el archivo ya existente.

```
1          d = {
2          'Dedo1': self.unpacked_data[0],
3          'Dedo2': self.unpacked_data[1],
4          'Dedo3': self.unpacked_data[2],
5          'Dedo4': self.unpacked_data[3],
6          'Dedo5': self.unpacked_data[4],
7
8          'AccX': self.unpacked_data[5],
9          'AccY': self.unpacked_data[6],
10         'AccZ': self.unpacked_data[7],
11         'GyroX': self.unpacked_data[8],
12         'GyroY': self.unpacked_data[9],
13         'GyroZ': self.unpacked_data[10],
14         'MagnX': self.unpacked_data[11],
15         'MagnY': self.unpacked_data[12],
16         'MagnZ': self.unpacked_data[13]
17         }
18         json_obj = json.dumps(d)
19         print(json_obj)
20         archivo = open("/var/www/html/archivo.json","w")
21         archivo.write(str(json_obj))
22         archivo.close()
```

Por último, se creará el servidor enlazándolo con el *host* “0.0.0.0” y el puerto 2320, manteniéndolo abierto hasta que se interrumpa el programa.

```
1  if __name__ == "__main__":
2      HOST, PORT = "0.0.0.0", 2320
3
4      with socketserver.TCPServer((HOST,PORT), MyTCPHandler) as server:
5          server.serve_forever()
```

```
172.16.0.200 wrote:
(37281.0, 33763.0, 34895.0, 34232.0, 32907.0, 1.2833575010299683, 1.053502440452
5757, -82.9776840209961, -0.0023183166049420834, -0.0009853846859186888, 0.00238
0901947617531, 0.0, 0.0, 0.0)
{"Dedo1": 37281.0, "Dedo2": 33763.0, "Dedo3": 34895.0, "Dedo4": 34232.0, "Dedo5"
: 32907.0, "AccX": 1.2833575010299683, "AccY": 1.0535024404525757, "AccZ": -82.9
776840209961, "GyroX": -0.0023183166049420834, "GyroY": -0.0009853846859186888,
"GyroZ": 0.002380901947617531, "MagnX": 0.0, "MagnY": 0.0, "MagnZ": 0.0}
```

Figura 4.9: Datos recibidos en el servidor TCP

Para finalizar este apartado, el último paso será configurar la Raspberry para que ejectute el *script* del servidor TCP de forma automática cada vez que se encienda:

En primer lugar se creará un archivo de servicio que contendrá las instrucciones necesarias para ejecutar el script (este archivo estará en la ruta /etc/systemd/system):

```
1  [Unit]
2  Description=Mi servicio SocketServer(TFG)
3  After=multi-user.target
4
5  [Service]
6  Type=simple
7  ExecStart=/usr/bin/python3 /home/pi/SocketServer.py
8
9  [Install]
10 WantedBy=multi-user.target
```

Este archivo creará un servicio llamado ”Mi servicio SocketServer(TFG)”. El tipo de servicio **simple** indica que el proceso comenzará con **ExecStart**, que es el principal proceso del servicio. Aquí se indicará la ruta al ejecutable.

Una vez creado, el siguiente paso será darle los permisos necesarios al archivo mediante el siguiente comando (permiso de lectura y escritura para el propietario, y solo de lectura para los demás):

```
1  sudo chmod 644 /etc/systemd/system/SocketServer.service
```

Mediante **systemctl** se actualizará la configuración que ha sido modificada y se activará el servicio:

```
1  sudo systemctl daemon-reload
2  sudo systemctl enable SocketServer.service
```

Por último, se comprobará con el siguiente comando que ha sido correctamente activado el archivo de servicio:

```
1 sudo systemctl status SocketServer.service
```

```
pi@raspberrypi:~ $ sudo systemctl status SocketServer.service
● SocketServer.service - Mi servicio SocketServer(TFG)
   Loaded: loaded (/etc/systemd/system/SocketServer.service; enabled; vendor pre
   Active: active (running) since Fri 2020-09-25 04:18:08 CEST; 8min ago
 Main PID: 664 (python3)
    Tasks: 1 (limit: 2065)
   CGroup: /system.slice/SocketServer.service
           └─664 /usr/bin/python3 /home/pi/SocketServer.py

sep 25 04:18:08 raspberrypi systemd[1]: Started Mi servicio SocketServer(TFG).
 ESCOC
```

Figura 4.10: Systemctl status

4.4. Servidor Web

Una vez establecida la comunicación entre el ESP32 y la Raspberry, el siguiente y último objetivo del proyecto será mostrar los datos adquiridos en un entorno interactivo, dónde pueda verse cómo varían los datos de los sensores en tiempo real. Para ello, se creará un Servidor Web en la Raspberry, que irá recogiendo la información de los sensores que llegan al servidor TCP y los mostrará en unas gráficas conforme vayan leyéndose los nuevos valores.

En primer lugar se creará un servidor web **apache** [19], instalando el paquete correspondiente:

```
1 sudo apt install apache2
```

Las rutas dónde se guardan las páginas web de apache son *www/html* y subcarpetas. Con el siguiente comando creamos el archivo **index.html** dónde estará definida la página web:

```
1 sudo vim /var/www/html/index.html
```

En este archivo **index.html** serán definidas las gráficas y el aspecto visual de la página web. Además, se creará el archivo **myChart.js** que utilizará la biblioteca **Chart.js** [20] perteneciente **JavaScript**. En este archivo se implementará la función correspondiente para la adquisición de los valores que recoge el servidor TCP y representarlos en las gráficas. Esta implementación del diseño de gráficas y la adquisición de datos se ha basado en un ejemplo [21] perteneciente a un repositorio en GitHub, pero modificando y añadiendo lo necesario para llevar a cabo el correcto funcionamiento que requiere el proyecto. (ver código en Apéndice B).

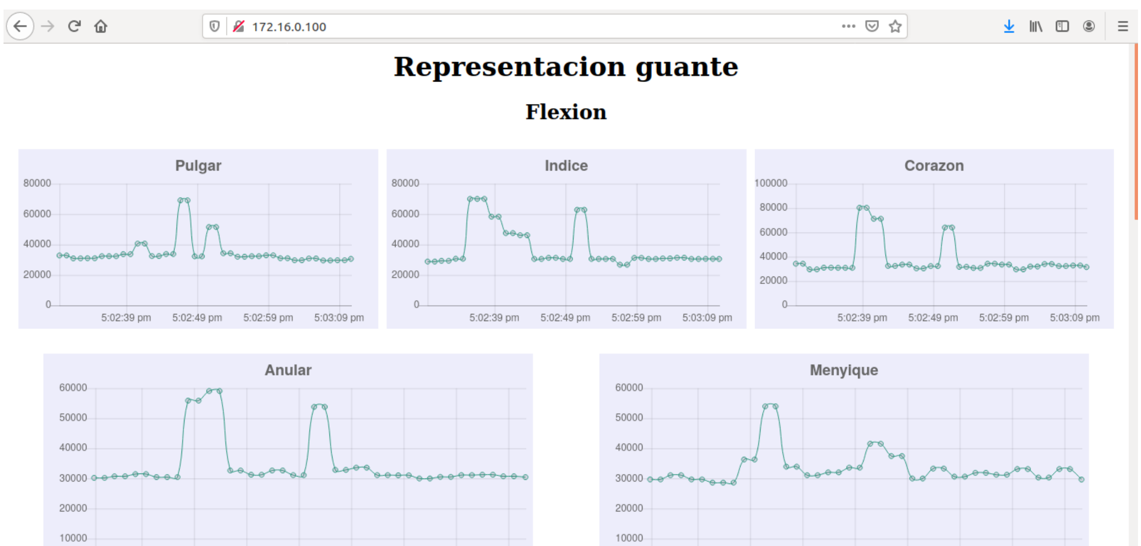

Figura 4.11: Gráficas servidor web

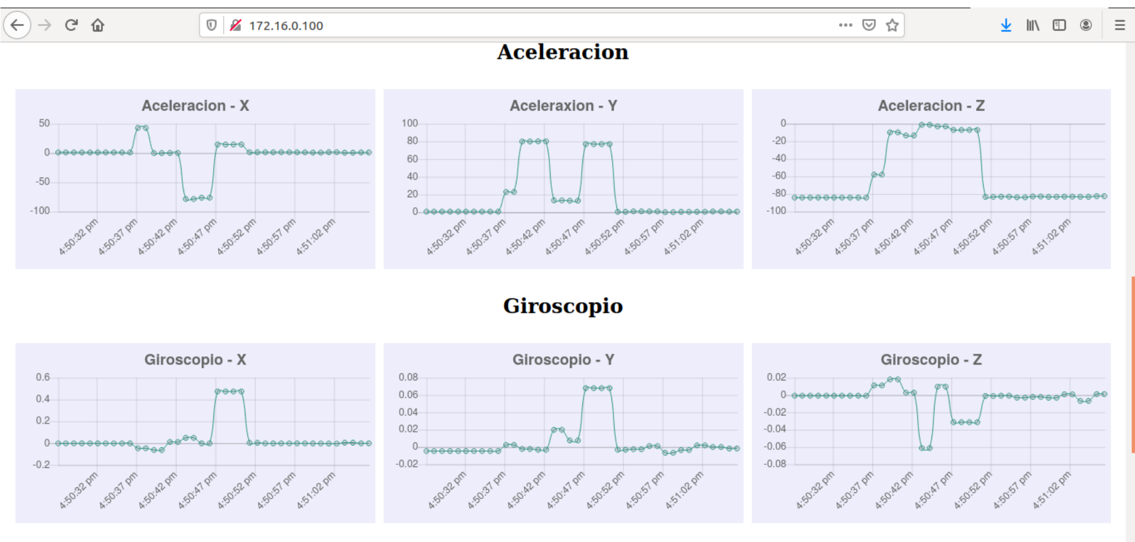

Figura 4.12: Gŕaficas servidor web

4.4.1. Demo

Para finalizar el desarrollo del proyecto, la última parte consistirá en una *demo* sencilla en la que se reconocerán algunos gestos. Para llevarlo a cabo, se ha hecho de forma similar al anterior apartado, utilizando el lenguaje de programación **JavaScript**, clasificando los datos recibidos y asignándolos a diferentes gestos. (ver código en Apéndice B).

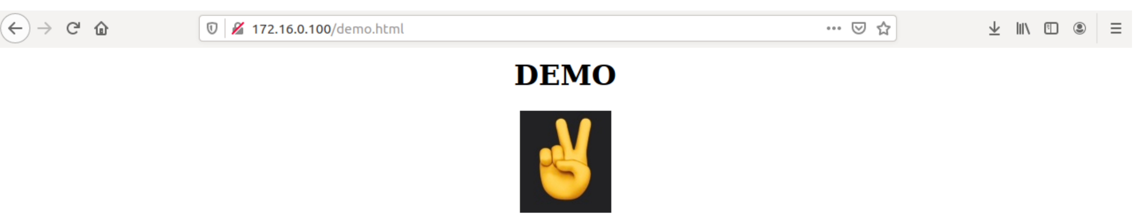

Figura 4.13: Demo página web

4.5. Diseño del guante interactivo

4.5.1. Esquemático protoboard

Previamente al diseño físico del guante, se utilizó una *protoboard* para montar el esquemático con los sensores y conexiones propias del diseño:

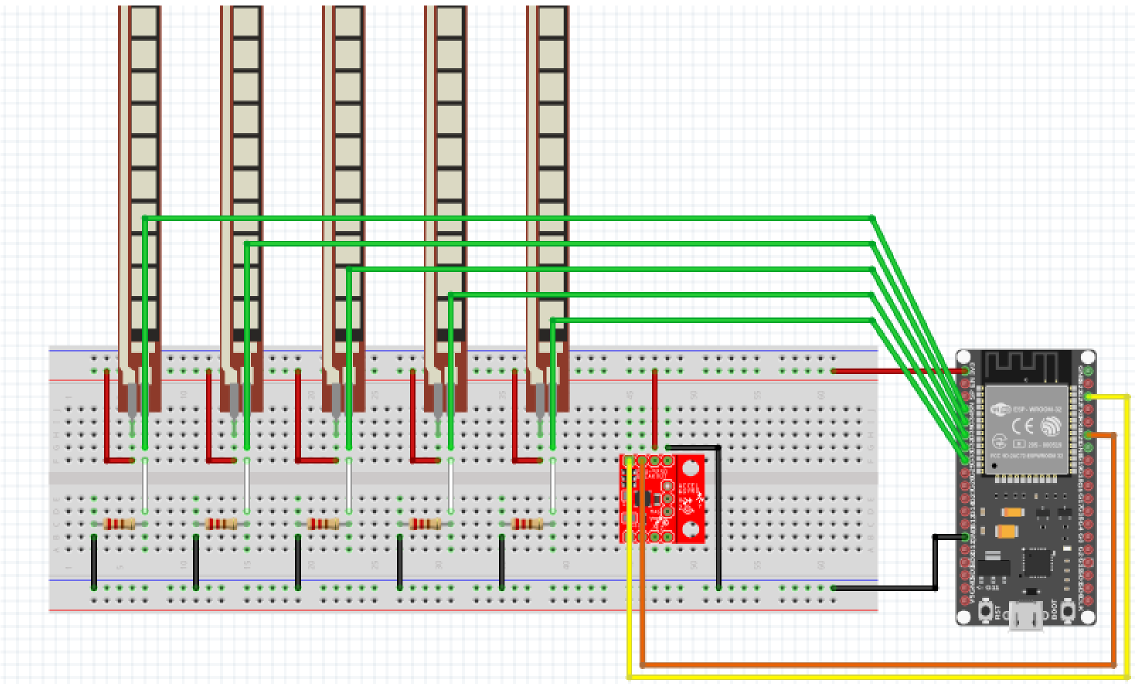

Figura 4.14: Esquemático protoboard

4.5.2. Guante interactivo

A continuación se mostrarán algunas imágenes del resultado final del diseño físico del guante:

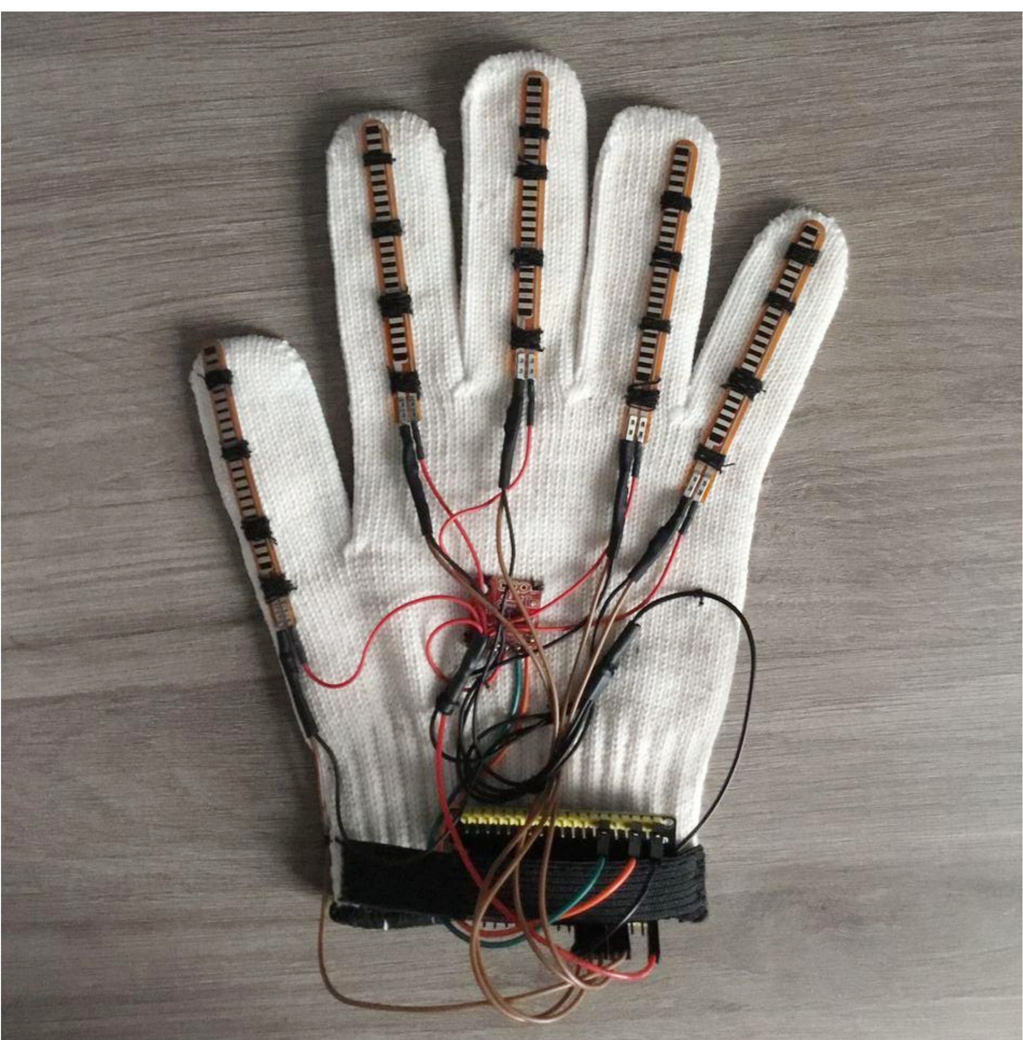

Figura 4.15: Guante interactivo (1)

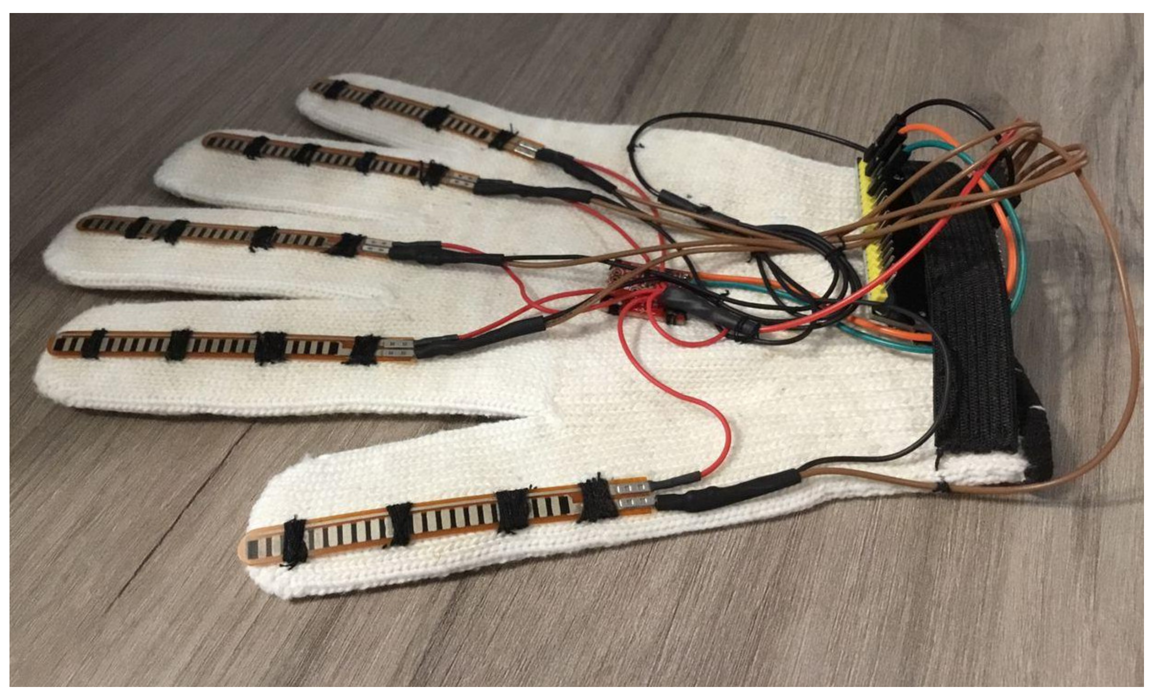

Figura 4.16: Guante interactivo (2)

En la siguiente imagen se pueden observar las conexiones de los cables al ESP32:

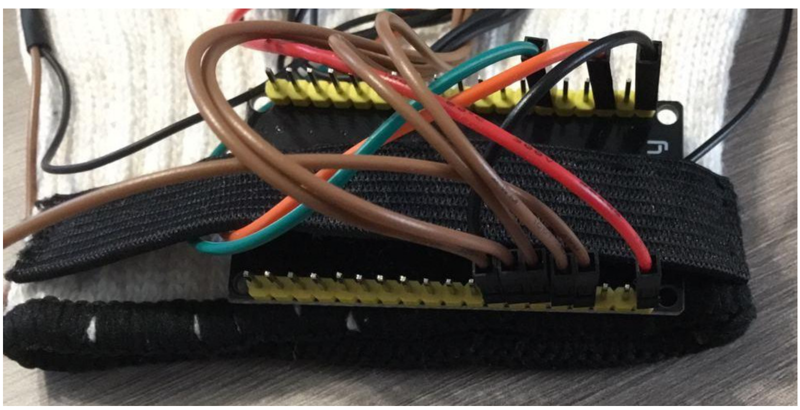

Figura 4.17: Guante interactivo (ESP32)

Las siguientes imágenes muestran de más cerca el sensor MPU-9250 y un sensor de flexión:

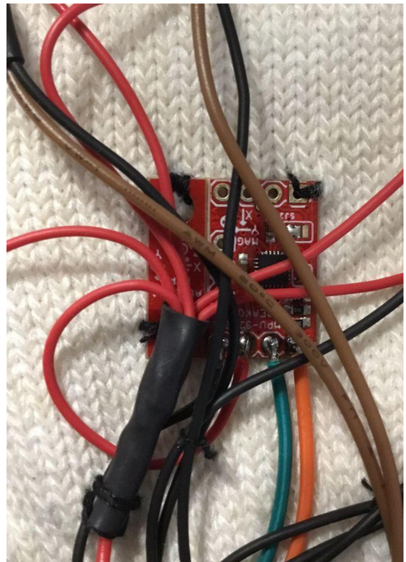

Figura 4.18: Guante interactivo (Sensor MPU-9250)

Figura 4.19: Guante interactivo (Sensor de flexión)

Capítulo 5

Presupuesto

En este capítulo se detallará el prepuesto necesario para desarrollar el proyecto. En primer lugar se han analizado los costes referentes al *hardware* utilizado:

	Raspberry Pi 3 Modelo B	37,44 €
	ESP32 AzDelivery Devkit	9,49 €
	MPU-9250	10,20 €
	Sensor de flexión	8,95 €
	Guante	2 €
	Resistencia 39k	0,08 €
	Cables	6,99 €

Tabla 5.1: Costes hardware

Teniendo en cuenta que los sensores flexibles y las resistencias son en total 5 de cada, los costes *hardware* suman un total de **38,03 €**, sin contar la Raspberry Pi. Contando este dispositivo, sumaría un total de **75,47 €**.

Además, también se han considerado los costes de personal:

	Tiempo dedicado (h)	Coste por unidad de tiempo (€)	Coste total (€)
Alumno	480	20 (*)	9600

Tabla 5.2: Costes personal

Lo que suma un total de **9 675,47 €.**

(*) Precio programador junior

Capítulo 6

Conclusión y líneas futuras

En este trabajado se ha desarrollado un sistema capaz de leer los valores que ofrecen los sensores que componen el guante interactivo, y de mandarlos a un dispositivo exterior de cómputo que será el encargado de procesar esos datos. Además, también se ha llevado a cabo del diseño físico del guante con los sensores y conexiones convenientemente soldadas. Como demostración, se puede observar el apartado 4.4, donde se muestran las gráficas en tiempo real con la variación de estos sensores. Además, para la presentación, se han preparado algunos gestos que serán reconocidos en una página web, al igual que las gráficas del apartado 4.4, como demostración de la lectura de los valores que ofrecen los sensores. Por lo que se puede confirmar que los objetivos han sido totalmente alcanzados.

En cuanto a los siguientes pasos que se podrían llevar a cabo en este proyecto, el primero de todos sería llevar a cabo la clasificación de los signos pertenecientes al lenguaje de signos mediante técnicas de *Machine Learning*. Aunque además de la finalidad del lenguaje de signos, el guante también podría utilizarse para muchas otras aplicaciones, como podría ser controlar la música que estés escuchando, utilizando gestos para pasar de canción, subir el volumen, etc.

Bibliografía

[1] Trello. https://trello.com/es.

[2] Git. https://git-scm.com/.

[3] ESP32 - Espressif. https://www.espressif.com/en/products/socs/esp32.

[4] ESP8266 - Espressif. https://www.espressif.com/en/products/socs/esp8266.

[5] ESP32-WROOM-32. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/modules-and-boards.html#esp32-wroom-32.

[6] Power Glove. https://es.wikipedia.org/wiki/Power_Glove.

[7] Nintendo. https://www.nintendo.es/.

[8] Raspberry Pi. https://www.raspberrypi.org/.

[9] PlatformIO. https://platformio.org/.

[10] Arduino Reference. https://www.arduino.cc/reference/en/.

[11] Biblioteca MPU9250 - GitHub. https://github.com/bolderflight/MPU9250.

[12] GitHub. https://github.com/.

[13] Hostapd. https://wiki.gentoo.org/wiki/Hostapd.

[14] Biblioteca WiFi - Arduino. https://www.arduino.cc/en/Reference/WiFi.

[15] Python. https://www.python.org/.

[16] Biblioteca SocketServer - Python. https://docs.python.org/es/3/library/socketserver.html#module-socketserver.

[17] Biblioteca Socket - Python. https://docs.python.org/es/3/library/socket.html#module-socket.

[18] Biblioteca Struct - Python. https://docs.python.org/es/3/library/struct.html#module-struct.

[19] Apache HTTP Server. https://httpd.apache.org/.

[20] Chart.js. https://www.chartjs.org/.

[21] Ejemplo gráficas con Chart.js - Repositorio gitHub. https://gist.github.com/MOAMIndustries/340442db992b2e6bfa77a85d752f2d52.

Parte II

Anexos

Apéndice A

Código principal

```
#include "MPU9250.h"
#include "WiFi.h"

#include "wifi.h"
#include "MPU.h"
#include "flex.h"
#include "datos.h"
#include "glove.h"


//WiFi
bool statusWifi; //comprobacion wifiConnect
IPAddress subnet(172,16,0,0);
IPAddress esp(172,16,0,200);
IPAddress rpi(172,16,0,100);
const int port = 2320; //puerto servidor

// Acelerometro
MPU9250 IMU(Wire,0x68);
bool statusMPU; //comprobacion initMPU

//estructuras
guante_t guante1;
acelerometro_t *acel = &guante1.acel;
flexibles_t *sflex = &guante1.sflex;

// Sensor flexible
extern float sflexCalLow[5];
extern float sflexCalHigh[5];

char txbuffer[sizeof(guante_t)]; //buffer para almacenar estructura guante1

```

```
33
34 void setup()
35 {
36     Serial.begin(115200);
37
38 //____________________ CONEXION WIFI
39     statusWifi = wifiConnect(esp, rpi, subnet);
40     if (!statusWifi) {
41         Serial.println("Connection error");
42         exit(1);
43     }
44
45     memset(&guante1, 0, sizeof(guante1)); //preparar memoria para estructura guante
46
47
48 //____________________ INICIALIZACION ACELEROMETRO
49     statusMPU = initMPU(&IMU);
50     if (!statusMPU) {
51         Serial.println("IMU initialization unsuccessful");
52         exit(1);
53     }
54     acel->IMU = &IMU; //meter objeto IMU en la estructura
55
56
57 //____________________ SENSOR FLEXIBLE
58     pinMode(FLEX_PIN1, INPUT);
59     pinMode(FLEX_PIN2, INPUT);
60     pinMode(FLEX_PIN3, INPUT);
61     pinMode(FLEX_PIN4, INPUT);
62     pinMode(FLEX_PIN5, INPUT);
63
64
65 //____________________ CALIBRACION SENSOR FLEXIBLE______________
66
67     //MANO ABIERTA
68     Serial.println("Modo calibracion mano abierta");
69     delay(2000);
70     calibrationFlex(sflexCalLow);
71
72     delay(2000);
73
74     //MANO CERRADA
75     Serial.println("Modo calibracion mano cerrada");
76     delay(2000);
77     calibrationFlex(sflexCalHigh);
78
79 }
80
```

```
81  void loop()
82  {
83      //ACELEROMETRO
84      getDataMPU(acel);
85      showDataStructMPU(acel);
86  
87      //SENSOR FLEXIBLE
88      getDataFlex(sflex);
89      showDataStructFlex(sflex);
90  
91      memcpy(txbuffer, &guante1, sizeof(txbuffer)); //copiar guante1 a buffer memoria
92  
93      // CLIENTE TCP
94      WiFiClient client;
95      if(!client.connect(rpi, port)) {
96          Serial.println("Connection to server failed");
97          delay(1000);
98          return;
99      }
100     Serial.println("Connected to server");
101     client.write(txbuffer, sizeof(txbuffer));
102     Serial.println("Disconnecting...");
103 
104     client.stop();
105 
106     delay(500);
107 }
```

Repositorio GitHub con el proyecto completo: Repositorio GitHub (esp32)

Apéndice B

Servidor TCP y Servidor Web

En el siguiente enlace se encuentran los archivos pertenecientes al Servidor TCP implementado en la Raspberry, y a los dos archivos **index.html** y **myChart.js** implementados para el Servidor Web. Además aparecen los archivos **demo.html** y **demoJS.js** pertenecientes a la *demo*.

Repositorio GitHub (raspberry)