Universidad Zaragoza
1542

Trabajo de Fin de Grado

Monitor de presión arterial y pulso

Blood pressure Monitor

Autor/es

Isabel Angélica Romeo

Director/es

Bonifacio Martín del Brío

Grado en ingeniería electrónica y automática

Departamento de Ingeniería electrónica y comunicaciones
Escuela de ingeniería y arquitectura de Zaragoza

Noviembre 2021

Resumen

En este documento se describe el proceso de investigación, diseño y desarrollo de un monitor de presión arterial y pulso. Esta clase de monitores se usan tanto en el área médica como en el hogar, de ahí la motivación de desarrollar un producto que se pueda usar en ambos casos.

Para desarrollar un monitor con estas funcionalidades primero se expone una serie de conceptos clave, como pueden ser la presión arterial y el método oscilométrico de medida, para dar una mejor base de conocimiento para la posterior explicación. El diseño del prototipo y la selección de componentes se hace en base a los requerimientos y prestaciones que se comentarán más adelante. El desarrollo del esquema del circuito se realiza mediante el software EasyEDA.

Una vez explicados esta serie de conceptos, el documento se centra en el esquema general del circuito y sus componentes más relevantes. Dos de los componentes más importantes del monitor son el microcontrolador y el sensor de presión. El microcontrolador utilizado para este prototipo es el ESP32, ya que ofrece elevadas prestaciones a un precio reducido.

Por otro lado, una de las partes críticas de este producto es la adquisición y procesamiento de los datos de presión. Para ello se ha elegido un sensor de presión de precisión del fabricante Honeywelll, además se utilizarán una serie de filtros digitales para obtener las señales necesarias para realizar la medición.

En cuanto al funcionamiento general del monitor, consiste en un sistema de tiempo real formado por una serie de tareas con distintas prioridades y que se comunican entre ellas. La tarea central es la máquina de estados del sistema, que es la que controla todo el funcionamiento del prototipo. Por otro lado, tenemos otras tareas que complementan a la central, como la tarea de adquisición de datos y la de visualización de resultados.

Para el procesado de las medidas, en una primera fase se utilizó el entorno de Matlab, donde se utilizan una serie de funciones para el cálculo de los resultados. Una vez validados los resultados sobre Matlab, las funciones empleadas para realizar el procesamiento de las señales en el microcontrolador se implementan en lenguaje C. Para esta implementación se utilizará el programa Eclipse IDE con el entorno de desarrollo que ofrece el fabricante del microcontrolador, Espressif IDF.

La validación de las mediciones del prototipo se realizó comparándolas con las obtenidas por el monitor TOPCOM Blood Pressure Monitor BPM Wrist 3311 y siguiendo una serie de condiciones que se comentan en el apartado de validación de medidas.

Por último se expondrán los resultados obtenidos y los problemas surgidos en el desarrollo de esta aplicación. Además se comentarán posibles mejoras e implementaciones futuras, que podrán mejorar la calidad del prototipo de monitor de tensión desarrollado.

Índice

1. Introducción

1.1 Motivación y objetivos del proyecto

Las enfermedades cardiovasculares (CVDs) están siendo la principal causa de muerte a lo largo del planeta durante los últimos 20 años. Representan el 32% de las muertes mundiales y un 29% de las muertes en España. Este tipo de enfermedades son causadas por desórdenes en el corazón y en los vasos sanguíneos, dentro de este grupo se incluyen enfermedades coronarias, cerebrovasculares, y otras. Más de 4 de cada 5 muertes causadas por CVDs son debidas a ataques de corazón y 1 de cada 3 muertes ocurre en personas menores de 70 años. [1]

Los factores más importantes que conducen a este tipo de enfermedades son una dieta no saludable, sedentarismo, tabaquismo y consumo excesivo de alcohol. Estos factores se ven reflejados en la salud de los individuos, como presión arterial elevada, alto nivel de glucosa y lípidos en sangre, sobrepeso y obesidad. Estos factores indicativos pueden ser medidos regularmente en instalaciones médicas para llevar un seguimiento y poder prever el riesgo de infarto, stroke, etc.

Uno de los factores indicativos más importantes es la presión arterial, ya que es un trastorno grave que aumenta significativamente el riesgo de sufrir cardiopatías y otras enfermedades del corazón. Se calcula que hay alrededor de 1.130 millones de personas con hipertensión y apenas una de cada cinco lo tienen controlado. [1]

A raíz de ahí surge la motivación de crear un prototipo con el que poder tomar medidas de presión arterial de forma fácil y sencilla en casa o en cualquier lugar, sin necesidad de desplazarse a unas instalaciones médicas. Esta accesibilidad permitirá llevar un control más continuo, tanto de los niveles de tensión arterial como de pulso.

Partiendo de estas ideas, los objetivos principales de este proyecto son:

- Entender el método oscilométrico y cómo aplicarlo para obtener las medidas de presión arterial.
- Diseñar un prototipo electrónico de monitor conforme a las prestaciones requeridas utilizando un ESP32 como microcontrolador.
- Programación en tiempo real del sistema de medida del monitor a partir de las librerías de Espressif en lenguaje C.
- Utilizar Matlab como paso intermedio para realizar una primera aproximación de los cálculos de las medidas, para más tarde adaptar el código a lenguaje C para su implementación en el microcontrolador.
- Diseño e implementación de los filtros adecuados para el procesamiento digital de señal.
- Validación de las medidas con un dispositivo homologado y obtención de las estadísticas de precisión del prototipo desarrollado.

1.2 Estructura del documento y cronograma

Este documento está organizado de tal forma que se expone el diseño y desarrollo de una aplicación en tiempo real en un sistema empotrado dentro de un microcontrolador. El código desarrollado se puede encontrar en uno de los anexos adjuntados con este documento.

La memoria se estructura de la siguiente manera. En el Capítulo 2 se explican una serie de conceptos clave sobre el funcionamiento del sistema cardíaco y arterial, además de exponer las patologías relacionadas con la presión arterial y los métodos empleados para la estimación de la tensión arterial y la frecuencia cardiaca. El Capítulo 3 se divide en dos partes, en la primera se muestra el esquema general del circuito, detallando la función de cada bloque y los componentes que lo forman. En la segunda parte del Capítulo 3 se muestra el proceso de filtrado digital, simulación de medidas en Matlab y por último se pasa a la explicación del software desarrollado. El Capítulo 4 se enfoca en la metodología de validación de las medidas y se muestra la precisión obtenida con el prototipo en diferentes fases de medida. Por último, en el Capítulo 5 se muestran las conclusiones obtenidas tras el desarrollo del prototipo, poniendo el foco en la precisión de los resultados. Por otro lado se proponen una serie de mejoras tanto para la calidad de las medidas como para ofrecer una mejor experiencia de uso al usuario.

La Fig. 1 muestra las principales actividades llevadas a cabo en este proyecto mediante un diagrama de Gantt. Entre estas actividades se encuentran: estudio de la metodología de obtención de las medidas, diseño del circuito e implementación del software en el microcontrolador.

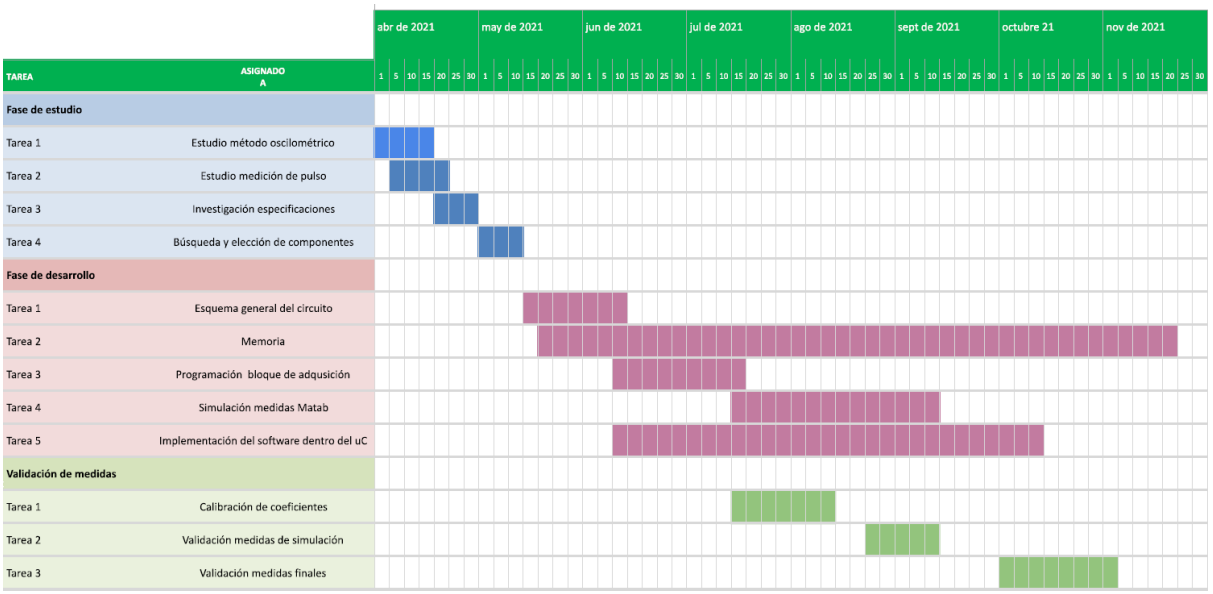


Fig. 1 Diagrama de Gantt

2. Marco de referencia. Medida de la presión arterial

2.1 Fisiología cardiovascular

El sistema cardiovascular está formado por el corazón y los vasos sanguíneos: una red de venas, arterias y capilares que transportan oxígeno desde los pulmones a los tejidos de todo el cuerpo a través de la sangre mediante el bombeo del corazón. Otra función del sistema cardiovascular es transportar dióxido de carbono, un producto de desecho, desde todo el cuerpo hasta el corazón y los pulmones y, finalmente, eliminar el dióxido de carbono mediante la respiración [12].

El ciclo cardíaco es la secuencia rítmica de contracción y relajación miocárdica (latido). A la contracción miocárdica se le llama sístole y durante ella se impulsa la sangre fuera del corazón. A la relajación miocárdica se le llama diástole y durante ella se llena de sangre el corazón.

El funcionamiento del corazón consiste en una serie ordenada de pasos (figura 2), empieza con la sangre desoxigenada regresando del resto del cuerpo al corazón por la *vena cava superior* (VCS) y la *vena cava inferior* (VCI), esta sangre entra entra en la aurícula derecha (AD) que se encuentra en estado de relajación (diástole), desde allí la sangre fluye a través de la *válvula tricúspide* (VT) hacia dentro del ventrículo derecho (VD). Teniendo en cuenta que las válvulas auriculo-ventriculares se encuentran cerradas, la presión sanguínea aumenta conforme se van llenando, de forma que se produce una sístole o contracción que provoca que se abran, mientras tanto la diástole ventricular sigue llenando los ventrículos y la sangre desoxigenada es bombeada a través de la *válvula pulmonar* (VP) hacia la arteria pulmonar principal (APP). Desde allí, la sangre fluye a través de las arterias pulmonares derecha e izquierda hacia adentro de los pulmones.

En los pulmones, se incorpora oxígeno y se retira dióxido de carbono a la sangre durante el proceso de respiración. Después de que la sangre recibe oxígeno en los pulmones, se llama sangre oxigenada.

En la segunda etapa la sangre oxigenada fluye desde los pulmones de vuelta a la aurícula izquierda (AI) a través de cuatro venas pulmonares, las válvulas aurículo-ventriculares se mantiene cerradas mientras se produce una sístole ventricular, la sangre oxigenada fluye a través de la válvula mitral (VM) hacia adentro del ventrículo izquierdo (VI).

El ventrículo izquierdo (VI) bombea la sangre oxigenada a través de la válvula aórtica (VAo) hacia la aorta (Ao), la principal arteria que transporta sangre oxigenada al resto del cuerpo.

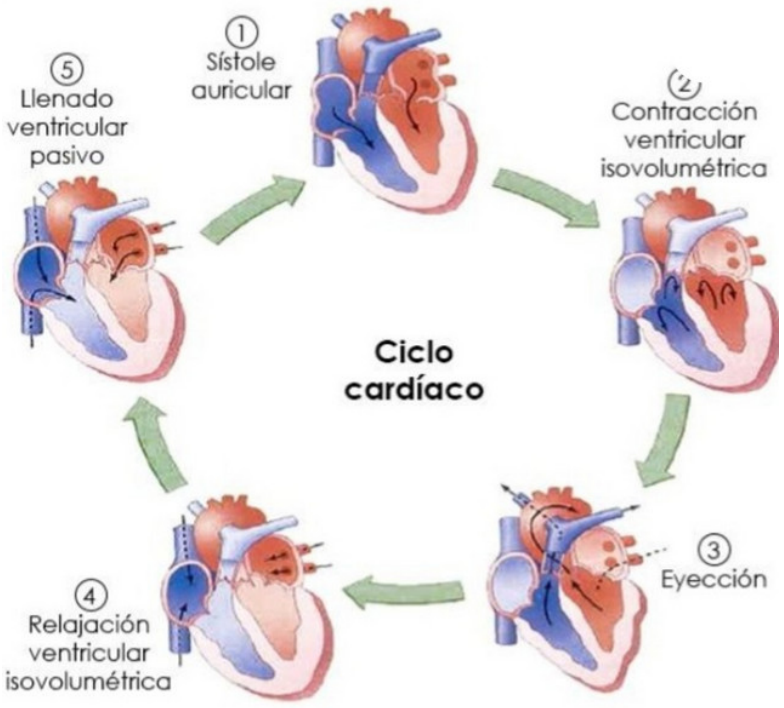


Fig. 2 Ciclo cardiaco
Fuente: ref [14]

2.1.1 Sistema arterial

El sistema arterial consiste en una serie de vasos, sucesivamente ramificados, que van desde las arterias de gran tamaño, como la aorta y la pulmonar, pasando por las de mediano, pequeño tamaño y arteriolas, hasta los capilares o vasos de intercambio. Las arterias encargadas de transportar la sangre al corazón son la pulmonar y la aorta. La arteria pulmonar es de recorrido corto y mide 3 cm de diámetro, comunica el ventrículo derecho con las arterias pulmonares derecha e izquierda. Por otro lado, la arteria aorta es de largo recorrido, tiene un diámetro igual que la pulmonar y sus cuatro divisiones principales son la aorta ascendente, el arco aórtico, la aorta torácica y la aorta abdominal. La aorta es el tronco principal de las arterias sistémicas [2].

Entre las arterias de las extremidades superiores está la arteria subclavia o también llamado axilar, al adentrarse en el brazo se denomina arteria braquial como se observa en la Figura 3, más tarde se ramifica en el antebrazo en radial y cubital, las cuales irrigan toda esa región.

La medición de parámetros como la presión arterial, se deben realizar en estado de reposo y el brazo izquierdo apoyado a la altura del corazón

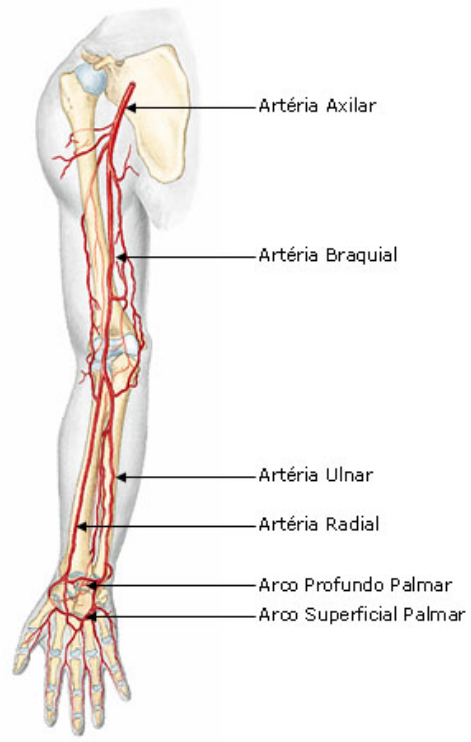


Fig. 3 Esquema arterial del brazo.
Fuente: ref [15]

2.2 Presión arterial

Por definición la presión arterial es la fuerza que ejerce contra la pared arterial la sangre que circula por las arterias. La presión arterial incluye dos mediciones: la presión sistólica, que se mide durante el latido del corazón (momento de presión máxima), y la presión diastólica, que se mide durante el descanso entre dos latidos (momento de presión mínima). La unidad de medida de la presión arterial es el mmHg. Primero se registra la presión sistólica y luego la presión diastólica, por ejemplo: 120/80. También se llama presión sanguínea arterial o tensión arterial [3].

Conceptualmente la presión arterial se diferencia de la tensión arterial, ya que la primera es la fuerza que ejerce la sangre que circula por las arterias, mientras que la tensión arterial es la tensión que realiza la sangre contra la pared de las arterias [4].

La ecuación que relaciona ambas expresiones es la ley de Laplace [5]:

$$P = \frac{T}{r}$$

Donde T es la tensión, P es la presión y $r$corresponde al radio de un vaso sanguíneo.
Cuando se tiene la medida aproximada de la presión arterial (PA), es posible identificar enfermedades de diferente índole, causadas por múltiples factores que se pueden identificar, de tal forma que sea posible brindar al paciente un tratamiento apropiado.

2.2.1 Hipertensión arterial

La Hipertensión Arterial (HTA) es la elevación continua de la PA por encima de los niveles normales, se considera como límites normales una Presión Arterial Sistólica (PAS) de 140 mmHg ó superior y Presión Arterial Diastólica (PAD) de 90 mmHg ó superior.

En la tabla 1 se muestra la clasificación que se debe tener en cuenta a la hora de diagnosticar a un paciente, estándar planteado por la OMS y JNC.

TABLA 1. **Clasificaciones de la HTA de la OMS y JNC VI**

	PAS		PAD
Clasificación de la HTA (OMS)			
Óptima	< 120		< 80
Normal	< 130		< 85
Normal-Alta	130-139		85-89
Grado 1, ligera	140-159		90-99
Subgrupo «limítrofe»	140-149		90-94
Grado 2, moderada	160-179		100-109
Grado 3, severa	≥ 180		≥ 110
HTA sistólica aislada	≥ 140		< 90
Subgrupo «limítrofe»	140-149		< 90
Clasificación de la HTA (JNC VI)			
Óptima	< 120	y	< 80
Normal	< 130	y	< 85
Normal alta	30-139	o	85-89
HTA o estadio 1	140-150	o	90-99
HTA o estadio 2	160-179	o	100-109
HTA o estadio 3	≥ 180	o	N ≥ 110

Algunos de los factores causantes de la prevalencia de esta patología son el tabaquismo, una dieta poco saludable, un estilo de vida sedentario, la obesidad y algunas alteraciones psicológicas(estrés, alteraciones emocionales, etc.). Por otro la edad y el sexo también son factores que diferencian la afectación de este trastorno.

2.2.2 Hipotensión arterial

La hipotensión arterial es la condición en la que se presenta una presión arterial baja continuada causada por la irrigación sanguínea deficiente, lo que provoca una afectación a la oxigenación y nutrición celular, pudiendo provocar síntomas como vértigo o mareo. Si se presenta una caída de presión de solo 20 mmHg puede ocasionar problemas en algunos órganos.

Existen tres tipos de hipotensión; Hipotensión Ortostática, Hipotensión Mediada Neuralmente (NMH) e Hipotensión grave producida por una pérdida súbita de sangre (shock), infección o reacción alérgica intensa.

- **La hipotensión ortostática** es producida por un cambio súbito en la posición del cuerpo, generalmente al pasar de estar tumbado a estar de pie y usualmente dura sólo unos pocos segundos o minutos. Si este tipo de hipotensión ocurre después de comer, se denomina hipotensión ortostática posprandial y afecta más comúnmente a los adultos mayores, aquellos con presión arterial alta.
- **La hipotensión mediada neuralmente** afecta con más frecuencia a adultos jóvenes y niños, y ocurre cuando una persona ha estado de pie por mucho tiempo.
- **El Shock** se define como la pérdida severa de la adecuada irrigación sanguínea a los órganos lo que disminuye el suministro de oxígeno y puede causar daños al organismo. La hipotensión severa se presenta con un descenso mayor a 40 mmHg de la presión sistólica, el tratamiento para este tipo de patología debe realizarse de forma simultánea con el de la enfermedad causante.

La presión arterial baja suele ser causada por fármacos como los ansiolíticos, antidepresivos, diuréticos, medicamentos para el corazón, entre ellos los que se utilizan para tratar la hipertensión arterial y la cardiopatía coronaria. Otras causas de presión arterial baja pueden ser la diabetes avanzada, anafilaxia (una respuesta alérgica potencialmente mortal), cambios en el ritmo cardíaco (arritmias), deshidratación, desmayo, etc [6].

2.3 Medición de la tensión arterial

La medición de la presión arterial (PA) se puede realizar de dos maneras distintas, mediante métodos directos e indirectos. La primera metodología, también conocida como invasiva, se debe tomar una muestra en el interior de la arteria por medio de un catéter, este procedimiento se utiliza únicamente con fines clínicos e investigativos.

Por otro lado, en el método indirecto, las mediciones se realizan mediante un esfigmomanómetro el cual obtienen resultados aproximados. Dentro del método indirecto se pueden encontrar otros tres métodos; palpatorio, auscultatorio y oscilométrico [7]:

- **Método palpatorio:** este método consiste en identificar el pulso ubicado a la altura de la muñeca cercano al dedo pulgar, utilizando un brazalete como instrumento de medición que se infla hasta que el pulso desaparezca. Posteriormente se desinfla paulatinamente hasta que el pulso vuelva a aparecer, en este momento se toma la medición de presión sistólica. Por último, cuando las pulsaciones vuelven a la normalidad, se toma la medición de la presión diastólica. Uno de los principales inconvenientes del método palpatorio es que es muy poco preciso, ya que se requiere de mucha práctica para realizar bien las tomas.

- **Método auscultatorio:** consiste en identificar los sonidos (sonidos de Korotkoff, Fig 4) que emite la arteria parcialmente ocluida y por consiguiente se determina el flujo arterial. Para realizar esta práctica se debe disponer de un estetoscopio y un brazalete. La medición se realiza en cinco fases; en la primera fase se identifica la presión sistólica y en la quinta la diastólica [8].

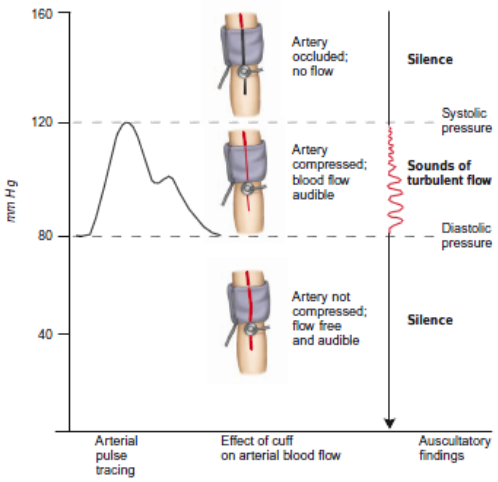


Fig. 4 Ruido de Korotkoff
Fuente: ref [16]

- **Método oscilométrico:** es la metodología indirecta más usada actualmente, el objetivo es monitorear las oscilaciones de la señal producida por la presión arterial. Igual que en los métodos anteriores se usa un brazalete para realizar la medidas y, mediante el procesamiento de la señal obtenida, se determina la presión sistólica (PS), diastólica (PD) y media (MAP). El método oscilométrico consiste en inflar el brazalete hasta llegar 40-50 mmHg por encima de la presión sistólica estándar (120 mmHg), esta presión es transmitida a través del brazo hasta las paredes de la arteria que pasará a estar más ocluida conforme aumente la presión. Una vez se llega a esta presión se pasa a desinflar el brazalete hasta bajar por debajo de la presión diastólica estándar (80 mmHg), en este tramo la arteria se va abriendo, la sangre empieza a circular de nuevo y las oscilaciones llegan a su máxima amplitud [9].

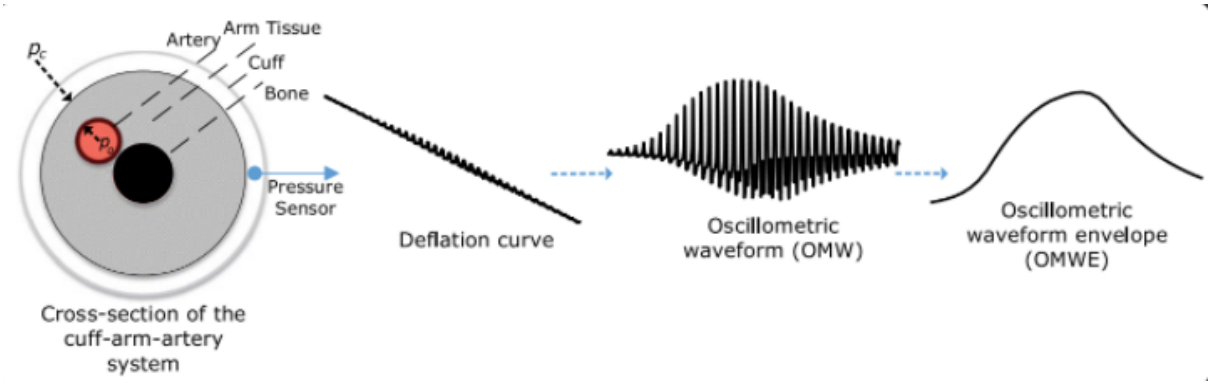


Fig. 5 Método oscilométrico
Fuente: ref [9]

EL cálculo de la presiones sistólica, diastólica y media se hace siguiendo estos tres pasos (Fig 5):

1. **Obtención de la curva de presión durante el desinflado:** durante el periodo de desinflado obtenemos la señal de presión a través del sensor de presión conectado con el brazalete. Esta señal tiene dos componentes importantes: la componente de baja frecuencia causada por el aumento paulatino de presión del brazalete y la componente de las oscilaciones de presión de la arteria. Esta última componente se llama onda oscilométrica(OMW) y será analizada para estimar la PS, PD, y MAP.
2. **Extracción de la onda oscilométrica(OMW):** para extraer esta onda a partir de la señal de presión del brazalete se utiliza un filtro digital de paso banda para eliminar la componente de presión del brazalete y permitir el paso de las oscilaciones de presión con unas frecuencias de corte de 0,5 Hz y 3,5 Hz .
3. **Creación de la envolvente de la onda oscilométrica(OMWE)**: la amplitud de los pulsos oscilométricos aumenta hasta llegar al máximo y luego desciende a medida que se desinfla el brazalete. Debido a que la amplitud de la onda oscilométrica carga con la mayoría de información para la estimación de la medida, muchos de los algoritmos oscilométricos utilizan la envolvente de la onda oscilométrica (OMWE). Esta envolvente se forma uniendo pico a pico los máximos locales separados por un número mínimo de muestras.

Algoritmos oscilométricos

Los algoritmos oscilométricos son utilizados para estimar la presión sistólica, diastólica y media mediante el análisis de los cambios en la morfología de las oscilaciones de presión. Estos algoritmos se pueden aplicar en diferentes estados de procesamiento de las señales grabadas y utilizan diferentes técnicas para la estimación de las medidas como el uso coeficientes empíricos, análisis de la pendiente de la OMWE, algoritmos de *machine learning* o modelado de la envolvente.

El algoritmo más popular y el que se implementa en este prototipo es el Algoritmo de Máxima Amplitud (MAA). Éste se basa en el supuesto de que la elasticidad arterial es máxima cuando la presión del brazalete es igual a la presión arterial, que se produce cuando las paredes de la arteria están mínimamente dilatadas. Basado en este supuesto la presión arterial media (MAP) se encuentra en la posición donde la envolvente de la señal de oscilaciones (OMWE) alcanza su máximo. La presión sistólica y diastólica se identifican cuando la amplitud de la oscilación alcanza unos ciertos ratios de la amplitud máxima (figura 6). Estos ratios se obtienen de forma empírica y varían en el rango de 0.45 a 0.73 para el coeficiente de presión sistólica, *rs* y de 0.69 a 0.83 para el coeficiente de presión sistólica *rd* [9].

$$\frac{Ad}{Am} = r_d \qquad \frac{As}{Am} = r_s$$

Ad: $Amplitud\ diastólica$
As: $Amplitud\ sistólica$
rd: $ratio\ de\ amplitud\ diastólica$
rs: $ratio\ de\ amplitud\ sistólica$

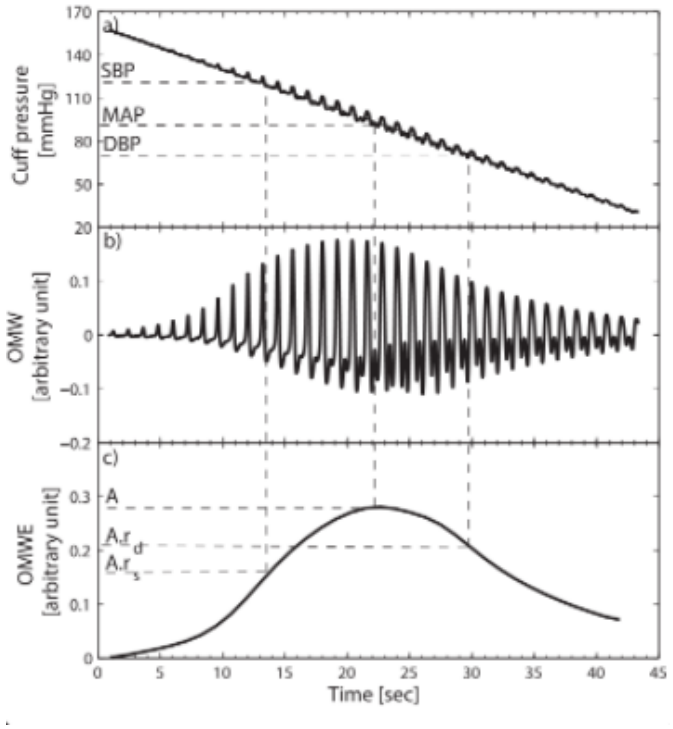


Fig 6. Algoritmo de Máxima Amplitud (MAA)
Fuente: ref [9]

2.4 Medición del pulso

En cuanto a la estimación de la frecuencia cardíaca se ha optado por analizar la señal de presión en el dominio de la frecuencia mediante la Transformada Rápida de Fourier (FFT) [10].

La FFT es un algoritmo que permite calcular la transformada de Fourier discreta (DFT) y su inversa cuando el número de muestras de la señal es una potencia de dos. Este algoritmo se emplea en una amplia variedad de aplicaciones, desde el tratamiento digital de señales y filtrado digital a la resolución de ecuaciones en derivadas parciales. El rango de frecuencias cubierto por el análisis FFT depende de la cantidad de muestras recogidas y de la proporción de muestreo. Para el cómputo de la DFT se utiliza:

$$x[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi kn}{N}}$$

Donde Xn es la señal de oscilaciones (OMW) y N es la longitud de la señal obtenida Xk y corresponde a las N muestras equiespaciadas de la transformada de Fourier. Se debe cumplir que $N \geq L$, siendo L el número de muestras de la señal Xn.

Una vez obtenida la secuencia de la transformada de Fourier de la señal de oscilaciones(OMW) , Xk, se busca el armónico más prominente dentro del rango de frecuencias cardiacas (0,8 - 2 Hz aprox) y se obtiene la frecuencia cardíaca usando la siguiente fórmula:

$$Heart\,rate \;=\; harmonic \;\times\; 60 \qquad (3)$$

En el caso de la Figura 7 se ha recortado un tramo de la señal de oscilaciones de presión (OMW) de 512 muestras al que se le aplica la transformada rápida de Fourier (FFT) obteniendo el armónico más prominente en 1,25 Hz lo que resulta en un pulso de 75 bpm.

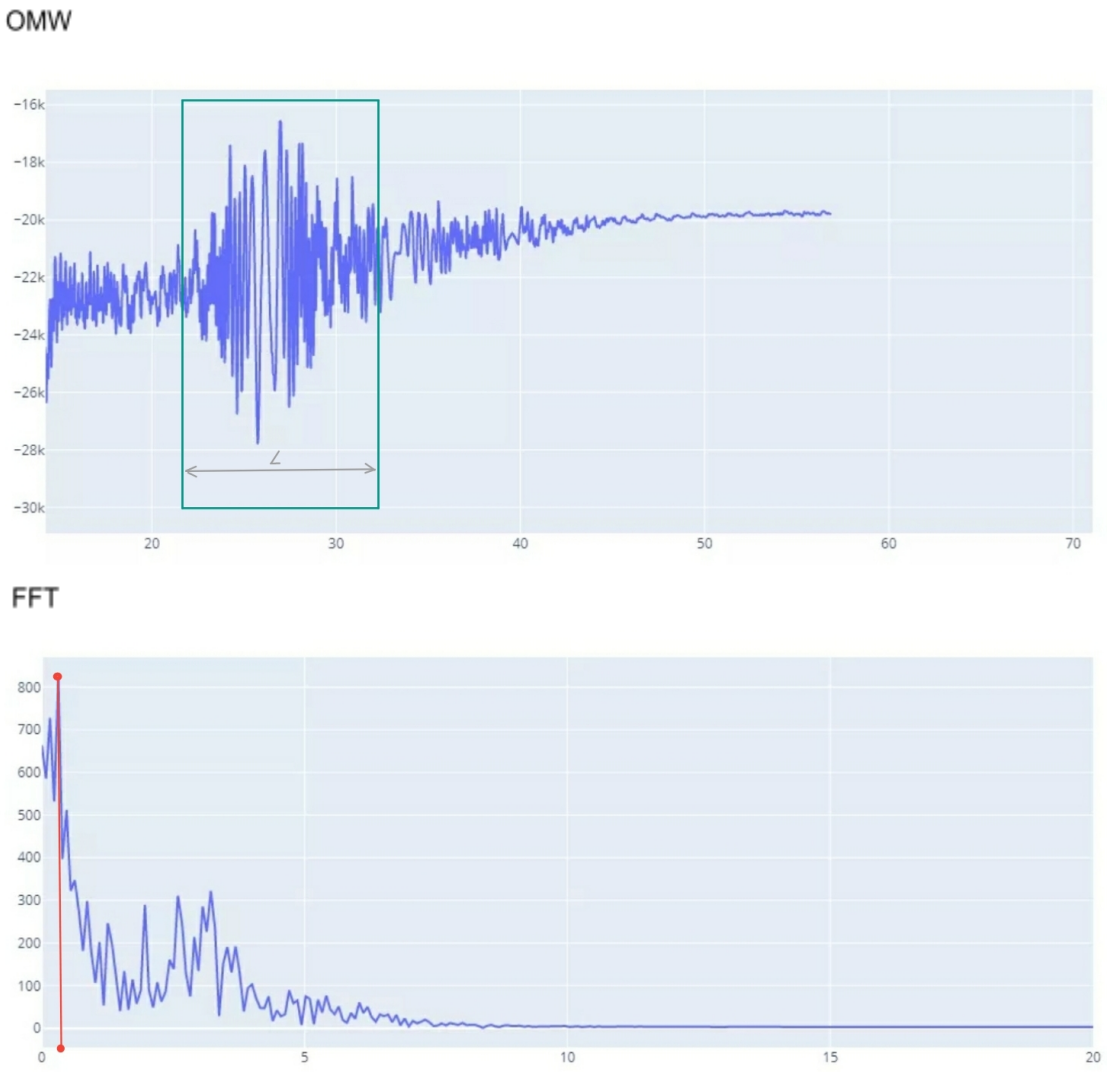


Fig 7. Aplicación de la FFT a la señal de oscilaciones de la presión arterial

3. Diseño y desarrollo del hardware

En este capítulo se lleva a cabo la descripción del diseño del prototipo hardware de monitor de presión arterial y sus diferentes partes. Para la elección de los componente de cada parte se han seguido una serie de requerimientos básicos:

- Bloque adquisición de datos:
  - Rango de presión mínimo: 0-200 mmHg (Absolutos)
  - Sensor calibrado.
  - Etapa amplificadora.

- Bloque neumático:
  - Caudal de aire mínimo: 2 L/min

- Bloque de control:
  - Módulo conversor A/D
  - Módulo PWM
  - Interfaz SPI
  - Interfaz I2C
  - Interfaz UART

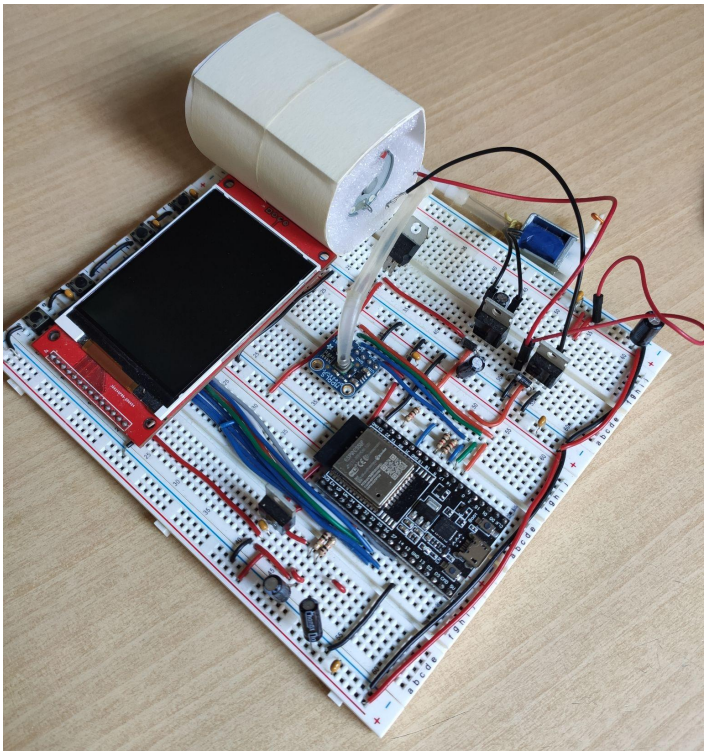


Fig 8. Circuito Monitor de tensión en protoboard

3.1 Esquema general del circuito

El diseño de este prototipo (Fig. 8) está compuesto por diversos bloques (Fig. 9): bloque de adquisición de datos (1), bloque neumático (2), bloque de alimentación (3), bloque de control (4) y bloque de visualización (5).

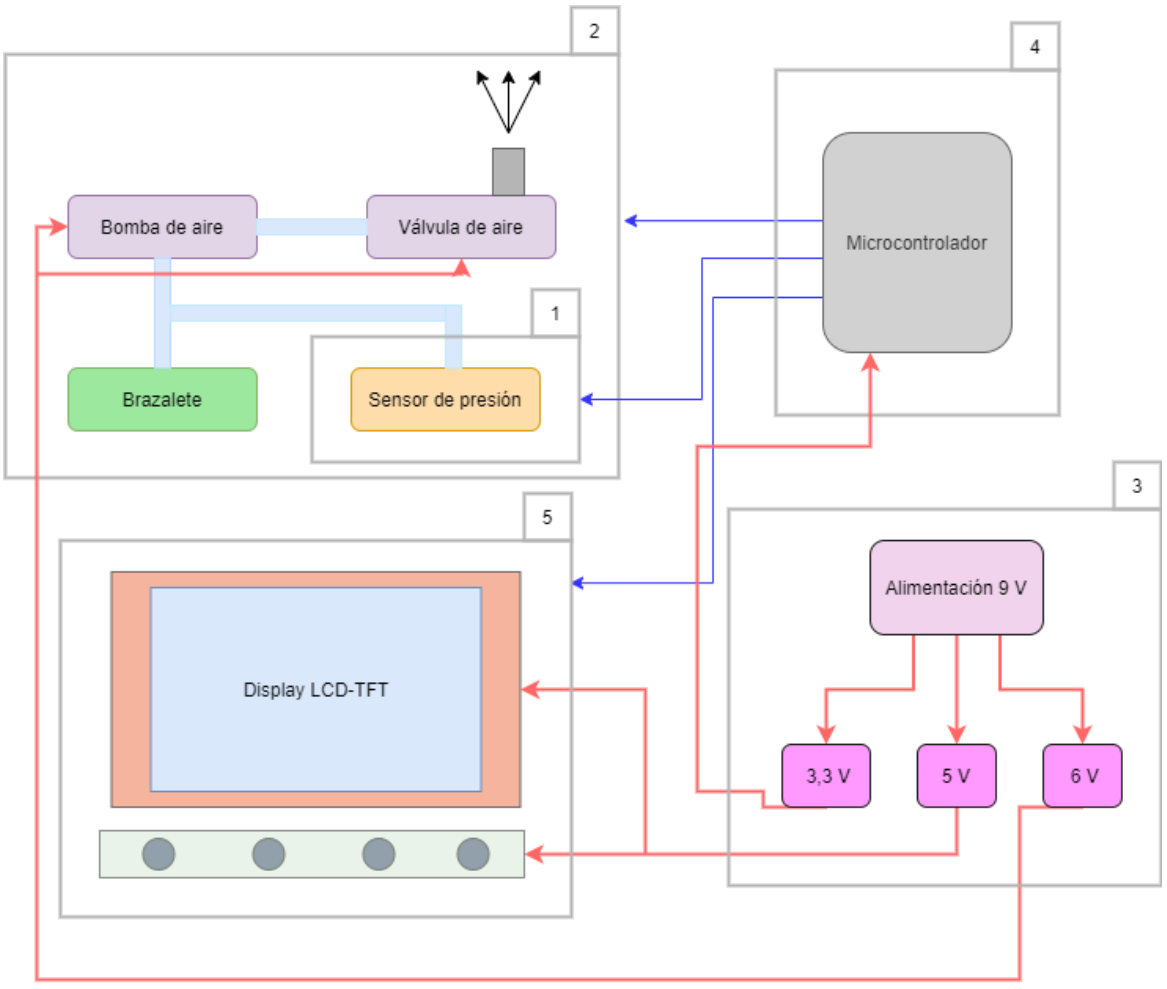


Fig 9. Diagrama de bloques esquema general del circuito

3.1.1 Bloque de adquisición de datos

El bloque de adquisición está formado por un módulo electrónico que cuenta con un sensor de presión de precisión, los amplificadores correspondientes, conversor A/D y los terminales necesarios para comunicación I2C (Fig 10 y 11). Inicialmente se utilizó un sensor de presión convencional conectado a una etapa amplificadora a su vez conectada con el módulo conversor A/D del micro, pero este sensor no estaba calibrado y no cumplía las especificaciones requeridas.

La banda de error total (TEB) es una especificación que incluye las principales fuentes de error del sensor de presión e indica el peor error que podría experimentar, 1.25% FFS (Full Scale Span) en el caso de este sensor [13]. La TEB no debe ser confundida con la precisión, ya que esta es en realidad es un componente de la TEB. El rango de presiones de este sensor es de 0-25 psi absolutos, suficiente para cubrir los requerimientos del monitor, ya que la máxima presión absoluta que va a soportar este sensor es de 200 mmHg (3,86 psi). Una de las peculiaridades de este sensor es que tiene un puerto de metal de 2,5 mm de diámetro al que conectamos el tubo que irá conectado al brazalete.

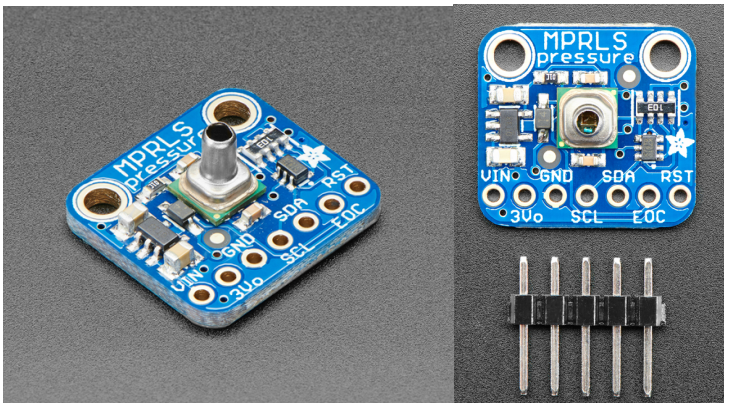


Fig 10. MPRLS ported pressure sensor Honeywell
Fuente: ref [17]

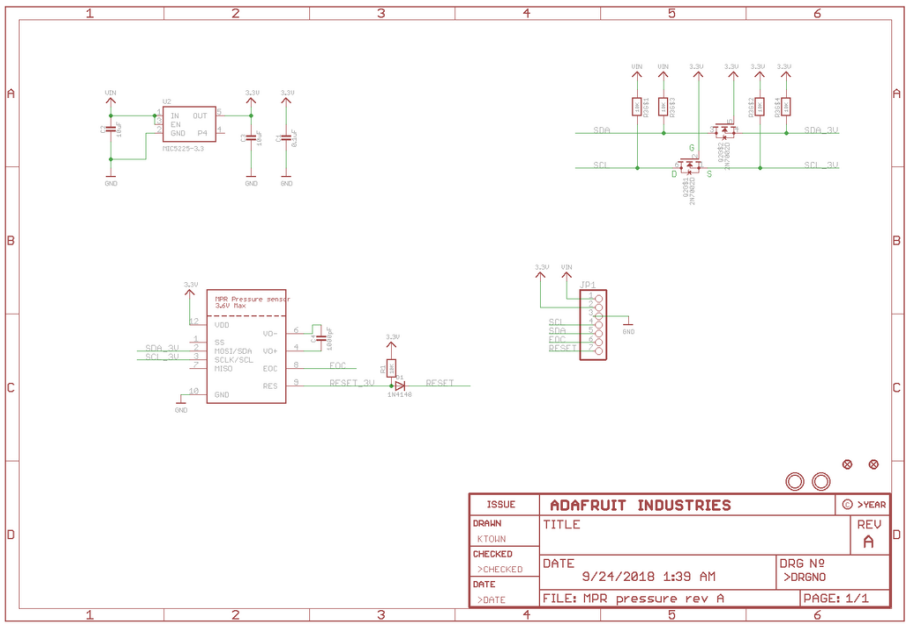


Fig 11. Esquema circuital del sensor de presión
Fuente: red [17]

3.1.2 Bloque neumático

Este bloque está formado por los elementos encargados de la regulación del aire (Fig. 11). Como elementos actuadores están la bomba de aire y la válvula de aire. Por otro lado, están los tubos de conexión que conectan los elementos actuadores con el brazalete y éste con el sensor de presión.

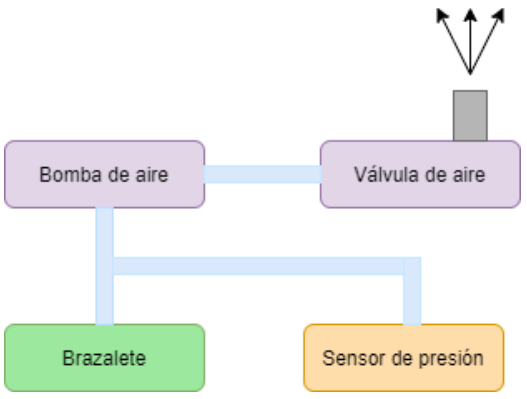


Fig 12. Diagrama del bloque neumático

La estructura neumática parte de la bomba, la cual se encarga de suministrar el aire que pasa a través de la válvula hasta llegar al brazalete. Del brazalete sale otra rama que lo une con el sensor de presión. Todas estas conexiones se han montado con un tubo de plástico flexible de 2,5 mm, excepto la conexión con el sensor de presión, que tiene un diámetro de 2 mm. Por otro lado, para la salida de aire de la válvula se ha utilizado un tubo de 0,5 mm,

ligeramente obstruido para conseguir una pendiente en la fase de desinflado de 2-3 mmHg/s [9].

La bomba debe tener un caudal que permita el inflado en un tiempo razonable y que tenga la suficiente potencia como para llegar a la presión de medida. Un tipo de bombas que cumplen estas características y son baratas son las llamadas bombas peristálticas, que utilizan rodillos para impulsar el fluido (en este caso aire) y pueden alcanzar un caudal de 3,4 l/m. El control de la bomba se realiza a través de un PWM.

Fig 13. Bomba peristáltica
Fuente: Aliexpress

Del mismo modo que se necesita inflar el brazalete, también se deberá desinflar después de su uso. De hecho, es la parte más importante de la medida, ya que la presión arterial se calcula durante el desinflado. Para ello se usa una válvula (Fig 14) de aire accionada electrónicamente, su funcionamiento es como el de un relé: cuando circula corriente en éste, activa un electroimán que hace un contacto mecánico, dejando pasar el aire y, por lo tanto, desinflando el brazalete.

Fig 14. Electroválvula
Fuente: Aliexpress

El brazalete se eligió con un tamaño de 22-48 cm de diámetro, por lo que este monitor se ha enfocado para el uso de personas adultas.

3.1.3 Bloque de alimentación

Es el bloque básico para el funcionamiento del aparato, ya que se encarga de suministrar energía tanto a los accionadores (bomba, electroválvula), como al control y a la interfaz con el usuario (botones y pantalla).

Para ello se requieren diferentes tensiones de alimentación, que vienen condicionadas por las tecnologías de construcción de los componentes:

- Bomba de aire y electroválvula: alimentación a 6 VDC
- Pantalla LCD y botones: alimentación a 5 VDC
- Unidad de control: alimentación a 3 VDC

Las posibles soluciones ante esta variedad de tensiones son muchas, pero principalmente tenemos dos: regulación por elevación de tensión o por reducción de la misma.

En el presente proyecto se ha escogido la segunda opción por su simplicidad al permitir emplear reguladores de tensión lineales, que aunque sean menos eficientes que otros de tipo conmutado, son mucho más sencillos, baratos y fáciles de adquirir. De esta forma, la alimentación viene de una fuente de 9V. El motivo de emplear una fuente de 9V, y no de otro tipo (por ejemplo, 4 baterías de 1,5V en serie) es la tensión de “dropout” de los reguladores lineales. Este parámetro nos impone una mínima diferencia de tensión entre la entrada y la salida del regulador, condición que no se puede satisfacer con baterías de 6V para el regulador de 5V. Existen reguladores llamados de “bajo dropout”, pero son más caros y pueden dar lugar a una reparación defectuosa en caso de ser sustituidos por uno convencional.

Así, desde la fuente se alimenta de forma independiente a cada uno de los circuitos integrados de los reguladores, y además se disponen condensadores de filtrado a la entrada y salida de los reguladores para filtrar ruidos de media y baja frecuencia.

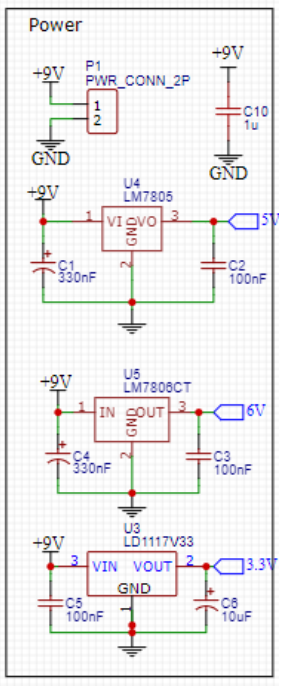


Fig 15. Esquema circuital del bloque de alimentación

3.1.4 Bloque de control

Constituye el elemento fundamental que controla el sistema y según su elección se deberán adoptar decisiones de diseño consecuentes. Existen componentes electrónicos integrados programables llamados microcontroladores que con un precio muy bajo y velocidad de desarrollo alta integran casi todos los elementos necesarios, es por lo tanto obvia la elección de este tipo de componente. En definitiva, un microcontrolador es un componente integrado que incluye en el chip: CPU, memoria, periféricos, etc. Dentro de la amplia variedad que existe se busca el más barato que incluya las siguientes especificaciones:

- Menor número de pines posibles (abarata costes).
- Capaz de entrar en modo de bajo consumo para guardar fecha y hora cumpliendo con la autonomía deseada.
- Largo ciclo de vida y soportado por el/los fabricante/s.
- Generador de pulsos con tecnología de modulación de ancho de pulso (PWM) integrado para el control de la bomba de aire.
- Frecuencia de reloj capaz de realizar las operaciones y mostrar la interfaz en plazo.

El microcontrolador elegido para este proyecto es el ESP32 de Espressif, montado en la placa de desarrollo WROOM 32D. Este microcontrolador es ampliamente usado en el desarrollo de prototipos electrónicos debido a su variedad de módulos, compatibilidad con diferentes lenguajes de programación, amplia documentación y bajo coste.

Algunas de sus características más destacables son:

- Single or Dual-Core 32-bit LX6, con frecuencia de reloj de hasta 240 MHz.
- 520 KB de SRAM, 448 KB de ROM y 16 KB de RTC SRAM.
- Conectividad Wi-Fi con velocidades de hasta 150 Mbps.
- Soporta Bluetooth v4.2 y BLE.
- 34 GPIOs programables.
- Hasta 18 canales de 12-bit SAR ADC y 2 canales de 8-bit DAC.
- 4 x SPI, 2 x I2C, 2 x I2S, 3 x UART.
- Ethernet MAC para conectividad LAN física.
- PWM para control motor y hasta 16 canales de PWM para LED.

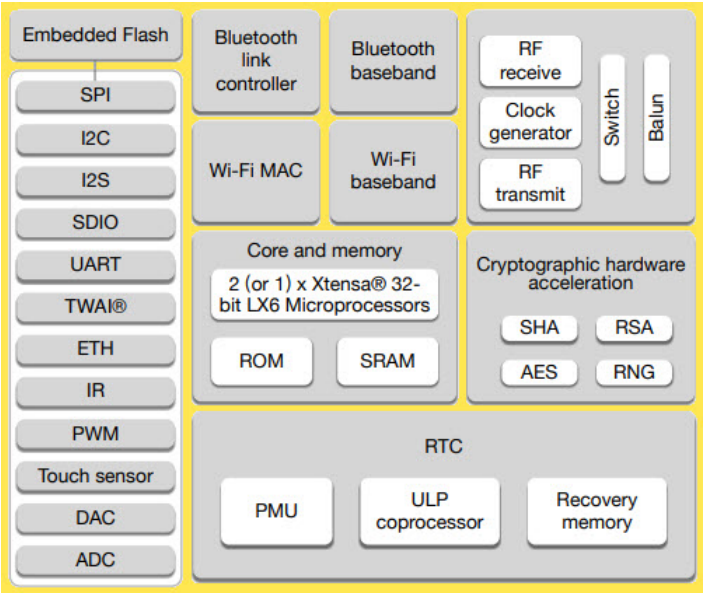


Fig 16. Diagrama de Bloques ESP32 wroom 32D
Fuente: ref [19]

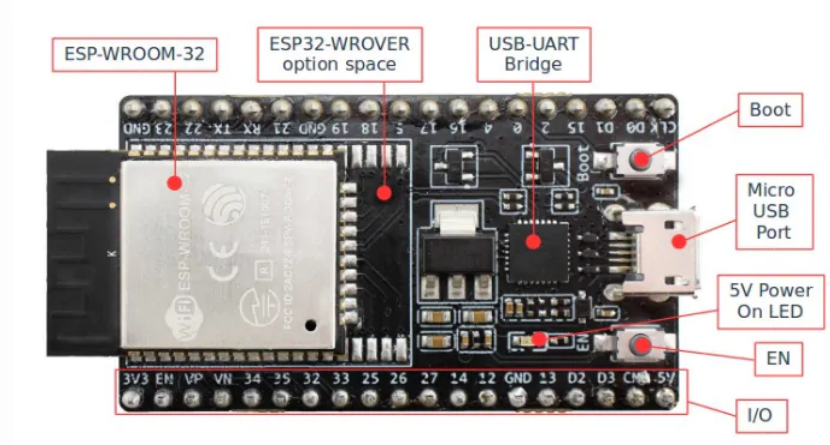


Fig 17. ESP32 wroom 32D
Fuente: ref [18]

3.1.5 Bloque de visualización

Para que el usuario visualice las medidas y pueda interactuar con el prototipo es necesario crear un bloque de visualización. Este bloque está compuesto por el *display* y los cuatro botones que lo controlan. El *display* es una pantalla LCD-TFT de 320x240 alimentado a 5V, controlado por un *driver* ILI9341 y conectado al microcontrolador vía SPI.

En cuanto a la botonera, lo más sencillo y eficiente es el uso de unos pocos botones, abarata el coste y su manejo es muy fácil e intuitivo. Por lo tanto, la botonera está compuesta por cuatro botones, tres de los cuales están reservados para la movilidad por la interfaz de usuario. El cuarto botón no tiene una función asignada todavía, por lo que se denomina auxiliar y se mantiene por si es necesario en mejoras futuras del prototipo.

Fig 18. Display LCD-TFT
Fuente: Aliexpress

3.1.6 Componentes auxiliares

Como puede observarse en la Fig 19, en el circuito se incluyen además algunos componentes estándar adicionales:

- Resistencias eléctricas, para limitar la corriente a ciertos componentes para que funcionen sin dañarse, como en el caso de los LED. También para regular parámetros, como la ganancia del amplificador del sensor de presión.

- Condensadores eléctricos: como es sabido, almacenar carga eléctrica durante cortos periodos de tiempo. Sirven para regular parámetros como la ganancia de amplificadores o para el desacoplo de señales. Esta última característica los convierte en un componente esencial cuando existen otros de carácter inductivo como la bomba o la válvula: cuando se aplica una tensión sobre un componente inductivo, la demanda de corriente es muy alta y, de no existir los condensadores de desacoplo, todo el sistema podría sufrir una bajada de tensión, dejando de funcionar el aparato.

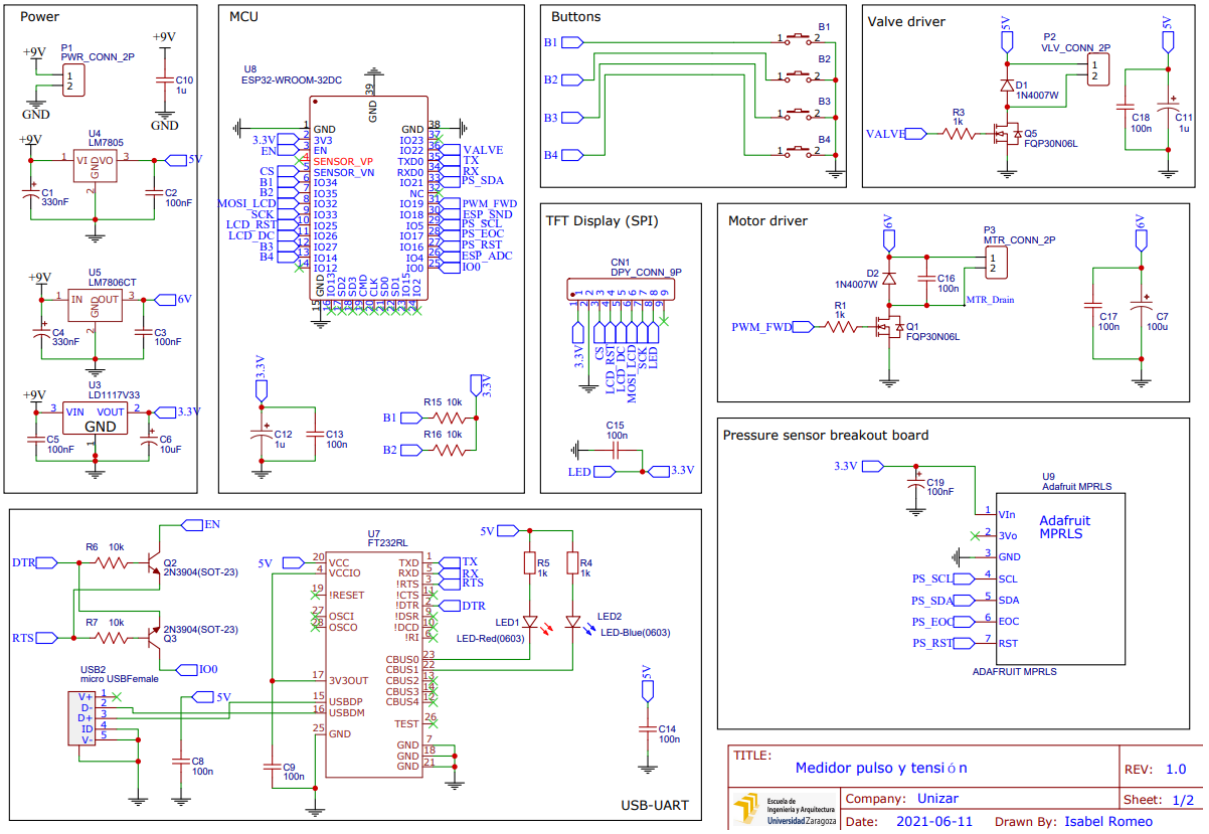


Fig 19. Esquema circuital completo

4. Procesado digital de la señal

4.1 Filtros digitales

Los filtros digitales son una parte fundamental del procesamiento digital de señales. Su función principal es modificar el espectro de una señal. Actualmente, existen una infinidad de aplicaciones, tales como eliminación de determinadas componentes espectrales, enfatizar o atenuar algunas componentes espectrales, o desfasar estas componentes, etc.

La principal diferencia entre un filtro digital y otro analógico es que los analógicos operan sobre señales continuas, mientras los digitales trabajan sobre una secuencia de datos (las muestras digitalizadas de la señal analógica).

Como se ha comentado en la sección del método oscilométrico, la señal de presión obtenida del sensor tiene dos componentes principales: la componente de baja frecuencia producida por el cambio de presión y la componente provocada por las fluctuaciones en la amplitud de las oscilaciones. Como la señal que interesa obtener es la producida por las oscilaciones de presión, se aplicará un filtro de paso de banda con frecuencias de corte de 0,5 Hz y 3,5 Hz, para eliminar el nivel de continua de la señal y, a la vez, permitir que pasen las fluctuaciones en la frecuencia cardíaca [9].

Para elegir qué tipo de filtro utilizar se ha realizado un *script* en python para analizar diferentes características de los filtros y elegir los adecuados. Debido a que la implementación de estos filtros se hace en una tarea con frecuencia de muestreo de 40 Hz dentro de un sistema de tiempo real, se ha decidido implementar un filtro *Butterworth* de orden dos, ya que filtros más complejos podrían aumentar el tiempo de ejecución de la tarea y bloquear el sistema.

La función de transferencia del filtro *Butterworth* de paso banda es la siguiente:

$$H(z) = \frac{0.1453z^4 - 0.2906z^2 + 0.1453}{z^4 - 2.2510z^3 + 2.3844z^2 - 1.1096z^1 + 0.2523}$$

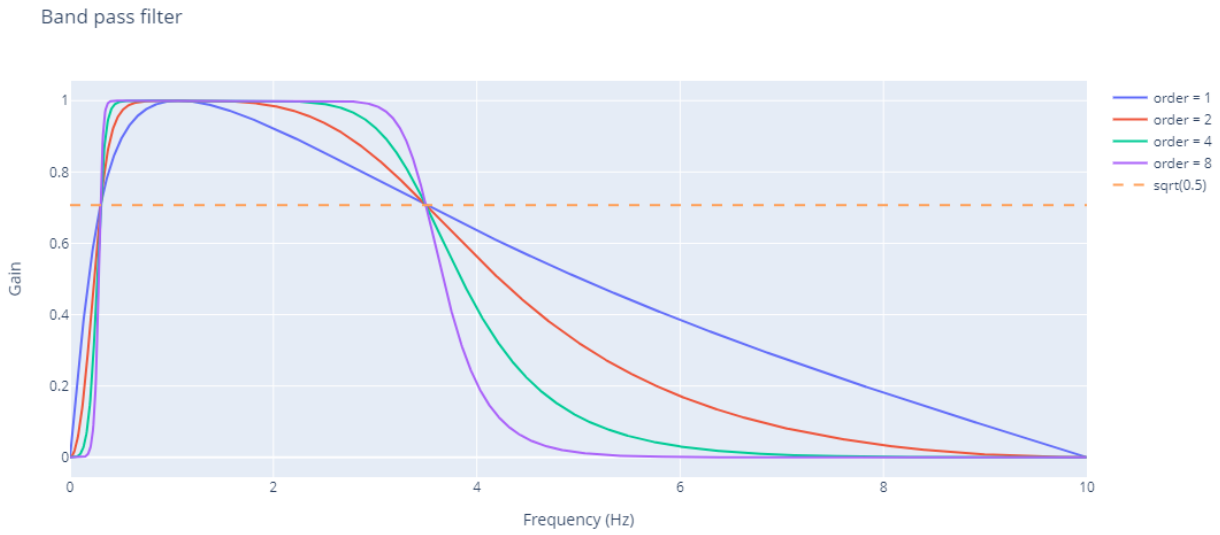


Fig 20. Diagrama Filtro paso banda

Los resultados obtenidos con el filtro *Butterworth* se pueden observar en las figuras 21 y 22. En la figura 21 se muestra la señal de presión del brazalete y en la figura 22 el resultado de pasarla a través del filtro paso banda, quedando así la señal de amplitud de las oscilaciones (OMW).

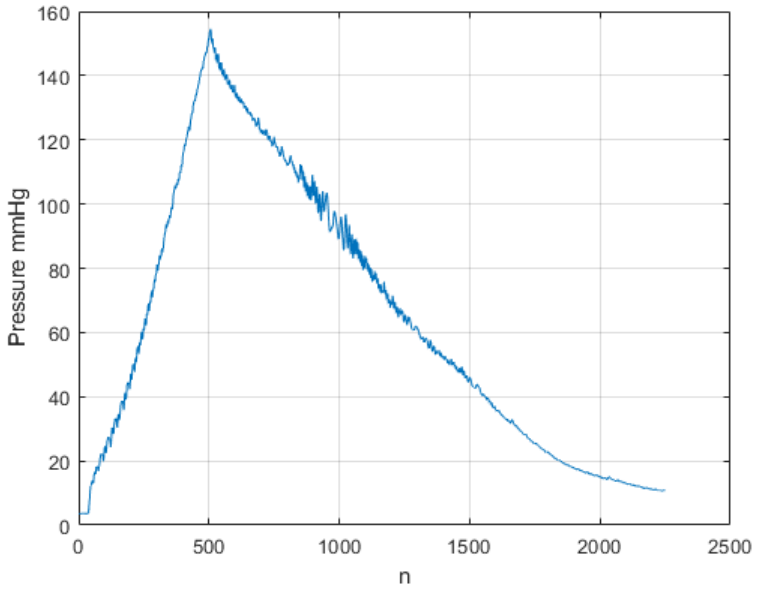


Fig 21. Señal de presión del brazalete

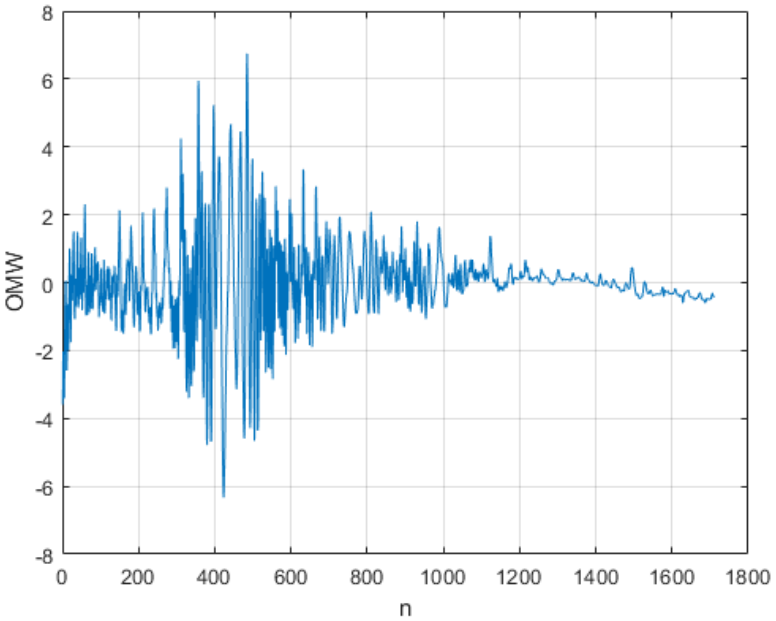


Fig 22. Señal de oscilaciones de presión (OMW)

4.2 Simulación de medidas en Matlab

La visualización de datos y gráficas en el entorno de programación del microcontrolador puede llegar a ser muy tedioso (a veces imposible), por lo que se ha optado por utilizar Matlab como entorno de simulación de las medidas y, una vez obtenido un resultado coherente en las medidas, se adaptará el código a nuestro microcontrolador con las librerías de Espressif.

El primer paso es obtener los datos de presión en un formato con el se pueda trabajar en Matlab, para ello se ha empleado el programa Megunolink que adquiere los datos a través de la interfaz UART, los representa en tiempo real y al finalizar la medida se exportan en un formato de valores delimitados por coma (*csv*).

Una vez obtenidos estos datos se importan a un fichero de Matlab en forma de *array*. Se va a trabajar con dos señales, la señal de presión y la señal de oscilaciones (OMW). Estas señales se recortan para obtener sólo la parte del descenso de presión. A continuación se utiliza la función envelope() para obtener la envolvente de la señal de oscilaciones (OMWE). Esta función realiza el cálculo de la envolvente en tres pasos:

1. Búsqueda de los máximos locales a lo largo del array de oscilaciones (Figura 24, proceso 1).
2. Selección de los máximos separados por más de la distancia mínima entre picos (Figura 24, proceso 2).
3. Interpolación lineal de los máximos seleccionados para crear la envolvente (Figura 25, proceso 3).

Una vez tenemos la envolvente de la señal sólo queda encontrar la presión arterial media (MAP) que se encuentra en el punto máximo de la envolvente (figura 25, proceso 4). Este valor se multiplica por los coeficientes de presión sistólica y diastólica (rs y rd) para obtener los valores de comparación de presión sistólica y diastólica (figura 26, proceso 5). A continuación se divide la envolvente en dos partes separadas por el punto máximo, ya que la presión sistólica se encuentra a la izquierda del presión arterial media (PAM) y la sistólica a la derecha, esta separación se encuentra indicada en la figura 26 con las flechas verde (parte sistólica) y roja (parte diastólica). Por último se van a buscar los valores de la envolvente que se parezcan más a los valores de comparación de presión sistólica y diastólica. En la parte de la izquierda se compara cada valor de la envolvente con el valor de comparación de presión sistólica y cuando se encuentra el más parecido guardamos el índice de localización de este valor; lo mismo se hace en la parte derecha con el valor de comparación de presión diastólica. Para obtener los valores de presión sistólica y diastólica se buscan los valores que corresponden a las localizaciones obtenidas en el array de presiones (figura 26). Todo este proceso se ilustra en las figuras 23, 24, 25 y 26.

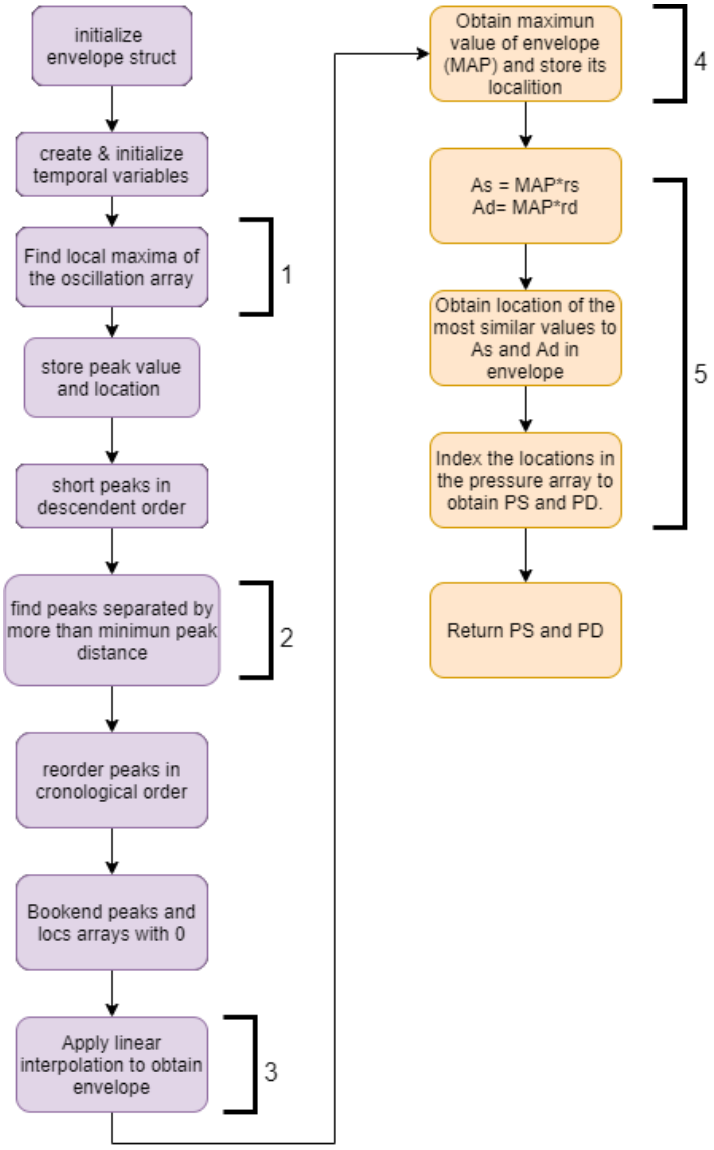


Fig 23. Diagrama de bloques del procesado de la medida

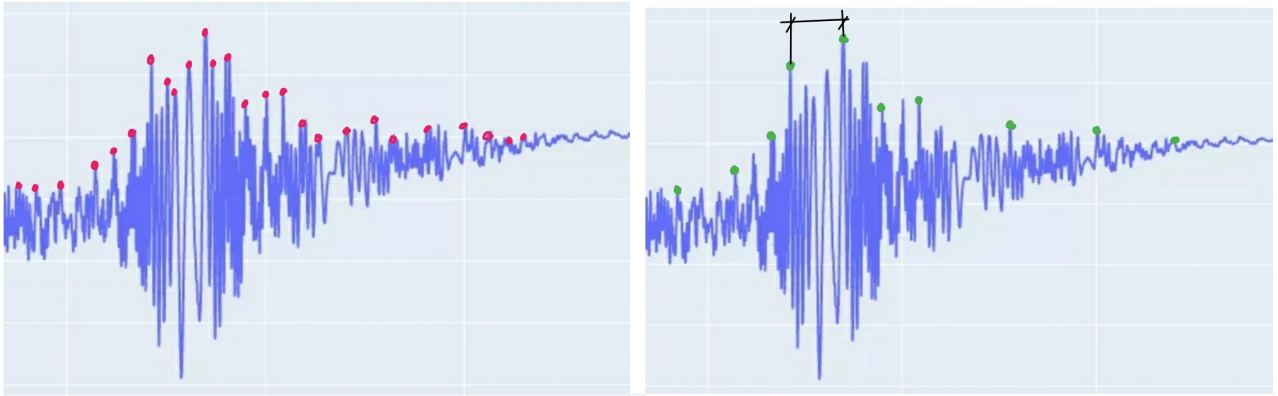


Fig 24. Procesos 1 y 2 de la Fig 18

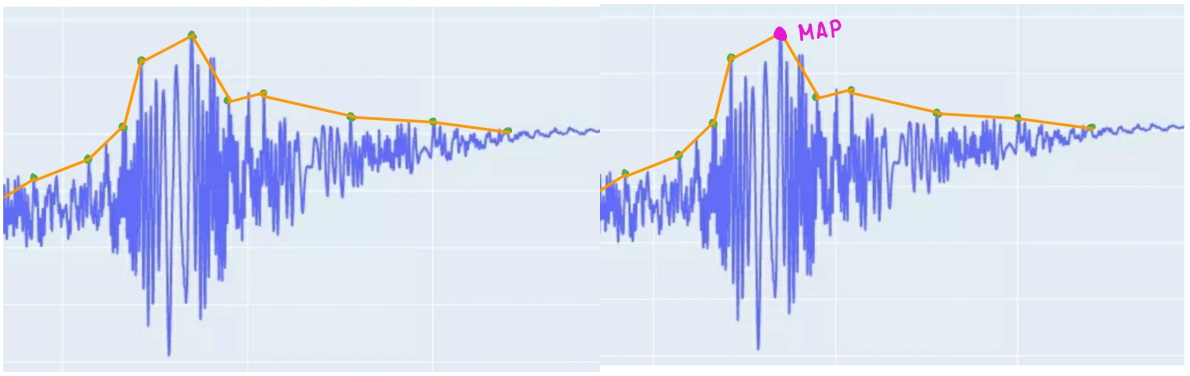


Fig 25. Procesos 3 y 4 de la Fig 18

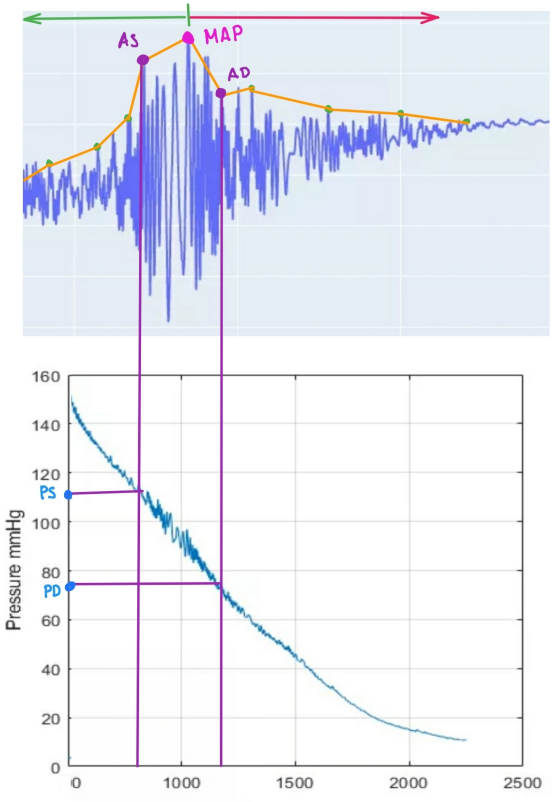


Fig 26: Proceso 5 de la Fig 18

5. Desarrollo del software

5.1 Entorno de desarrollo

En cuanto a la implementación del software en el microcontrolador, existen varios entornos en los que desarrollar aplicaciones con el ESP32, como VisualStudio, EclipseIDE o Arduino IDE. Para esta aplicación se ha escogido EclipseIDE, una vez instalado el entorno de desarrollo se debe añadir el *framework* que utiliza Espressif para la programación de sus dispositivos, ESP-IDF.

5.2 Estructura del software

La aplicación desarrollada para el prototipo de monitor de tensión arterial es un sistema de tiempo real que se basa en una conjunto de tareas que se van ejecutando periódicamente donde cada una de ellas tiene asignada una prioridad distinta en función de cual sea su objetivo.

La tarea principal se llama `pressure_control_handler` y consiste en una máquina de estados que controla el funcionamiento del prototipo. Por otro lado la tarea de adquisición de datos (`read_pressure_task`) es la tarea de mayor prioridad, ya que tiene el periodo de muestreo más pequeño (25 ms) para conseguir la precisión adecuada en la señal de presión. Por último, para el control del bloque de visualización se han desarrollado las tareas de interfaz de usuario (`gui_manager_task`) y de botones (`button_task`), las dos con la menor prioridad, ya que son las que menor uso de la CPU hacen.

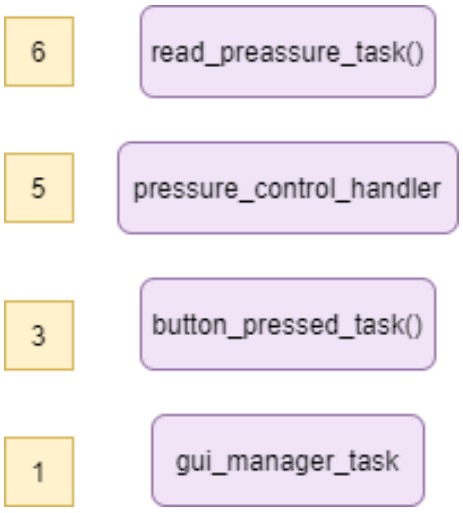


Fig 27. Diagram de bloques de tareas

5.3 Tarea adquisición datos

Esta tarea (Fig 29), como ya se ha comentado, es una de las más importantes y tiene tres funciones principales: es la encargada de recibir los datos de presión del sensor a través del protocolo de comunicación I2C, pasar estos datos por los filtros adecuados y almacenar estos datos en dos arrays circulares distintos.

El protocolo de comunicación serie I2C define la trama de datos y las conexiones físicas para transferir bits entre 2 dispositivos digitales. El puerto I2C incluye dos cables de comunicación, SDA (Serial Data) y SCL (Serial Clock). El protocolo permite conectar hasta 127 dispositivos esclavos con esas dos líneas, con velocidades de 100, 400 y 1000 kbits/s.

El protocolo I2C es uno de los más utilizados para comunicarse con sensores digitales, ya que a diferencia del puerto serie, su arquitectura permite tener una confirmación de los datos recibidos dentro de la misma trama, entre otras ventajas.

En el caso de I2C se diferencian dos elementos básicos, un MAESTRO y un ESCLAVO. La Figura X, muestra una conexión típica de tres dispositivos, el bus consiste en dos líneas llamadas, Serial Data (SDA) y Serial Clock (SCL).

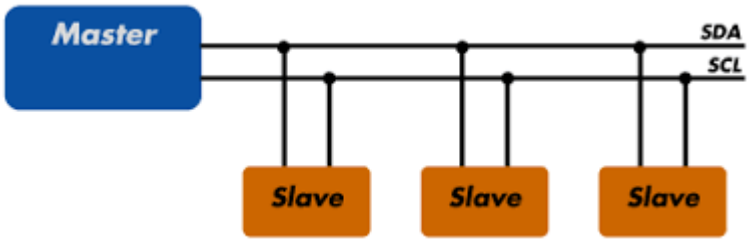


Fig 28. Interfaz I2c

Fuente: ref [20]

El MAESTRO I2C se encarga de controlar la línea de reloj, además de iniciar y parar la comunicación. La información binaria serie se envía a través de la línea SDA. Puede funcionar de dos maneras, como maestro-transmisor o maestro-receptor y sus funciones principales son:

- Iniciar la comunicación – S
- Enviar 7 bits de dirección – ADDR
- Generar 1 bit de Lectura ó Escritura – R/W
- Enviar 8 bits de dirección de memoria
- Transmitir 8 bits de datos –
- Confirmar la recepción de datos – ACK – ACKnowledged
- Generar confirmación de No-recepción, NACK – No-ACKnowledged
- Finalizar la comunicación

El esclavo generalmente suele ser un sensor. Este elemento es el encargado de suministrar la información de interés al MAESTRO. Puede actuar de dos formas: esclavo-transmisor ó esclavo-receptor. Sus funciones principales son:

- Enviar información en paquetes de 8 bits.
- Enviar confirmaciones de recepción, llamadas ACK

Para establecer conexión entre el puerto I2C del microcontrolador (Maestro) y el sensor de presión (esclavo) se han creado tres funciones `read_MPRLS_status()`, `read_MPRLS_data()` y `read_MPRLS_pressure()`. La primera función es la encargada de leer el byte de estado que aporta la información que se muestra en la tabla 2. La segunda se encarga de establecer conexión con el sensor, mandando primero la dirección y comando

de medida, a continuación se espera hasta que se ponga a cero la bandera de dispositivo ocupado y, entonces, se manda la dirección con el comando de lectura para obtener la medida. Por último `read_MPRLS_pressure()` transforma el valor obtenido del sensor a un valor de presión dentro del rango de 0 a 25 psi.

TABLE 15. I²C STATUS BYTE EXPLANATION

BIT (MEANING)	STATUS	COMMENT
7	always 0	—
6 (Power indication)	1 = device is powered 0 = device is not powered	Needed for the SPI Mode where the Master reads all zeroes if the device is not powered or in power-on reset (POR).
5 (Busy flag)	1 = device is busy	Indicates that the data for the last command is not yet available. No new commands are processed if the device is busy.
4	always 0	—
3	always 0	—
2 (Memory integrity/error flag)	0 = integrity test passed 1 = integrity test failed	Indicates whether the checksum-based integrity check passed or failed; the memory error status bit is calculated only during the power-up sequence.
1	always 0	—
0 (Math saturation)	1 = internal math saturation has occurred	—

Tabla 2. Status Byte I2C [13]

Una vez obtenidos los datos de presión, se procede al filtrado de la señal. De la misma forma que un filtro analógio tiene su modelo matemático expresado en ecuaciones diferenciales, los filtros digitales tienen su representación matemática a través de ecuaciones lineales en diferencias con coeficientes constantes, por lo general esta ecuación parte de la siguiente expresión:

$$\sum_{k=0}^{N} a_k \, y[n-k] = \sum_{k=0}^{M} b_k \, x[n-k]$$

Donde $x[n]$es la señal de entrada, $y[n]$ la señal de salida y a_iy $b_j$son los coeficientes del filtro.

Al tratarse de un sistema lineal invariante en el tiempo, es decir, está en reposo en el estado inicial, su función de transferencia va a ser de tipo racional:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{\sum_{k=0}^{N} a_k \, z^{-k}}$$

Por lo tanto para realizar el filtrado de la señal de entrada, $x[n]$, a través del filtro definido por los coeficientes a_iy b_j se emplea la siguiente igualdad recursiva:

$$y[n] = \frac{1}{a_0}\left\{\sum_{k=0}^{M} b_k \, x[n-k] - \sum_{k=1}^{N} a_k \, y[n-k]\right\}$$

De esta forma para implementar el filtro *Butterworth* desarrollado en el apartado de Procesamiento digital se han utilizado las siguientes ecuaciones:

$$x_k = raw\ pressure$$
$$y_k = (b_0 * x_k + b_2 * x_{k-2} + b_4 * x_{k-4}) - (a_0 * y_k + a_1 * y_{k-1} + a_2 * y_{k-2} + a_3 * y_{k-3} + a_4 * y_{k-4})$$
$$y_{k-4} = x_{k-3}\,;\ \ y_{k-3} = y_{k-2}\,;\ \ y_{k-2} = y_{k-1}\,;\ \ y_{k-1} = y_k$$
$$x_{k-4} = x_{k-3}\,;\ \ x_{k-3} = x_{k-2};\ \ x_{k-2} = x_{k-1};\ \ x_{k-1} = x_k$$

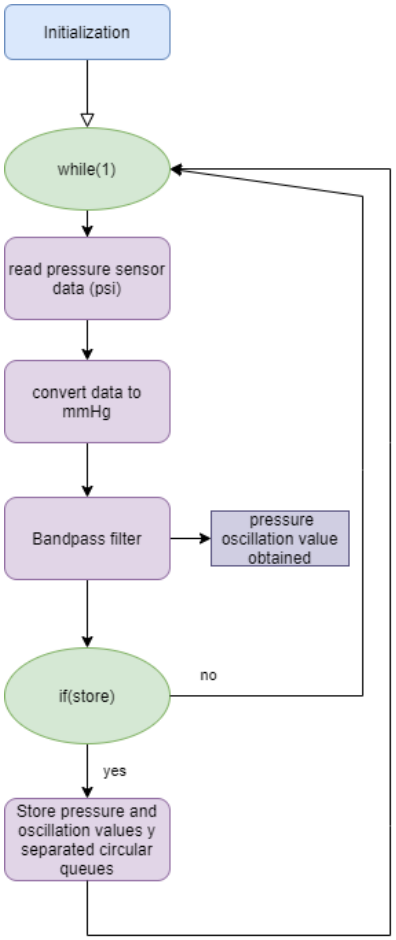


Fig 29. Diagrama de bloques de la tarea de adquisición de datos

Para terminar con la tarea de adquisición de datos, se almacenarán los datos en dos colas circulares. El código para la implementación de estas colas se ha extraído de un artículo de StackOverflow [11] al que se le ha hecho alguna modificación.

Se ha elegido esta implementación porque almacena una serie de datos sobre cada cola como cuántos elementos tiene, en qué índice está el último y primer elemento, etc. Esta información es imprescindible para la obtención de la medida.

5.4 Tarea botones

El control de la interfaz de usuario (Fig 30) se hace a través de 3 botones, dejando un cuarto botón de auxiliar por si es necesario en un futuro. Para facilitar la navegación por la pantalla esta botonera está programada como si fuera un encoder, de forma que dos de los tres botones se utilizan para incrementar y decrementar el contador del encoder, mientras que el último botón sirve para seleccionar el objeto seleccionado en la pantalla. Dentro del fichero `display_gui.c`, donde se desarrolla la programación de la pantalla, se encuentra la `función get_encoder_button_data_cb()` que es la encargada de actualizar la información del driver del encoder.

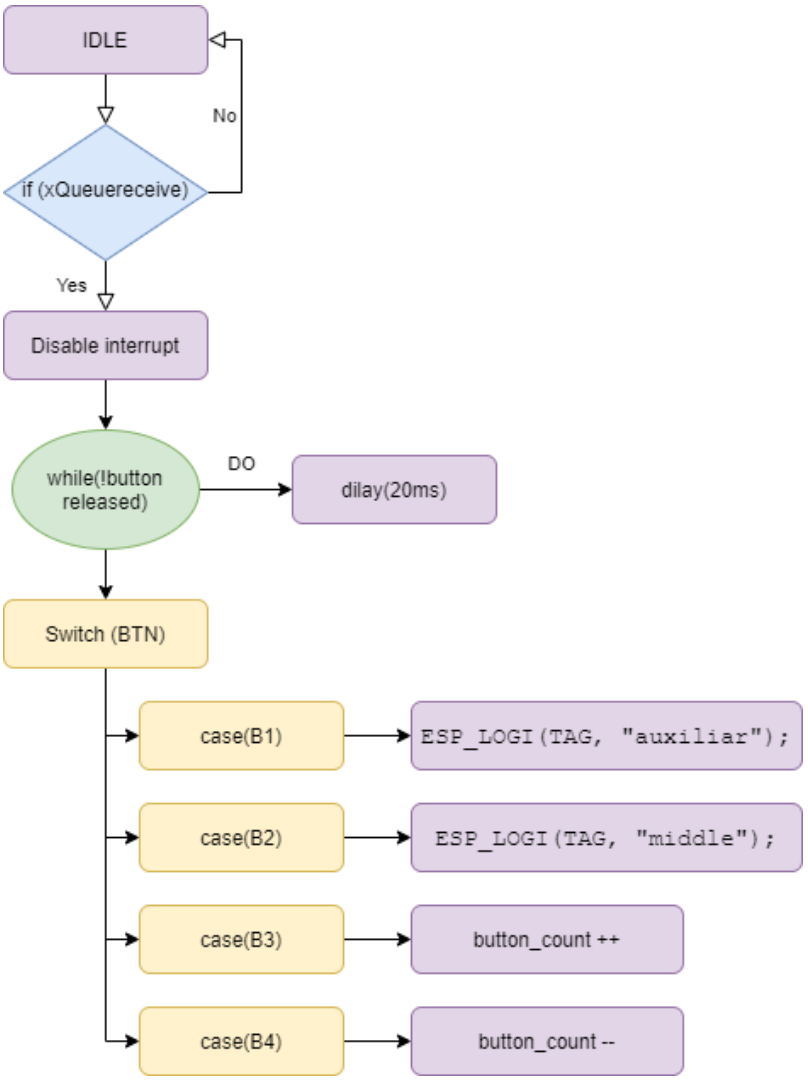


Fig 30. Diagrama de bloques tarea de botonera

5.5 Tarea de procesado de la medida

Otra de las tareas más importantes es la tarea de control (`pressure_control_handler()`), la cual se encarga de organizar el proceso de la medida desde el inicio hasta la visualización de los resultados. Este control se hace a través de una máquina de estados con seis estados, los cuales podemos ver en la figura 31.

- MEASURE_IDLE: Este es el estado inicial o de reposo, en el que lo único que se hace es abrir la válvula de aire.
- SCHEDULED: cuando se selecciona el botón Measure en la pantalla se pasa al estado de planificación en el que se cierra la válvula y se activa el PWM que controla el motor para comenzar a inflar el brazalete. Además se almacena el primer valor de presión (presión inicial) para, más tarde, restarlo al resultado de la medida. Por otro lado se activa una variable lógica llamada *store* con la que controlamos el almacenamiento de los datos de presión y oscilaciones en la tarea de adquisición. Por último se pasa el estado a INFLATING.
- INFLATING: en este estado se va actualizando el último valor de presión adquirido y se compara con el límite de presión suprasistólica (190 mmHg), una vez se alcance dicho valor se pasa al estado DEFLATING.
- DEFLATING: una vez se llega a este estado se abre la válvula y se cambia la referencia del PWM para que llegue gradualmente a cero en un espacio de 35 segundos, de esta forma se asegura que el brazalete se desinfla lentamente y la señal se almacenada correctamente. Por último se pone la variable *store* a false para dejar de almacenar datos y cambiamos el estado a DONE.
- DONE: este es el estado en el que se procesa la señal para obtener las medidas de presión sistólica, diastólica y pulso. Para ello se utiliza la función `init_envelope()` para inicializar la variable donde se van a almacenar los valores de la envolvente. Por otro lado se utiliza la función `find_peaks()` que se encarga de todo el procesamiento de la señal, desde la generación de la envolvente hasta la obtención de los resultados de la medida de la presión, el diagrama de funcionamiento de esta función se puede ver en la figura 18. A continuación se realiza el cálculo del pulso, para ello transformamos la señal de oscilaciones en un array complejo al que aplicaremos la transformada rápida de Fourier con la función fft(). Más tarde, con la función `get_highest_harmonic()` se obtiene el armónico más prominente del rango de frecuencias cardiacas y, por último, se cambia el estado a WRITE_STATUS.
- WRITE_STATUS: una vez terminado el procesado de las medidas, se escriben los tres resultados por pantalla a través de la función `write_measure_bpm()`.

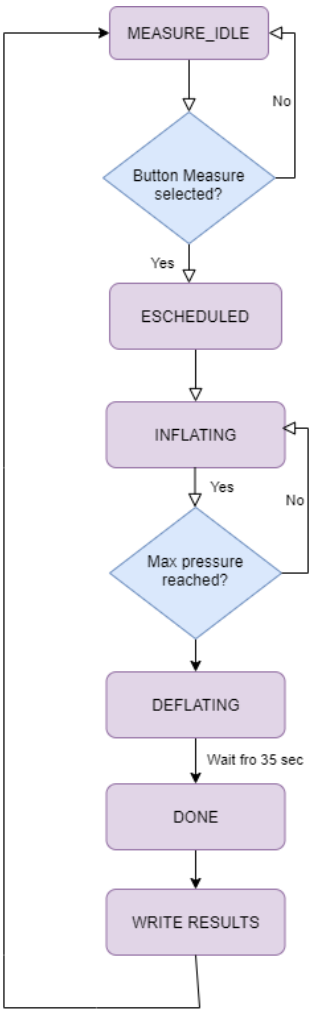


Fig 31. Diagrama de bloque tarea de control

5.6 Tarea UI/Visualización

En la tarea de interfaz de usuario (Fig 33) se muestran una serie de pantallas que parten de un menú principal como se observa en la figura 32. La pantalla principal es la MAIN MENU, una vez seleccionado el botón MEASURE se procede a realizar la medida de presión y se muestran los resultados.
Por otro lado, las dos pantallas restantes, USER WINDOW y RECORDS WINDOW, servirían para elegir el usuario y acceder al historial de medidas del mismo pero todavía no han sido desarrolladas. Para la creación de esta interfaz se ha usado la librería gráfica LVGL, con la que se han creado todas las pantallas y sus elementos dentro de la función `pp_ui()` que se ejecuta periódicamente dentro de esta tarea. Esta librería gráfica es muy útil para este tipo de proyectos, ya que contiene gran cantidad de elementos interactivos y personalizables para crear interfaces únicas.

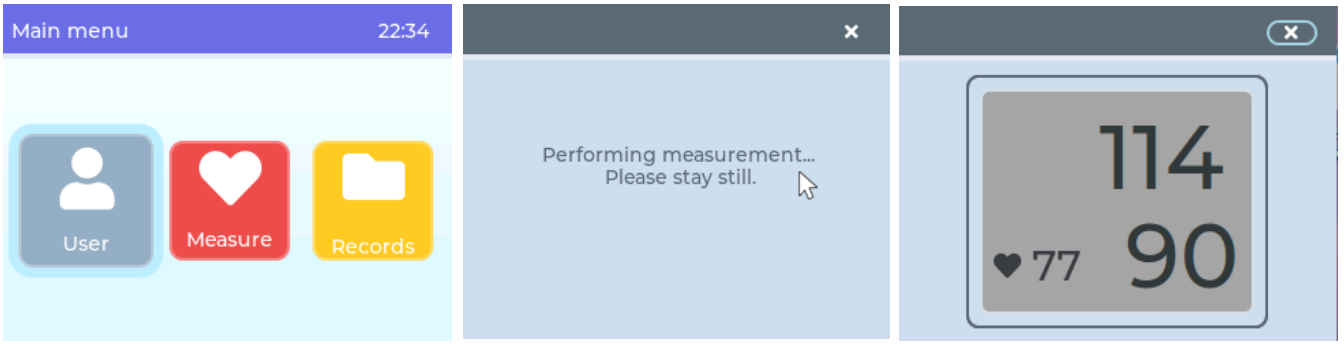


Fig 32. Pantallas de la interfaz de usuario

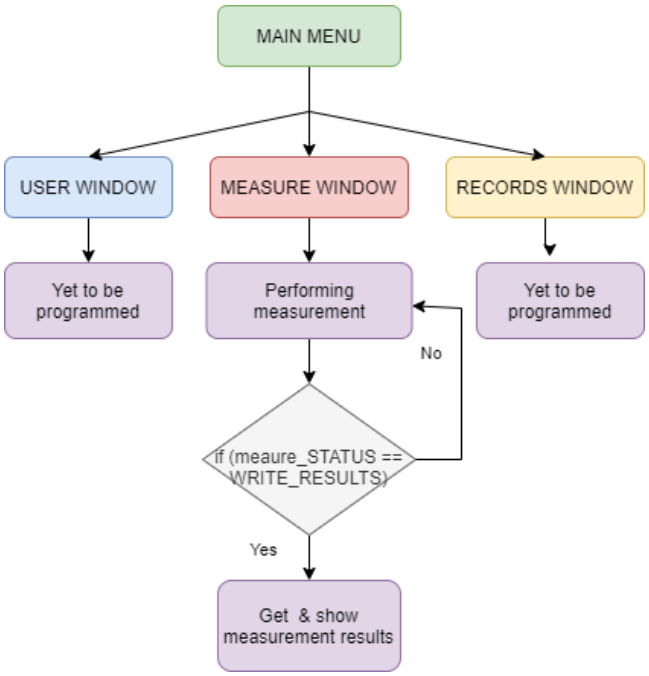


Fig 33. Diagrama de bloques tarea de interfaz de usuario

6. Análisis de los resultados.

Durante el procesado de las señales y el cálculo de las medidas en Matlab se utilizó el tensiómetro de muñeca *TOPCOM Blood Pressure Monitor BPM Wrist 3311* para comparar las medidas obtenidas con nuestro prototipo, con las obtenidas con un tensiómetro homologado.

Las medidas se realizan en estado de reposo, con el brazo izquierdo extendido y apoyado a la altura del corazón. Primero se realiza la medida con el prototipo desarrollado en este proyecto y dos minutos después con el tensiómetro homologado, ya que no se pueden realizar las dos medidas a la vez porque pueden afectarse mutuamente.

6.1 Validación de las medidas de las presiones sistólica y diastólica

Antes de poder validar las medidas se debe realizar una fase de calibración de los coeficientes de presión sistólica y diastólica, para ello se empieza eligiendo dos valores centrados dentro del rango de coeficientes empíricos [9]. Con estos coeficientes se realizan 6 medidas, se observa la precisión de las mismas y si es necesario se reajustan los coeficientes. Este proceso de ajuste se repite hasta que ya no se pueda mejorar la precisión. Los coeficientes elegidos son: $rs = 0.5;\ rd = 0.75$

rs = 0.6	rd = 0.8						
Ps Matlab	Ps real	Pd Matlab	Pd real	ΔPs	ΔPd	std dev ΔPs	std dev ΔPd
103	100	87	70	3	17	5,7	6,3
101	97	81	87	4	6		
93	100	84	69	7	15	mean ΔPs	mean ΔPd
118	110	70	73	8	3	5,8	6,8
115	107	72	88	8	16		
104	85	80	85	19	5	mean err % Ps	mean err % Pd
						5,8	8,6
rs = 0.55	rd = 0.77						
Ps Matlab	Ps real	Pd Matlab	Pd real	ΔPs	ΔPd	std dev ΔPs	std dev ΔPd
95	100	83	70	5	13	1,4	5,5
91	84	87	87	7	0		
101	97	73	69	4	4	mean ΔPs	mean ΔPd
118	114	82	73	4	9	5,4	5,9
91	97	75	88	6	13		
104	97	82	85	7	3	mean err % Ps	mean err % Pd
						5,2	7,6
rs = 0.5	rd = 0.75						
Ps Matlab	Ps real	Pd Matlab	Pd real	ΔPs	ΔPd	std dev ΔPs	std dev ΔPd
105	103	77	80	2	3	1,8	1,2
106	107	79	84	1	5		
105	111	78	83	6	5	mean ΔPs	mean ΔPd
98	95	86	83	3	3	2,6	3,3
113	117	72	69	4	3		
97	99	75	73	2	2	mean err % Ps	mean err % Pd
						2,1	4,1

Tabla 3. Medidas para la fase de calibración de coeficientes

Una vez calibrados los coeficientes se pasa a la fase de validación de los cálculos en Matlab donde se realizaron 24 medidas a tres pacientes diferentes y se obtuvieron los resultados que pueden verse en la Tabla 4.:

Paciente 1							
Ps Matlab	Ps real	Pd Matlab	Pd real	ΔPs	ΔPd	std dev ΔPs	std dev ΔPd
103	100	77	70	3	7	1,8	2,3
94	97	83	87	3	4		
101	100	77	69	1	8	mean ΔPs	mean ΔPd
110	110	85	80	0	5	2,7	3,4
111	107	86	88	4	2		
109	105	72	75	4	3	mean err % Ps	mean err % Pd
105	100	78	76	5	2	2,7	4,2
90	85	82	79	5	3		
Paciente 2							
Ps Matlab	Ps real	Pd Matlab	Pd real	ΔPs	ΔPd	std dev ΔPs	std dev ΔPd
95	100	80	79	5	1	2,3	1,3
113	107	79	77	6	2		
97	100	74	79	3	5	mean ΔPs	mean ΔPd
110	110	81	83	0	2	2,7	1,8
103	97	85	88	6	3		
90	88	72	75	2	3	mean err % Ps	mean err % Pd
116	115	70	69	1	1	2,6	2,4
103	99	75	73	4	2		
Paciente 3							
Ps Matlab	Ps real	Pd Matlab	Pd real	ΔPs	ΔPd	std dev ΔPs	std dev ΔPd
81	90	66	60	9	6	2,4	4,1
79	87	66	57	8	9		
71	75	62	59	4	3	mean ΔPs	mean ΔPd
78	80	51	63	2	12	4,5	6,9
80	87	52	68	7	16		
78	84	58	65	6	7	mean err % Ps	mean err % Pd
81	85	66	55	4	11	5,5	11,5
71	75	54	60	4	6		

Tabla 4. Medidas de la fase de cálculo con Matlab

Como se puede observar en esta tabla, los dos primeros pacientes tienen una presión arterial dentro de los límites normales (120-80 mmHg) y las medidas obtenidas con Matlab no se alejan mucho de la medida con el tensiómetro homologado. En estos dos casos la tolerancia de la medida simulada está alrededor de ±2-3 mmHg (2,4-4,2% error absoluto). Sin embargo, el tercer paciente tiene una presión arterial por debajo de la normal, rozando los límites de la hipotensión y el algoritmo implementado en MATLAB estima peor la medida. En este caso la tolerancia de la medida es de ±4-6 mmHg (5,5-11,5% error absoluto).

Una vez se termina la programación del prototipo se lleva a cabo la última fase de validación en la que se realizaron otras 24 medidas a los mismos pacientes. Como era de esperar tras los resultados obtenidos con MATLAB, las medidas realizadas con el prototipo a los pacientes con tensiones dentro del rango normal tiene más precisión que las realizadas al paciente con hipotensión. Se observa que los resultados obtenidos con el prototipo tiene una ligera peor precisión que los obtenidos en Matlab, esto puede deberse a que las operaciones en Matlab se realizan en coma flotante (permite operar con más precisión) mientras que en en el microcontrolador se realizan en coma fija, o simplemente por que aunque la implementación de las funciones de Matlab en el microcontrolador se ha

realizado siguiendo el mismo algoritmo, alguno de los cálculo se puede llevar a cabo de forma distinta.
En cuanto a los errores de medida totales, tanto en los cálculos con Matlab como con el prototipo final, se atribuyen a la vulnerabilidad del prototipo ante ruido y perturbaciones provocadas por movimientos de elementos externos como el motor.

Paciente 1							
Ps prototipo	Ps real	Pd prototipo	Pd real	ΔPs	ΔPd	std dev ΔPs	std dev ΔPd
97	94	84	80	3	4	1,2	1,2
103	99	80	87	4	7		
99	93	75	79	6	4	mean ΔPs	mean ΔPd
107	110	84	88	3	4	3,1	4,2
94	97	78	75	3	3		
109	106	77	73	3	4	mean err % Ps	mean err % Pd
97	94	85	80	3	5	3,0	5,1
111	113	92	96	2	4		
Paciente 2							
Ps prototipo	Ps real	Pd prototipo	Pd real	ΔPs	ΔPd	std dev ΔPs	std dev ΔPd
105	100	85	83	5	2	1,4	1,8
101	97	86	80	4	6		
98	100	80	79	2	1	mean ΔPs	mean ΔPd
90	92	86	83	2	3	2,8	2,1
113	110	74	75	3	1		
95	90	85	88	5	3	mean err % Ps	mean err % Pd
99	97	90	87	2	3	2,7	2,6
105	107	84	79	2	5		
Paciente 3							
Ps prototipo	Ps real	Pd prototipo	Pd real	ΔPs	ΔPd	std dev ΔPs	std dev ΔPd
80	84	66	59	4	7	2,6	1,9
77	75	70	73	2	3		
88	79	69	63	9	6	mean ΔPs	mean ΔPd
78	80	65	58	2	7	3,3	5,9
80	87	68	63	7	5		
92	88	73	65	4	8	mean err % Ps	mean err % Pd
75	70	69	60	5	9	4,1	9,1
73	75	68	61	2	7		

Tabla 5. Medidas última fase de validación

6.2 Validación de la medición del pulso

En cuanto a la validación de las medidas del pulso cardíaco, sólo se realizó en una fase y fue una vez terminada la programación de todo el prototipo. La precisión de esta medida se ve afectada por el tamaño de ventana de la transformada rápida de Fourier, ya que el rango de frecuencias dentro del límite cardiaco se ve reducido. Las condiciones de realización de estas medidas son las mismas que en las medidas de presión arterial. Dentro de esta fase se realizaron 24 medidas de 3 pacientes diferentes. Como se puede observar en la tabla 6, los errores medios absolutos se encuentran en el rango de 4-9%, superando el límite máximo de 5% de error que se establece para este tipo de dispositivos.

Paciente 1			Paciente 2			Paciente 3		
BPM proto	BPM real	ΔBPM	BPM proto	BPM real	ΔBPM	BPM proto	BPM real	ΔBPM
76	80	4	65	59	6	72	75	3
76	70	6	69	65	4	75	80	5
79	85	6	70	79	9	65	59	6
74	86	12	79	85	6	73	78	5
72	77	5	69	75	6	72	75	3
73	65	8	71	75	4	75	80	5
76	70	6	75	75	0	76	80	4
65	59	6	78	84	6	72	75	3
mean ΔBPM	6,0		mean ΔBPM	5,5		mean ΔBPM	4,0	
mean err %	9,0		mean err %	7,1		mean err %	5,2	

Tabla 6. Medidas de validación de frecuencia cardiaca.

7. Conclusiones y trabajo futuro.

La implementación de aplicaciones médicas en sistemas empotrados basados en microcontroladores requiere de un estudio intensivo del tipo de señales con las se va a trabajar, ya que tratan con la salud de las personas.

La aparición en los últimos años de microcontroladores más potentes ha favorecido el aumento del desarrollo de este tipo de sistemas, haciéndolos cada vez más compactos, accesibles y con más funcionalidades. Los primeros aparatos de medición de la presión eran manuales y dependían de la habilidad del profesional que realizaba la medida. Sin embargo, una vez aparecieron los monitores de tensión digitales, los factores de error humano desaparecieron y las personas comenzaron a poder llevar un control de sus niveles de tensión sin necesidad de ir a un centro de salud.

De ahí el objetivo principal de este proyecto: desarrollar un prototipo de monitor de presión que sea accesible, portable y que proporcione medidas precisas. Como se puede ver en el Anexo 3, el precio de este prototipo, como simple suma de precio de componentes, es de 42,76€, si bien el precio se reduciría bastante en el caso de una empresa que compra cientos de componentes.

El desarrollo de este proyecto se ha llevado a cabo realizando un estudio previo tanto del método oscilométrico y sus algoritmos de estimación de la presión arterial, como de las técnicas de análisis de señales en el dominio de la frecuencia para la estimación del pulso cardiaco.

Como se ha visto en el apartado de validación, la precisión de las medidas es aceptable para individuos con presión arterial dentro del rango normal (120-80 mmHg), sin embargo las medidas realizadas en el paciente con presión arterial baja tienen menos precisión y son menos constantes. Por otro lado, los datos obtenidos para la frecuencia cardiaca no tienen la precisión que sería deseable. Hay que tener en cuenta que el prototipo está construido en

una placa de desarrollo electrónico (protoboard), lo que lo hace más vulnerable a ruidos, movimientos e interferencias de su alrededor. Además, la validación se ha realizado con datos tomados de tan solo 3 personas.

Para una validación más adecuada, habría que tomar datos de muchas más personas y sobre un prototipo realizado en una PCB diseñada para minimizar interferencias y ruido.

No obstante, los resultados obtenidos no son malos. El error normal de la medición de presión arterial en monitores homologados está entre 2-3 mmHg, y el del pulso alrededor del 5%. El monitor desarrollado en este proyecto todavía no cumple estos límites, por esta misma razón a continuación se comentaran una serie de posibles mejoras y nuevas funcionalidades que se pueden aplicar para mejorar la calidad del monitor de tensión desarrollado.

En cuanto al trabajo futuro, primero se deberá centrar el foco en mejorar la precisión de las medidas, tanto de presión arterial como de pulso. Para ello se proponen una serie de mejoras:

- Construcción del prototipo en PCB, diseñado para minimizar ruido e interferencias.
- Estudio de algoritmos oscilométricos más precisos, como el de oscilometría derivativa, que en vez de basarse en el uso de coeficientes empíricos realiza la medida llevando a cabo un análisis de la pendiente de la envolvente. Otra técnica interesante sería el uso de redes neuronales que reciben como entrada la envolvente de la señal de oscilaciones (OMWE) y establecen relaciones complejas y no lineares entre la envolvente y la presión arterial.
- Análisis de filtros complejos para obtener una mejor señal de oscilaciones y por tanto una envolvente con más resolución.
- Añadir un sensor ECG (electrocardiograma) para mejorar la precisión de la medida de la frecuencia cardiaca.

Por otro lado, una vez se haya conseguido una precisión de medida dentro de los rangos exigidos, se puede mejorar la calidad y versatilidad del producto implementando una serie de nuevas funcionalidades como:

- Añadir una opción para la elección de usuario antes de realizar la medida, para posteriormente añadir una sección de historial donde cada usuario guarde sus medidas y éstas puedan ser exportadas vía USB al ordenador.
- Desarrollo de una aplicación de smartphone en la que se muestren gráficas y estadísticas de las mediciones de cada usuario. La sincronización del prototipo con el smartphone se haría vía Bluetooth, de esta manera se actualizarán los datos de las medidas realizadas por el monitor.

Referencias

1. “Cardiovascular Diseases (Cvds).” *World Health Organization*, 11 Jun. 2021, https://www.who.int/news-room/fact-sheets/detail/cardiovascular-diseases-(cvds)

2. “Sistema Arterial.” *Clase De Anatomía*, 18 Dec. 2020, https://www.auladeanatomia.com/novosite/es/sistemas/sistema-cardiovascular/vasos-sanguineos/sistema-arterial/.

3. “Diccionario De Cáncer Del NCI.” *Instituto Nacional Del Cáncer*, 6 May. 2021 https://www.cancer.gov/espanol/publicaciones/diccionarios/diccionario-cancer/def/presion-arterial.

4. Amadeo Morera, et al. “Diferencias Entre Presión y Tensión Arterial” *Mi Tensiómetro*, 13 Dec. 2019, https://mitensiometro.com/diferencias-presion-y-tension-arterial/.

5. M Olmo R Nave. “Tensión En Paredes Arteriales.” *Hyperphisics*, 27 Jul. 2021 http://hyperphysics.phy-astr.gsu.edu/hbasees/ptens3.html.

6. Dr. Jeovhanni Nieves Rivera, et al. *Módulo Instruccional Hipertensión e Hipotensión Arterial*. Mar. 2016, https://gurabo.uagm.edu/sites/default/files/uploads/EducacionContinua/pdf/2016-2/MOD-HIPER-HIPOTENSION-JA.pdf.

7. Simarro Blasco, J.A., Noheda Blasco, M.C., Bascuñana Blasco, M., Noheda Recuenco, M., Tolmo Aranda, I. Romero Carralero, M.I. (2011). Estudio comparativo de la presión arterial invasiva frente a la presión arterial no invasiva: Valoración de la diferencia. *Enfermería Global*, vol *10* n.24 https://dx.doi.org/10.4321/S1695-61412011000400006

8. F. Barranco Ruiz, J. Blasco Morilla, et al. *1.16.2. Toma De La Presion Arterial*, 15 Sep. 2021 https://uninet.edu/tratado/c011602.html.

9. M. Forouzanfar, H.R. Dajani, V.Z. Groza, M. Bolic, S. Rajan, and I Batkin, “Oscillometric blood pressure estimation: past, present, and future,” IEEE Reviews in Biomedical Engineering, vol. 8, pp. 44-63, May 2015.

10. Sani, Hac & Mansor, Wahidah & Lee, Yoot Khuan & Zainudin, N. & Mahrim, Syamsul Adlan. (2015). Determination of heart rate from photoplethysmogram using Fast Fourier Transform. 168-170. 10.1109/ICBAPS.2015.7292239.

11. Seamus. “How Do You Make a FIFO Array in C.” *Stack Overflow*, 25 Sep. 2019, https://stackoverflow.com/questions/59023297/how-do-you-make-a-fifo-array-in-c.

12. Dr. Emiliano Fdez-Obanza Windscheid. “El Sistema Cardiovascular.” *Sociedade Galega de Cardioloxía*, 25 Apr. 2019, https://www.sogacar.com/el-sistema-cardiovascular/.

13. Honeywell, “MPR SERIES MicroPressure Board Mount Pressure Sensors Compact, High Accuracy, Compensated/Amplified”, 32332628 Issue I datasheet, Jul. 2021.

14. “La Historia Del Ciclo Cardiaco Timeline.” *Timetoast Timelines*, 4 Mar. 2021, https://www.timetoast.com/timelines/la-historia-del-ciclo-cardiaco.

15. *“Sistema Cardiovascular.” Aula De Anatomia*, 31 Aug. 2021, https://www.auladeanatomia.com/.

16. Rivas, Iván. “Ruidos De Korotkoff.” *Gastro Mérida*, 4 Aug. 2018, https://www.ivanrivasmd.com/ruidos-de-korotkoff/.

17. Ada, Lady. “Adafruit MPRLS Ported Pressure Sensor Breakout.” *Adafruit Learning System*, July 2021, https://learn.adafruit.com/adafruit-mprls-ported-pressure-sensor-breakout.

18. “ESP32-DEVKITC Development Boards.” *Mouser*, 10 July 2018, https://www.mouser.es/new/espressif/espressif-esp32-devkitc-boards/.

19. Teja, Ravi. “Introduction to ESP32: Specifications, ESP32 Devkit Board, Layout.” *Electronics Hub*, 7 Sept. 2021, https://www.electronicshub.org/getting-started-with-esp32/.

20. “El Bus I2C.” *Dignal.com*, 25 Mar. 2015, https://dignal.com/el-bus-i2c/.

Anexos

Anexo 1: Código desarrollado en Eclipse IDE

1. main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "freertos/queue.h"
#include "esp_log.h"
#include "esp_types.h"
#include "driver/i2c.h"
#include "fifo_queue.h"
#include "driver/ledc.h"
#include "findPeaks.h"
#include "fft.h"
#include "display_gui.h"
#include "lvgl/lvgl.h"
#include "lvgl_helpers.h"


#define B1      34 // Aux button

#define B2      35 // Middle button

#define B3      27 // Right button

#define B4      14 // Left button
#define EOC_PS     17 // End of Conversion prs sensor
#define RST_PS     16 // Reset prs sensor
#define SDA_PS     21 //
End of Conversion prs
sensor #define SCL_PS 5
// End of Conversion prs
sensor
#define MPRL_ADDRESS 0x18//I2C adress
#define I2C_MASTER_FREQ_HZ 100000//SCK maste freq 100KHz

//Display
#define LCD_CS     14
#define LCD_RST 25
#define LCD_DC     26
#define LCD_MOSI 32
#define LCD_SCK 33

// i2c
#define ACK_CHECK_EN 0x1                    /*!< I2C master
will check ack from slave*/
#define ACK_CHECK_DIS 0x0                   /*!< I2C master
will not check ack from slave */
#define ACK_VAL 0x0                         /*!< I2C ack value */
```

```
#define NACK_VAL 0x1                            /*!< I2C nack value */

// Pressure related
#define COUNTS_224 (16777216L)       ///< Constant: 2^24
#define PSI_to_HPA (68.947572932) ///< Constant: PSI to HPA conversion factor
#define PSI_to_ATM (0.068046)
#define PSI_to_mmHg (51.715) ///< Constant: PSI to mmHg conversion factor
#define MPRLS_OUTPUT_min (uint32_t)((float)COUNTS_224 * (10 / 100.0) + 0.5)
#define MPRLS_OUTPUT_max (uint32_t)((float)COUNTS_224 * (90 / 100.0) + 0.5)
#define MPRLS_PSI_min 0
#define MPRLS_PSI_max 25

//typedef enum{MEASURE_IDLE, SCHEDULED, INFLATING, DEFLATING, DONE}
measure_stage;

#define VALVE_PIN 22
#define FWD_PIN 19
//#define BWD_PIN 18
#define PUMP_CHANNEL_A LEDC_CHANNEL_1
//#define PUMP_CHANNEL_B LEDC_CHANNEL_2
#define TMR_RES LEDC_TIMER_12_BIT
//display
#define LV_TICK_PERIOD_MS 1
```

```
/* User global variables */

static TaskHandle_t button_task;
static QueueHandle_t button_queue;

static TaskHandle_t pressure_task;
static QueueHandle_t pressure_queue;

static TaskHandle_t pwm_task;
//static QueueHandle_t pressure_queue;

static TaskHandle_t gui_task;
static SemaphoreHandle_t xGuiSemaphore; int32_t btn_encoder_count = 0;

bool valve_state = false; bool store = false;
struct fifo_queue oscil_queue;
struct fifo_queue press_queue;

struct envelope env; float first_pressure = 0;


uint32_t button_count;
static const char* TAG= "ButtonInfo";



/* Task declarations */

static void button_pressed_task(void *params); static void read_pressure_task(void *params); static void
lv_tick_task(void *arg);
static void gui_manager_task(void *pvParameter); static void IRAM_ATTR gpio_isr_handler(void *args){

        uint32_t pin_number = (uint32_t)args;
        xQueueSendFromISR(button_queue, &pin_number, NULL);
}
void pressure_control_handler(void *params);

void app_main(void){
        /*Button config*/
        gpio_config_t config;
        config.intr_type =
        GPIO_INTR_NEGEDG
        E; config.mode =
        GPIO_MODE_INPUT;
        config.pull_down_en =
        false; config.pull_up_en
        = true;
        config.pin_bit_mask = ((1ULL<<B1) | (1ULL<<B2) | (1ULL<<B3) | (1ULL<<B4));

        gpio_config(&config);
        button_queue = xQueueCreate(4,sizeof(uint32_t));
        xTaskCreate(button_pressed_task, "button pushed", 2048, NULL,
        3, &button_task);

        gpio_install_isr_service(0);
        gpio_isr_handler_add(B1,
        gpio_isr_handler, (void *)B1);
        gpio_isr_handler_add(B2,
        gpio_isr_handler, (void *)B2);
        gpio_isr_handler_add(B3,
        gpio_isr_handler, (void *)B3);
        gpio_isr_handler_add(B4,
```

```
gpio_isr_handler, (void *)B4);
```

```
/*I2C config*/

gpio_pad_select_gpio(RST_PS);
gpio_set_direction(RST_PS,
GPIO_MODE_OUTPUT);
gpio_set_level(RST_PS, 1);

gpio_pad_select_gpio(EOC_PS);
gpio_set_direction(EOC_PS,
GPIO_MODE_INPUT);


i2c_config_t i2c_config = {
.mode = I2C_MODE_MASTER,
.sda_io_num = SDA_PS,
.scl_io_num = SCL_PS,
.sda_pullup_en = GPIO_PULLUP_ENABLE,
.scl_pullup_en = GPIO_PULLUP_ENABLE,
.master.clk_speed =
I2C_MASTER_FREQ_HZ};
i2c_param_config(I2C_NUM_0,
&i2c_config);
i2c_set_timeout(I2C_NUM_0,
1048575);
i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0);
xTaskCreate(read_pressure_task, "read pressure", 4096, NULL, 5,
&pressure_task);

ESP_LOGI(TAG, "Setup done");

// Configure PWM
ledc_timer_config_t pwm_timer_config = {
.speed_mode = LEDC_LOW_SPEED_MODE,
.duty_resolution = TMR_RES,
.timer_num = LEDC_TIMER_1,
.freq_hz = 15000,
.clk_cfg = LEDC_AUTO_CLK};
ledc_timer_config(&pwm_timer_config)
;

ledc_channel_config_t pump_channel_config = {
.gpio_num = FWD_PIN,
.speed_mode = LEDC_LOW_SPEED_MODE,
.channel = PUMP_CHANNEL_A,
.timer_sel = LEDC_TIMER_1,
.duty = 0,
.hpoint = 0};
ledc_channel_config(&pump_channel_
config);

xTaskCreate(pressure_control_handler, "pwm", 14*2048, NULL, 5, &pwm_task);

// Configure screen
xTaskCreate(gui_manager_task, "gui", 4096*2, NULL, 1, &gui_task);
// see gui_manager_task() and pp_ui() for the layout
//xTaskCreatePinnedToCore(gui_manager_task, "gui", 4096*2, NULL, 0, &gui_task, 1);

// Configure valve
gpio_config_t valve_config = {
```

```
    .mode = GPIO_MODE_OUTPUT,
    .pull_up_en = GPIO_PULLUP_DISABLE,
    .pull_down_en = GPIO_PULLDOWN_DISABLE,
    .pin_bit_mask = (1ULL<<VALVE_PIN),
    .intr_type = GPIO_INTR_DISABLE
    };
    ESP_ERROR_CHECK(gpio_config(&valve_config));




}
```

```
static void button_pressed_task(void *params){

        uint32_t pin_number=0;

        while(1)
        {
          if (xQueueReceive(button_queue, &pin_number, portMAX_DELAY))
        {

        // disable the interrupt
        gpio_isr_handler_remove(pin_number);

        // wait some time while we check for the button to be released
        do
        {
          vTaskDelay(20 / portTICK_PERIOD_MS);
        } while(gpio_get_level(pin_number) == 0);

        //do some work
                switch(pin_number) {
                        case B1: // Aux button
                        printf("Boton auxiliar ya
                        le daremos algun uso");
                        ESP_LOGI(TAG,
                        "auxiliar");
                        break;

                        case B2: // Middle button
                        ESP_LOGI(TAG, "middle");
                        break;

                        case B3: // Right button
                        //measure_STATUS = DEFLATING;
                        btn_encoder_count++;
                        ESP_LOGI(TAG, "encoder++, value : %d \n", btn_encoder_count);
                        break;

                        case B4: // Left button
                        //measure_STATUS = SCHEDULED;
                        btn_encoder_count--;
                        ESP_LOGI(TAG, "encoder++, value
                        : %d \n",btn_encoder_count); break;
                }

        // re-enable the interrupt
        gpio_isr_handler_add(pin_number, gpio_isr_handler, (void *)pin_number);



        }
        }
}


uint8_t read_MPRLS_status(){

        uint8_t status_data;

        i2c_cmd_handle_t cmd = i2c_cmd_link_create(); i2c_master_start(cmd);
```

```
        i2c_master_write_byte(cmd, (MPRL_ADDRESS << 1) | I2C_MASTER_READ,
        ACK_CHECK_EN); i2c_master_read_byte(cmd, &status_data, NACK_VAL);
        i2c_master_stop(cmd);
        ESP_ERROR_CHECK(i2c_master_cmd_begin(I2C_NUM_0, cmd, 1000 /
        portTICK_RATE_MS)); i2c_cmd_link_delete(cmd);

        return status_data;

}
```

```
uint32_t read_MPRLS_data(){

        uint8_t query_command[3] = {0xAA, 0x00, 0x00};

        // Ask for data
        i2c_cmd_handle_t cmd =
        i2c_cmd_link_create();
        i2c_master_start(cmd);
        i2c_master_write_byte(cmd, (MPRL_ADDRESS << 1) | I2C_MASTER_WRITE,
        ACK_CHECK_EN); i2c_master_write(cmd, query_command, 3,
        ACK_CHECK_EN);
        i2c_master_stop(cmd);
        ESP_ERROR_CHECK(i2c_master_cmd_begin(I2C_NUM_0, cmd, 1000 /
        portTICK_RATE_MS)); i2c_cmd_link_delete(cmd);

        // Wait for data
        TickType_t t=
        xTaskGetTickC
        ount(); uint8_t
        last_status;
        while ((last_status = read_MPRLS_status()) & 0x20) { // Device busy flag

                //printf("status: 0x%X, \r\n", last_status);
                float ellapsed_wait_time_ms = (xTaskGetTickCount() - t) / portTICK_RATE_MS;
                if (ellapsed_wait_time_ms > 20){
                        //printf("TIMEOUT BUSY FLAG \r\n");
                        return 0xFFFFFFFF; // timeout
                  }
        }


        // Read data
        //ESP_LOGI(TAG, "Reading data");
        const size_t N_PRESSURE_BYTES = 4;
        uint8_t pressure_raw_data[N_PRESSURE_BYTES];

        cmd = i2c_cmd_link_create();
         i2c_master_start(cmd);
        i2c_master_write_byte(cmd, (MPRL_ADDRESS << 1) | I2C_MASTER_READ,
        ACK_CHECK_EN); i2c_master_read(cmd, pressure_raw_data, N_PRESSURE_BYTES - 1,
        ACK_VAL); i2c_master_read_byte(cmd, pressure_raw_data + N_PRESSURE_BYTES - 1,
        NACK_VAL); i2c_master_stop(cmd);
        ESP_ERROR_CHECK(i2c_master_cmd_begin(I2C_NUM_0, cmd, 1000 /
        portTICK_RATE_MS)); i2c_cmd_link_delete(cmd);

        //printf("READ status: 0x%X, \r\n", pressure_raw_data[0]);
        uint32_t pressure_raw = (pressure_raw_data[1] << 16) | (pressure_raw_data[2] << 8) |
        (pressure_raw_data[3]);
        return pressure_raw;
}

float read_MPRLS_pressure(){

        uint32_t raw_psi = read_MPRLS_data();
        float psi = (raw_psi - MPRLS_OUTPUT_min) * (MPRLS_PSI_max -
        MPRLS_PSI_min); psi /= (float)(MPRLS_OUTPUT_max -
        MPRLS_OUTPUT_min);
        psi += MPRLS_PSI_min;
```

```
        return psi;

}

static void read_pressure_task(void *params){

        // Initialization
        gpio_set_level(RST_PS, 0);
        vTaskDelay(10/ portTICK_RATE_MS); gpio_set_level(RST_PS, 1);
        vTaskDelay(10/ portTICK_RATE_MS); // Startup timing
        //init pressure queue
        init_fifo_queue( &press_queue, 3000);
        init_fifo_queue( &oscil_queue, 3000);
        // Bandpass pass
        //const float b0 = 0.1453, b2 = -0.2906, b2 = 0.2929;
        //const float a2 = 0.1716; // a0=1, a1 = 0
        const float float b0 = 0.1453, b1 =0, b2 = -0.2906,b3 = 0, b4 = 0.1453;
        const float a4= 1, a3 = -2.521, a2 = 2.3844, a1 = -1.1096, a0=0.2523;
        float xk1=0, xk2=0, xk3=0, xk4=0, yk1=0, yk2=0,
        yk2=0, yk3=0, yk4=0; float xk, yk;

        TickType_t last_wake_time = xTaskGetTickCount();
        while(1){

                //uint32_t pressure_raw = read_MPRLS_data();
                float psi = read_MPRLS_pressure();
                float mmHg = psi * PSI_to_mmHg;
```

```
                // Band pass filter

                xk = mmHg;
                yk = b0*xk + b1*xk1 + b2*xk2 + b3*x3 + b4*x4 +
                (a0*yk + a1*yk1 + a2*yk2 + a3*yk3 + a4*yk4);

                yk4 = yk3; yk3 = yk2; yk2 = yk1; yk1 = yk;
                xk4 = xk3; xk3= xk2; xk2 = xk1; xk1 = xk;

                if(store == true){
                        enqueue(&oscil_queue, yk);
                         enqueue(&press_queue, mmHg);
                        //for data aquisition
                        //printf("{TIMEPLOT|data|RawPressure|T|%d}\n", (uint32_t)((yk - 740)*1000)); //
                        DEBUG
                        //printf("{TIMEPLOT|data|OscilAmplitude|T|%d}\n", (uint32_t)((yk_h*1000)));
                }

                vTaskDelayUntil(&last_wake_time, pdMS_TO_TICKS(25));
                }


}


void pressure_control_handler(void *params){ TickType_t last_wake_time = xTaskGetTickCount();

        // Array for bpm measurement
        const float Ts = 0.025;
         measure_stage measure_STATUS;
        const uint32_t N = 512;
        int offset = 500;
        float max_harmonic=0;
        measure_STATUS = MEASURE_IDLE;
        float sys_r = 0;
        float dias_r = 0;
```

```
while(1){

        measure_STATUS = read_measure_status();
        switch(measure_STATUS){

        case MEASURE_IDLE:;
        // PWM Control
        ESP_LOGI(TAG, "Ready for measurement");
        //float ref_pressure = 0;
        // Valve control
        gpio_set_level(VALVE_PIN, 0);//valveopen
        break;

        case SCHEDULED:
        ESP_LOGI(TAG, "Scheduled measurement");
        ledc_fade_func_install(0);
        measure_STATUS = INFLATING; ESP_LOGI(TAG, "Proceeding to inflate");
        gpio_set_level(VALVE_PIN, 1);//valve closed
        //Comenzamos a guardar los datos
        store = true;
        ledc_set_fade_time_and_start(LEDC_LOW_SPEED_MODE, PUMP_CHANNEL_A,
        2800, 1500, LEDC_FADE_WAIT_DONE);

        first_pressure = last_queue_value(&press_queue);
        break;

        case INFLATING:;

        float current_pressure = last_queue_value(&press_queue);
        printf("Pressure measured %f \r\n", current_pressure);
        if (current_pressure > 890.){
            measure_STATUS = DEFLATING;

        }

        break;

        case DEFLATING:;
        gpio_set_level(VALVE_PIN, 0);
        ESP_LOGI(TAG, "Proceeding to deflate");
        ledc_set_fade_time_and_start(LEDC_LOW_SPEED_MODE,
        PUMP_CHANNEL_A, 0, 35000, LEDC_FADE_WAIT_DONE);

        store = false;
        measure_STATUS = DONE; break;

        case DONE:
        gpio_set_level(VALVE_PIN, 0);
        ESP_LOGI(TAG, "Measurement done");
        vTaskDelay(pdMS_TO_TICKS(500));

        //BUSCAMOOS LOS MAXIMOS LOCALES CON ENVPEAKS
        int minD=30; //minima distancia entre picos

        init_envelope(&env, minD, 2800);
        findPeaks(&env, &oscil_queue);
        int sys= env.As_idx;
        int dias= env.Ad_idx;
        sys_r = (press_queue.arr[sys])-740;
```

```
dias_r = (press_queue.arr[dias])-740;

//Calculamos BPM
complex *ordered_bpm_arr = (complex *) malloc(N*sizeof(complex));
complex *temp = (complex *) malloc(N*sizeof(complex));

for (size_t j = 0; j < N; j++){
    ordered_bpm_arr[j].Im = 0.;
    ordered_bpm_arr[j].Re = oscil_queue.arr_norm[offset + j];

}

fft(ordered_bpm_arr, N, temp); // Use unordered array as scratch
max_harmonic = get_highest_harmonic(ordered_bpm_arr, N, 1/Ts);
```

```
                free(ordered_bpm_arr); free(temp);

                printf("Highest harmonic is %4.4f \n", max_harmonic);
                printf("BPM: %f \n", max_harmonic*60);


                free_envelope(&env);
                free_queue(&press_queue);
                free_queue(&oscil_queue); measure_STATUS = WRITE_RESULTS;
                break;
                case WRITE_RESULTS:

                printf("sys Pressure: %f \n", sys_r); printf("dias Pressure: %f \n", dias_r);
                printf("first Press %f: \n", first_pressure);
                printf("Highest harmonic is %4.4f \n", max_harmonic);
                printf("BPM: %f \n", max_harmonic*60);
                write_measured_bpm(60*max_harmonic, sys_r,dias_r);
                ESP_LOGI(TAG, "write_result_status");



                break; default:
                break;

                }
                write_measure_status(measure_STATUS);
                vTaskDelayUntil(&last_wake_time, pdMS_TO_TICKS(50));
        }

}

static void gui_manager_task(void *pvParameter) {
 (void) pvParameter;

TickType_t last_wake_time = xTaskGetTickCount();
lv_init();
lvgl_driver_init();

// Use double buffered when not working with monochrome displays
```

```
static lv_color_t buf1[DISP_BUF_SIZE];
static lv_color_t buf2[DISP_BUF_SIZE]; static lv_disp_buf_t disp_buf;
uint32_t size_in_px = DISP_BUF_SIZE;
lv_disp_buf_init(&disp_buf, buf1, buf2, size_in_px);

lv_disp_drv_t disp_drv; lv_disp_drv_init(&disp_drv);
disp_drv.flush_cb = disp_driver_flush;
disp_drv.buffer = &disp_buf;
lv_disp_drv_register(&disp_drv);

// Create and start a periodic timer interrupt to call lv_tick_inc
const esp_timer_create_args_t periodic_timer_args = {
.callback = &lv_tick_task,
.name = "periodic_gui"
};
esp_timer_handle_t periodic_timer;
ESP_ERROR_CHECK(esp_timer_create(&periodic_timer_args, &periodic_timer));
ESP_ERROR_CHECK(esp_timer_start_periodic(periodic_timer, LV_TICK_PERIOD_MS
* 1000));

// Create the demo application
pp_ui();

while (1) {
        // Delay 1 tick (assumes FreeRTOS tick is 10ms
        vTaskDelay(pdMS_TO_TICKS(10));

        lv_task_handler();

}

vTaskDelayUntil(&last_wake_time, pdMS_TO_TICKS(100));
}

static void lv_tick_task(void *arg) {
        (void) arg;
        lv_tick_inc(LV_TICK_PERIOD_MS);
}
```

2. findPeaks.c

```
/*
*	findPeaks.c
*
*	Created on: 5 oct. 2021
*	Author: romeoisabel
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <math.h>
#include "fifo_queue.h"
#include "findPeaks.h"
void init_envelope(struct envelope *env, int d, int nelems){

env->MAXSIZE = nelems;

env->y = malloc(nelems*sizeof(float));
env->minD=d;
env->x = malloc(nelems*sizeof(float));
env->MAP = 0.;
env->MAP_idx = 0;
env->As= 0.;
env->As_idx= 0;
env->Ad= 0.;
env->Ad_idx= 0;
//env->iPk = malloc(nelems*sizeof(float));

}
void free_envelope(struct envelope *env){
free(env->y);
free(env->x);
env->MAXSIZE=0;
env->minD=0;
env->MAP = 0.;
env->MAP_idx = 0;
env->As= 0.;
env->As_idx= 0;
env->Ad= 0.;
env->Ad_idx= 0;


}



void findPeaks(struct envelope *env, struct fifo_queue *oscil_norm){

	float *yTemp;
	int oscil_size = oscil_norm->size;
	printf("oscil_size= %d \n", oscil_size );
	yTemp = malloc((oscil_size +2)*sizeof(float));

	for(int i=0; i<(oscil_size +2); i++){

		if(i==0){
```

```
            yTemp[i]=0;
          }
          else if(i==oscil_size +1){
            yTemp[i] = 0;
          }
          else {
            yTemp[i] = oscil_norm->arr_norm[i-1];
            }
}

//Hacemos resta de los valores adyacentes a lo largo del array
//y sacamos el signo de cada valor
float *diff;
signed
int *s;
diff = malloc((oscil_size+2)*sizeof(float)); s = malloc((oscil_size+2)*sizeof(int)); printf("g \n" );
for(int i=0; i<(oscil_size+2); i++){
          if(i>0){
          diff[i-1] = yTemp[i]-yTemp[i-1];
          }
}

for(int i=0; i<(oscil_size+2); i++){
          if(diff[i]>0){ s[i] = 1;
          }
          else if(diff[i]<0){
           s[i] = -1;
          }
          else s[i] = 0;
  }


//Buscamos los maximos locales (su localizacio, indice)
//para ello hacemos la resta de los valores adyacentes a lo largo de s
//y buscamos donde sea menor de 0
signed int *diffS;
diffS = malloc((oscil_size+2)*sizeof(int));

struct fifo_queue iMax;
init_fifo_queue( &iMax, (oscil_size+2));

for(int i=0; i<(oscil_size+2); i++){
if(i>0){
diffS[i-1] = s[i]-s[i-1];
  }
}

for(int i=0; i<(oscil_size+2); i++){
if(diffS[i]<0){
 enqueue(&iMax, i);

  }
}

struct fifo_queue iPk;
init_fifo_queue( &iPk, (oscil_norm->size+2));
int size_iMax = iMax.size;
for(int i=0; i<=size_iMax; i++){
   enqueue(&iPk, i-1);
```

```
}
```

```
//findPeaksSeparatedByMoreThanMinPeakDistance
//guardamos peaks y locs en dos variables
int size_iMax = iMax.size;
float pks[size_iMax];
float locs[size_iMax];
//locs=iPk;
printf("k \n" );
for(int i=0; i<size_iMax; i++){
  pks[i] = oscil_norm->arr_norm[(int)iMax.arr[i]];/// //////////////arr_norm
  locs[i] = iMax.arr[i];
}

//ordenamos los Peaks de mayor a menor y guardamos
//los idx ordenados tmbn en otro array
printf("size_iMax= %d \n", size_iMax );

struct sort_queue
{
  float value;
  int idx;
};

struct sort_queue sortIdx[size_iMax];

for (int i = 0; i < size_iMax; i++)
{
   sortIdx[i].value = pks[i]; sortIdx[i].idx = (int)locs[i];
}

//decending function
int comp(const void *a, const void *b){
          struct sort_queue *a1 = (struct sort_queue *)a;
          struct sort_queue *a2 = (struct sort_queue *)b;
           if ((*a1).value > (*a2).value)) {
              return -1;
          }
          else if ((*a1).value < (*a2).value) {
             return 1;
          }
          else return 0;
}
```

```
qsort(sortIdx, size_iMax, sizeof(sortIdx[0]), comp);

//una vez tenemos el array de idx ordenado obtenemos el
//array de x indexando sortIdx

for(int i=0; i<size_iMax; i++){
  locs[i] = sortIdx[i].idx;
}
int idelete[size_iMax];
for(int i=0; i<size_iMax; i++){
   idelete[i]=0;
}


//If the peak is not in the neighborhood of a larger peak, find
//secondary peaks to eliminate.
int masc1[size_iMax];
int masc2[size_iMax];
for(int i=0; i<size_iMax; i++){
        if(!idelete[i]){

        for(int k=0; k<size_iMax; k++){
        if(locs[k]>=(locs[i]-env->minD)){
        masc1[k]=1;
          } else masc1[k]=0;

        if(locs[k]<=(locs[i]+env->minD)){
        masc2[k]=1;
        } else masc2[k]=0;

}

for(int j=0; j<=size_iMax; j++){
   idelete[j] = idelete[j] | (masc1[j] & masc2[j]);
}
idelete[i] = 0; //keep current peak
}
}
```

```
//Para cribar los maximos solo nos quedaremos con los maximos
//en los que el valor en el indice correspondiente en el array idelete
//sea 0. -> Ej: sortValue=[5, 3, 2 ,1] sortIdx=[2, 3, 1, 0] idelete=[0, 0, 1, 0]
//nos quedaremos con los maximos situados en 2 3 y 0

//Para ello crearemos otro struct sort con los valores definitivos

//Debemos contar cuantos maximos quitamos para saber el size del nuevo
//struct de valores
int cnt=0;
for(int i=0; i<size_iMax; i++){
        if(idelete[i]==1){
            cnt++;
        }
}

int size_def = size_iMax-cnt;
struct sort_queue def[size_def];
cnt=0;

for(int i=0; i<size_iMax; i++){
    if(idelete[i]==0){
        def[cnt].value = sortIdx[i].value;
        def[cnt].idx = sortIdx[i].idx;
        cnt++;
        //guardamos valores definitivos
        }
}

cnt=0;

//Una vez tenemos seleccionados los maximos los volvemos a reordenar
//en orden cronologico
int comp2(const void *a, const void *b)
{
        struct sort_queue *a1 = (struct sort_queue *)a;
        struct sort_queue *a2 = (struct sort_queue *)b;
        if ((*a1).idx > (*a2).idx)
        return 1;
        else if ((*a1).idx < (*a2).idx)
        return -1; else return 0;
}
qsort(def, size_def, sizeof(def[0]), comp2);



//Una vez lo tenemos reordenado guardamos los idx
//hacemos bookend con 0 a los arrays de valores e indices finales para interpolar bien

  for(int i=0; i<size_def+2; i++){
        if((i==0) | (i==(size_def+1))){

        env->y[i]=0;
        env->x[i]=0;
        }
        else {
         env->y[i]=def[i-1].value;
        env->x[i]=def[i-1].idx;
        }
```

```
}
```

```
float xi[oscil_size +2];
float yi[oscil_size +2];
//Inicializamos el array de valores de x de la iterpolacion
for(int i=0; i<oscil_size+2; i++){
   xi[i] = i;

}

//Algoritmo 2

for(int i=0; i<(size_def+1); i++){
          int x0 = env->x[i];
          int x1 = env->x[i+1];
          float y0 = env->y[i]; float y1 = env->y[i+1]; yi[x0]=y0;
          yi[x1]=y1;


          for(int j=x0+1; j<x1; j++){
                    int xp = xi[j];
                   float yp = y0+ ((y1-y0)/(x1-x0))*(xp-x0);
                   yi[xp]=yp;

            }


}


//Una vez tenemos la envelope hecha solo queda buscar el maximo que sera el MAP
//Una vez tengamos el MAP las medidas de SYS y DIAS seran unos porcentaje fijos
//del MAP(por arriba o por abajo)
float max=0; int max_idx=0;
//Para Buscar el MAP buscaremos a partir del indice 500 + o - mpara evitar los
//maximos del principio
for(int i=500; i<(oscil_size-100); i++){

          if((yi[i])>max){
          max=(yi[i]); max_idx=i;
          }
}




float MAP=max;
printf("MAP %f \n", MAP );
printf("MAP_idx %d \n", max_idx );
float Ad=0.5*MAP;
float As=0.75*MAP;
//Buscamos el idx donde se encuentra As y Ad
float d=5; float d_min=5;

int x_min_sys=0;
int x_min_dias=0;

for(int i=500; i<max_idx; i++){
```

```
                d=abs((yi[i])-abs(As));
                if(d<d_min){
                  d_min=d;
                  x_min_sys=i;
                }

        }

        d=5;
        d_min=5;
        for(int i=max_idx; i<1500; i++){
                d=abs((yi[i])-abs(Ad));
                if(d<d_min){
                  d_min=d;
                  x_min_dias=i;
                }

        }
        env->MAP = MAP;
        env->As = As; env->Ad = Ad;
        env->MAP_idx = max_idx; env->As_idx = x_min_sys; env->Ad_idx = x_min_dias;

        printf("SYS_idx %d \n", env->As_idx );
        printf("DIAS_idx %d \n", env->Ad_idx );

        //Obtenemos los valores de presion dias y sys con los indices obtenidos
        //float press_sys=press_queue.arr[x_min_sys];
        //float press_dias=press_queue.arr[x_min_dias];

        //Vaciamos las colas

        free(yTemp); free(diff); free(s);
        free_queue(&iMax);


}
```

3. display_gui.c

```
/*
*         display_gui.c
*
*         Created on: 21 oct. 2021
*         Author: romeoisabel
*/

#include "display_gui.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
 #include "driver/gpio.h"
 #include "freertos/queue.h"
#include "esp_log.h"
#include "esp_types.h"
#include "driver/i2c.h"

/**** User variables ****/

#define B2          35 // Middle button

// Main menu and globals static lv_group_t* pp_group;
static lv_obj_t *btn_settings;
 static lv_obj_t *btn_measure;
static lv_obj_t *btn_record;
static lv_style_t window_bg_style;
static lv_style_t win_btn_style;

// Settings menu
static lv_obj_t* devmode_switch;
static lv_obj_t* settings_menu;
static lv_obj_t *btn_settings_close;
static lv_obj_t* set_devmode_btn;
static lv_obj_t* user1_btn;
static lv_obj_t* user2_btn;
static lv_obj_t* user3_btn;
static lv_obj_t* set_time_btn;
static lv_obj_t* motor_control_btn;
static lv_obj_t* view_graph_btn;
static bool dev_state = false;
static lv_style_t btn_settings_style;

// Measure window
static lv_obj_t *btn_measure_close; lv_task_t *get_measure_status_task;
measure_stage measure_STATUS = MEASURE_IDLE;
static lv_obj_t * measure_status_label;
static float measured_bpm = 0.;
static float measured_sys = 0.;
```

```
static float measured_dias = 0.;

//Results window
static lv_obj_t *btn_results_close;
static lv_obj_t * sys_label ;
static lv_obj_t * dias_label ;
static lv_obj_t * bpm_label ;
static lv_obj_t * par;

/**** User function prototypes ****/
#if !SIMULATION
int32_t btn_encoder_count;
bool get_encoder_button_data_cb(lv_indev_drv_t *indev_drv,
lv_indev_data_t *data);
uint8_t pwm_power;
#endif
static void btn_settings_cb(lv_obj_t * btn, lv_event_t event);

// Settings menu
static void settings_window();
static void btn_devmode_cb(lv_obj_t * btn, lv_event_t event);
 static void close_win_settings_cb(lv_obj_t *btn, lv_event_t event);
static void btn_view_graph_cb(lv_obj_t *btn, lv_event_t event);
static void add_dev_settings();
static void btn_motor_control_cb(lv_obj_t *btn, lv_event_t event);

// Measure window
static void measure_window();
static void btn_measure_cb(lv_obj_t * btn,
lv_event_t event);
 static void close_win_measure_cb(lv_obj_t *btn,
lv_event_t event);
 static void
get_measure_status_cb(lv_task_t
*task);
//void write_measured_bpm(float bpm);

// Measure result window
static void measure_result_window();
static void close_win_results_cb(lv_obj_t *btn, lv_event_t event);
extern void write_measured_bpm(float bpm, float sys, float dias);

void pp_ui(void){


        pp_group = lv_group_create();

        /* Initialize input driver(encoder) */
        v_indev_drv_t enc_drv;
        lv_indev_drv_init(&enc_drv);
        enc_drv.type = LV_INDEV_TYPE_ENCODER;
         #if SIMULATION
        enc_drv.read_cb =
        mousewheel_read;
        #else
        enc_drv.read_cb =
        get_encoder_button_data_cb;
        #endif
```

```
lv_indev_t *enc_indev = lv_indev_drv_register(&enc_drv);
lv_indev_set_group(enc_indev, pp_group);

/* style win bg */
lv_style_init(&window_bg_style);
lv_style_set_bg_color(&window_bg_style, LV_STATE_DEFAULT,
lv_color_hex(0xFFFFFF)); lv_style_set_bg_grad_color(&window_bg_style,
LV_STATE_DEFAULT, lv_color_hex(0xdff9fb));
lv_style_set_bg_grad_dir(&window_bg_style, LV_STATE_DEFAULT,
LV_GRAD_DIR_VER); lv_style_set_bg_main_stop(&window_bg_style,
LV_STATE_DEFAULT, 10);
lv_style_set_bg_grad_stop(&window_bg_style, LV_STATE_DEFAULT, 200);
/* Create main window */
lv_obj_t *win_main = lv_win_create(lv_scr_act(), NULL);
lv_win_set_title(win_main, "Main menu");
lv_win_set_header_height(win_main, 40);
lv_obj_set_style_local_text_color(win_main, LV_WIN_PART_HEADER,
LV_STATE_DEFAULT, lv_color_hex(0xffffff));
lv_obj_set_style_local_bg_color(win_main, LV_WIN_PART_HEADER,
LV_STATE_DEFAULT, lv_color_hex(0x686de0)); // f2f9fa
lv_obj_add_style(win_main, LV_WIN_PART_BG, &window_bg_style);

/* label main window */
lv_obj_t *label_win_main = lv_label_create(lv_scr_act(), NULL);
lv_label_set_text(label_win_main, "22:34");
lv_obj_set_style_local_text_color(label_win_main, LV_LABEL_PART_MAIN,
LV_STATE_DEFAULT, lv_color_hex(0xFFFFFF)); lv_obj_align(label_win_main, win_main,
LV_ALIGN_IN_TOP_RIGHT, -20, 12);

/* Add buttons to main window */

/* button style */
static lv_style_t main_btn_style;
 lv_style_init(&main_btn_style);
lv_style_set_radius(&main_btn_style, LV_STATE_DEFAULT, 10);
lv_style_set_outline_color(&main_btn_style, LV_STATE_DEFAULT,
lv_color_hex(0xFFFFFF));
lv_style_set_border_color(&main_btn_style, LV_STATE_DEFAULT,
lv_color_hex(0xFFFFFF));
lv_style_set_border_opa(&main_btn_style, LV_STATE_DEFAULT,
LV_OPA_30); lv_style_set_text_color(&main_btn_style,
LV_STATE_DEFAULT, lv_color_hex(0xFFFFFF));
lv_style_set_outline_width(&main_btn_style, LV_STATE_FOCUSED, 7);
lv_style_set_outline_color(&main_btn_style, LV_STATE_FOCUSED,
lv_color_hex(0xb0eaff)); lv_style_set_outline_opa(&main_btn_style,
LV_STATE_FOCUSED, LV_OPA_80);
lv_style_set_transform_width(&main_btn_style, LV_STATE_FOCUSED, 5);
lv_style_set_transform_height(&main_btn_style, LV_STATE_FOCUSED, 5);
/* label style */
static lv_style_t main_label_style;
lv_style_init(&main_label_style);
lv_style_set_text_color(&main_label_style, LV_STATE_DEFAULT, lv_color_hex(0xffffff));
/* heart icon style */
static lv_style_t heart_icon_style; lv_style_init(&heart_icon_style);
lv_style_set_text_color(&heart_icon_style, LV_STATE_DEFAULT, lv_color_hex(0xffffff));
lv_style_set_text_font(&heart_icon_style, LV_STATE_DEFAULT, &heart_44);
static lv_style_t user_icon_style; lv_style_init(&user_icon_style);
lv_style_set_text_color(&user_icon_style, LV_STATE_DEFAULT, lv_color_hex(0xffffff));
lv_style_set_text_font(&user_icon_style, LV_STATE_DEFAULT, &user_44);
/* icon style */
```

```
static lv_style_t main_icon_style; lv_style_init(&main_icon_style);
lv_style_set_text_color(&main_icon_style, LV_STATE_DEFAULT, lv_color_hex(0xffffff));
lv_style_set_text_font(&main_icon_style, LV_STATE_DEFAULT, &lv_font_montserrat_44);


const uint8_t padding_btn = 16;
const uint8_t button_wh = 85;

/* btn settings */
btn_settings = lv_btn_create(win_main, NULL);
lv_obj_set_size(btn_settings, button_wh, button_wh);
lv_obj_align(btn_settings, NULL, LV_ALIGN_IN_LEFT_MID, padding_btn, 0);
lv_obj_set_style_local_bg_color(btn_settings, LV_BTN_PART_MAIN,
LV_STATE_DEFAULT, lv_color_hex(0x95afc0));
lv_obj_set_style_local_bg_color(btn_settings, LV_BTN_PART_MAIN,
LV_STATE_PRESSED, lv_color_hex(0xc3dbeb)); lv_group_add_obj(pp_group,
btn_settings);
lv_obj_add_style(btn_settings, LV_BTN_PART_MAIN, &main_btn_style);
lv_obj_set_event_cb(btn_settings, btn_settings_cb);
/* icon settings */
lv_obj_t *icon_settings = lv_label_create(btn_settings, NULL);
lv_label_set_text(icon_settings, "\uf007" );
lv_obj_set_style_local_pad_left(icon_settings, LV_LABEL_PART_MAIN,
LV_STATE_DEFAULT, 2);
//lv_obj_set_style_local_pad_top(icon_settings, LV_LABEL_PART_MAIN,
LV_STATE_DEFAULT, 20);
lv_obj_add_style(icon_settings, LV_LABEL_PART_MAIN, &user_icon_style);
/* label settings */
lv_obj_t *label_settings = lv_label_create(btn_settings, NULL);
lv_label_set_text(label_settings, "User");
lv_obj_add_style(label_settings, LV_LABEL_PART_MAIN, &main_label_style);
```

```
/* btn measure */
btn_measure = lv_btn_create(win_main,
NULL); lv_obj_set_size(btn_measure,
button_wh, button_wh);
lv_obj_align(btn_measure, btn_settings, LV_ALIGN_OUT_RIGHT_MID, padding_btn, 0);
lv_obj_set_style_local_bg_color(btn_measure, LV_BTN_PART_MAIN, LV_STATE_DEFAULT,
lv_color_hex(0xeb4d4b));

lv_obj_set_style_local_bg_color(btn_measure, LV_BTN_PART_MAIN, LV_STATE_PRESSED,
lv_color_hex(0xff7979)); lv_group_add_obj(pp_group, btn_measure);
lv_obj_add_style(btn_measure, LV_BTN_PART_MAIN,
&main_btn_style); lv_obj_set_event_cb(btn_measure,
btn_measure_cb);
/* icon measure */
lv_obj_t *icon_measure = lv_label_create(btn_measure, NULL);
//lv_label_set_text(icon_measure, LV_SYMBOL_PLAY);
lv_label_set_text(icon_measure, "\uf004");
lv_obj_set_style_local_pad_left(icon_measure, LV_LABEL_PART_MAIN, LV_STATE_DEFAULT, 2);
//lv_obj_add_style(icon_measure, LV_LABEL_PART_MAIN, &main_icon_style);
lv_obj_add_style(icon_measure, LV_LABEL_PART_MAIN, &heart_icon_style);
/* label measure */
lv_obj_t *label_measure =
lv_label_create(btn_measure, NULL);
lv_label_set_text(label_measure,
"Measure");
lv_obj_add_style(label_measure, LV_LABEL_PART_MAIN, &main_label_style);

/* btn records */
btn_record = lv_btn_create(win_main, NULL);
lv_obj_set_size(btn_record, button_wh,
button_wh);
lv_obj_align(btn_record, btn_measure, LV_ALIGN_OUT_RIGHT_MID, padding_btn, 0);
lv_obj_set_style_local_bg_color(btn_record, LV_BTN_PART_MAIN, LV_STATE_DEFAULT,
lv_color_hex(0xf9ca24)); lv_obj_set_style_local_bg_color(btn_record, LV_BTN_PART_MAIN,
LV_STATE_PRESSED, lv_color_hex(0xf6e58d)); lv_group_add_obj(pp_group, btn_record);
lv_obj_add_style(btn_record, LV_BTN_PART_MAIN, &main_btn_style);
/* icon records */
lv_obj_t *icon_record = lv_label_create(btn_record, NULL);
lv_label_set_text(icon_record, LV_SYMBOL_DIRECTORY);
lv_obj_set_style_local_pad_left(icon_record, LV_LABEL_PART_MAIN,
LV_STATE_DEFAULT, 2); lv_obj_add_style(icon_record,
LV_LABEL_PART_MAIN, &main_icon_style);
/* label records */
lv_obj_t *label_record =
lv_label_create(btn_record, NULL);
lv_label_set_text(label_record,
"Records");
lv_obj_add_style(label_record, LV_LABEL_PART_MAIN, &main_label_style);

/* window buttons style */
lv_style_init(&win_btn_style);
lv_style_set_radius(&win_btn_style,
LV_STATE_DEFAULT, 10);
lv_style_set_outline_color(&win_btn_style, LV_STATE_DEFAULT,
lv_color_hex(0xFFFFFF)); lv_style_set_border_color(&win_btn_style,
LV_STATE_DEFAULT, lv_color_hex(0xFFFFFF));
lv_style_set_border_opa(&win_btn_style, LV_STATE_DEFAULT,
LV_OPA_30); lv_style_set_text_color(&win_btn_style, LV_STATE_DEFAULT,
lv_color_hex(0xFFFFFF)); lv_style_set_outline_width(&win_btn_style,
LV_STATE_FOCUSED, 2); lv_style_set_outline_color(&win_btn_style,
```

```
LV_STATE_FOCUSED, lv_color_hex(0xb0eaff));
lv_style_set_outline_opa(&win_btn_style, LV_STATE_FOCUSED,
LV_OPA_80); lv_style_set_transform_height(&win_btn_style,
LV_STATE_DEFAULT, -10);
lv_style_set_transform_width(&win_btn_style, LV_STATE_DEFAULT, -10);

}

#if !SIMULATION
bool get_encoder_button_data_cb(lv_indev_drv_t *indev_drv, lv_indev_data_t *data){
        static int32_t last_encoder_diff = 0;
        int32_t encoder_val = btn_encoder_count;
        int32_t encoder_diff = encoder_val - last_encoder_diff;
        data->enc_diff = encoder_diff;
        last_encoder_diff = encoder_val;

        //btn select=B2 (middle)
        if (!gpio_get_level(B2)) data->state = LV_INDEV_STATE_PR;
        else data->state = LV_INDEV_STATE_REL;

        return false;
}
#endif

static void btn_settings_cb(lv_obj_t * btn, lv_event_t event){
if(event == LV_EVENT_SHORT_CLICKED) {
  settings_window();
}
static void btn_measure_cb(lv_obj_t * btn, lv_event_t event){
if(event == LV_EVENT_SHORT_CLICKED){
  measure_STATUS = SCHEDULED;
  measure_window();
}
}

void write_measure_status(measure_stage stage){
  measure_STATUS = stage;
}

measure_stage read_measure_status(){
  return measure_STATUS;
}
```

```
static void measure_window(){

/* Remove items from scrolling group */
lv_group_remove_all_objs(pp_group);

/* Create window */
lv_obj_t *win_measure =
lv_win_create(lv_scr_act(),
NULL);
lv_win_set_title(win_measur
e, "");
lv_win_set_header_height(w
in_measure, 40);
lv_obj_set_style_local_text_color(win_measure, LV_WIN_PART_HEADER,
LV_STATE_DEFAULT, lv_color_hex(0xffffff));
lv_obj_set_style_local_bg_color(win_measure, LV_WIN_PART_HEADER,
LV_STATE_DEFAULT, lv_color_hex(0x5a6975));
//lv_obj_add_style(win_measure, LV_WIN_PART_BG, &window_bg_style);
lv_obj_set_style_local_bg_color(win_measure, LV_WIN_PART_BG, LV_STATE_DEFAULT,
lv_color_hex(0xcedfed));
//0x130f40
lv_win_set_layout(win_measure, LV_LAYOUT_COLUMN_MID);
/* Close button */
btn_measure_close =
lv_win_add_btn_right(win_measure, "");
lv_obj_add_style(btn_measure_close,
LV_BTN_PART_MAIN, &win_btn_style);
lv_obj_set_event_cb(btn_measure_close,
close_win_measure_cb);
lv_obj_t *close_btn_label =
lv_label_create(btn_measure_clo
se, NULL);
lv_label_set_text(close_btn_label,
LV_SYMBOL_CLOSE);
lv_obj_set_style_local_text_color(close_btn_label, LV_LABEL_PART_MAIN,
LV_STATE_DEFAULT, lv_color_hex(0xffffff)); lv_group_add_obj(pp_group,
btn_measure_close);

/* Label */
lv_obj_t * measure_label = lv_label_create(win_measure, NULL);
lv_label_set_text(measure_label, "\n\n\nPerforming measurement...\n
Please stay still."); lv_label_set_align(measure_label,
LV_LABEL_ALIGN_CENTER);

get_measure_status_task = lv_task_create(get_measure_status_cb, 500,
LV_TASK_PRIO_MID,NULL);


}
```

```
static void measure_result_window(){

// Remove items from scrolling group
lv_group_remove_all_objs(pp_group);

// Create window
lv_obj_t *win_result =
lv_win_create(lv_scr_act(), NULL);
lv_win_set_title(win_result, "");
lv_win_set_header_height(win_result, 40);
lv_obj_set_style_local_text_color(win_result, LV_WIN_PART_HEADER, LV_STATE_DEFAULT,
lv_color_hex(0xffffff)); lv_obj_set_style_local_bg_color(win_result, LV_WIN_PART_HEADER,
LV_STATE_DEFAULT, lv_color_hex(0x5a6975));
//lv_obj_add_style(win_measure, LV_WIN_PART_BG, &window_bg_style);
lv_obj_set_style_local_bg_color(win_result, LV_WIN_PART_BG, LV_STATE_DEFAULT,
lv_color_hex(0xcedfed)); //0x130f40
//lv_win_set_layout(win_result, LV_LAYOUT_COLUMN_MID);

// Close button
btn_results_close = lv_win_add_btn_right(win_result, "");
lv_obj_add_style(btn_results_close, LV_BTN_PART_MAIN,
&win_btn_style); lv_obj_set_event_cb(btn_results_close,
close_win_results_cb);
lv_obj_t *close_btn_label =
lv_label_create(btn_results_close, NULL);
lv_label_set_text(close_btn_label,
LV_SYMBOL_CLOSE);
lv_obj_set_style_local_text_color(close_btn_label, LV_LABEL_PART_MAIN, LV_STATE_DEFAULT,
lv_color_hex(0xffffff)); lv_group_add_obj(pp_group, btn_results_close);

/* heart icon style */
static lv_style_t heart_icon_style; lv_style_init(&heart_icon_style);
//lv_style_set_text_color(&heart_icon_style, LV_STATE_DEFAULT, lv_color_hex(0xffffff));
lv_style_set_text_font(&heart_icon_style, LV_STATE_DEFAULT, &heart_20);

/*mini window style*/
static lv_style_t par_style; lv_style_init(&par_style);
lv_style_set_bg_color(&par_style,
LV_STATE_DEFAULT,lv_color_hex(0xa6a6a6));
lv_style_set_outline_width(&par_style, LV_STATE_DEFAULT, 2);
lv_style_set_outline_color(&par_style, LV_STATE_DEFAULT,
lv_color_hex(0x5a6975)); lv_style_set_outline_pad(&par_style,
LV_STATE_DEFAULT, 8);


par = lv_obj_create(win_result, NULL); /*Create a parent object on the current screen*/
lv_obj_set_size(par, 200, 160);
lv_obj_align(par,NULL,
LV_ALIGN_CENTER,0,0);
lv_obj_add_style(par, LV_LABEL_PART_MAIN, &par_style);
lv_group_add_obj(pp_group, par);
//lv_obj_set_color(par, lv_color_hex(0xa6a6a6));
```

```
/*result style*/
static lv_style_t result_style; lv_style_init(&result_style);
lv_style_set_text_font(&result_style, LV_STATE_DEFAULT, &montserrat_64);
static lv_style_t bpm_style; lv_style_init(&bpm_style);
lv_style_set_text_font(&bpm_style, LV_STATE_DEFAULT,&lv_font_montserrat_30);

/* Label SYS*/
sys_label = lv_label_create(par, NULL);
lv_obj_add_style(sys_label, LV_LABEL_PART_MAIN, &result_style);
lv_obj_align(sys_label, par,
LV_ALIGN_CENTER, 40,-30);


/* Label DIAS*/
dias_label = lv_label_create(par, NULL);
lv_obj_add_style(dias_label,
LV_LABEL_PART_MAIN,
&result_style);
lv_obj_align(dias_label, par,
LV_ALIGN_CENTER, 45,40);

/* Label BPM*/
bpm_label = lv_label_create(par, NULL);
lv_obj_add_style(bpm_label, LV_LABEL_PART_MAIN, &bpm_style);
lv_obj_align(bpm_label, par, LV_ALIGN_CENTER, -45,45);

/*Label heart*/
lv_obj_t * heart_label = lv_label_create(par, NULL);
lv_obj_add_style(heart_label, LV_LABEL_PART_MAIN, &heart_icon_style);
lv_label_set_text(heart_label, "\uf004");
lv_obj_align(heart_label, par, LV_ALIGN_CENTER, -80,45);

if(measure_STATUS == WRITE_RESULTS){
        char buf1[30];
        sprintf(buf1, "%2d", (int)(measured_bpm));
        char buf2[30];
        sprintf(buf2, "%2d", (int)(measured_sys));
        char buf3[30];
        sprintf(buf3, "%2d", (int)(measured_dias));
        lv_label_set_text(bpm_label, buf1);
        lv_label_set_text(sys_label, buf2);
        lv_label_set_text(dias_label, buf3);

        if (!gpio_get_level(B2)){
                uint32_t btn_id = 0;
                lv_event_send(btn_results_close, LV_EVENT_RELEASED, &btn_id);

                lv_win_close_event_cb(btn_results_close, LV_EVENT_RELEASED);

                /* Re-add settings group items */
                lv_group_add_obj(pp_group, btn_settings);
                lv_group_add_obj(pp_group, btn_measure);
                lv_group_add_obj(pp_group, btn_record);
                measure_STATUS = MEASURE_IDLE;

        }
}
```

```
}

static void close_win_results_cb(lv_obj_t *btn, lv_event_t event){
        if(event == LV_EVENT_RELEASED){

                lv_win_close_event_cb(btn, event);

                /* Re-add settings group items */
                lv_group_add_obj(pp_group, btn_settings);
                lv_group_add_obj(pp_group, btn_measure);
                lv_group_add_obj(pp_group, btn_record);
                measure_STATUS = MEASURE_IDLE;
        }
}
static void close_win_measure_cb(lv_obj_t *btn, lv_event_t event){
        if(event == LV_EVENT_RELEASED){
                lv_win_close_event_cb(btn, event);
                /* Re-add settings group items */
                lv_group_add_obj(pp_group, btn_settings);
                lv_group_add_obj(pp_group, btn_measure);
                lv_group_add_obj(pp_group, btn_record);


        }
}


static void get_measure_status_cb(lv_task_t *task){

#if SIMULATION
static uint8_t i = 5;
if (i== 0){
  measure_STATUS = DONE;
  write_measured_bpm(77, 114, 90);
}
i--;
#endif

if(measure_STATUS == WRITE_RESULTS){

  measure_result_window();
}
#if SIMULATION
if(kk==1)
measure_S
TATUS =
MEASURE
_IDLE;
#endif
}

void write_measured_bpm(float bpm, float sys, float dias){
measured_bpm = bpm;
measured_sys = sys; measured_dias = dias;
}

static void settings_window(){

/* Remove items from scrolling group */
```

```
lv_group_remove_all_objs(pp_group);

/* Create window */
lv_obj_t *win_settings =
lv_win_create(lv_scr_act(),
NULL);
lv_win_set_title(win_settings
, "Settings");
lv_win_set_header_height(w
in_settings, 40);
lv_obj_set_style_local_text_color(win_settings, LV_WIN_PART_HEADER,
LV_STATE_DEFAULT, lv_color_hex(0xffffff)); lv_obj_set_style_local_bg_color(win_settings,
LV_WIN_PART_HEADER, LV_STATE_DEFAULT, lv_color_hex(0x686de0)); // f2f9fa
lv_obj_add_style(win_settings, LV_WIN_PART_BG, &window_bg_style);
/* Close button */
btn_settings_close =
lv_win_add_btn_right(win_settings, "");
lv_obj_add_style(btn_settings_close,
LV_BTN_PART_MAIN, &win_btn_style);
lv_obj_set_event_cb(btn_settings_close,
close_win_settings_cb);
lv_obj_t *close_btn_label =
lv_label_create(btn_settings_close, NULL);
lv_label_set_text(close_btn_label,
LV_SYMBOL_CLOSE);
lv_obj_set_style_local_text_color(close_btn_label, LV_LABEL_PART_MAIN,
LV_STATE_DEFAULT, lv_color_hex(0xffffff)); lv_group_add_obj(pp_group,
btn_settings_close);


/* Create settings menu */
settings_menu = lv_list_create(win_settings, NULL);
lv_obj_set_size(settings_menu, lv_obj_get_width_fit(lv_scr_act())-30,
lv_obj_get_height_fit(lv_scr_act())-70);

/* Button style */
lv_style_init(&btn_settings_style);
lv_style_set_outline_color(&btn_settings_style,
LV_STATE_FOCUSED, lv_color_hex(0x22a6b3));
lv_style_set_radius(&btn_settings_style,
LV_STATE_FOCUSED, 10);


/* User select Buttons */
set_devmode_btn = lv_list_add_btn(settings_menu,NULL,"User 1");
lv_obj_add_style(set_devmode_btn, LV_BTN_PART_MAIN,
&btn_settings_style); devmode_switch = lv_switch_create(win_settings, NULL);
lv_obj_align(devmode_switch, devmode_switch, LV_ALIGN_IN_TOP_LEFT,
220, 6); lv_obj_set_event_cb(set_devmode_btn,btn_devmode_cb);


set_time_btn = lv_list_add_btn(settings_menu,LV_SYMBOL_EDIT,"Set time");
lv_obj_add_style(set_time_btn, LV_BTN_PART_MAIN, &btn_settings_style);

lv_group_add_obj(pp_group, set_devmode_btn);
lv_group_add_obj(pp_group, set_time_btn);
if(dev_state){
        add_dev_settings();
        lv_switch_on(devmode_switch, LV_ANIM_OFF);
  }
```

```
}
```

```
        static void btn_devmode_cb(lv_obj_t * btn, lv_event_t event){
                if(event == LV_EVENT_SHORT_CLICKED) {
                  lv_switch_toggle(devmode_switch, LV_ANIM_ON);

                if (dev_state){
                  dev_state = false;
                  lv_list_remove(settings_menu, lv_list_get_size(settings_menu)-1);
                  lv_list_remove(settings_menu, lv_list_get_size(settings_menu)-1);

                } else{
                  dev_state = true; add_dev_settings();
                }

                }
        }


        static void close_win_settings_cb(lv_obj_t *btn, lv_event_t event){
                 if(event == LV_EVENT_RELEASED){
                          lv_win_close_event_cb(btn, event);
                         /* Re-add settings group items */
                         lv_group_add_obj(pp_group, btn_settings);
                         lv_group_add_obj(pp_group, btn_measure);
                         lv_group_add_obj(pp_group, btn_record);
                }
}
```

Anexo 2: Presupuesto

0. Introducción

En este documento se presenta el presupuesto del proyecto, dividido en diferentes categorías. Todos los precios del documento están en euros.
Por otro lado, los precios señalados en este documento no incluyen el impuesto I.V.A, éste se añadirá en el cálculo total.

1. Materiales y componentes

1.1 Componentes electrónicos

Nombre	Designante	Descripción	Cantidad	Fabricante	Proveedor	Precio (€)	Precio unitario (€)	Precio unitario ($)
LM7805	U4	Regulador de tensión	1	TF	LCSC	0,08531892	0,08531892	0,105332
2N3904(SOT-23)	Q3,Q2	Transistor	2	KEC	LCSC	0,009801	0,0049005	0,0121
K2-3.6×6.1_SMD	B1,B2,B4,B3	Botón	4		LCSC	0,03645	0,0091125	0,045
10uF	C6	Condensador	1	ValuePro	LCSC	0,008424	0,008424	0,0104
PWR_CONN_2P	P1	Conectores	1	Skywin	Aliexpress	0,5751	0,5751	0,71
MTR_CONN_2P	P3	Conectores	1	Skywin	Aliexpress	0,5751	0,5751	0,71
VLV_CONN_2P	P2	Conectores	1	Skywin	Aliexpress	0,5751	0,5751	0,71
100n	C17,C13,C9,C15,C8,C18,C14	Condensador	8	LCSC	LCSC	0,008991	0,001123875	0,0111
1u	C10	Condensador	1	ValuePro	LCSC	0,016119	0,016119	0,0199
DPY_CONN_9P	CN1	Conectores	1	Ningbo Xinlaiya Elec.	LCSC	0,137214	0,137214	0,1694
FQP30N06L	Q5,Q1	Transistor	2	VBsemi Elec	LCSC	0,164618973	0,0823094865	0,2032333
MPRLS0025A	U6	Sensor de presión	1	Honeywell	Adafruit	12,1095	12,1095	14,95
ESP32-WROOM-32DC	U8	Microcontrolador	1	Espressif Systems	LCSC	2,31463656	2,31463656	2,857576
1u	C11,C12	Condensaores	2	ReliaPro	LCSC	0,09153	0,045765	0,113
100u	C7	Condensaores	1	AVX	LCSC	0,155925	0,155925	0,1925
LED-Red(0603)	LED1	Diodo LED	1	Hubei KENTO Elec	LCSC	0,00324	0,00324	0,004
LED-Blue(0603)	LED2	Diodo LED	1	EVERLIGHT	LCSC	0,012312	0,012312	0,0152

Nombre	Designante	Descripción	Cantidad	Fabricante	Proveedor	Precio (€)	Precio unitario (€)	Precio unitario ($)
LD1117V33	U3	Regulador de tensión	1	STMicroelectronics	LCSC	0,28998	0,28998	0,358
1k	R12,R5,R4	Resistencias	3	Guangdong Fenghua Advanced Tech	LCSC	0,001944	0,000648	0,0024
22k	R8,R2	Resistencias	2	Uniroyal Elec	LCSC	0,001539	0,0007695	0,0019
10k	R10,R13,R15,R16,R7,R9,R6	Resistencias	7	Uniroyal Elec	LCSC	0,024705	0,003529285714	0,0305
100k	R11,R14	Resistencias	2	Uniroyal Elec	LCSC	0,006885	0,0034425	0,0085
1k	R3,R1	Resistencias	2	Guangdong Fenghua Advanced Tech	LCSC	0,001944	0,000972	0,0024
LM7806CT	U5	Regulador de tensión	1	Unisonic Tech	LCSC	0,195048	0,195048	0,2408
100nF	C2,C3,C5	Condensadores	3	LCSC	LCSC	0,008991	0,002997	0,0111
1N4007W	D1,D2	Diodo	2	BLUE ROCKET	LCSC	0,006885	0,0034425	0,0085
330nF	C4,C1	Condensadores	2	ValuePro	LCSC	0,005994	0,002997	0,0074
LM324	U2	Regulador de tensión	1	PUOLOP	LCSC	0,061236	0,061236	0,0756
smd button	B5,B6,B7,B8	Botones	4	TLZWLA	Aliexpress	1,35	0,3375	1,666666667
BTN_CONN_4P	CN2	Conectores	1	Skywin	Aliexpress	0,64	0,64	0,7901234568
DPY_CONN_9P	U8	Pantalla TFT	1	TZT	Aliexpress	4,83	4,83	5,962962963
Motor		Bomba de aire	1	World Driven	Aliexpress	5,04	5,04	6,222222222
Válvula		Válvula	1	Shenzhen	Aliexpress	1,16	1,16	1,432098765

1.2 Componentes no electrónicos

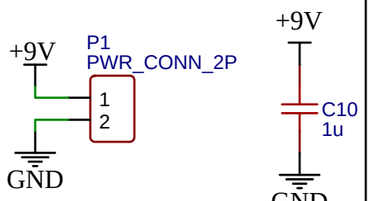
Nombre	Descripción	Cantidad	Fabricante	Proveedor	Precio (€)
Brazalete	Brazalete de presión	1	Salorie	Aliexpress	4,84

2. Presupuesto global

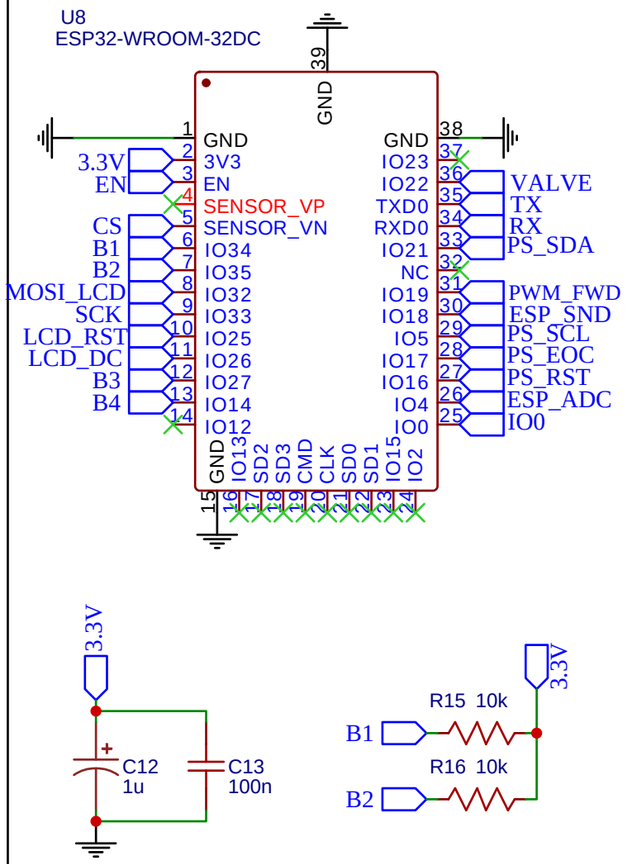
Partida		Precio unitario (€)
Partida de Materiales y Componentes	Componentes electrónicos	30,50
	Componentes no electrónicos	4,84
	Precio Total (sin I.V.A)	35,34
	Precio Total (con I.V.A)	42,76

Anexo 3: Esquema general del circuito

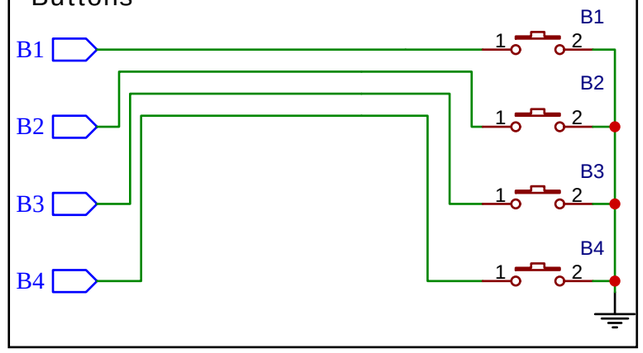
Power



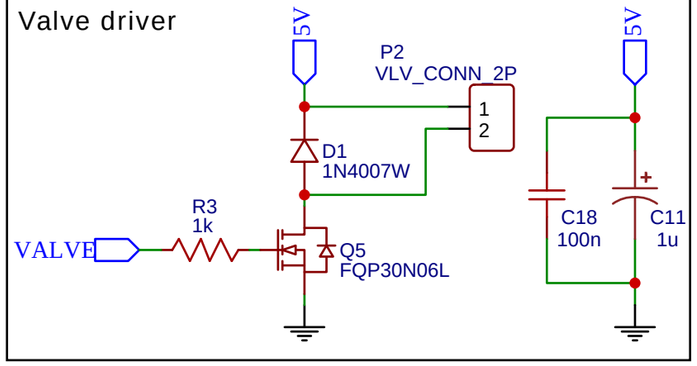
MCU



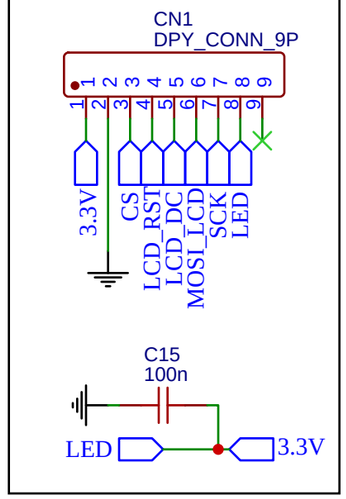
Buttons



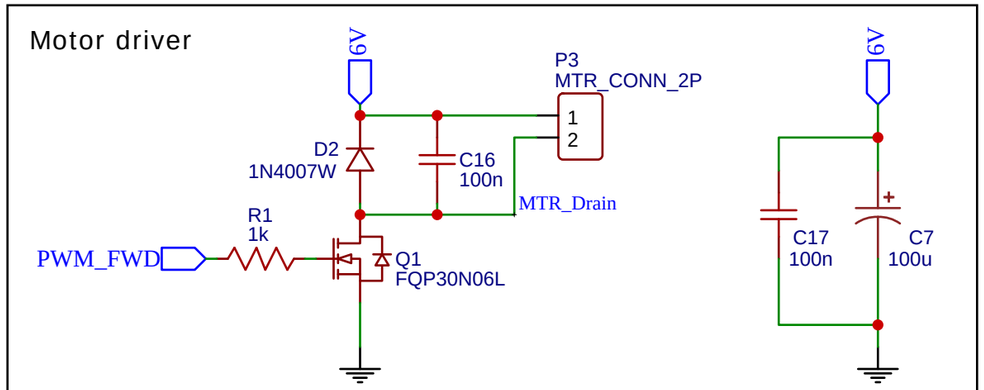
Valve driver



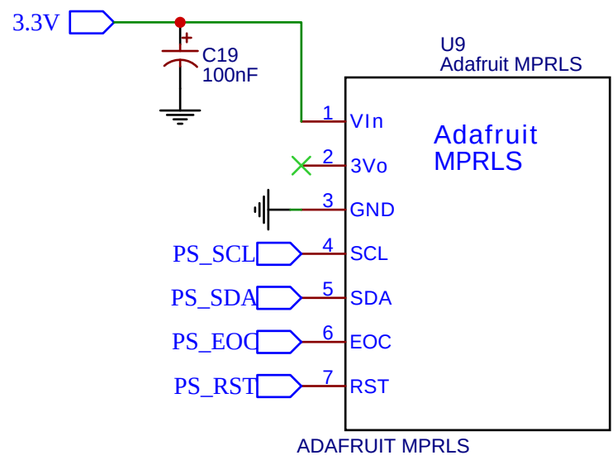
TFT Display (SPI)



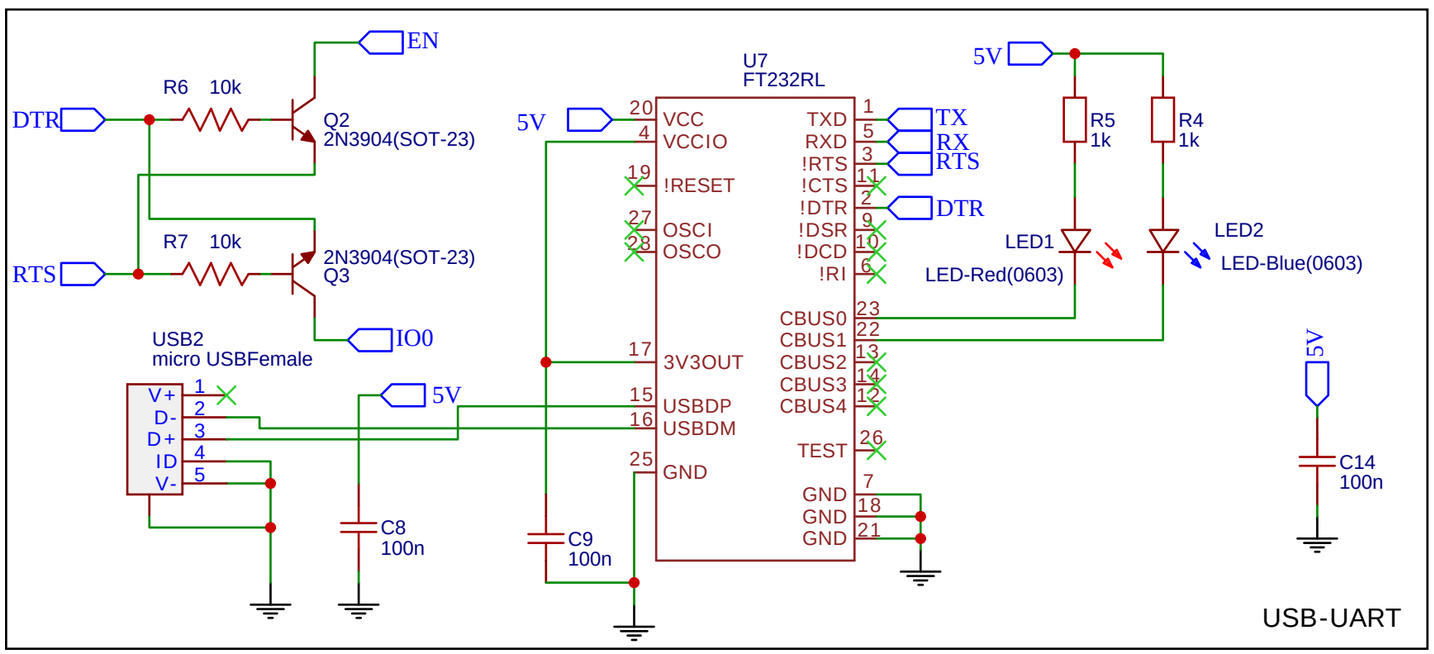
Motor driver



Pressure sensor breakout board



USB-UART



TITLE: Medidor pulso y tensió n		REV: 1.0
Escuela de Ingeniería y Arquitectura Universidad Zaragoza	Company: Unizar	Sheet: 1/2
	Date: 2021-06-11 Drawn By: Isabel Romeo	