Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería Electrónica Industrial y Automática

Seguimiento de Constantes Vitales

Autor:

Díez Arias, Alberto

Tutor:

Díez Muñoz, Pedro Luis
Dpto. de Tecnología Electrónica

Valladolid, Julio 2020.

Índice

Tabla de contenido

Tabla de imágenes

Tabla de tablas

Resumen/Abstract

El objetivo de este trabajo es diseñar un dispositivo portable capaz de tomar mediciones de constantes médicas y vitales de una persona y enviarlas a un dispositivo móvil, donde se mostrarán las lecturas tomadas.

El trabajo pretende abarcar los diferentes aspectos aprendidos durante el grado, desde los fundamentos teóricos, pasando por la electrónica, la programación y el desarrollo de una aplicación para dispositivos móviles. Se ha planteado el trabajo como un proyecto de aprendizaje en el cual se explicarán los distintos pasos que se han ido dando, así como los problemas y errores surgidos en el mismo, así como sus posibles correcciones.

- Palabras clave

Pulsioximetría, Android, bluetooth, dispositivo portable.

The main goal of this work is to design a portable device able to take medical and vital signs measurements from a person and send them to a mobile device, where those data readings would be displayed.

This work tries to embrace the different aspects learnt during the grade, from the theorical foundations, through electronics, the programming and the developement of an app for mobile devices. The work has been conceived as a learning Project in which the different steps of the same would be explained, and the problems and errors that has come up from it, such as the posible fixes aswell.

- *Keywords*

Pulse oximetry, Android, bluetooth, portable device.

1. Introducción

Nuestro objetivo es desarrollar un dispositivo portable, como por ejemplo una pulsera o brazalete. Se pretende que el dispositivo sea lo más pequeño y cómodo posible y con la menor cantidad de cableado a los sensores, con los métodos menos invasivos posibles.

La memoria se divide en varias partes, cada una enfocada a uno de los diferentes bloques que componen el proyecto: fundamentos teóricos, esquema electrónico, programación del dispositivo y aplicación para Android.

Se procede ahora a comentar ciertos aspectos del proyecto que condicionarán el desarrollo del mismo.

1.1. Signos vitales

Se definen como signos vitales a las medidas fisiológicas que sirven para valorar las funciones corporales básicas. Existen cuatro constantes vitales básicas consensuadas, además de algunas adicionales que también se consideran como tal. Las constantes vitales básicas son:

- Tensión arterial.
- Frecuencia cardíaca o pulso.
- Temperatura corporal.
- Frecuencia respiratoria.

Algunas otras fuentes añaden otras, entre las cuales podemos encontrarnos el nivel de glucosa en sangre o la saturación de oxígeno, hablándose así de cinco y hasta seis signos vitales.

Los signos vitales que tomaremos serán saturación de oxígeno y frecuencia cardíaca, puesto que su medición está altamente relacionada entre sí. El método que permite determinar el porcentaje de saturación de oxígeno en sangre con la ayuda de métodos fotoeléctricos se conoce como

pulsioximetría, y es un método no invasivo. Esto es, no necesitaremos penetrar o romper la piel para obtener los datos requeridos.

Además, en la medida de lo posible, intentaremos diseñar un sensor lo más ergonómico y discreto posible.

De forma adicional, la aplicación en Android tomará los datos del paciente para calcular el índice de masa corporal y mostrarlo como una variable más.

1.2. Microcontrolador

Para el desarrollo de nuestro dispositivo es necesario un microcontrolador con el que controlar los sensores, programar la rutina del dispositivo, procesar los datos y enviarlos a través de un puerto serie al dispositivo móvil. Elegiremos para ello un procesador de la familia ARM.

Los microcontroladores de la familia ARM son unos procesadores avanzados basados en arquitectura RISC (Reduced Instruction Set Computer) que actualmente podemos encontrar en muchos dispositivos electrónicos actuales, como smartphones, Smart TVs, smartwatches... ARM licencia su tecnología a otros fabricantes, como ATMEL o Samsung.

Basaremos el desarrollo del proyecto en una placa de desarrollo Arduino DUE, basada en el procesador SAM3X8E ARM Cortex-M3. Las ventajas que nos proporciona el microcontrolador ARM del Arduino DUE frente a otros microcontroladores, como pueden ser los AVR como el Atmega328P de placas como el Arduino UNO o Arduino Nano, son:

- Mayor velocidad de procesamiento.
- Frecuencia mayor del reloj interno.
- Resolución de hasta 12 bits para las lecturas analógicas.
- Hasta 4 puertos serie.
- Alimentación de 3,3V.

Puesto que la alimentación del microcontrolador es de 3,3V, deberemos diseñar el circuito electrónico con estas especificaciones y limitaciones.

1.3. Comunicación

Actualmente la comunicación entre dispositivos es algo habitual gracias al pequeño tamaño y gran capacidad de las nuevas tecnologías sobre microprocesadores y microelectrónica. Existen dos principales métodos de intercomunicación entre dispositivos: la tecnología Bluetooth y el concepto del internet de las cosas, o *Internet of things* (IoT por sus siglas en inglés).

Bluetooth es una especificación industrial para Redes Inalámbricas de Área Personal (WPAN) que permite la comunicación y transmisión de datos mediante radiofrecuencia en la banda ISM de los 2,4GHz. Esta tecnología está ampliamente extendida en dispositivos móviles y la gran mayoría de móviles o tabletas admiten conexión por Bluetooth. Es un protocolo diseñado para dispositivos de bajo consumo. Su principal desventaja es que es una comunicación a corta distancia. Conectarse a un dispositivo requiere conocer la dirección MAC del mismo.

El internet de las cosas es un concepto que se ha popularizado en los últimos años y que consiste en la interconexión de dispositivos, generalmente objetos cotidianos, mediante internet. Para ello, el dispositivo tendría que encontrarse dentro de una red local para así intercomunicarnos con él mediante otro dispositivo, si bien perteneciente a la misma red o bien externo a la misma y pudiendo acceder a la red. Esto es posible gracias al desarrollo y popularización actual de los dispositivos móviles inteligentes. La principal desventaja de este método es que se debe conocer la dirección IP de la red en la que se encuentra el dispositivo con el que queremos comunicarnos, así que éste ha de pertenecer siempre a la misma red.

El enfoque que se ha hecho al proyecto es la de un dispositivo de bajo

coste y portable que pueda ser empleado en cualquier lugar, por lo que se ha optado por una transmisión de datos a través de Bluetooth. A pesar de que esto requiere una conexión de corto alcance entre el dispositivo y un móvil o tableta, este último puede ser empleado también para transmitir los datos mediante internet a otro dispositivo lejano, por lo que salvamos la principal desventaja de la conexión Bluetooth. Si se quisiera orientar el proyecto de forma que el dispositivo permaneciera siempre en una red, como podría ser el caso de un dispositivo pensado para usar en hospitales u otros centros públicos, podría considerarse el empleo del concepto del internet de las cosas. Como ejemplo serviría un hospital en el cual se pueden monitorizar los signos vitales de un paciente por parte del personal médico del centro.

1.4. Regulación y normativa

Todo producto sanitario o médico incluidos dispositivos electrónicos o programas informáticos desarrollados en España deben estar regulados tanto por la legislación española como por la normativa de la Comisión Europea.

Existe una clasificación en 4 clases de todo producto sanitario o médico a nivel global en función del riesgo que presenten. La Organización Mundial de la Salud (OMS) establece las siguientes herramientas y requerimientos regulatorios de los dispositivos médicos en su publicación *Medical Device Regulations, global overview and guiding principles*:

Table 2. Tools and general requirements of the five members of the GHTF

COUNTRY/REGION	PRE-MARKET	PLACING ON-MARKET		POST MARKET
	Product control Tools for acknowledging product cleared for the market	Medical device establishment control	Advertising control	Vendor after-sale obligations Examples of common requirements
Australia*	ARTG number	Enterprise Identification (ENTID)	Generally, prohibition of advertisement before a device is cleared to enter the market. Prohibition of any misleading or fraudulent advertisement	1. Problem reporting 2. Implant registration 3. Distribution records 4. Recall procedure 5. Complaint handling
Canada	Device licence	Establishment licence		
European Union	Compliance label (CE mark)	Responsible person registration		
Japan**	Shounin (approval) or Todokede (notification)	Seizo-Gyo (Manufacturer Licence) Yunyu Hanbai-Gyo (Import Licence) Hanbai Todoke (Sales notification)		
United States of America	Approval Letter (PMA) or Marketing Clearance (510k)	Establishment registration		

* Australia's new medical devices legislation was passed by the Australian Parliament in April 2002 (see www.health.gov.au/tga/)
** Japan's PAL (Pharmaceutical Administration Law) revision is scheduled for 2005.

Tabla 1. Herramientas y requerimientos regulatorios según la OMS.

Además establece la normativa y los estándares de calidad requeridos para las distintas regiones, siendo la normativa ISO13485 e ISO13488 las aplicables dentro del ámbito de la Unión Europea. La norma ISO14971 además es una normativa internacional sobre la gestión de riesgos de productos sanitarios.

Table 3. Quality system standards used by different authorities

COUNTRY/REGION	STANDARDS/REGULATIONS	CONFORMITY ASSESSMENT
Australia	ISO13485 or EN46001* ISO13488 or EN46002*	Government and Third party
Canada	ISO13485, ISO13488	Third party
European Union	EN46001* or ISO13485 EN46002* or ISO13488	Third party
Japan	GMP #40 ordinance GMPI #63 ordinance QS Standard for medical devices #1128 notice	Government
United States	QS (21 CFR part 820)	Government

* EN46001 and EN46002 are being phased out by the end of March 2004.

Tabla 2. Normativa y estándares regulatorios por región según la OMS.

Dentro del marco de la Unión Europea, la Comisión Europea regula los productos médicos a través de las directivas 90/385/EEC, 93/42/ECC y 98/79/EC, corregidas y revisadas a 5 de Abril de 2017 mediante las regulaciones 2017/745 y 2017/746. En este caso, la normativa que nos compete es la 2017/745, referente a dispositivos médicos. Los requisitos generales de seguridad y funcionamiento pueden encontrarse en el Anexo I de la misma.

Por último, en España los dispositivos médicos están regulados por la normativa establecida por la Agencia Española de Medicamentos y Productos Sanitarios (AEMPS), una agencia estatal adscrita al Ministerio de Sanidad, Consumo y Bienestar Social del gobierno de España. La legislación vigente se encuentra descrita por el Real Decreto 1591/2009, del 16 de octubre.

Por ejemplo, para nuestro proyecto, deberíamos atenernos al artículo 12 y sub-artículos del mismo, sobre dispositivos electrónicos equipados con una fuente de energía, como es nuestro caso. Si bien no es el objetivo de este trabajo el estudio y cumplimiento de la normativa vigente con respecto a dispositivos médicos, sino más bien el estudio y desarrollo desde el punto de vista técnico, esta normativa es de obligado cumplimiento si deseáramos desarrollar y comercializar nuestro producto.

2. Conceptos Teóricos

Como hemos mencionado anteriormente en el apartado 1.1, los signos vitales son las medidas fisiológicas que sirven para valorar las funciones corporales básicas. En nuestro caso hemos establecido que los signos vitales a medir en nuestro dispositivo son la frecuencia cardíaca y la saturación de oxígeno en sangre.

Se define como pulsioximetría como el método no invasivo que permite determinar el porcentaje de saturación de oxígeno en sangre de un paciente con la ayuda de métodos fotoeléctricos. La encargada de transportar el oxígeno O_2 desde los pulmones hasta los tejidos del resto del cuerpo es la hemoglobina, una hemoproteína que se encuentra en la sangre y que confiere a la misma de un color rojo característico. La hemoglobina está compuesta por cuatro cadenas polipeptídicas llamadas globinas a las cuales se les une un grupo hemo, con un átomo de hierro que permite crear un enlace reversible con las moléculas de oxígeno, el cual permite transportar el oxígeno a través del torrente sanguíneo.

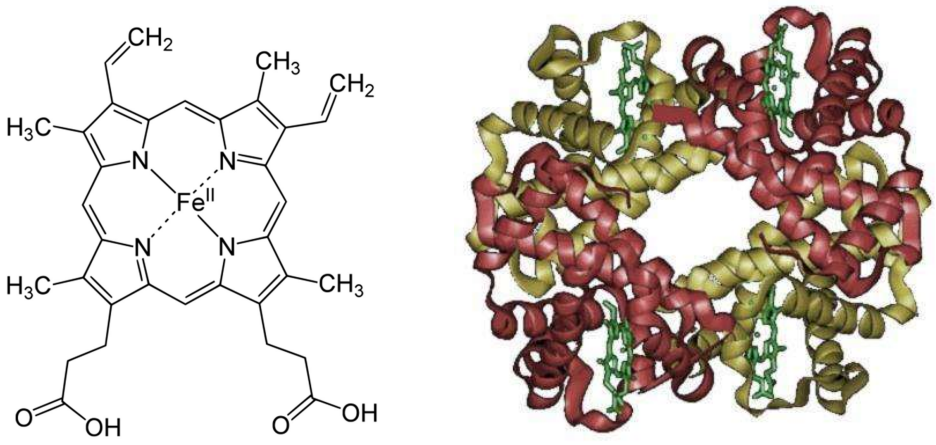


Ilustración 1. A la izquierda, estructura molecular del grupo hemo. A la izquierda, representación virtual de una molécula de hemoglobina.

La sangre se satura de oxígeno a su paso por los pulmones gracias a la hemoglobina. Cuando la molécula de oxígeno se enlaza con oxígeno gracias a

los átomos de hierro (hasta 4 moléculas de oxígeno por molécula de hemoglobina), se la denomina oxihemoglobina o hemoglobina oxigenada.

$$Hb + 4O_2 \rightarrow Hb(O_2)_4$$

A medida que la sangre recorre el resto del cuerpo va perdiendo el oxígeno. Cuando la hemoglobina pierde todo el oxígeno se la lama desoxihemoglobina o hemoglobina reducida. Mientras que la hemoglobina oxigenada presenta un color rojo intenso, característico de la sangre arterial, al convertirse en hemoglobina reducida pasa a tomar un color rojo más oscuro, característico de la sangre venosa. Es esta diferencia de color la que nos permitirá medir la saturación de oxígeno en sangre mediante fotopletismografía.

La fotopletismografía es una técnica médica mediante la cual se obtiene un fotopletismograma (PPG por las siglas del inglés *photoplethysmography*), una medición de los cambios de volumen de un órgano, en este caso de la sangre, mediante haces de luz.

La variación de volumen de sangre en la sangre se detecta iluminando una sección de piel mediante un diodo emisor de luz (LED) y midiendo la cantidad de luz que llega a un fotodiodo al otro lado de dicha sección. Para que el receptor reciba suficiente luz y permitir la toma correcta de datos, la medición debe tomarse en partes del cuerpo lo suficientemente translúcidas. Las mediciones de oxígeno en sangre mediante oxímetros suele tomarse en el extremo de los dedos de la mano, o en los lóbulos de las orejas.

La determinación de la saturación en sangre mediante fotopletismografía se basa en la ley de Beer-Lambert, que relaciona la cantidad de luz absorbida con las propiedades del material que atraviesa. La cantidad de luz de una determinada longitud de onda absorbida depende de la cantidad de especie absorbente con la que se encuentra la luz al pasar por la muestra.

Según esta ley, podemos determinar la absorbancia (A) de una

muestra por la relación entre el espacio recorrido por la luz a través de la misma (d), la concentración de la especie absorbente (c) y el coeficiente de atenuación molar o absortividad de la especie (ε).

$$A = \varepsilon dc$$

De igual manera, se puede calcular la absorbancia a partir de la relación entre la intensidad de luz entrante (I_0) y saliente (I_1) al medio.

$$A = -log_{10}\frac{I_1}{I_0}$$

La hemoglobina absorberá la luz de forma distinta en función de si está oxigenada o no. Midiendo la cantidad de luz absorbida por oxihemoglobina y por desoxihemoglobina podemos calcular el ratio entre ambas, o lo que es lo mismo, la saturación de la hemoglobina o de oxígeno en sangre. Se indica SaO_2 para la saturación de oxígeno arterial y SpO_2 para la saturación de oxígeno capilar periférica:

$$SaO_2 = \frac{HbO_2}{HbO_2 + Hb}$$

De la misma forma que oxihemoglobina y desoxihemoglobina no absorben la misma cantidad de luz, tampoco responden de la misma manera a diferentes longitudes de onda. Se puede observar su comportamiento a través de un gráfico espectral de absorción de luz de ambas.

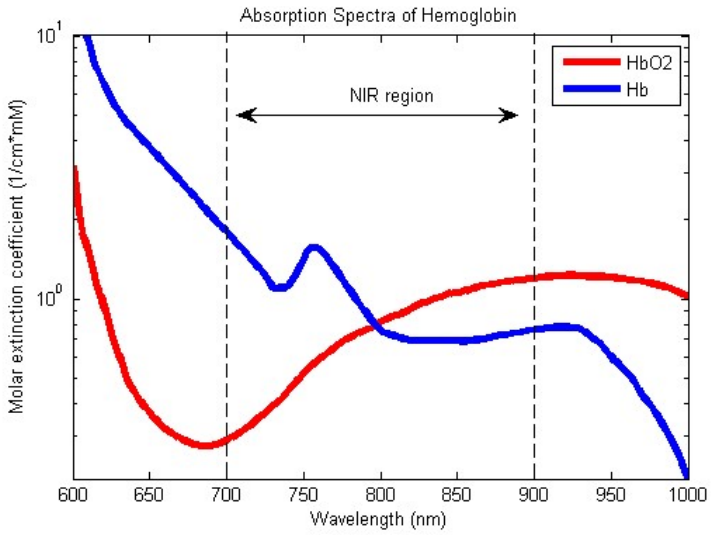


Ilustración 2. Espectro de absorción de la oxihemoglobina y la hemoglobina reducida.

Existe un punto para cada una de ellas en la cual la diferencia de absorción entre ambas se maximiza, de tal forma que para una de ellas la absorción es máxima y para la otra es mínima. Esos valores corresponden a una longitud de onda de 940nm para la oxihemoglobina y de 660nm para la hemoglobina reducida.

Estudiar la absorción de hemoglobina reducida y oxihemoglobina a longitudes de 660nm y 940nm nos permite determinar la cantidad absorbida de hemoglobina reducida y oxihemoglobina respectivamente, de forma que obviamos la cantidad absorbida por la otra, aislando así lo máximo posible la cantidad de luz absorbida por ella.

Existe una problemática a la hora de medir la luz a través del cuerpo, como por ejemplo a través de un dedo, ya que el resto de tejidos, como piel, músculo o masa ósea, van a absorber parte de la luz. Además la fisionomía propia difiere mucho no solo entre cada persona sino también de la posición en que se coloque el transductor del pulsioxímetro.

Takuo Aoyagi descubrió que la luz que atravesaba a través del cuerpo presentaba ondas pulsátiles puesto que, mientras que la luz absorbida por otros tejidos era constante, la absorbida por la hemoglobina era variable y pulsante. Esto permite obviar la cantidad de luz absorbida de forma absoluta

y así solamente es necesario medir la cantidad pulsante.

Para el diseño de nuestro dispositivo, leeremos mediante un fotodiodo la luz absorbida a través del dedo y que emiten dos LED, uno rojo de longitud de onda de 660nm y otro infrarrojo con una longitud de onda de 940nm. Cada LED emitirá luz de forma alterna y la variación de intensidad a través del fotodiodo nos permitirá tomar lecturas de voltaje proporcionales a la luz absorbida al pasar el dedo, proporcionándonos un fotopletismograma. Mediremos la variación pulsante relativa para cada una, y la relación entre ambas nos permitirá establecer el nivel de saturación en sangre cotejándola con los valores de un oxímetro de pulso comercial.

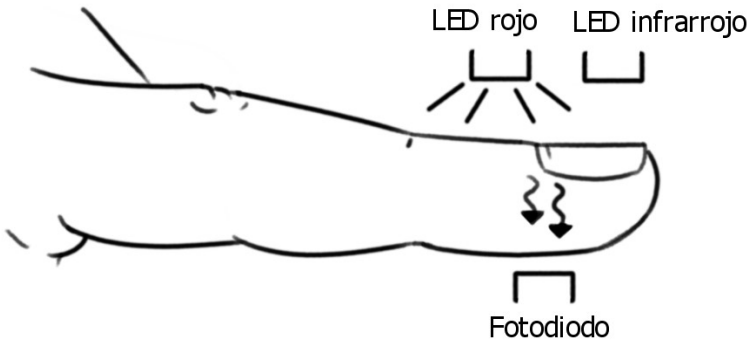


Ilustración 3. Esquema del funcionamiento de un oxímetro.

Los valores de saturación en sangre normales oscilan entre 96% y 99%, y deberían ser superiores a 94%. Un nivel de saturación en sangre inferior a 90% puede considerarse hipoxia, comúnmente producida por una anemia.

Algunas abreviaturas para referirnos al nivel de saturación de oxígeno en sangre son:

Abreviaturas comunes por la saturación de oxígeno de diferentes tipos

sO2	Saturación de oxígeno en general
SaO2	Saturación de oxígeno arterial (medida en muestra de sangre arterial)[3]
SpO2	Saturación de oxígeno capilar periférica (casi-arterial)[4]
SvO2	Saturación de oxígeno venosa[3 5]
ScvO2	Saturación de oxígeno venosa central[5]
S$\bar{v}$O2	Saturación de oxígeno venosa mixta[6]

Tabla 3. Abreviaturas comunes para la saturación de oxígeno en sangre.

El nivel de oxígeno en sangre, además de medir el porcentaje de oxihemoglobina como se ha descrito anteriormente, se puede medir según la presión arterial del oxígeno en milímetros de mercurio. La equivalencia entre el porcentaje de saturación en sangre y la presión arterial del oxígeno quedaría:

Saturación de oxígeno	Presión arterial de oxígeno
100%	228 mmHg
98,4%	100 mmHg
95%	80 mmHg
90%	59 mmHg
80%	48 mmHg
73%	40 mmHg
60%	30 mmHg
50%	26 mmHg
40%	23 mmHg
35%	21 mmHg
30%	18 mmHg

Tabla 4. Equivalencia entre saturación y presión arterial de oxígeno.

Como hemos visto, la medición mediante fotometría nos proporciona un fotopletismograma pulsante función de la variación de volumen de la sangre. Esos pulsos corresponden a cada uno de los pulsos cardíacos, lo que, además de permitirnos calcular la saturación de oxígeno en sangre, nos permitirá calcular el segundo signo vital que debemos determinar: la frecuencia cardíaca.

Se define como frecuencia cardíaca o pulso a la cantidad de contracciones de corazón o pulsaciones por unidad de tiempo, generalmente por minuto en pulsaciones por minuto (ppm, bpm por las siglas en inglés *beats per minute*). La medición del pulso suele tomarse en partes del cuerpo donde las arterias están más expuestas, como en el cuello o las muñecas, o en la caja torácica donde las pulsaciones del corazón son más notables.

Para la toma de frecuencia cardíaca se tiene en cuenta la actividad o las condiciones en las que se toman, en reposo o en actividad. Ésta depende de diferentes factores, desde genéticos hasta el estado físico de la persona.

La frecuencia cardíaca máxima (FC depende del sexo y de la edad:

	Frecuencia cardíaca máxima
Hombre	220-edad (años)
Mujer	215-edad (años)

Tabla 5. Frecuencia cardíaca máxima en función del sexo y la edad.

Existen fórmulas más precisas para el cálculo de la frecuencia cardíaca máxima que tienen en consideración la edad y el peso. La frecuencia cardíaca mínima no es inferior de 59 pulsaciones por minuto en reposo, aunque en el caso de deportistas puede llegar a ser de 39.

newborn (0–3 months old)	infants (3 – 6 months)	infants (6 – 12 months)	children (1 – 10 years)	children over 10 years & adults, including seniors	well-trained adult athletes
99-149	89–119	79-119	69–129	59–99	39–59

Tabla 6. Frecuencias máximas y mínimas por rango de edad.

Puesto que el dispositivo ha de proporcionar una respuesta en tiempo real, contabilizar las pulsaciones en un minuto no es una opción viable. Se tomará por tanto el tiempo entre pulsos según la onda fotopletismográfica y con el período de cada pulso se calculará la frecuencia del mismo en ppm.

La frecuencia cardíaca es variable en cada pulso, así que se tomará la frecuencia cardíaca ponderada en un período de tiempo lo suficientemente corto para permitir que los datos sean mostrados en tiempo real y lo suficientemente largo como para que compense las posibles variaciones puntuales de pulsaciones.

Finalmente, se define como índice de masa corporal (IMC) a la relación matemática entre altura y masa corporal de un individuo. Fue ideado por el estadístico belga Adolphe Quetelet, por lo que también se conoce como índice de Quatelet. Se puede calcular mediante tablas o a partir la siguiente expresión:

$$IMC = \frac{masa}{estatura^2}$$

El IMC es un indicador del estado corporal de una persona y según su valor se suele dividir en tres categorías: infrapeso, peso normal, sobrepeso y obesidad. EL IMC no tiene en cuenta el sexo o la edad, aunque existen tablas por percentiles de sexo y edad, generalmente para el cálculo del mismo en niños y adolescentes.

Clasificación de la OMS del estado nutricional de acuerdo con el IMC[6]

Clasificación	IMC (kg/m²)	
	Valores principales	Valores adicionales
Bajo peso	**<18,50**	**<18,50**
Delgadez severa	<16,00	<16,00
Delgadez moderada	16,00 - 16,99	16,00 - 16,99
Delgadez leve	17,00 - 18,49	17,00 - 18,49
Normal	18,5 - 24,99	18,5 - 22,99
		23,00 - 24,99
Sobrepeso	**≥25,00**	**≥25,00**
Preobeso	25,00 - 29,99	25,00 - 27,49
		27,50 - 29,99
Obesidad	**≥30,00**	**≥30,00**
Obesidad leve	30,00 - 34,99	30,00 - 32,49
		32,50 - 34,99
Obesidad media	35,00 - 39,99	35,00 - 37,49
		37,50 - 39,99
Obesidad mórbida	≥40,00	≥40,00

Tabla 7. Clasificación de pesos en función del IMC.

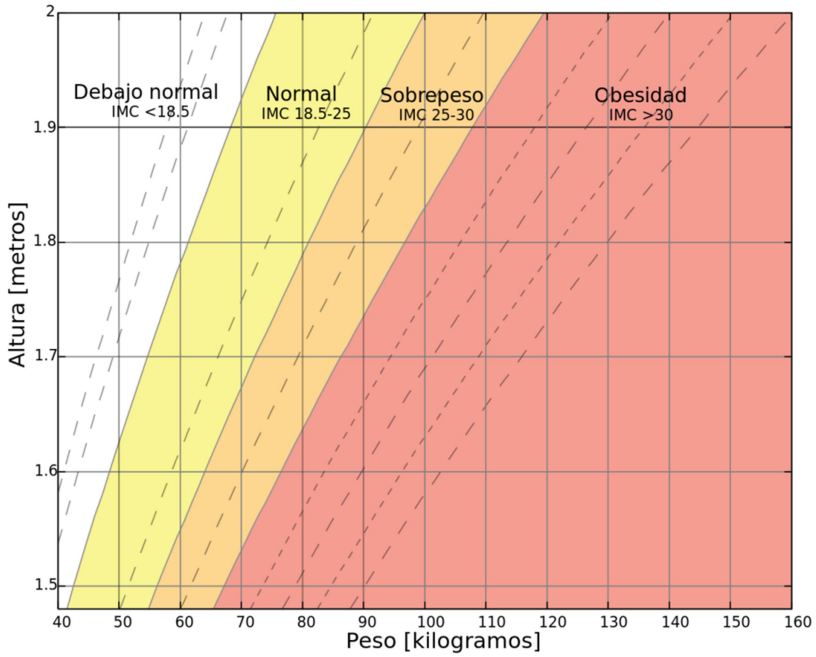


Ilustración 4. Tabla para el cálculo del índic de masa corporal.

Una vez detallados los conceptos médicos teóricos hay que recordar que, a pesar de ser necesaria la explicación de los mismos para el diseño del dispositivo, el objetivo del proyecto se centra en la parte técnica del mismo y no en la médica, y por tanto no se evaluará en ningún caso el resultado de las mediciones.

3. Diseño electrónico

3.1. Arduino DUE

Como se ha comentado, el dispositivo se va a diseñar para ser controlado por un microcontrolador que emplee tecnología ARM. Para desarrollar el dispositivo nos ayudaremos de una placa de desarrollo Arduino DUE, basada en un microprocesador ARM de 32 bits Atmel SAM3X8E ARM Cortex-M3.

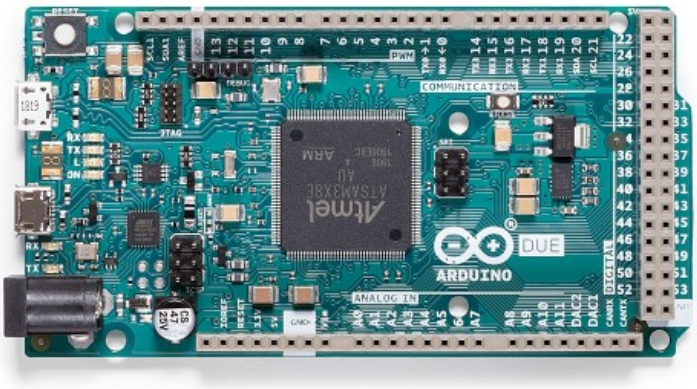


Ilustración 5. Placa de desarrollo Arduino DUE.

La placa dispone de 54 pines digitales de entrada y salida, 12 entradas analógicas con hasta 12 bits de resolución, 4 puertos de serie UART y reloj de 84MHz. Además, la placa cuenta con dos puertos micro USB para programarlo:

- El programming port está conectado a un microcontrolador ATmega16U2 que hace las veces de intermediario. El ATmega16U2 se conecta al SAM3X8E a través los puertos RX0 y TX0 del puerto serie UART0 que permiten la programación Serial-USB, además de permitir la comunicación entre el computador y la placa.

- El native port se conecta directamente con el SAM3X8E y se programa a través de él al estar conectado al puerto UOTGID. Además, este puerto sirve para conectar periféricos a la placa Arduino.

La placa puede alimentarse de forma externa mediante una fuente mediante 6 y 20V, entre 7 y 12V recomendados, sin embargo, a diferencia de otras placas Arduino, el microcontrolador se alimenta mediante una tensión de 3,3V, y la tensión máxima tolerada en cualquiera de sus entradas o salidas es de 3,3V. Por tanto, esta tensión máxima es la que determinará el futuro desarrollo del dispositivo.

La máxima corriente soportada por la placa es de 800mA, y la máxima soportada en pines es de 15mA o 3mA dependiendo del pin (ver tabla de correspondencia de pines para el Arduino DUE adjunta en bibliografía).

Para programarlo, emplearemos el entorno de desarrollo integrado (IDE) de Arduino, así como la versión online de Arduino Create.

3.2. Sensor

El sensor corresponde a la parte que se ajustará, en este caso, al dedo del paciente y que contendrá los dos LED y el fotodiodo.

Un fotodiodo es un diodo que, al incidir sobre él la luz, produce una corriente eléctrica proporcional a la luz que incide sobre el mismo. A diferencia de una fotorresistencia LDR, en el fotodiodo no se produce una caída de tensión entre sus extremos.

Como se ha comentado anteriormente, emplearemos dos LED: uno rojo con una longitud de onda de 660nm y otro infrarrojo con una longitud de onda de 940nm. El encendido y apagado de los LED lo controlaremos mediante el microcontrolador a través de los pines 23 y 24, ambos con una intensidad de salida máxima de 15mA. En serie a cada uno de los LED

añadiremos una resistencia para limitar la intensidad.

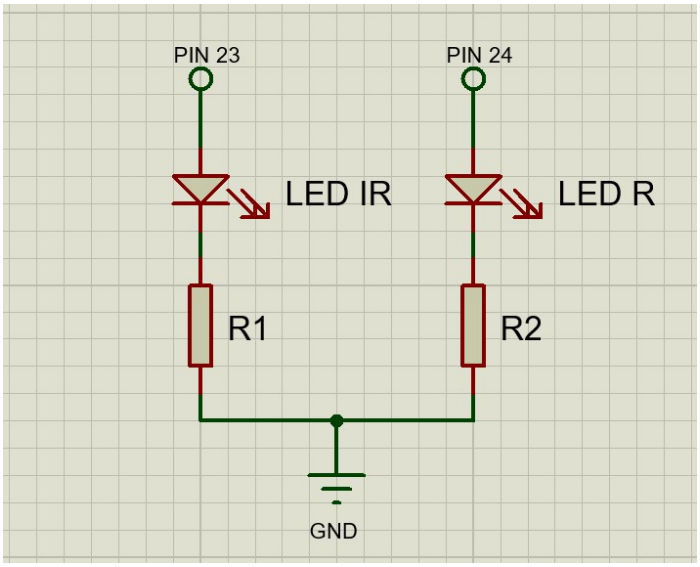


Ilustración 6. Circuito inicial de los LED.

El valor mínimo de esas resistencias será:

$$R_{mín} = \frac{3.3V - V_{LED}}{15 \cdot 10^{-3} A}$$

El valor máximo de la intensidad limita la luminosidad de los LED. Se busca maximizar la luminosidad optimizando la intensidad que circula por los LED. Para ello los dispararemos de forma indirecta a mediante transistores NPN como se indica en el siguiente esquema:

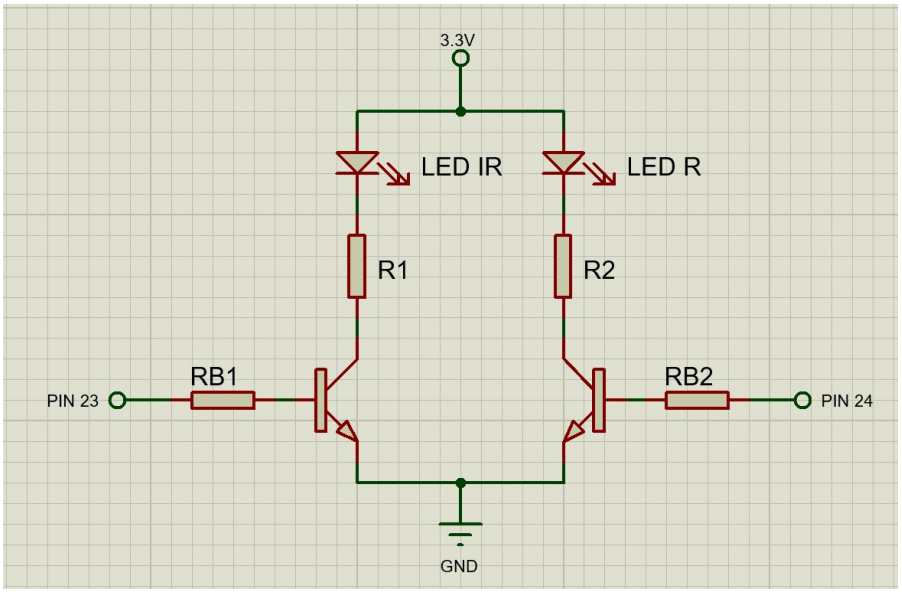


Ilustración 7. Circuito final de los LED.

El valor mínimo de R1 y de R2 viene ahora determinada por la intensidad máxima soportada por los LED. Se ajustará el valor de forma que la emisión de luz sea adecuada al rango de tensiones que permite la lectura (entre 0 y 3.3V) para el modelo concreto de LED que se vaya a emplear.

$$R_{mín} = \frac{3.3V - V_{LED} - V_{CE(sat)}}{I_{LED\ máx}}$$

El valor mínimo de las resistencias de disparo RB1 y RB2:

$$R_{B\ mín} = \frac{3.3V - V_{BE(on)}}{15 \cdot 10^{-3} A}$$

Puesto que no es necesario maximizar la corriente de base para disparar el transistor, se ha optado por elegir unas resistencias de base elevadas con el fin de minimizar el consumo del dispositivo. Se ha elegido un valor de 10kΩ para ambos casos.

Para el fotodiodo, al no generar una caída de tensión en sus extremos, dispondremos una resistencia en serie para poder medir la tensión producida por la corriente generada. Mediremos esa tensión mediante el pin A0 del Arduino.

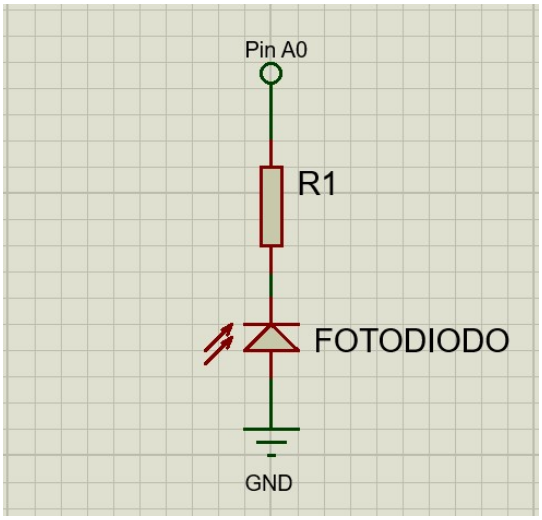


Ilustración 8. Circuito inicial del fotodiodo.

El resultado de la medición mediante este método es impreciso. El método óptimo para adecuar la señal de un fotodiodo es mediante un amplificador de transimpedancia. De forma práctica se probaron diferentes configuraciones para un amplificador TLC271. La configuración con la que se obtuvo una medición correcta y limpia fue la siguiente:

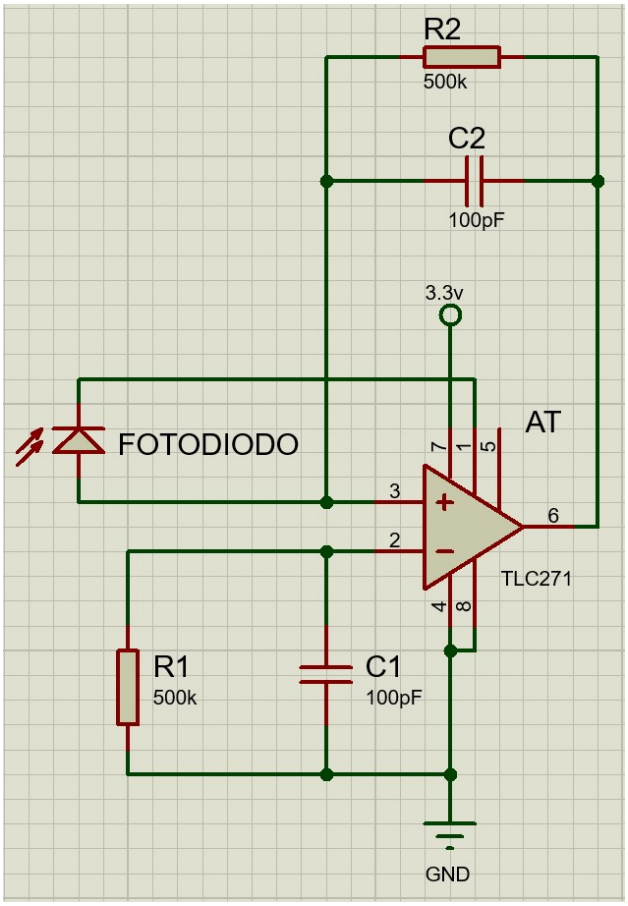


Ilustración 9. Fotodiodo y amplificador de transimpedancia.

Dispondremos el fotodiodo polarizado inversamente entre la patilla 1 de *offset* y la entrada positiva del amplificador.

La ganancia de tensión del amplificador responde a la expresión:

$$A_V = \frac{V_o}{V_i} = \frac{R2}{R1}$$

El amplificador satura a una tensión aproximada de 2.6V. Puesto que el rango de tensión es pequeño, se cambiará la resolución de la lectura analógica del Arduino mediante el comando *AnalogResolution(12).* Con esto reservamos 12 bits para la lectura de tensión Analógica. A diferencia de otros modelos de Arduino, la referencia de tensión para las mediciones analógicas es de 3.3V, lo que nos otorga una resolución de 0.8mV.

Para el desarrollo del sensor primero se diseñó un prototipo mediante

una pinza de ropa de plástico a la cual se la efectuaron dos oquedades en ambas partes. Se introdujeron en dichas aberturas dos LED, uno rojo y otro infrarrojo, un L934LSRD y un TSAL4400 respectivamente, y un fotodiodo LL304PDC2E en el extremo opuesto. Como resistencias en serie a los LED se eligieron resistencias de 50Ω.

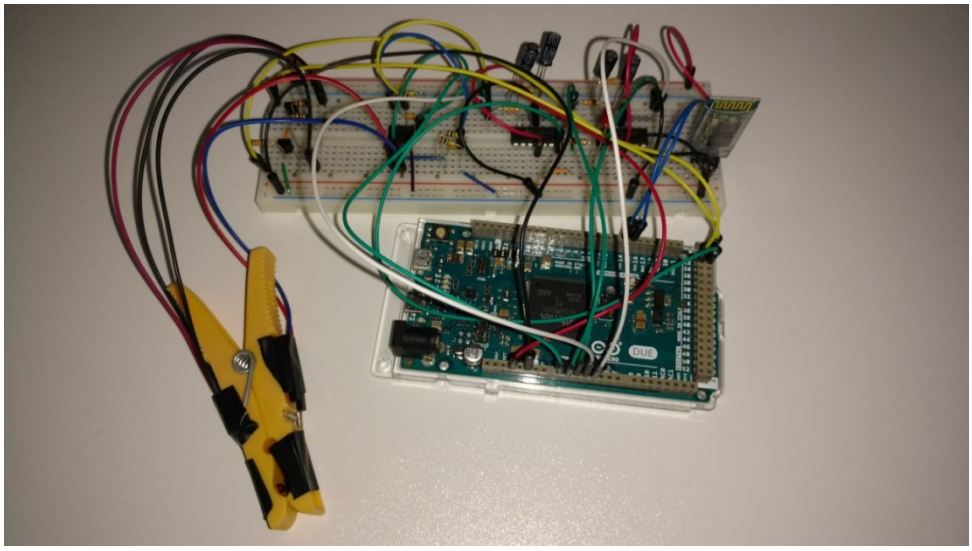


Ilustración 10. Primer prototipo para el sensor empleando una pinza de ropa.

Si bien el resultado obtenido al tomar las mediciones en la fascia entre el dedo índice y pulgar, el sensor de pinza y los componentes elegidos fueron insuficientes para la lectura correcta en los dedos, siendo la luz emitida demasiado débil o el sensor incapaz de detectarla. Además, la presión de la pinza resulta molesta y dificulta el movimiento.

Se propone como objetivo que el sensor sea lo más cómodo y menos incapacitante posible. Los oxímetros comerciales o empleados

Se diseñó un segundo prototipo empleando componentes SMD. Se sustituyó la pinza por una tira de velcro ajustable y unas plantillas de goma eva perforadas, donde se introducen los componentes y aíslan el cableado como se muestra en las siguientes fotografías. La tira de velcro, además de ajustarse al dedo, aísla al sensor de luz ambiental, resultando una medición más adecuada.

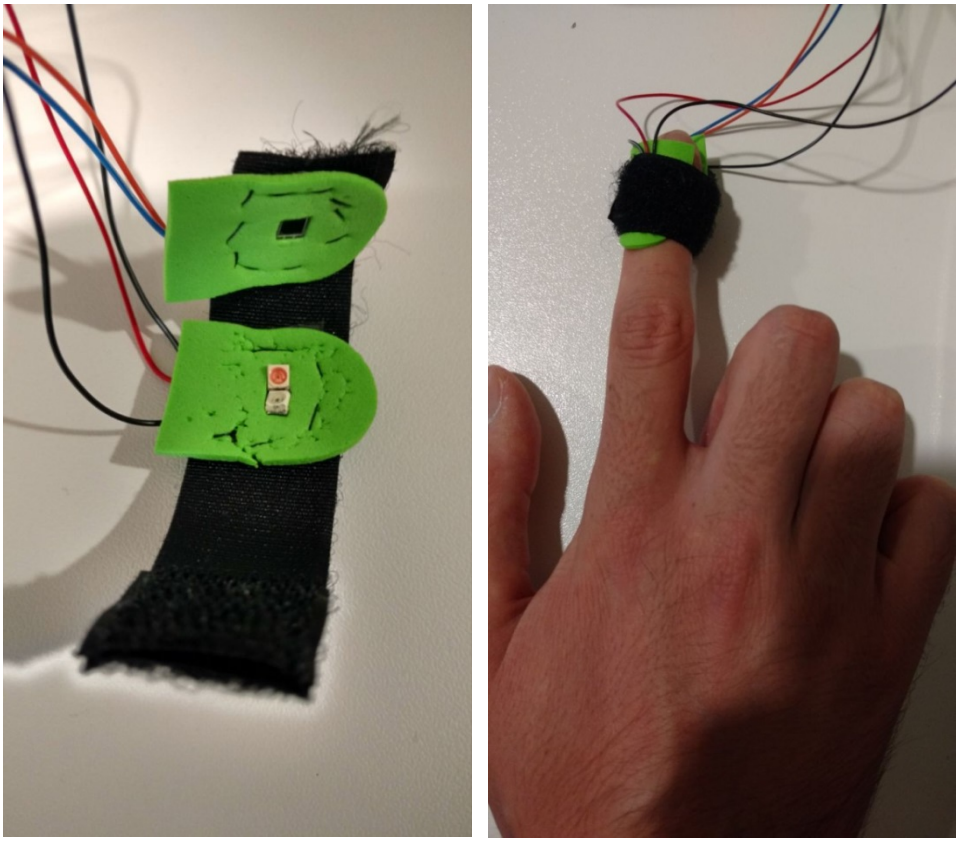


Ilustración 11. Prototipo mejorado del sensor.

En este caso se emplearon un LED rojo KA-3528SRT, un LED infrarrojo VSMB3940X01 y un fotodiodo BPW 34 S. Para estos componentes:

$$I_{R\ DC\ máx} = 30mA \rightarrow R_{R\ mín} = \frac{3.3V - V_{LED} - V_{CE(sat)}}{I_{R\ DC\ máx}} = \frac{3.3V - 1.85V - 0.09V}{0.03A}$$

$$= 45.33\Omega$$

$$I_{R\ pico\ máx} = 155mA \rightarrow R_{R\ mín} = \frac{3.3V - V_{LED} - V_{CE(sat)}}{I_{R\ pico\ máx}}$$

$$= \frac{3.3V - 1.85V - 0.09V}{0.155A} = 8.77\Omega$$

$$I_{IR\ DC\ máx} = 100mA \rightarrow R_{IR\ mín} = \frac{3.3V - V_{LED} - V_{CE(sat)}}{I_{R\ DC\ máx}}$$
$$= \frac{3.3V - 1.15V - 0.09V}{0.1A} = 20.6\Omega$$

$$I_{IR\ pico\ máx} = 200mA \rightarrow R_{IR\ mín} = \frac{3.3V - V_{LED} - V_{CE(sat)}}{I_{R\ pico\ máx}}$$
$$= \frac{3.3V - 1.15V - 0.09V}{0.2A} = 10.3\Omega$$

El valor de la intensidad de pico del LED infrarrojo se ha tomado con un ciclo de servicio de 0.5 y un pulso de 100µs, mientras que para el LED rojo el valor de la intensidad de pico se ha tomado para un ciclo de servicio de 1/10 con un tiempo de pulso de 0.1ms.

3.3. Filtros

Con el objetivo de adecuar la señal de salida del sensor y eliminar posibles ruidos, se pretende diseñar un filtro de paso banda para eliminar toda frecuencia por encima y por debajo de la señal a estudiar.

Como se ha comentado anteriormente, el pulso o frecuencia cardíaca varía a lo sumo entre 39 pulsaciones por minuto y 220 pulsaciones por minuto, por tanto, toda frecuencia por encima o por debajo no es objeto de medida del dispositivo. Ambas frecuencias se corresponden en hertzios a 0.65 y 3.67 respectivamente.

Se ha diseñado un filtro de cuatro etapas de tipo Sallen-Key, dos de paso bajo y dos de paso alto. Como frecuencia de corte para los filtros de paso bajo se tomará una frecuencia de 5Hz, y para la frecuencia de corte en los filtros de paso alto una frecuencia de 0.5Hz.

- **Filtro de paso bajo:**

  Se trata de un filtro de dos etapas de tipo Sallen-Key que permite el paso de frecuencias por debajo de la frecuencia de corte f_c. La

configuración de paso bajo Sallen-Key es la que sigue:

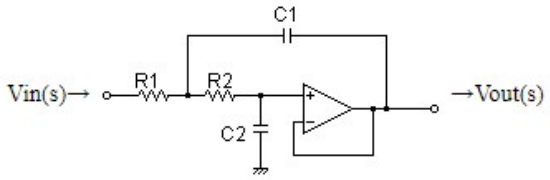


Ilustración 12. Configuración de paso bajo Sallen-Key.

La frecuencia de corte para este filtro, así como el factor de calidad Q vienen determinadas por la siguiente expresión:

$$f_c = \frac{1}{2\pi\sqrt{R_1 R_2 C_1 C_2}}$$

$$Q = \frac{R_2 \frac{C_1 C_2}{C_1 + C_2}}{\sqrt{R_1 R_2 C_1 C_2}}$$

Para una frecuencia de corte de 5Hz y un factor de calidad Q=0.5 tenemos así los siguientes resultados obtenidos de forma simulada:

R_1=33kΩ

R_2=33kΩ

C_1=1μF

C_2=1μF

$$G(s) = \frac{918.27364554637}{s^2 + 60.606060606061s + 918.27364554637}$$

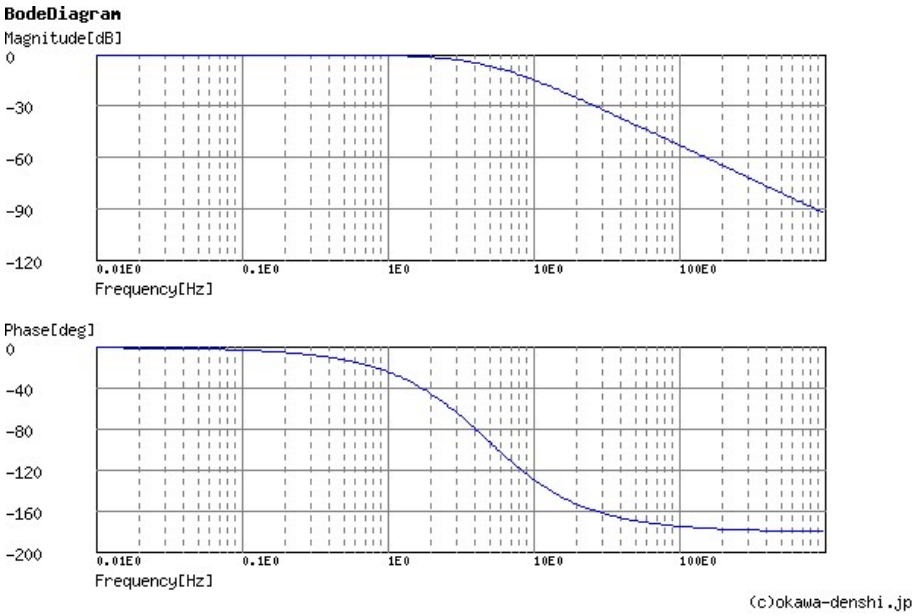


Ilustración 13. Diagrama de Bode para el filtro paso bajo.

- **Filtro de paso alto:**

  El filtro de paso alto permitirá el paso de las frecuencias por encima de la frecuencia de corte. La expresión para el cálculo de la frecuencia de corte y el factor de calidad son iguales, pero la configuración del filtro en cambio es diferente:

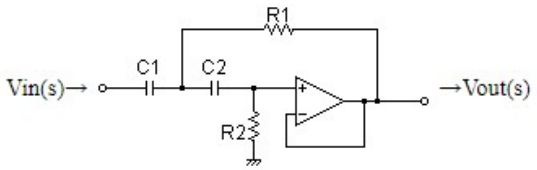


Ilustración 14. Configuración de paso alto Sallen-Key.

Para una frecuencia de corte de 0.5Hz y un factor de calidad 0.5, los resultados y la simulación obtenida son los siguientes:

R_1=33kΩ

R_2=33kΩ

C_1=10μF

C_2=10μF

$$G(s)=\frac{s^2}{s^2+6.0606060606061s+9.1827364554637}$$

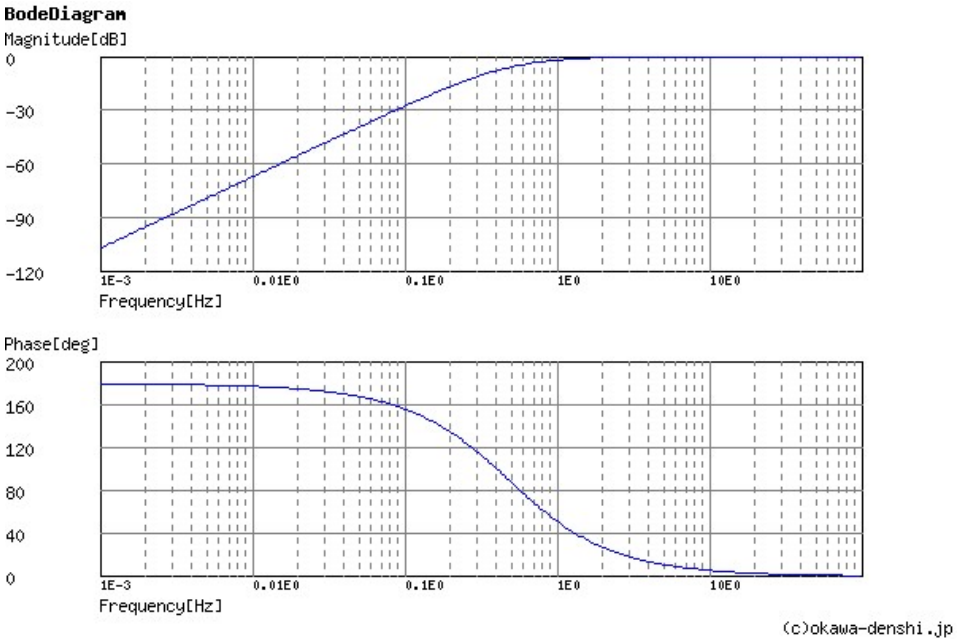


Ilustración 15. Diagrama de Bode para el filtro de paso alto.

Para el diseño de los filtros emplearemos un circuito integrado LM324 que dispone de cuatro módulos de amplificadores operacionales de baja tensión. Las cuatro etapas de filtros quedan de la siguiente forma:

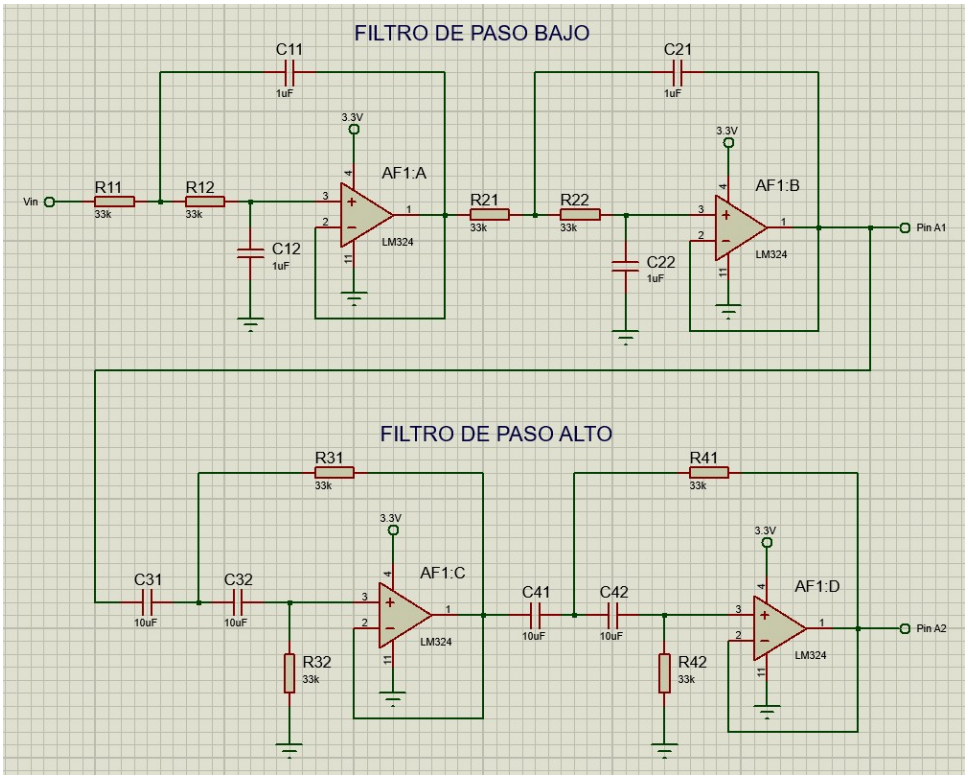


Ilustración 16. Filtro paso banda en cuatro etapas.

A continuación se van a realizar diferentes pruebas para estudiar el comportamiento de los filtros. Se tomarán medidas a la salida del amplificador de transimpedancia, a la salida de la etapa de paso bajo y finalmente a la salida de las cuatro etapas de filtro tras la etapa de paso alto.

Se harán pruebas del funcionamiento de los LED tanto para emisión continua de uno de ellos como para emisión en pulsos de forma individual y de forma alterna. La secuencia de apagado y encendido de los LED será en períodos alternos de 1ms y con un ciclo de servicio de 0.1ms como se indica en la figura. El ciclo de servicio podrá modificarse para aumentar o disminuir la luminosidad en caso de que lo necesitemos.

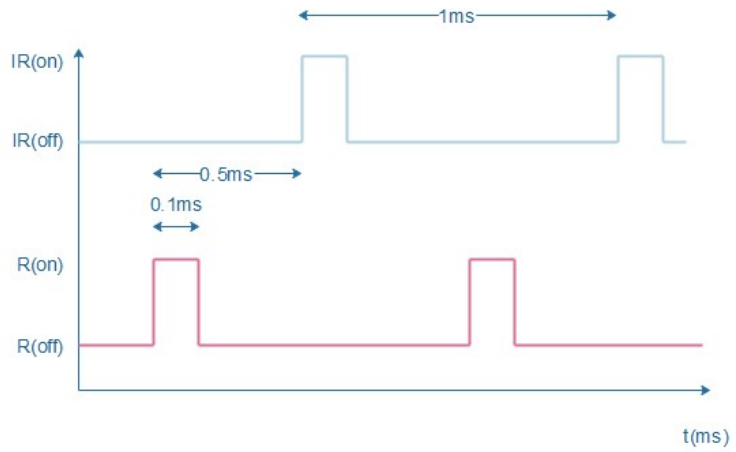


Ilustración 17. Secuencia de encendido de los LED.

- **Emisión continua.**

  Para esta prueba comprobamos el funcionamiento de los filtros empleando un único LED (en este caso el LED infrarrojo) en modo de emisión continua.

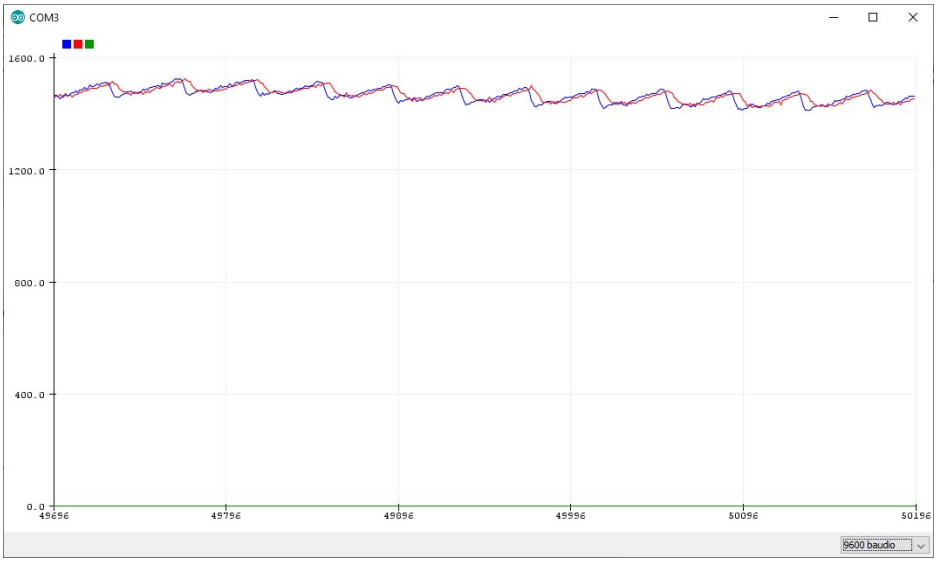


Ilustración 18. Señal en modo de emisión continua antes y después de los filtros.

La señal azul muestra la salida de tensión tras el amplificador de transimpedancia, la señal roja tras los dos filtros de paso bajo y la señal verde tras los dos filtros de paso alto. Como vemos, tras la

salida de la primera etapa de paso bajo del filtro, la señal se suaviza pero mantiene su forma característica, aunque es posible que al suavizarse se pierda información como la diferencia exacta entre máximos y mínimos. La salida tras el filtro de paso alto es nula.

Eliminamos uno de los filtros de paso bajo para ver el comportamiento de una sola etapa del mismo.

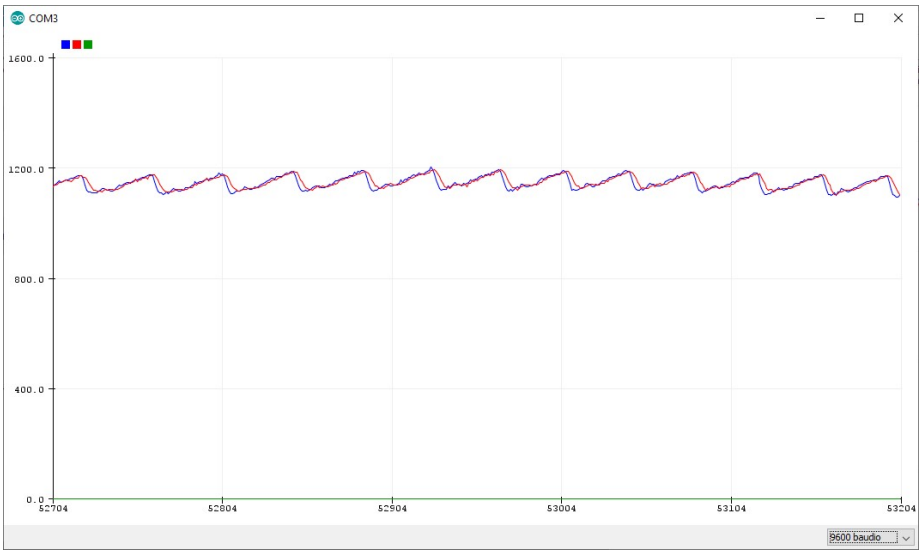


Ilustración 19. Señal en modo de emisión continua antes y después de los filtros, habiendo eliminado una etapa del filtro de paso bajo.

La señal tras el filtro de paso bajo se suaviza menos que en el caso anterior, semejándose así más a la señal original.

- **Emisión pulsante 1 LED.**

  Para este caso emitiremos mediante el LED infrarrojo en pulsos de 0.1 ms y un período de 1ms como el descrito anteriormente. Realizaremos las lecturas tan solo en los pulsos para no ensuciar la representación de la señal ya que la información de tensión durante el resto del período no nos proporciona ningún tipo de información.

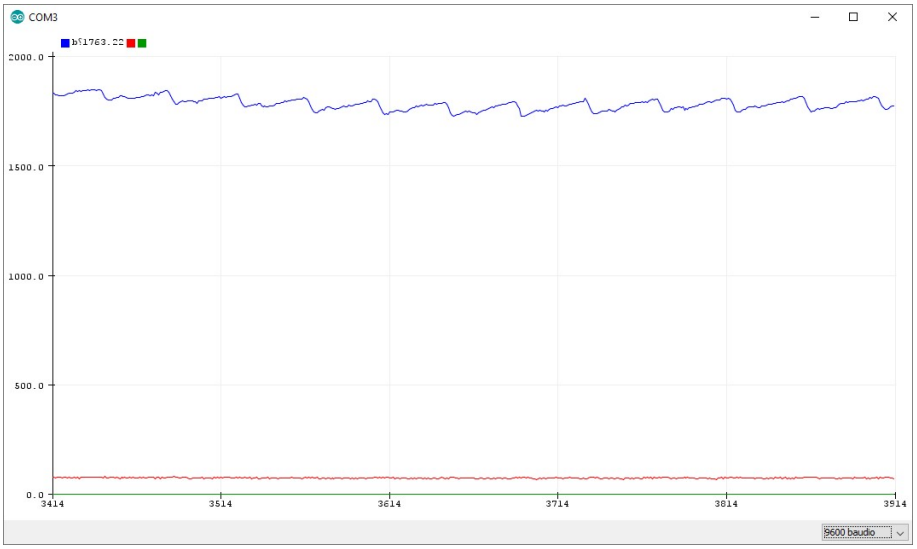


Ilustración 20. Señal en modo de emisión pulsante mediante un solo LED antes y después de los filtros.

En azul vemos la señal tras el amplificador de transimpedancia, en rojo tras la etapa de paso bajo y la verde tras la etapa de paso alto. La señal tras los filtros cae con respecto al modo de emisión continua.

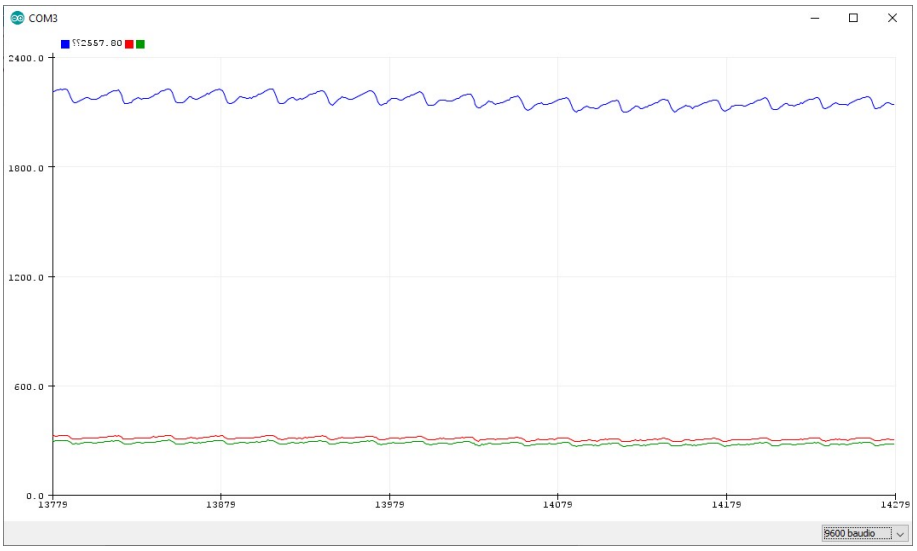


Ilustración 21. Señal en modo de emisión pulsante mediante un solo LED antes y después de los filtros habiendo eliminado una etapa de paso bajo.

Si eliminamos una de las etapas de paso bajo vemos como, en este caso, la señal tras el filtro de paso bajo presenta la forma de pulsos de la original, aunque tras una bajada de tensión considerable y

estar más atenuada. La señal tras la etapa de paso alto es prácticamente idéntica a la anterior en este caso, ligeramente menor en cuanto a tensión, aunque apenas apreciable.

- **Emisión pulsante 2 LED.**

  En este caso alternaremos los LED rojo e infrarrojo como se ha descrito anteriormente y con un tiempo de servicio de 0.2ms por cada LED. De igual forma tomaremos medidas únicamente durante los pulsos y las representaremos gráficamente simultáneamente tras cada ciclo global.

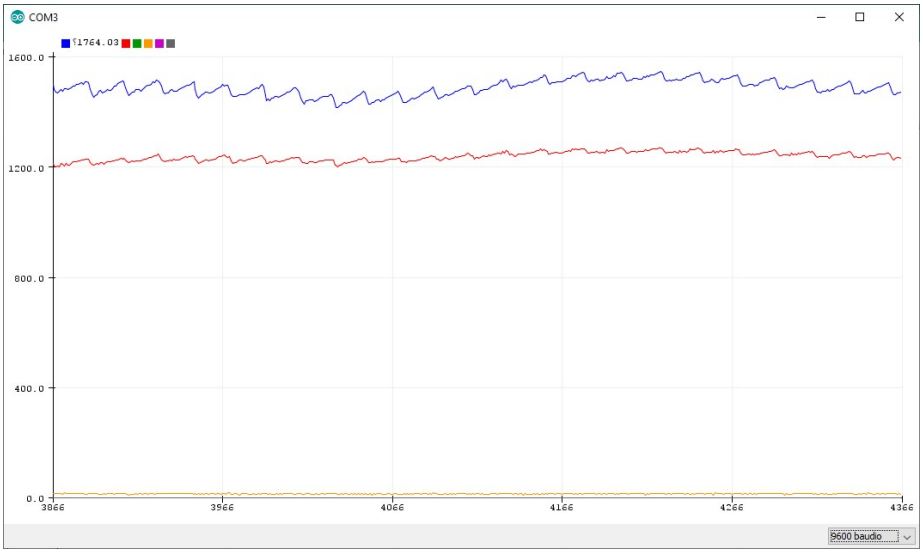


Ilustración 22. Señal en modo de emisión pulsante de los dos LED antes y después de los filtros.

En la imagen observamos las dos señales infrarroja y roja, señal azul y roja respectivamente. Por lo general, la señal roja es menor y menos pulsante que la infrarroja. Las señales verde y amarillo corresponden a las señales infrarroja y roja respectivamente tras la etapa de paso bajo. Finalmente las señales morada y se gris corresponden a las señales infrarroja y roja respectivamente tras los filtros de paso alto.

Como vemos las señales caen a valores de tensión prácticamente nulos.

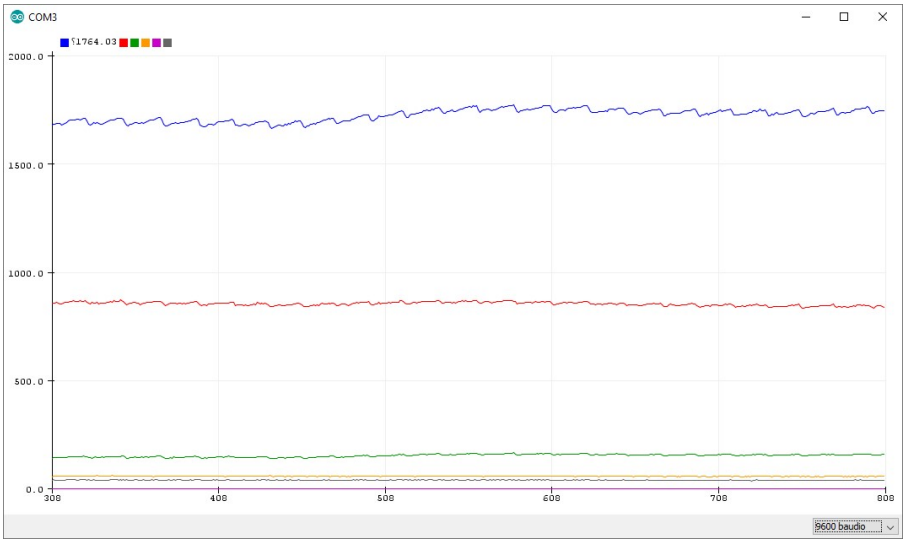


Ilustración 23. Señal en modo de emisión pulsante de los dos LED antes y después de los filtros habiendo eliminado una etapa de paso bajo.

Eliminando una de las etapas de paso bajo como en casos anteriores observamos una ligera mejora en las señales tras los filtros, sobre todo en la salida infrarroja en la que vemos su forma pulsante tras la primera etapa, pero a pesar de ello resultando una señal insuficiente.

El objetivo de un filtro es adecuar una señal librándola de ruido para una lectura óptima con la mayor precisión posible. En nuestro caso disponemos de un filtro aplicado a un solo fotodiodo, a pesar de que medimos dos señales que se intercalan. El resultado es un incorrecto uso del filtro, que filtrará ambas señales sin discriminarlas como si fuera una única señal.

Podría pensarse en emplear dos fotodiodos y duplicar los filtros, uno para cada uno de los fotodiodos. Sin embargo no hay que olvidar que los dos fotodiodos recibirían las mismas señales lumínicas y, por tanto, responderán de la misma forma y la señal será la misma independientemente de haber usado dos filtros.

Una posible solución es emplear optoacopladores para conectar de forma alterna las dos señales provenientes del fotodiodo con los dos filtros,

uno para cada señal de LED, y así filtrar las señales de forma independiente.

Sin embargo, en vista de los resultados, la señal sin filtrar en cualquiera de los casos es suficientemente aceptable para tomar mediciones a partir de la misma sin necesidad de filtrarla. Además, la señal filtrada se atenúa demasiado, perdiéndose información y precisión a la hora de medir los puntos máximos y mínimos. Se intentará filtrar de forma digital la señal a través del código del microcontrolador como se explicará en los correspondientes apartados más adelante.

Se debe añadir que para la realización de estas mediciones se ha empleado una velocidad de transmisión de 9600 baudios. Esto limita la velocidad de emisión y luminosidad de los LED, aunque no altera las conclusiones tomadas del ensayo.

3.4. Módulo bluetooth

Para la comunicación bluetooth se ha elegido un módulo HC-06, alimentado a 3.3V. El módulo se conecta directamente al puerto serie del microcontrolador. En nuestro caso hemos empleado el serial 3, correspondiente a los pines TXD2 (Pin PA11) y RXD2 (Pin PA10).

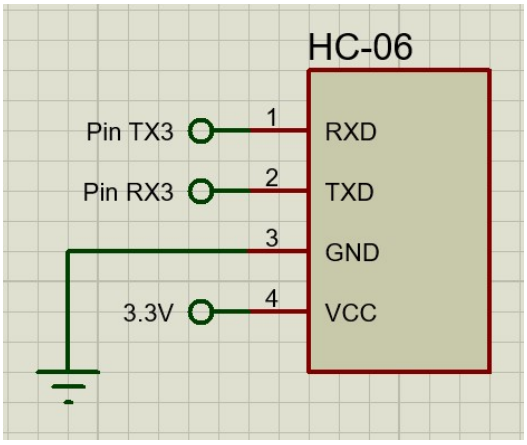


Ilustración 24. Conexión del módulo bluetooth HC-06.

El módulo ha de ser configurado a través de los comandos AT de forma que se ajuste a la velocidad de transmisión del puerto serie elegido. Para ello,

conectamos el módulo como se ha indicado y, mediante un programa que permita el envío de mensajes al correspondiente puerto serie, escribiremos el comando AT+UART=n, siendo n la velocidad en baudios a la que queremos transmitir.

Además, para permitir reconocer el dispositivo cuando nos conectemos a él, se ha cambiado el nombre del módulo a OXIMETRO mediante AT+NAME:nombre.

Si la comunicación es correcta, el módulo enviará el mensaje OK al introducir los comandos.

AT COMMAND LISTING

	COMMAND	FUNCTION
1	AT	Test UART Connection
2	AT+RESET	Reset Device
3	AT+VERSION	Querry firmware version
4	AT+ORGL	Restore settings to Factory Defaults
5	AT+ADDR	Query Device Bluetooth Address
6	AT+NAME	Query/Set Device Name
7	AT+RNAME	Query Remote Bluetooth Device's Name
8	AT+ROLE	Query/Set Device Role
9	AT+CLASS	Query/Set Class of Device CoD
10	AT+IAC	Query/Set Inquire Access Code
11	AT+INQM	Query/Set Inquire Access Mode
12	AT+PSWD	Query/Set Pairing Passkey
13	AT+UART	Query/Set UART parameter
14	AT+CMODE	Query/Set Connection Mode
15	AT+BIND	Query/Set Binding Bluetooth Address
16	AT+POLAR	Query/Set LED Output Polarity
17	AT+PIO	Set/Reset a User I/O pin
18	AT+MPIO	Set/Reset multiple User I/O pin
19	AT+MPIO?	Query User I/O pin
20	AT+IPSCAN	Query/Set Scanning Parameters
21	AT+SNIFF	Query/Set SNIFF Energy Savings Parameters
22	AT+SENM	Query/Set Security & Encryption Modes
23	AT+RMSAD	Delete Authenticated Device from List
24	AT+FSAD	Find Device from Authenticated Device List
25	AT+ADCN	Query Total Number of Device from Authenticated Device List
26	AT+MRAD	Query Most Recently Used Authenticated Device
27	AT+STATE	Query Current Status of the Device
28	AT+INIT	Initialize SPP Profile
29	AT+INQ	Query Nearby Discoverable Devices
30	AT+INQC	Cancel Search for Discoverable Devices
31	AT+PAIR	Device Pairing
32	AT+LINK	Connect to a Remote Device
33	AT+DISC	Disconnect from a Remote Device
34	AT+ENSNIFF	Enter Energy Saving mode
35	AT+EXSNIFF	Exit Energy Saving mode

Tabla 8. Tabla de comandos AT para el módulo HC-06.

3.5. Fuente de alimentación

Como hemos comentado, todo el circuito se alimentará con una tensión de 3.3V. El dispositivo ha de ser autónomo y por tanto alimentado mediante una batería interna. Por ser una tensión pequeña se puede

alimentar mediante una pila alcalina de botón de 1.5V con el fin de reducir el espacio.

Se ha escogido un convertidor de tensión LTC3525-3.3 para garantizar una tensión estable de 3.3V. En la documentación, adjunta en los anexos, se sugiere unas configuraciones recomendadas, entre ellas para una pila alcalina de 1.5V.

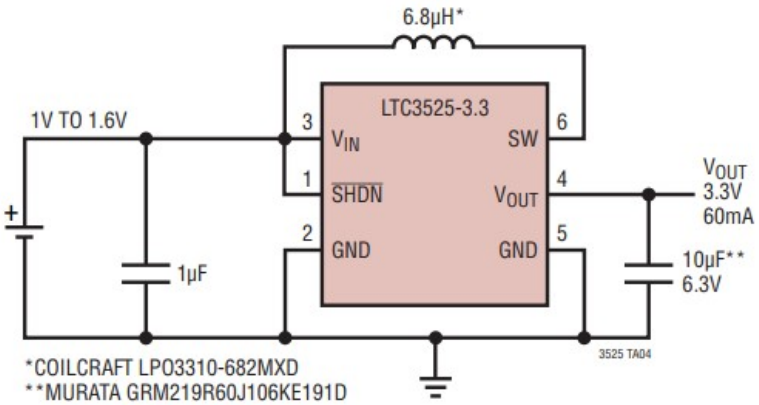


Ilustración 25. Esquema de la fuente de alimentación.

3.6. Conexión USB

Aunque el Arduino DUE puede programarse a través del puerto de programación, programar el microcontrolador de esta forma supone añadir un microcontrolador ATmega16U2 adicional al circuito y programarlo. El SAM3X8E cuenta con varias formas de programarlo, entre las que se encuentran la conexión nativa por el puerto UOTGID a través de USB, la programación mediante JTAG y finalmente por SWD.

El SAM3X8E viene con un programa bootloader de fábrica llamado SAM-BA instalado en la ROM en lugar de la memoria flash como en el caso de los microcontroladores AVR, lo que permite programarlo desde el puerto nativo UOTGID sin necesidad de instalar nosotros el bootloader. El bootloader se cargará al pulsar el botón de borrado *erase* (pin PC0).

La conexión recomendada por el fabricante para el caso en que el

microcontrolador sea alimentado de forma autónoma como es nuestro caso, es la siguiente:

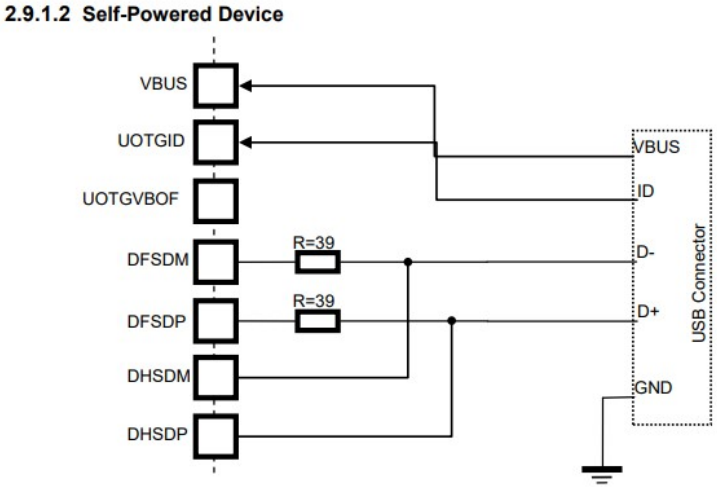


Ilustración 26. Conexión USB recomendada.

Por el número de terminales sugeridos por el esquema se ha de usar un USB de tipo micro o mini. En nuestro caso usaremos un micro USB.

Además de estas conexiones, se debe acoplar un filtro RC al pin VBG como se indica en la imagen:

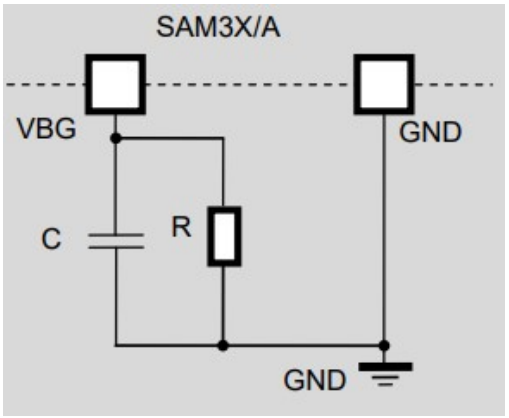


Ilustración 27. Filtro RC en el pin VBG.

3.7. Conexión JTAG

A pesar de que el objetivo es programar el microcontrolador a través del puerto nativo USB mediante el Arduino IDE, la implementación de un terminal de depuración (debug) JTAG es recomendada. Además de la conexión al terminal, se deben conectar resistencias de 100kΩ de pull-up en los pines TCK (PB28), TMS (PB31) y TDI (PB29) además de a los respectivos terminales del conector JTAG. El pin TDO (PB30) se conecta directamente con el terminal.

Para el terminal RESET del terminal se conectará un botón a tierra paray una resistencia en serie para limitar la corriente.

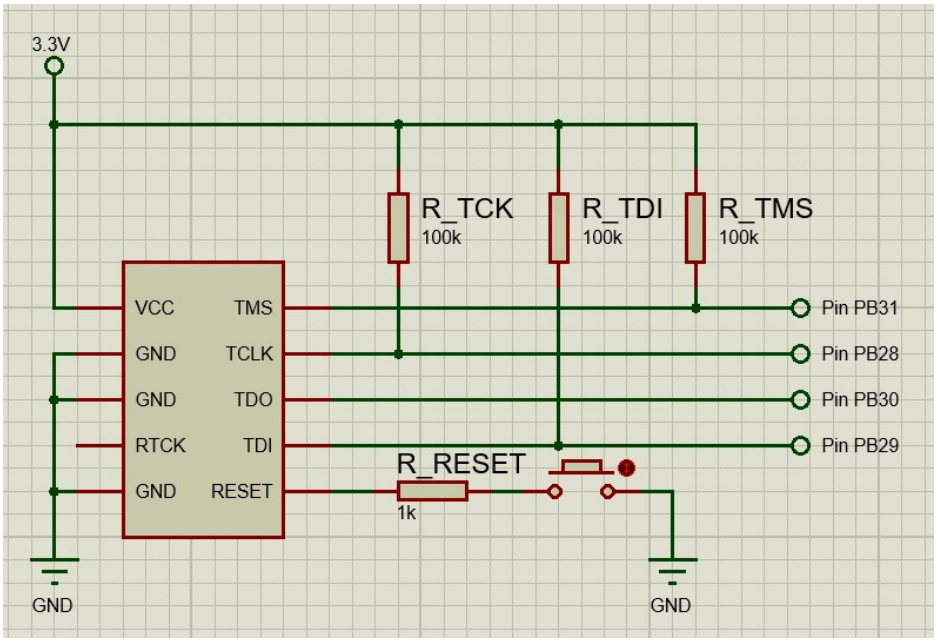


Ilustración 28. Conexión del terminal JTAG.

El pin JTAGSEL debe conectarse a tierra puesto que no le emplearemos.

3.8. Reloj y osciladores

El SAM3X8E dispone de dos pares de pines dedicados para los dos

relojes internos del oscilador, los pines XIN y XOUT para el reloj principal y XIN32 y XOUT32 para el reloj de baja frecuencia.

- XIN y XOUT

  El microcontrolador está diseñado para trabajar con cristales de entre 3 y 20MHz. Según el fabricante, el bootloader SAM-BA soporta comunicación serial a través de UART y USB cuando trabajamos con cristales de 12MHz.

  La conexión del cristal queda como sigue:

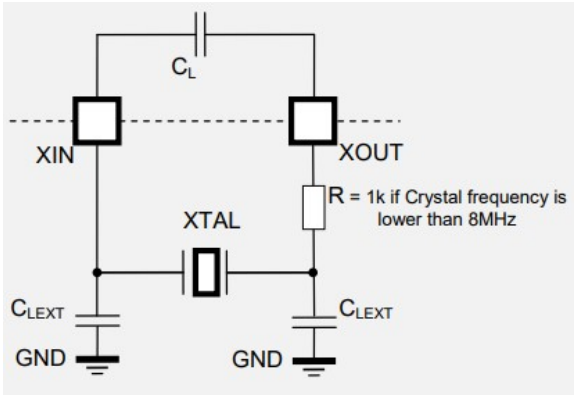


Ilustración 29. Conexión del oscilador principal.

Teniendo en cuenta que:

$$C_{LEXT} = 2(C_{Cristal} - C_L - C_{PCB})$$

Donde C_{PCB} es la capacidad de la placa de circuito impreso desde el pin del cristal hasta el pin del microcontrolador y C_L es la capacidad de carga interna que equivale a 9.5pF.

- XIN32 y XOUT32

  Para el oscilador de baja frecuencia se ha de emplear un cristal de 32.768kHz, siendo la configuración del mismo análoga a la anterior:

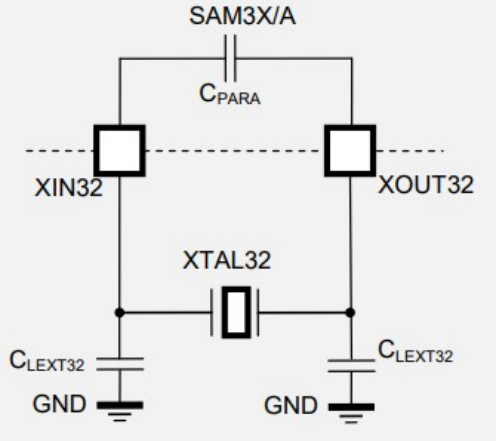


Ilustración 30. Conexión del oscilador de baja frecuencia.

$$C_{LEXT32} = 2(C_{Cristal} - C_{PCB} - C_{Parasita})$$

Donde la capacidad parásita equivalente es de 1.4pF.

3.9. Alimentación del microcontrolador

El SAM3X8E dispone de varios pines de alimentación que deben conectarse según el siguiente esquema y las indicaciones del manual.

☑	Signal Name	Recommended Pin Connection	Description
	VDD_{IN}	1.8V to 3.6V Voltage range Decoupling/Filtering Capacitor (CD_{IN}) (10µF or higher ceramic capacitor) [1,2]	Powers the internal voltage regulator
	VDD_{IO}	Voltage ranges from 1.62V to 3.6V Decoupling/Filtering Capacitors (4.7µF on the main branch and 100nF on each pin as described in the figure) [1,2]	Powers the Peripherals I/O lines
	VDD_{ANA}	Voltage ranges from 2.0V to 3.6V for 10-bits resolution **Warning:** Voltage ranges from 2.4V to 3.6V for 12-bits resolution. Decoupling/Filtering Capacitors (100nF) [1,2]	Powers the ADC and DAC cells
	VDD_{UTMI}	Voltage ranges from 3.0V to 3.6V, 3.3V nominal Decoupling/Filtering Capacitors (100nF) [1,2] Additional RLC circuit (R = 1; L = 10µH; C = 4.7µF) [1,3]	Powers the UTMI+ interface Supply ripple must not exceed 10mV
	VDD_{BU}	Voltage ranges from 1.62V to 3.6V Decoupling/Filtering Capacitors (100nF) [1,2]	Powers the Slow Clock oscillator and a part of the System Controller
	VDD_{OUT}	Decoupling/Filtering Capacitor (CD_{OUT}) (4.7µF or higher ceramic capacitor) [1,2]	It is the output of the voltage regulator. Decoupling/Filtering capacitors must be added to Guarantee stability.
	VDD_{CORE}	Voltage ranges from 1.62V to 1.95V Decoupling/Filtering Capacitors (100nF on each pin) [1,2]	Power the core, the embedded memories and the peripherals
	VDD_{PLL}	Voltage ranges from 1.62V to 1.95V Decoupling/Filtering Capacitors (100nF) [1,2] Additional RLC circuit (R = 1; L = 10µH; C = 4.7µF) [1,3]	Powers the PLL A, UPLL, and 3-20MHz oscillator. Supply ripple must not exceed 10mV.
	GND	Grounded	Ground pins GND are common to VDD_{IN}, VDD_{IO}, and VDD_{CORE}
	GND_{BU}	Grounded	Dedicated VDD_{BU} ground pin
	GND_{PLL}	Grounded	Dedicated VDD_{PLL} ground pin
	GND_{UTMI}	Grounded	Dedicated VDD_{UTMI} ground pin
	GND_{ANA}	Grounded	Dedicated VDD_{ANA} ground pin

Ilustración 31. Instrucciones de conexión de los pines de alimentación del microcontrolador.

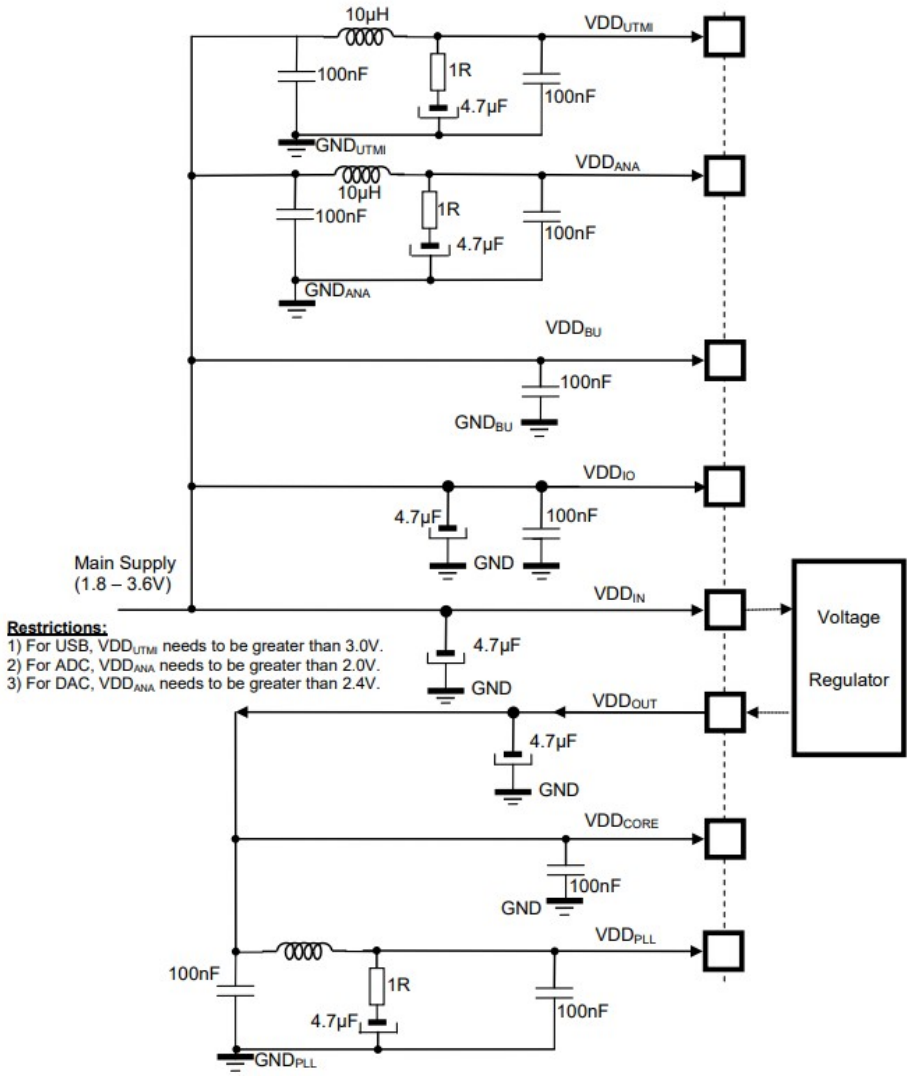


Ilustración 32. Esquema de conexión de los pines de alimentación del microcontrolador.

Las conexiones de los condensadores se han de realizar lo más próximo posible a sus respectivos pines como se especifica en la documentación.

3.10. Otras conexiones

Además de las descritas, se deben realizar las siguientes conexiones al microcontrolador:

- ADC

Para permitir el funcionamiento del convertidor ADC con una resolución de 12 bits, se debe conectar el pin ADVREF a V_{VDDANA}, siendo esta superior a 2.4V.

- **Erase**

  Para poder cargar el bootloader como se ha indicado anteriormente, se debe añadir un botón de *erase* para el borrado de la memoria flash y que indicará al bootloader que debe cargarse. Conectaremos a la alimentación un pulsador al pin de *erase* (PC0) en serie a una resistencia para regular la intensidad.

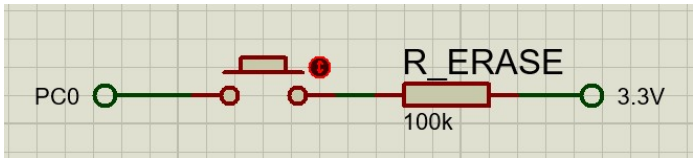


Ilustración 33. Conexión del botón de erase.

- **Reset/Test**

  La documentación del microcontrolador sugiere conectar los pines NRST, NRSTB y TST como se indica a continuación:

  - Dejaremos NRST al aire, puesto que permitiremos el reset mediante NRSTB, que además incluye reset de la región de backup.
  - NRSTB se debe conectar a una capacidad de 10nF de pull-up y al botón de reset que añadimos en el terminal JTAG.
  - TST será conectado a tierra.

- **Shutdown/Wakeup logic**

  El pin SHDN se mantendrá desconectado, mientras que el pin FWUP se conectará mediante una resistencia de pull-up al pin VDDBU.

3.11. Diseño de la placa de circuito impreso

Se ha diseñado el circuito para su fabricación en una placa de circuito impreso. Debido a la naturaleza portable del dispositivo se han elegido componentes con las características dimensionales más reducidas posibles. El diseño puede variar ligeramente si se eligen componentes con diferentes *footprints* a los propuestos.

El dispositivo se alimentará con una pila de botón LR44 de 1.5V como se especificó en el apartado de la fuente de alimentación y el regulador de tensión. Se han distribuido las vías de tensión por separado a los distintos componentes para evitar pérdidas entre nodos.

Aunque el tamaño de la placa es reducido, éste está limitado por el tamaño de los componentes empleados: el módulo bluetooth, el tamaño del microcontrolador empleado o la existencia de componentes dedicados para la programación del dispositivo impiden reducir aún más su tamaño.

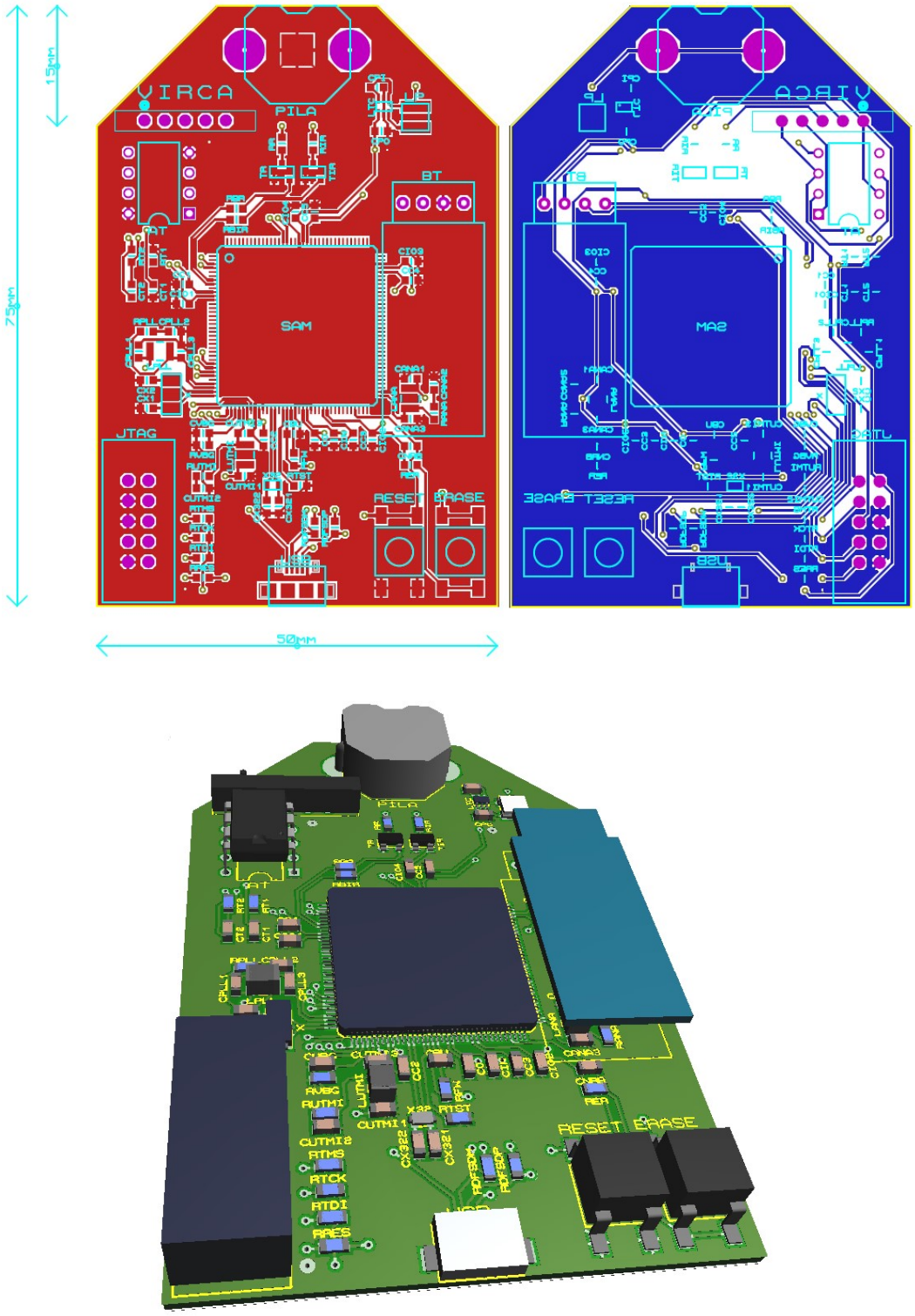


Ilustración 34. Diseño de la placa de circuito impreso. Vista superior (superior izda.), vista inferior (superior dcha.) y modelo 3D de la placa (abajo).

4. Programa del microcontrolador

4.1. Código del programa

El programa se ha desarrollado en Arduino IDE mediante la plataforma online Arduino Create, que incorpora tecnología en la nube. El lenguaje empleado es el de Arduino, que se basa en el lenguaje C++ e implementa una serie de funciones propias.

Puesto que nos comunicaremos con una aplicación Android y ésta a su vez está escrita en JAVA, emplearemos un formato JSON para el mensaje enviado. Por ello incluiremos la librería ArduinoJson en nuestro programa.

Inicialmente se partió del programa de prueba de LED para el modo de pulsos alternos. En primer lugar se inicializan las variables y constantes globales. Dentro de setup() configuramos los pines que vamos a emplear, así como los puertos serie, su velocidad de transmisión, la resolución del convertidor ADC y el formato del mensaje JSON que vamos a enviar.

```
void setup()
{
  while (!Serial) continue;                    //Esperamos a que se inicialice el puerto serie
  while (!HC06) continue;                      //Esperamos a que se inicialice el puerto serie3 del bluetooth

  delay(1000);
  Serial.begin(B_RATE_LENTO);
  HC06.begin(B_RATE_RAPIDO);

  pinMode(LED_IR, OUTPUT);
  pinMode(LED_R, OUTPUT);

  pinMode(Vo0, INPUT);

  analogReference(AR_DEFAULT);                 //Referencia de AnalogRead a 3.3V
  analogWriteResolution(10);                   //10 bits (0-1023) para valores de tensión de salida
  analogReadResolution(12);                    //12 bits (0-4095) para lectura de tensión

  digitalWrite(LED_IR, LOW);
  digitalWrite(LED_R, LOW);


  //Inicializamos el mensaje con formato json

  datos["ppg"]=0;
  datos["oxigeno"]=0;
  datos["pulso"]=0;
}
```

Ilustración 35. Función setup() del programa del microcontrolador.

La función Pulso_LED() toma como argumento el LED que vamos a encender y lo enciende durante el período indicado, en el cual se toma una lectura analógica de tensión en milivoltios. El período de encendido y apagado

de los LED es de 15ns, mientras que la velocidad de lectura de los Arduinos basados en tecnología ATmega (UNO, Nano, Mini y Mega) es de 100µs. A pesar de que en el caso de Arduino DUE realiza esta función en un tiempo aproximado de 40µs, la duración del pulso se ha aumentado a 200µs en vez de los 100µs de los que se partieron.

La señal de tensión leída para cada LED se guarda en sus respectivas variables, una para la señal infrarroja y otra para la roja.

```
void Pulso_LED(int led)
{
  unsigned long t=micros();
  float lectura;

  digitalWrite(led, HIGH);

  while((micros()-t)<200)
  {
    lectura = Lectura_Vo0();
  }

  digitalWrite(led, LOW);

  if(led == LED_IR)
    V_IR = lectura;
  else
    V_R = lectura;
}
```

Ilustración 36. Función Pulso_LED() del programa del microcontrolador.

El principal motivo de este cambio es la de aumentar la luminosidad de los LED y la de dar un tiempo de medida suficiente, que experimentalmente se comprobó más eficiente. El ajuste entre resistencias en los LED y tiempo de pulso depende de un proceso experimental de gran variabilidad de resultado, puesto que se trata de ajustar un único transductor a una infinita diversidad de dedos y colocaciones del mismo, así como la posibilidad de realizar la medición en ambientes con una contaminación lumínica que puede afectar a la toma de datos, pese a que el diseño del sensor ha tenido como objetivo el aislamiento de LED y fotodiodo máximo posible.

Además, las dos señales de tensión no sólo deben situarse en el rango deseado, entre los 0V y los 2.6V aproximados del valor de saturación del amplificador de transimpedancia, valores ya de por sí limitantes y que requieren la mayor sensibilidad posible, sino que además deben ser

suficientemente claras como para ver en ellas las señales pulsantes y permitir la toma clara de máximos y mínimos, de forma que cuanto mayor sea esa diferencia, mejor.

Para asegurarnos de que siempre nos mantenemos en un rango de luminosidad óptima se añade una función de chequeo que compruebe si estamos dentro de un rango de lecturas entre 400 y 2400mV. La metodología para conseguirlo es realizar pulsos a intervalos de medio segundo en los que se encienden los LED secuencialmente durante 10ms mediante una función similar a la de Pulso_LED() llamada DetectarDedo().

```
float DetectarDedo(int led)
{
  unsigned long t=millis();
  float lectura = 0;
  int i = 1;
  digitalWrite(led, HIGH);
  while ((millis() - t) < 10)
  {
    lectura = (Lectura_Vo0() + (i - 1) * lectura) / i;              //Calculamos el valor medio de la lectura
    i++;
  }
  digitalWrite(led, LOW);
  return(lectura);
}
```

Ilustración 37. Función DetectarDedo() del programa del microcontrolador.

Si la media de medición durante ese tiempo está por debajo de los límites antes descritos, el dispositivo se queda en ese estado intermitente de detección hasta que se detecte una señal en el rango que indique que el sensor va a ser utilizado. Esto se comprueba mediante la función Check(), que devuelve *true* o *false* en función de si estamos en el rango o no.

```
bool Check()
{
  unsigned long t = millis();
  float lectura_R, lectura_IR;

  lectura_R = DetectarDedo(LED_R);
  lectura_IR = DetectarDedo(LED_IR);

  if ((lectura_R < SENSIBILIDAD_ALTA) && (lectura_IR < SENSIBILIDAD_ALTA) && (lectura_R > SENSIBILIDAD_BAJA) && (lectura_IR > SENSIBILIDAD_BAJA))
  {
    return (true);
  }
  else
  {
    return (false);
  }
}
```

Ilustración 38. Función Check() del programa del microcontrolador.

Una vez determinadas las señales infrarroja y roja, se representan gráficamente. En sus gráficas se puede observar la forma característica de las mismas. En primer lugar la onda pulsante crece hasta un máximo relativo

formando una especie de joroba para después subir hasta su máximo. Ahí cae bruscamente hasta el inicio del siguiente pulso.

A pesar de la forma individual de cada onda, la gráfica continuada de los pulsos puede ser variable, si bien por una variación en la posición del dedo, como de la propia absorción de luz por parte de la hemoglobina a cada pulso cardíaco. Por lo general y en condiciones estables, la onda tiene una tendencia a mantenerse, aunque puede presentar tendencias a la alza o a la baja.

Es por tanto que estudiar los máximos y los mínimos absolutos de la señal no tiene sentido, primero porque la señal, desde su inicio, presenta variaciones, puesto que no existe una transición instantánea entre el estado de detección y el estado de lectura correcta. Esto es debido a que colocarse el sensor de forma correcta en el dedo lleva unos segundos.

Puesto que la función de la señal infrarroja alcanza valores mayores que la señal de luz roja y además a priori es la más marcada en cuanto a forma, de aquí en adelante es la que tomaremos para el estudio de la naturaleza de la misma y la detección de pulsos cardíacos.

En primer lugar se diseñó un programa que tomara los valores máximos y mínimos de la gráfica correspondiente a la lectura infrarroja a intervalos fijos suficientemente largos como para captar los máximos y mínimos de al menos un pulso en estado de reposo. El tiempo máximo entre pulsos corresponde a la frecuencia cardíaca de deportistas entrenados, que puede ser de hasta 39ppm (ver Tabla 6). Esto es un pulso cada 1.53 segundos. Se tomó así un intervalo de 2 segundos para tomar los máximos y mínimos.

Para detectar cada pulsación se deben buscar los picos o los valles. Para ello se calculaba el valor situado a 2/3 entre el máximo y el mínimo leído en cada intervalo de 2 segundos. El motivo de esto era encontrar los picos de la función de onda de forma que se discriminen los pequeños valles antes

descritos (que suelen tener lugar entre 1/3 y 1/2 de la onda).

Comparando ese valor límite con el valor actual de la onda, podemos determinar si estamos en la región de pico y, en caso afirmativo, marcar el momento en que esto sucede e iniciar un contador hasta el siguiente momento en que entremos en una región de pico. Midiendo tiempos entre picos estamos midiendo el tiempo entre pulsos cardíacos, con lo que hallar la frecuencia cardíaca.

La principal problemática de este método es la posible existencia de valores variables en tiempos pequeños o de ruido. Un valor que supere el valor límite y un valor consecuente con un valor por debajo del mismo darán como resultado la detección de un falso pico para consecuentemente detectar otro pico debido a la naturaleza creciente de la onda en la región.

Esto dará como resultado tiempos de pulso muy pequeños. La solución a esta problemática es sencilla, filtrar aquellos tiempos de pulso inferiores al tiempo mínimo de pulso posible. Teniendo en cuenta que según las tablas la frecuencia cardíaca máxima posible es de 220 pulsaciones, toda pulsación de una duración inferior a 0.27 segundos está por debajo del umbral de lo que es un pulso normal.

Debemos además considerar que un tiempo de pulso de esa duración puede ser posible y que eso no implique una frecuencia cardíaca alta, como por ejemplo puede ser un pulso aislado menor de lo habitual. Es por ello que se establece un margen a ese tiempo de pulso mínimo.

Además, otra problemática que surge mediante este método es la tendencia ascendente o descendente de la onda. Durante el intervalo de 2 segundos la onda puede salir del rango establecido por el máximo y el mínimo, pudiendo resultar imprecisa la toma de tiempos entre picos.

Se cambió el programa partiendo de la misma premisa. Analizando la forma de la onda podemos distinguir un segmento fácilmente distinguible, que es el flanco de bajada de la onda. Es un descenso abrupto que sucede en

un período muy breve de tiempo. Esta situación es única en la onda, lo que la convierte en una buena referencia para identificar cada pulsación.

Para identificar este segmento vamos a estudiar el incremento y decremento de la función de onda. Para ello el programa se escribió para almacenar los últimos valores de las lecturas y calcular su valor mínimo, que corresponde a un valor negativo variable en función de la pendiente de la onda. El tamaño del vector en el que se almacenan los últimos valores depende de la sensibilidad que queramos darle al algoritmo, de forma que para valores mayores filtramos posibles efectos de ruido, pero perdemos sensibilidad.

```
void ActualizaVectorPulso(int nuevo_valor)
{
  int i;
  int aux;
  for(i=2; i>=0; i--)
  {
    aux=vectorPulsaciones[i];
    vectorPulsaciones[i]=nuevo_valor;
    nuevo_valor=aux;
  }
}
```

Ilustración 39. Función ActualizaVectorPulso() del programa del microcontrolador.

Tras almacenar los valores y detectar los mínimos de la función de la pendiente y cada segundo, se calcula un valor situado a 1/3 de dicho mínimo, considerándose así que todos los valores por debajo de ese límite corresponden a una caída de pendiente suficiente para considerar que estamos en la región deseada. El motivo por el cual este valor ha de resetearse en un intervalo de tiempo fijo, y no cada pulso, es que precisamente este valor es requerido para detectar el pulso y por tanto no puede depender de él. Tras colocar el transductor en el dedo, las señales varían de forma dispar hasta establecerse, lo que hace variar la función de pendiente de forma arbitraria hasta su establecimiento. Tras unos pocos segundos los valores mínimos de la misma son suficientemente buenos como para detectar los pulsos como se ha descrito.

En la siguiente imagen pueden verse las señales infrarroja (azul) y roja

(roja) junto con la gráfica de la pendiente (verde). La gráfica amarilla representa el valor situado a 1/3 del mínimo, que se restablece cada segundo. La gráfica morada corresponde a una variable introducida para visualizar cuando la pendiente pasa a valer menos que el límite de detección, viéndose claramente cómo se corresponden los flancos de subida de esta variable con el máximo de la onda infrarroja.

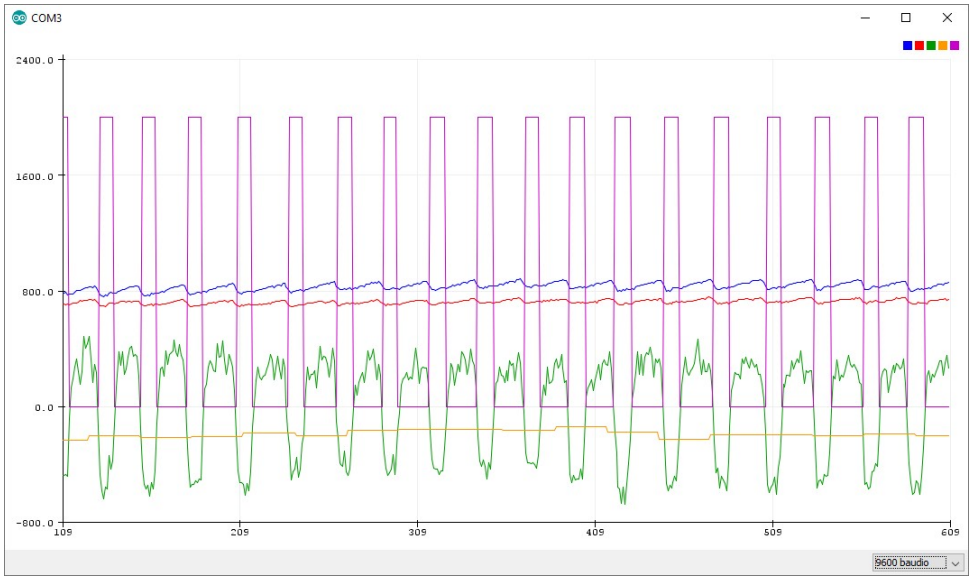


Ilustración 40. Detección de máximos de pulso.

Una vez identificados de forma correcta los pulsos, podemos medir el tiempo entre ellos. Se crea una variable booleana que sea true o false en función de si estemos en la región de pico de forma análoga a la de la variable auxiliar antes descrita. Tras cada ciclo de encendido de los LED se comprueba mediante la función DetectarPulso() si hemos pasado de un estado *false* a *true* y, en caso afirmativo calculamos el tiempo transcurrido desde el anterior pulso e iniciamos la cuenta de nuevo hasta el siguiente.

Además, filtramos ruidos de la misma forma que se hizo en el anterior método, filtrando pulsaciones de menor duración de 250 milisegundos y aquellas que duren más de 2 segundos. Si la duración está en ese rango, almacenamos su valor en un vector que contenga las duraciones de las 3 últimas pulsaciones con las que se calculará el valor de la frecuencia cardíaca media entre esas tres a través de la función CalculaPPM(). El motivo de esto

es que, como hemos dicho antes, un pulso de duración anormal puede no ser significativo, es por tanto que se eligió realizar la media con las tres últimas mediciones para que la medición fuera lo más rápida (en tiempo real) exacta posible. El valor del vector puede cambiarse para cotejar sus resultados a diferentes tamaños del vector.

```
int CalculaPPM()
{
    float pulso1 = 60000/vectorPulsaciones[0];
    float pulso2 = 60000/vectorPulsaciones[1];
    float pulso3 = 60000/vectorPulsaciones[2];

    return (int)(((pulso1+pulso2+pulso3)/3)+0.5);       //Devolvemos el promedio de los tres pulsos, redondeando al entero más próximo
}
```

Ilustración 41. Función CalculaPPM() del programa del microcontrolador.

Para calcular el nivel de oxígeno en sangre, esta vez se reiniciaran los valores máximo y mínimo a sus valores extremos en cada pulsación, de forma que se calculen los máximos y mínimos relativos de cada onda. Cuando la función DetectarPulso() reinicia el contador y detecta un nuevo pulso correcto calcula la relación entre la variación relativa de cada tensión como se explica en el apartado 2 de fundamentos teóricos. De igual forma que se calcula la frecuencia cardíaca con las tres últimas duraciones de pulso, se calculará la relación media de las tres últimas relaciones de tensiones para hallar el nivel de oxígeno correspondiente a tal relación.

```
float CalculaR()
{
  float r = (((extremos_R[1]-extremos_R[0])*extremos_IR[0])/((extremos_IR[1]-extremos_IR[0])*extremos_R[0]));
  extremos_IR[0]=2400;
  extremos_IR[1]=0;
  extremos_R[0]=2400;
  extremos_R[1]=0;

  return r;
}
```

Ilustración 42. Función CalculaR() del programa del microcontrolador.

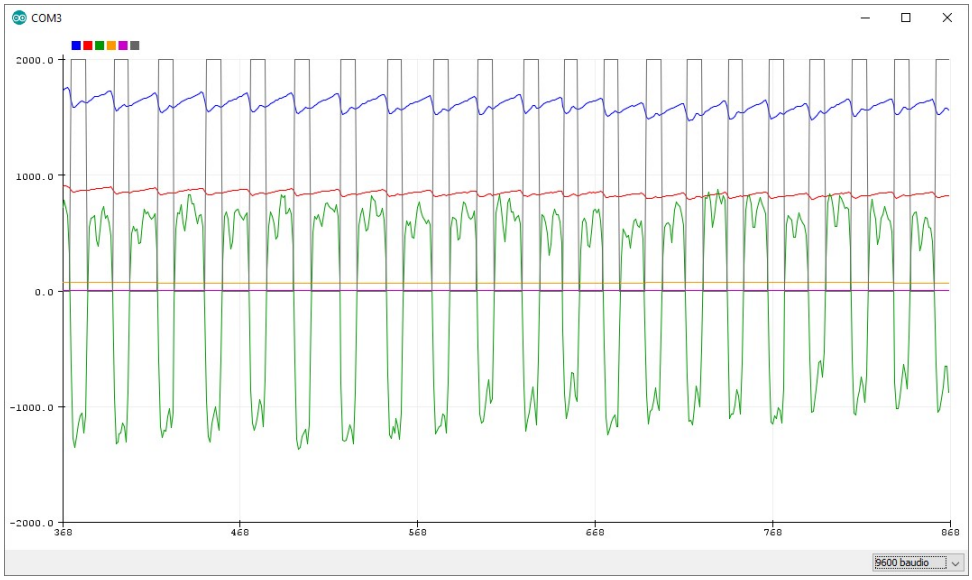
De nuevo, el motivo de tomar el valor medio para la relación de tensiones vuelve a ser que la razón de tensiones relativas para un único pulso no es significativa. Además, teniendo en cuenta los rangos de tensión tan pequeños entre los que nos movemos y la gran variabilidad y sensibilidad de la lectura, el cálculo de este valor resulta complejo, dando resultados dispares a cada pulsación, y por tanto se realiza una estimación media de las relaciones relativas.

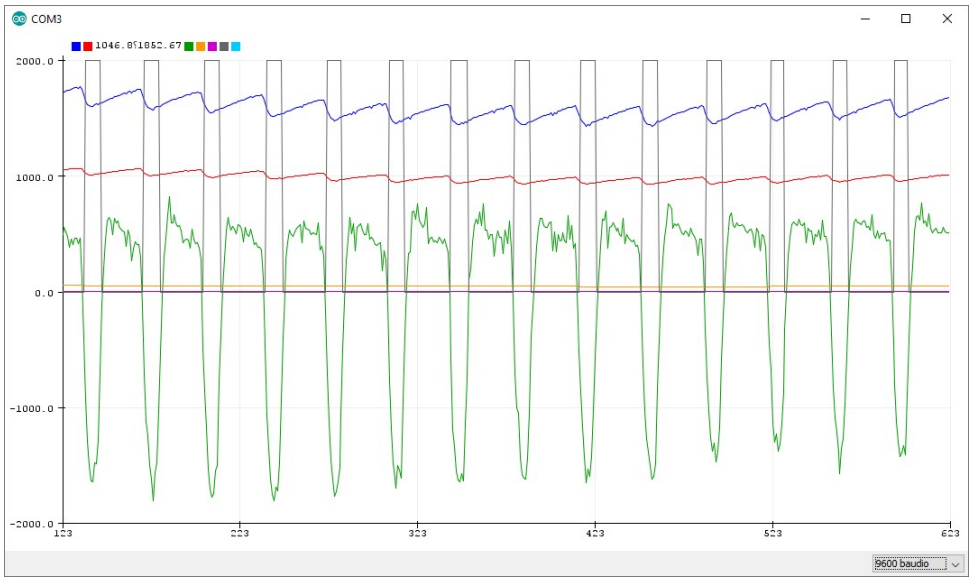
Por último, la función DetectarPulso() actualiza los valores del mensaje JSON a enviar por el serial del bluetooth.

```
void DetectaPulso()
{
  pulsoAnterior=pulso;

  if(pendiente < limiteDeteccion)
    pulso = true;
  else
    pulso = false;

  if((!pulsoAnterior) && pulso)                          //Si detectamos pulso
  {
    duracion_pulso=millis()-comienzo_pulso;
    comienzo_pulso=millis();

    if(duracion_pulso > 250)                            //Si la variación no es fruto de ruidos
    {
      ActualizaVectorPulso(duracion_pulso);
      ActualizaVectorR(CalculaOxigeno());
      ppm=CalculaPPM();
      R=(vectorR[0]+vectorR[1]+vectorR[2])/3;

      datos["ppg"]=V_IR;
      datos["oxigeno"]=R;
      datos["pulso"]=ppm;
    }
  }
}
```

Ilustración 43. Función CalculaPulso() del programa del microcontrolador.

Cada 1 segundo, coincidiendo con el período de restablecimiento del límite de la pendiente, enviaremos el mensaje en formato JSON a través del serial3, que se ha llamado HC06, que se ha configurado a una velocidad de transmisión de 115200 baudios (bits por segundo). Esto equivale a una velocidad de 14400 bytes por segundo. Por lo tanto, por cada ciclo de encendido de los LED se podría enviar un mensaje de un tamaño de 14,4 bytes.


COM3
2000.0
1000.0
0.0
-1000.0
-2000.0
360
460
560
660
760
860
9600 baudio


COM3
2000.0
1000.0
0.0
-1000.0
-2000.0
123
223
323
423
523
623
9600 baudio

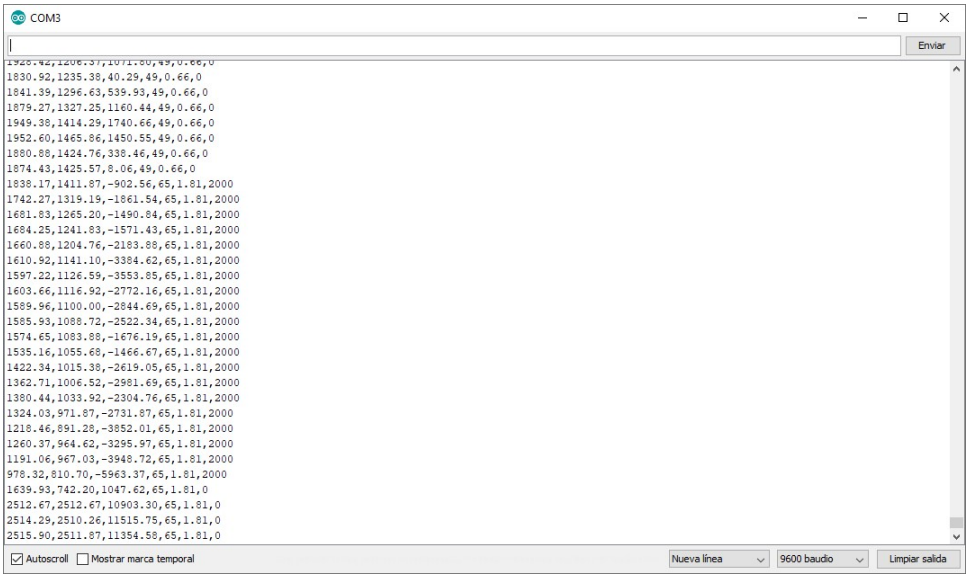


Ilustración 44. Dos ejemplos de señal de onda (arriba y medio) y transmisión del vector de datos representado (abajo).

Al declarar la variable JSON del mensaje se debe reservar el tamaño del mismo, y para ello se reservaron 200 bytes, un tamaño con un margen suficiente para el mensaje a enviar. El modelo normalizado del mensaje a enviar, en el formato JSON es el siguiente:

```
{"datos":{"ppg":0,"oxigeno":0,"pulso":0}}
```

En el mensaje se ha incluido la variable ppg correspondiente a la señal fotopletismográfica de la medición infrarroja. El motivo de esto es porque en un principio se pensó monitorizar su valor, y finalmente se descartó su uso por varios motivos. En primer lugar, el valor de la señal en sí no aporta ningún tipo de información, puesto que se corresponde al valor de la tensión producida en el fotodiodo en milivoltios, valor arbitrario cuyo valor absoulto carece de significado real. Otra posibilidad sería graficar el valor de la señal, pero de nuevo no deja de ser una señal sin información, a pesar de que gráficamente puedan apreciarse los pulsos. Además, teniendo en cuenta que la variación relativa de los pulsos respecto del total de la señal es bastante pequeña, por lo que la visualización sería pobre. En adición, la aplicación ha sido diseñada para tratar todas las variables como valores a representar numéricamente en una vista de tipo lista única para cada variable, lo cual

implicaría realizar una excepción únicamente con este valor para disponerlo en una vista distinta a la vista de constantes vitales (ver el apartado referente a la clase ConstanteVital). Por último, la velocidad de transmisión influye en la posibilidad de representar gráficamente la señal. Puesto que el mensaje se envía cada 1 segundo, es imposible reconstruir la gráfica a esa frecuencia.

Para un mensaje de 200bytes a una velocidad de 115200 baudios podemos enviar un mensaje cada 13,9 milisegundos. A esta frecuencia sería posible reconstruir la señal a partir de los datos enviados. El motivo de enviar los datos a intervalos tan grandes como un segundo es que las señales que vamos a monitorizar no necesitan de un refresco rápido, puesto que el nivel de oxígeno en sangre o la frecuencia cardíaca no varían de forma rápida (milisegundos). Además, debemos dar tiempo a los múltiples procesos paralelos de la aplicación Android para extraer y procesar la información recibida en el mensaje.

Aunque finalmente se decidiera no emplear la variable fotopletismográfica, se ha mantenido en el contenido del mensaje como ejemplo de la versatilidad en el uso del mensaje JSON como formato de transmisión de datos, así como de su interpretación de forma normalizada en la aplicación Android, como se verá más adelante.

4.2. Calibración y toma de datos

Para conocer el valor del nivel de oxígeno que le corresponde a cada uno de los valores calculados a partir de la relación entre extremos relativos de las dos señales, cotejamos los resultados con los niveles de oxígeno que nos proporcionará un oxímetro comercial modelo HealthForYou del fabricante SilverCrest.

Ilustración 45. Modelo del oxímetro comercial empleado para realizar la calibración de nuestro dispositivo.

Se tomarán datos de correspondencia entre el valor R devuelto por el programa y los niveles de saturación de oxígeno medidos simultáneamente mediante el oxímetro comercial. Además, se compararán los valores de frecuencia cardíaca proporcionados por ambos dispositivos, para verificar que nuestro algoritmo devuelve unos valores que se corresponden con la medida exacta.

Como se ha descrito en la sección anterior, se envían los valores promedio de las tres últimas mediciones, tanto para el valor de la frecuencia cardíaca como para el valor de la relación entre señales.

PPM	
Dispositivo	Oxímetro comercial
62	62
67	67
66	67
75	73
62	63
71	71
78	83
82	80
60	61
74	74
63	64
69	73
61	61
52	52
52	49
56	55
50	50

Tabla 9. Tabla de mediciones de frecuencia cardíaca para una ponderación de 3 valores.

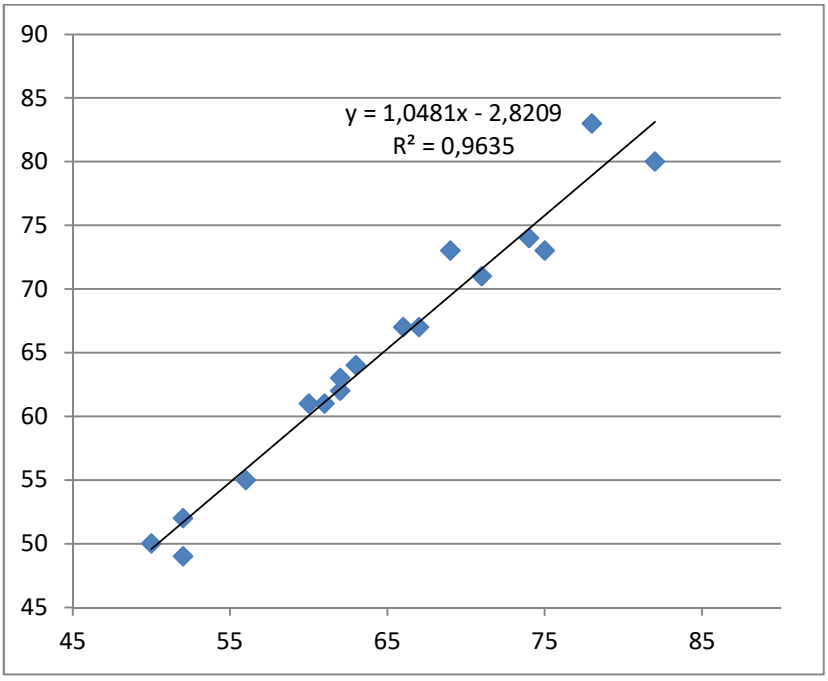


Ilustración 46. Comparación de medidas de frecuencia cardíaca con una ponderación de 3 valores.

Saturación de oxígeno	
Dispositivo	**Oxímetro comercial**
0,67	97
0,68	97
0.70	96
0.56	97
0,57	97
0,66	96
0,7	95
0,68	97
0,73	95
0,67	95
0,86	94
0.57	95
0.77	97
0,73	94
0,67	96
0,58	98
0,67	94
0,66	95
0,64	96
0,69	95

Tabla 10. Tabla de mediciones de saturación de oxígeno para una ponderación de 3 valores.

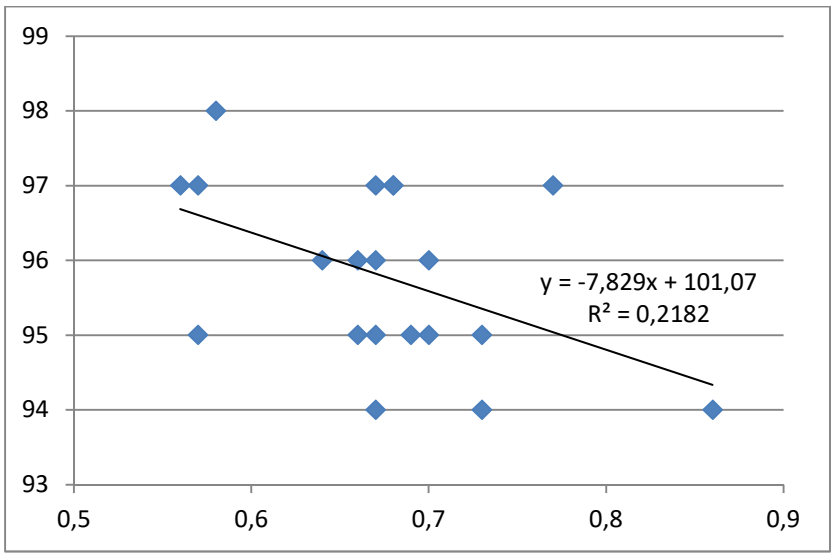


Ilustración 47. Comparación de mediciones de saturación de oxígeno con una ponderación de 3 valores.

En la gráfica referente a la frecuencia cardíaca observamos un ajuste bastante aceptable de los resultados. Durante la toma de datos se observó una gran variabilidad de los valores con respecto del tiempo, es decir, la señal varía demasiado rápido en cada pulso.

Esto es debido a que el peso relativo que tiene cada medición es muy alto teniendo en cuenta que cada medición representa un tercio del promedio, lo que hace que una medida algo diferente (bien por un pulso anormalmente más corto o bien por un error al identificar el pulso) haga fluctuar el promedio en gran medida.

Algo similar ocurre con la toma de datos para el cálculo de la saturación de oxígeno. La relación entre pulsos puede variar entre cada pulso y como consecuencia su valor promedio fluctúa mucho. Además, observamos el principal problema de la toma de datos que es la mala adecuación de las mediciones a una recta.

Los motivos de ello pueden ser el poco rango numérico en el que los valores de saturación de oxígeno pueden tomar. Por otra parte, la variabilidad del oxmetro comercial conlleva bastante incertidumbre con la toma de datos: desde medidas por debajo de los niveles saludables, hasta grandes variaciones en un intervalo pequeño de tiempo que se añaden a la alta variabilidad de las mediciones tomadas de nuestro sensor como se ha comentado anteriormente. No se han considerado como válidas las mediciones por debajo de 94% por considerarse inequívocamente erróneas.

A pesar de ello se intuye una relación proporcionalmente inversa entre los datos cotejados.

Para intentar corregir los errores derivados de la variabilidad de los datos se decide cambiar los vectores de toma de datos a un tamaño de 10 y realizar la ponderación a través de estas 10 mediciones.

En primer lugar se han definido esos tamaños como constantes globales en nuestro programa, para permitir la modificación de las

ponderaciones en posibles cambios sucesivos. Se han modificado las funciones ActualizaVectorPulso(), CalculaPPM() y ActualizarVectorR() para que recorran el vector con el nuevo tamaño establecido. Además se ha añadido la función CalculaMediaR() para devolver la media de los valores del vector de relaciones entre extremos.

```
float CalculaMediaR()
{
  int i;
  float suma_R;

  for(i=0; i<SIZE_VECTOR_MEDIA; i++)
    {
      suma_R+=vectorR[i];
    }

  return (suma_R/SIZE_VECTOR_MEDIA);
}
```

Ilustración 48. Función CalculaMediaR() del programa del microcontrolador.

Con estas modificaciones tomamos nuevamente mediciones tanto de la frecuencia cardíaca como de los niveles de saturación de oxígeno en sangre.

PPM	
Dispositivo	Oximetro comercial
70	70
88	88
68	68
66	67
79	79
68	68
74	75
81	81
58	58
72	72
75	75
68	68
74	75
73	73
60	60
61	61
67	66
82	82
62	62
70	71
90	90
80	82
62	62
80	81
57	59
86	86
64	63
56	56
53	52
53	53
67	68

Tabla 11. Tabla de mediciones de frecuencia cardíaca para una ponderación de 10 valores.

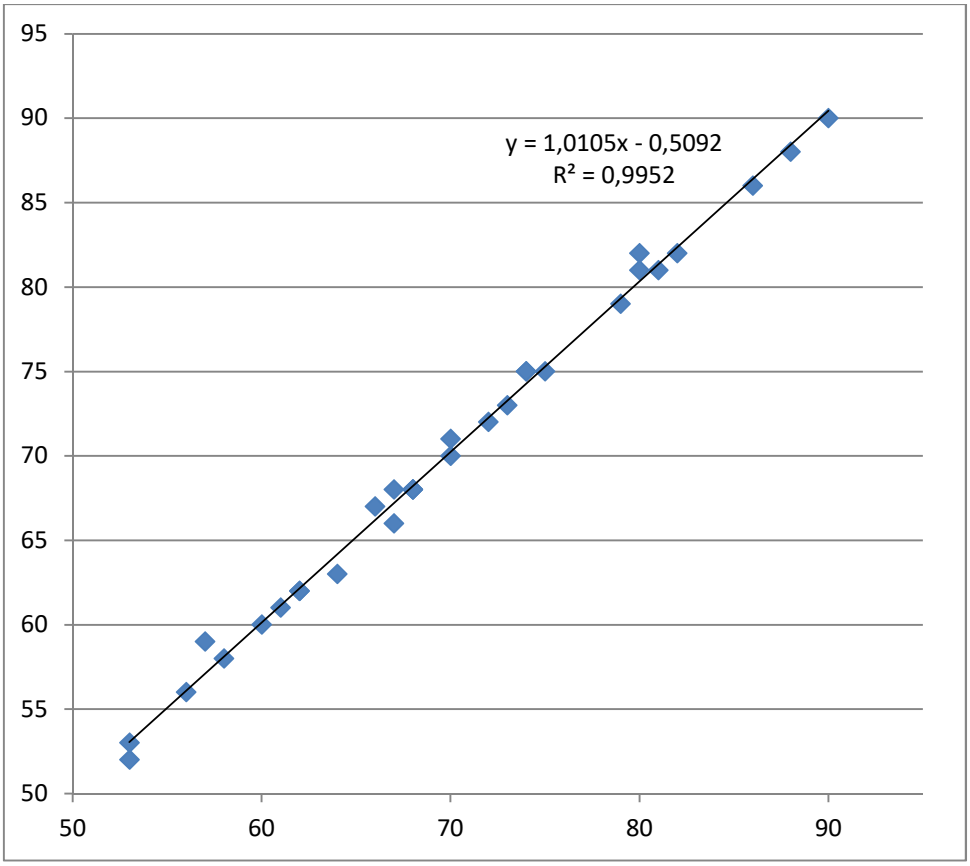


Ilustración 49. Comparación de mediciones de frecuencia cardíaca con una ponderación de 10 valores.

Saturación de oxígeno	
Dispositivo	Oximetro comercial
0,8	98
0,78	96
0,83	95
0,95	94
0,82	97
0,86	95
0,6	96
0,72	97
0,68	98
0,66	97
0,65	97
0,79	94
0,74	95
0,79	94
0,71	95
0,77	94
0,85	94
0,91	95
0,71	95
0,87	95
0,71	93
0,92	92
0,72	95
0,78	94
0,86	96
0,77	94
0,94	93
0,93	95
0,67	95
0,74	94
0,65	98
0,71	98

Tabla 12. Tabla de mediciones de saturación de oxígeno para una ponderación de 10 valores (en gris los valores descartables inferiores a 94%).

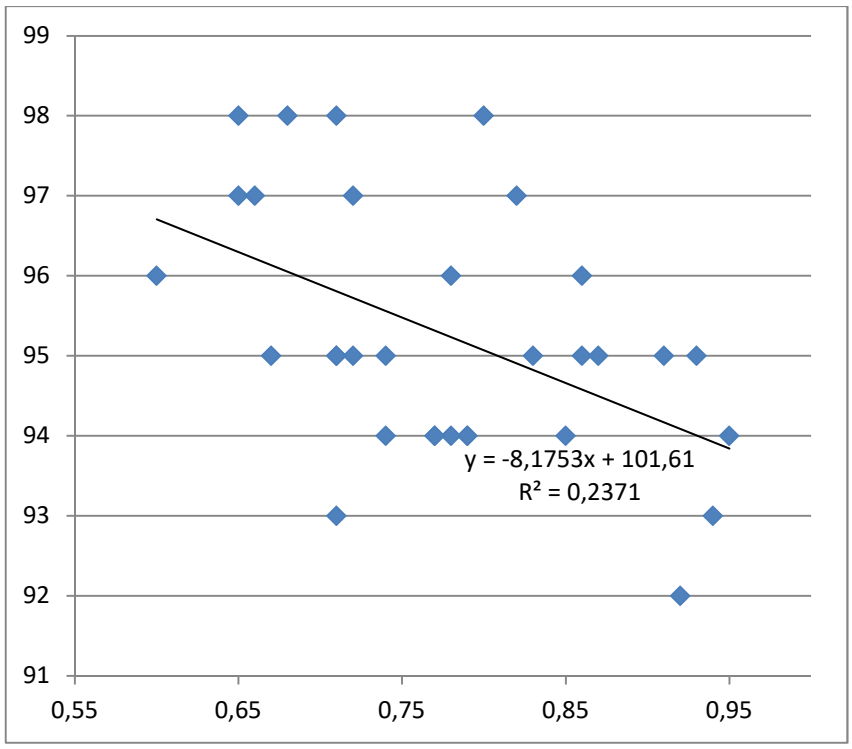


Ilustración 50. Comparación de mediciones de saturación de oxígeno con una ponderación de 10 valores, incluidos los valores descartables.

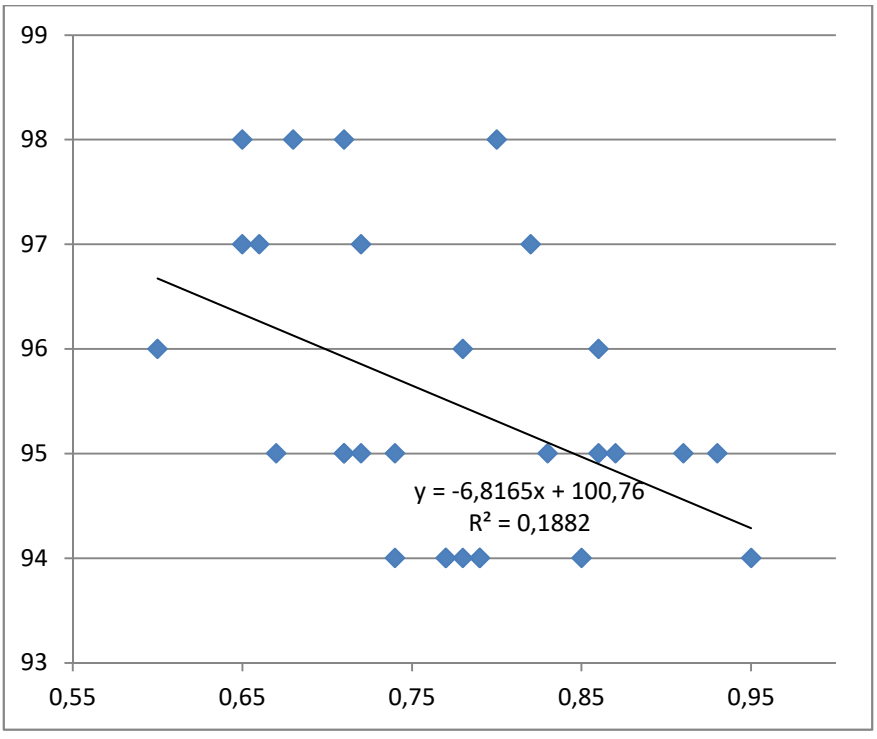


Ilustración 51. Comparación de mediciones de saturación de oxígeno con una ponderación de 10 valores, sin incluir valores descartables.

La ponderación a 10 consigue una medición más estable que en el caso anterior. Las medidas se han tomado en momentos en la que ambos sensores permanecían estables. La frecuencia cardíaca se ajusta de forma correcta a las mediciones tomadas con el oxímetro comercial.

Para la toma de datos de saturación de oxígeno vuelve a presentarse la problemática de la gran variabilidad de la toma de datos, si bien existe una tendencia descendente. Podemos afirmar además que la correlación de datos no parece diferir de la toma de datos para una ponderación de 3 elementos. La problemática del limitado rango de datos posibles y la disparidad de medidas del oxímetro comercial dificultan un mejor ajuste.

Para intentar ajustar mejor los datos se han aceptado esta vez valores menores a 94% de saturación de oxígeno. En ese caso, la ecuación que representa la correlación de datos es la siguiente:

$$y = -8{,}1753x + 101{,}61$$

Construimos una función en nuestro programa que devuelva el valor correspondiente de saturación de oxígeno, redondeando al entero más próximo.

```
int CalculaSpO2(float r)
{
  return (int)(-8.1753*r + 101.61 + 0.5);    //rednodeamos al entero más cercano
}
```

Ilustración 52. Función CalculaSpO2() del programa del microcontrolador.

4.3. Mejoras del programa

A continuación añadiremos una serie de cambios realizados en el programa del microcontrolador tras la calibración del dispositivo.

En primer lugar filtramos los datos a enviar, no enviando los datos hasta que el vector de pulsos se rellene completamente, dando un tiempo de

establecimiento hasta el inicio del envío de datos. Igualmente, si el valor de saturación de oxígeno supera el 100% o está por debajo de 90% descartamos el envío de datos. Además, si la duración del pulso es demasiado alta se considerará como una mala toma de mediciones, por lo que se reiniciarán los vectores de pulsos y relaciones hasta recuperar una buena toma de medidas.

```
void DetectaPulso()
{
  pulsoAnterior=pulso;

  if(pendiente < limiteDeteccion)
    pulso = true;
  else
    pulso = false;

  if((!pulsoAnterior) && pulso)                              //Si detectamos pulso
  {
    duracion_pulso=millis()-comienzo_pulso;
    comienzo_pulso=millis();

    if((duracion_pulso > 250) && (duracion_pulso < 1500))           //Si la variación no es fruto de ruidos
    {
      ActualizaVectorPulso(duracion_pulso);
      ActualizaVectorR(CalculaR());
      ppm=CalculaPPM();
      R=CalculaMediaR();
      oxigeno = CalculaSpO2(R);

      if(vectorPulsaciones[0]==0 || oxigeno>100 || oxigeno < 90)
      {
        datos["ppg"]=V_IR;
        datos["oxigeno"]=0;
        datos["pulso"]=0;
      }
      else
      {
        datos["ppg"]=V_IR;
        datos["oxigeno"]=oxigeno;
        datos["pulso"]=ppm;
      }
    }
    else if (duracion_pulso >1500)
    {
      for(int i=0; i<SIZE_VECTOR_MEDIA; i++)      //Reiniciamos los vectores
      {
        vectorPulsaciones[i]=0;
        vectorR[i]=0;
      }
    }
  }
}
```

Ilustración 53. Función DetectaPulso() del programa del microcontrolador.

Se envía el valor V_IR puesto que puede ser de interés a la hora de conocer por qué los datos no han sido enviados.

Enviamos un mensaje nulo también en el caso de que el sensor esté en modo de detección. Si entramos en modo detección reiniciamos los vectores de pulso y relaciones hasta que el sensor vuelva a detectar de forma correcta.

```
else
{
  long unsigned t = millis();

  for(int i=0; i<SIZE_VECTOR_MEDIA; i++)      //Reiniciamos los vectores
  {
    vectorPulsaciones[i]=0;
    vectorR[i]=0;
  }

  datos["ppg"]=0;
  datos["oxigeno"]=0;
  datos["pulso"]=0;

  enviaBluetooth();             //Enviamos un mensaje nulo para indicar que no estamos midiendo correctamente

  while (millis() - t < 500) continue;
}
```

Ilustración 54. Secuencia de código del modo detección en el programa del microcontrolador.

Puesto que la toma de datos se ha realizado mediante el puerto serie 0 a 9600 baudios, la frecuencia de emisión de los LED se reduce al ser dependiente de la velocidad de transmisión. Al dejar de transmitir datos al IDE de Arduino, la frecuencia de encendido y apagado de los LED vuelve a su valor teórico.

Al hacerlo el resultado varía, para observar las variaciones aumentamos la velocidad de transmisión de datos a 115200 baudios y representamos los valores por el puerto serie al IDE de Arduino. Cuanto mayor es la frecuencia de muestreo observamos una mayor fluctuación de la variación de pendiente de la señal infrarroja. Para atenuar este fenómeno, restingimos las mediciones de tensión que se incluyen en el vector de señal cada 10ms.

```
float ActualizaVectorSenal(float nuevo_valor)
{
  int i;
  float aux;
  if(millis()%10==0)
  {
    for (i = (SIZE_VECTOR_PENDIENTE - 1); i >= 0; i--)
    {
      aux = vector[i];
      vector[i] = nuevo_valor;
      nuevo_valor = aux;
    }
  }
  return ((vector[SIZE_VECTOR_PENDIENTE - 1] - vector[0]) * 10);
}
```

Ilustración 55. Función ActualizaVectorSenal del microcontrolador con muestreo cada 10ms.

Con esta secuencia de código mejoramos la detección de pulsos considerablemente como se ve a continuación.

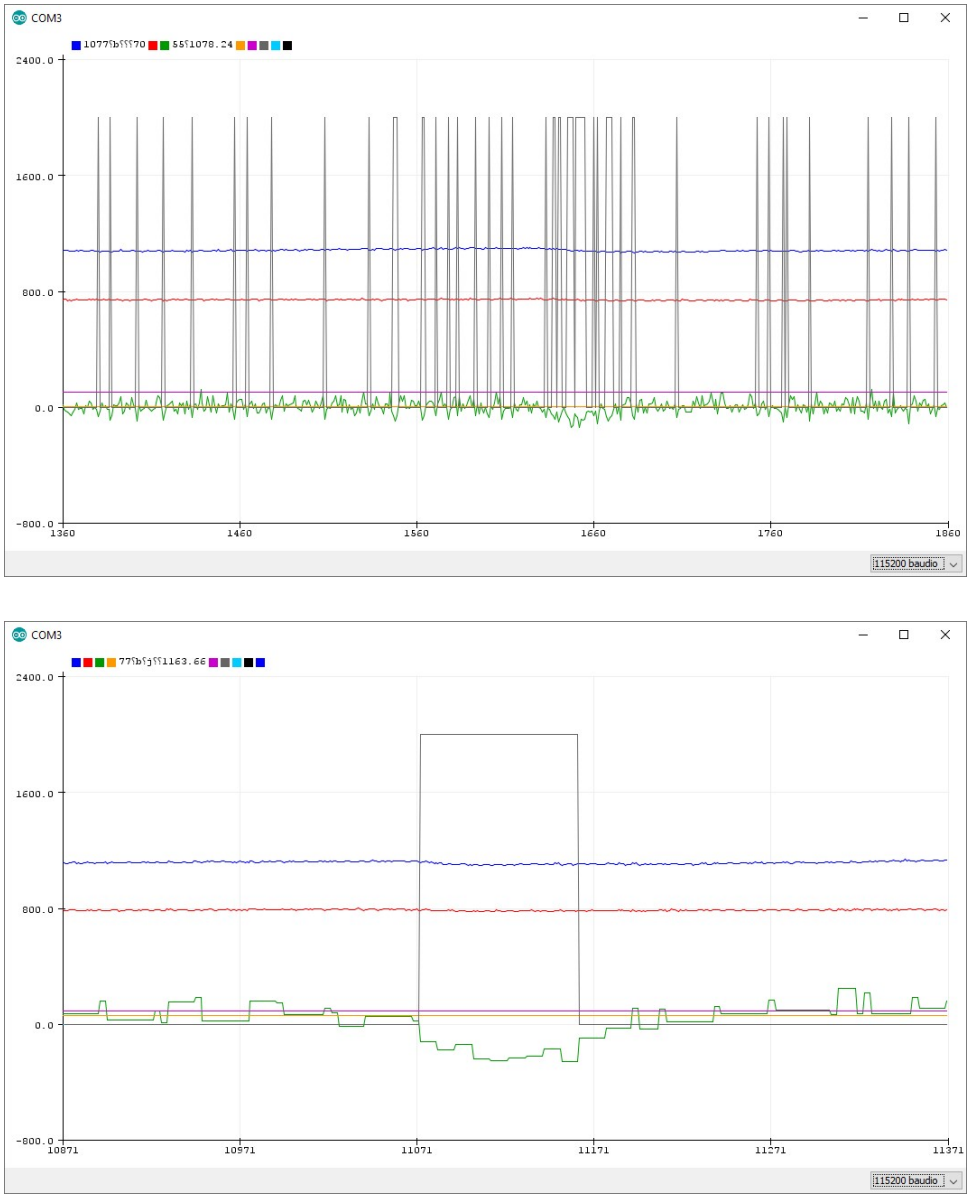


Ilustración 56. Comparativa del muestreo a la frecuencia de transmisión de datos de 115200 baudios (arriba) y tras la modificación de la función con muestreo cada 10ms (abajo).

5. Programa de Android

En primer lugar, Android es un sistema operativo desarrollado por la empresa Google pensado principalmente para dispositivos móviles y tabletas y, a grandes rasgos, se basa en el lenguaje de programación Java y XML. Java es un lenguaje orientado a objeto bastante parecido a C++, y el funcionamiento de la aplicación se basa en este lenguaje, mientras que XML es un lenguaje que sirve entre otras cosas como base de la apariencia e interfaz gráfica del programa en cuestión, así como del uso de recursos (idiomas, imágenes, audios…).

Inicialmente se consideró dedicar menos tiempo y esfuerzo en la aplicación de Android, recurriendo a la plataforma online Android Inventor, desarrollada por el *Massachusetts Institute of Tecnology* (MIT) y orientado al ámbito de la educación. La plataforma dispone de una serie de instrucciones o pseudo-código en forma de bloques conectables con una interfaz intuitiva y bastante flexible.

El objetivo del programa es simple: conectarse al dispositivo mediante bluetooth, recibir los datos de forma correcta y finalmente mostrarlos por pantalla. El dispositivo debe conectarse a la dirección MAC del módulo bluetooth empleado, que en nuestro caso es 00:02:5B:00:A0:18. Una vez establecida la conexión de forma correcta, un botón permitirá iniciar la transmisión de datos desde el dispositivo. De la misma forma, se permitirá detener la transmisión de datos.

En un principio, se pensó en desarrollar un programa sencillo para este proyecto, y la plataforma Android Inventor es una herramienta ideal para ello, capaz de desarrollar aplicaciones sencillas sin necesitar profundizar en el aprendizaje de los lenguajes empleados por Android. El mensaje no se enviaba en formato JSON sino como una cadena de caracteres separada mediante un carácter de separación, de forma que la interpretación de los datos era sencilla pero menos robusta y normalizada.

Finalmente y a la vista de la gran importancia que los dispositivos móviles como los basados en tecnología Android tienen hoy en nuestra vida diaria, se tomó la oportunidad que este proyecto brindaba para aprender y profundizar sobre el mismo. Se decidió dejar la aplicación de Android Inventor como herramienta para comprobar si el dispositivo envía correctamente el mensaje, de forma que nos sirva de apoyo a la hora de desarrollar la aplicación y depurar el funcionamiento del programa en Java que desarrollemos.

Se modificó ligeramente en su versión final para recibir una cadena en formato JSON en lugar de variables separadas por caracteres específicos.

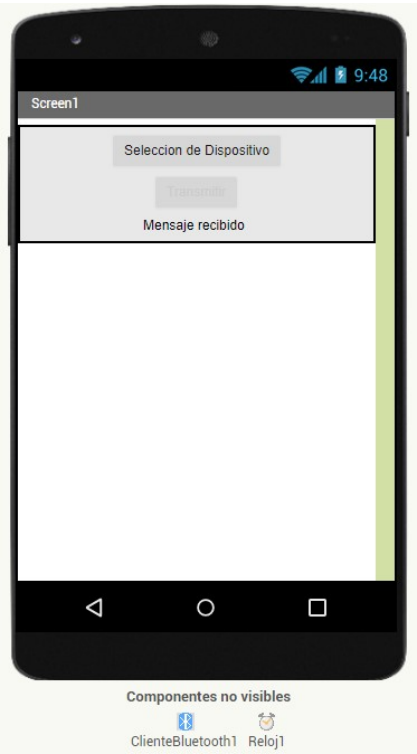


Ilustración 57. Representación de las vistas de la aplicación de comprobación.

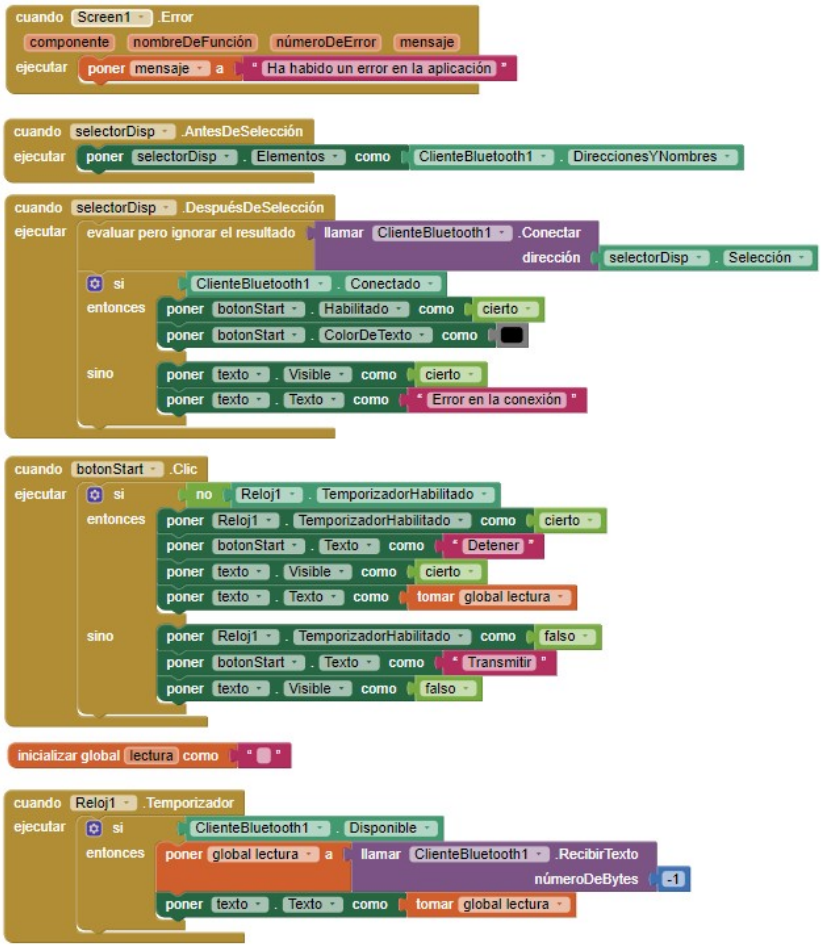


Ilustración 58. Vista de bloques de pseudo-código de la aplicación de comprobación.

Para el desarrollo de la aplicación se empleó Android Studio, un software desarrollado por Google y que permite la creación de proyectos. En esta memoria se procederá a comentar tan sólo los principales archivos dentro del proyecto, obviando otros como los referentes a plantillas (layouts) o recursos.

Aunque no se entrará a explicar toda la terminología y fundamentos de la programación en Android en esta memoria, hay que comentar una serie de términos o conceptos para poder entender los sucesivos apartados. En primer lugar y como se ha dicho anteriormente, Android se fundamenta en los lenguajes Java y XML. Java es un lenguaje orientado a objeto, y por tanto se construye en torno a elementos como clases y a conceptos como el de herencia o métodos. Una de las clases que emplearemos con Android es la

clase *view*. Podemos entender una *view* como un "bloque" o área rectangular que en función de su contenido define una subclase: puede ser un texto (*TextView*), una imagen (*ImageView*), una lista de otras *views* (*ListView*), un botón (*Button*) o servir como plantillas que contienen otras *views* recibiendo el nombre de *layout* (*ViewGroup*). Una serie de atributos definen su posición, su tamaño o su contenido. La disposición y configuración inicial de esas *views* se determinará en los archivos XML de la aplicación, y podrán referenciarse en el código del programa en Java de la aplicación.

Otro concepto que hay que entender antes de continuar es el de actividades. Una aplicación de Android puede componerse de diferentes pantallas, cada una con su propia disposición de *views* diferente e independiente y con unas funciones y contexto completamente diferentes. A cada una de esas pantallas se le asocia una clase específica a las que se las llama actividades, y a cada una le corresponde un fichero en Java y otro en XML. Ambos ficheros están relacionados de forma que el fichero en XML se asocia al contexto del fichero en Java, correspondiéndose así un entorno gráfico y todas sus vistas a un código de programa específico.

El programa se divide en dos actividades principales, la primera es una pantalla introductoria en la que se recogerán los datos del paciente: el nombre, la edad, la altura y el peso. Dichos datos se guardaran dentro de la clase *Paciente* y se emplearán para el cálculo del índice de masa corporal. Al pulsar un botón de continuar, la aplicación nos lleva a la segunda actividad en la que la pantalla nos muestra por pantalla un resumen desplegable con los datos del paciente introducidos anteriormente, un botón para conectarnos al dispositivo y una *ListView* en la que se visualizan de forma sintética todos los signos vitales a monitorizar.

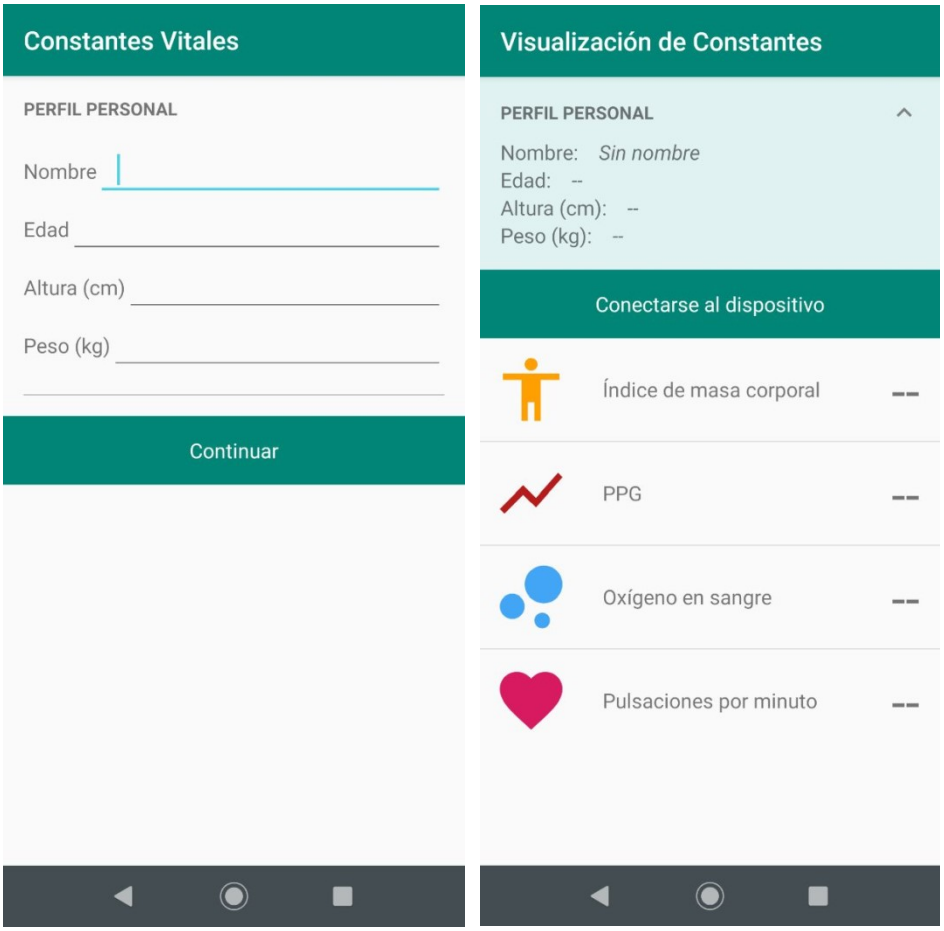


Ilustración 59. Vista de las dos actividades de la aplicación de Android. A la izquierda la actividad de inicio. A la derecha la actividad principal de visualización de constantes.

Al pulsar el botón de conectar al dispositivo, el hilo principal del programa llama a un segundo hilo de conexión y, si la conexión se establece correctamente, el hilo de conexión termina e inicia un segundo hilo indefinido que recibe y decodifica los datos para que se puedan mostrar en tiempo real en la *ListView* sin que el resto de la aplicación se quede bloqueada. Tras pulsar el botón de desconectar terminamos el hilo y la actividad vuelve al estado inicial de la misma, pudiendo pulsarse el botón de conexión nuevamente.

El desarrollo de la aplicación fue un desarrollo iterativo en la que algunas de las funcionalidades antes descritas se modificaron o evolucionaron para adaptarse a las problemáticas o mejorar su diseño frente al planteamiento inicial. A continuación se pasa a describir con mayor

detenimiento los principales puntos del programa.

5.1. Clases Paciente y ConstanteVital

Para el programa se han construido dos clases para almacenar los valores del usuario y sus signos vitales a monitorizar, Paciente y ConstanteVital. La estructura básica de una clase en Java la componen una serie de atributos privados (es decir, unas variables internas a las que no se pueden acceder desde fuera de la clase), un constructor con el mismo nombre que la clase y que llamaremos cada vez llamemos a la clase y mediante el cual introduciremos los argumentos de la misma, y una serie de métodos a los que podremos llamar tanto interna como externamente (dependiendo de si son públicos o privados) y que pueden trabajar o no con otros métodos o con los atributos de la clase. En otras palabras podemos entender los métodos como funciones internas que podremos llamar desde fuera o no y que, al igual que cualquier otra función, puede devolver un valor o no.

- Paciente

  Esta clase guarda los valores del nombre, edad, altura en centímetros y peso en gramos del paciente como atributos. Los valores se introducen como argumentos a través de su constructor de clase. La clase dispone de cuatro métodos getNombre(), getEdad(), getAltura() y getPeso(), que devuelven los valores de los atributos del nombre, la edad, la altura y el peso respectivamente.

  Al declarar un objeto o una variable dentro de una actividad, el dominio de ésta es tan sólo la de la propia actividad. La clase declarada no existe en otra actividad, y por tanto no podremos acceder a ella. Si queremos pasar los datos guardados en la clase paciente en la actividad de inicio a la actividad de visualización de constantes debemos crear un *intent* para transmitir dicha clase de una actividad a otra. Esto se verá al analizar más detenidamente la

actividad de inicio más adelante, pero para poder transmitir esa clase a través del *intent* es necesario que nuestra clase herede el contenido de la clase *Serializable* mediante el uso de *extends*. Cuando una clase hereda los métodos y los datos de otra clase podemos emplearlos en nuestra clase como si pertenecieran a ésta. Éste es uno de los pilares fundamentales en los que se basa la programación orientada a objetos, permitiendo así crear nuevas clases sin tener que programar todas sus funcionalidades desde cero, permitiendo así heredar el contenido de otras ya existentes y permitiendo su simplificación y versatilidad.

Extendiendo la herencia de *Serializable* a nuestra clase podremos pasarla como atributo al *intent* que permitirá transmitirla de la actividad inicial a la de monitorización de constantes.

```
package com.example.android.constantesvitales;

import java.io.Serializable;

/serial/
public class Paciente implements Serializable {
    private String mNombre;
    private int mEdad;
    private int mAltura;
    private int mPeso;

    Paciente(String nombre, int edad, int altura, int peso){
        mNombre=nombre;
        mEdad=edad;
        mAltura=altura;
        mPeso=peso;
    }

    public String getNombre(){

        return mNombre;
    }

    public int getEdad() {

        return mEdad;
    }

    public int getAltura() {

        return mAltura;
    }

    public int getPeso() {

        return mPeso;
    }
}
```

Ilustración 60. Clase Paciente.

- **ConstanteVital**

  Esta clase sufrió modificaciones desde su planteamiento inicial. Inicialmente la clase serviría para almacenar todas las variables a monitorizar: el oxígeno en sangre, la frecuencia cardíaca, el valor de la señal fotopletismográfica y el índice de masa corporal. Puesto que son variables que se correlacionan con el propio paciente, se pensó también en incluir estos valores en la clase Paciente.

  Para entender el resultado final es necesario conocer algunos aspectos del programa. Al recibir los datos a través de la conexión Bluetooth y reconstruir el mensaje JSON de forma correcta, lo pasamos al método getListaConstantes() de la clase ObtenciónDatos, que decodifica el mensaje y toma sus valores para después añadirlos como un nuevo elemento de la *ListView* que mostrará cada una de las variables a monitorizar.

  Cada elemento de la lista se compone de tres elementos: el nombre del signo vital que vamos a monitorizar, su valor numérico y un icono o imagen representativa para que se pueda identificar de forma rápida y a primera vista.

Ilustración 61. Plantilla de un elemento de la lista de visualización de signos vitales.

  Si guardamos en una misma clase, bien sea en la propia clase Paciente o en una nueva, habremos de tener 4 atributos para almacenar el valor de cada una de las variables a visualizar, además de otros 4 atributos que se corresponderían al identificador del icono correspondiente (ese identificador es una variable de tipo *int*). A estos atributos habría que sumarles nuevos métodos que devolvieran el valor de los mismos. Si quisiéramos introducir una nueva variable a monitorizar en nuestro programa tendríamos modificar la clase en cuestión y añadir dos nuevos

atributos, con sus respectivos métodos.

Un siguiente planteamiento sería el de observar una equivalencia entre todas las constantes. Puesto que todas siguen la estructura de nombre, valor e imagen, es lógico pensar que la mejor solución es la de crear una clase versátil que sirva para cualquier variable que queramos mostrar. Esto además facilita la modificación del programa en caso de añadir nuevos signos vitales a mostrar, de forma que normalizamos la clase.

Por tanto, la clase final cuenta con tres atributos: nombre, valor y un identificador para el icono que emplearemos. Puesto que el valor de la constante puede ser un entero o un *float* crearemos dos constructores diferentes dependiendo del tipo de variable de forma que al llamar a la clase podremos introducir como argumento tanto un valor entero como uno decimal, y el código automáticamente será capaz de determinar que constructor usar.

```
public class ConstanteVital {
    private String mNombre;
    private int mValor;
    private double mValorDouble;
    private int mIcono;

    public ConstanteVital(String nombre, int valor, int icono) {
        mNombre = nombre;
        mValor = valor;
        mIcono = icono;
    }

    public ConstanteVital(String nombre, double valorDouble, int icono) {
        mNombre = nombre;
        mValorDouble=valorDouble;
        mIcono = icono;
    }

    public String getNombre() {

        return mNombre;
    }

    public int getValor() {

        return mValor;
    }

    public double getValorDouble() {

        return mValorDouble;
    }

    public int getIcono()   {

        return mIcono;
    }
}
```

Ilustración 62. Clase ConstanteVital.

5.2. Clase ObtencionDatos

Esta clase proporciona un método para convertir un mensaje en el formato JSON antes descrito y extraer los valores de todas las variables a monitorizar. El método, llamado getListaConstantes(), toma como argumento el mensaje en cuestión, la clase paciente con los datos que hemos tomado en la pantalla de inicio y los recursos de la aplicación para poder acceder a ellos.

La clase a su vez devuelve una lista de objetos ConstanteVital. Para cada variable del mensaje JSON se añade a la lista una clase a la que se pasan como argumentos el valor extraído, el nombre de la variable en cuestión y el icono que queremos asociarle. Para la variable correspondiente al índice de masa corporal, puesto que no se toma del mensaje, el método calculaImc() toma la clase paciente introducida como argumento al llamar a la

clase y lo calcula a partir de los datos del paciente. Tras ello, añade un elemento a la lista como se ha descrito anteriormente con el valor calculado. Si alguno de los datos necesarios para el cálculo son 0 o menores que 0, devolvemos el valor -1.

```
private static double calculaImc(@org.jetbrains.annotations.NotNull Paciente paciente){
    double altura = paciente.getAltura();
    double peso = paciente.getPeso();
    if(altura>0 && peso>0)
        return  peso/(altura*altura/10000);
    else
        return -1;
}
```

Ilustración 63. Método calculaIMC() de la clase ObtencionDatos.

Así pues tendremos una lista de todas las constantes vitales que vamos a monitorizar, cada una con su nombre, valor e icono representativo. Si quisiéramos modificar nuestro dispositivo tan solo tendríamos que añadir o eliminar elementos al método en función del mensaje JSON enviado, permitiendo así flexibilidad al programa a la par que dándole robustez, siendo esta la única parte del código que tendríamos que modificar si quisiéramos cambiar las variables a mostrar.

```
public static ArrayList<com.example.android.constantesvitales.ConstanteVital> getListaConstantes
        (Paciente paciente, String mensaje, Resources r){

    ArrayList<com.example.android.constantesvitales.ConstanteVital> listaConstantesVitales =
            new ArrayList<>();

    try {
        JSONObject jsonObject = new JSONObject(mensaje);
        JSONObject datos = jsonObject.getJSONObject("datos");

        listaConstantesVitales.add(new com.example.android.constantesvitales.ConstanteVital
                ("Body mass index", calculaImc(paciente), R.drawable.ic_imc_foreground));
        listaConstantesVitales.add(new com.example.android.constantesvitales.ConstanteVital
                (r.getString(R.string.ppg), datos.getInt( name: "ppg"),
                        R.drawable.ic_ppg_foreground));
        listaConstantesVitales.add(new com.example.android.constantesvitales.ConstanteVital
                ("Oxygen in blood", datos.getInt( name: "oxigeno"),
                        R.drawable.ic_oxigeno_foreground));
        listaConstantesVitales.add(new com.example.android.constantesvitales.ConstanteVital
                ("Beats per minute", datos.getInt( name: "pulso"),
                        R.drawable.ic_corazon_foreground));
    }
    catch (JSONException e){
        Log.e( tag: "ObtencionDatos",  msg: "Error en la extracción del formato JSON", e);
    }

    return listaConstantesVitales;
}
```

Ilustración 64. Método getListaConstantes() de la clase ObtencionDatos.

5.3. ActividadInicio

Como se ha indicado anteriormente, una actividad en Java es una clase determinada. En otras palabras, una actividad en Android es una clase que hereda de la clase AppCompactActivity. Como se ha dicho antes, una clase de tipo actividad determina el funcionamiento de cada una de las pantallas de nuestra aplicación, y a ésta se le asocia un fichero XML de plantilla o *layout*. Como cualquier otra clase, una actividad se compone de atributos, constructores, métodos…

Inicialmente se planteó la aplicación como una única actividad, al igual que la desarrollada con App Inventor. En dicha actividad se mostrarían unas *views* de tipo EditText para rellenar un formulario con los datos del paciente: nombre, edad, altura y peso. A continuación, un botón serviría para conectar al dispositivo y mostrar los valores de las variables médicas a monitorizar.

El principal problema de esto fue de diseño, puesto que el formulario de toma de datos ocupa demasiado espacio en pantalla, desplazando la lista hacia la parte inferior. De forma ideal, las constantes han de ser visibles sin necesidad de tener que desplazar la pantalla. Además, aunque menos importante, de forma intuitiva, al rellenar valores de un formulario, se espera un botón de introducción o de aceptación de datos.

Es por tanto que se decidió dividir la aplicación en dos actividades, la primera para la recogida de datos del formulario y la segunda para visualizar las constantes. La primera deberá, no solo ser capaz de tomar los datos del paciente y almacenarlos en una clase Paciente, sino que además tendrá que enviar dicha clase a la siguiente actividad.

La clase ActividadInicio en un primer lugar inicializa las *views* del *layer* que vamos a emplear a través de sus identificadores. Un *TextView* hará las veces de botón, al asociársele un *listener* de tipo *OnClick*. Esta decisión es puramente estética. Cualquier *view* puede trabajar como botón al asociársele

un *listener*. Además a la view en el fichero XML se le añade un atributo de efecto de onda al pulsar.

El método que ejecuta el *listener* al ejecutarse por el evento de pulsación en primer

En primer lugar, el *listener* guarda los datos de los *EditText* en un objeto de tipo Paciente. Esto lo hace a través de una función llamada devuelveDatos(), que convierte las cadenas en el tipo de variable correspondiente y las guarda en sus respectivas variables. Además, en caso de que el usuario no introduzca algún valor, la función devolverá unos valores por defecto.

```
/*Devuelve una clase Paciente con los datos que el usuario ha introducido en los campos de
texto*/

private Paciente devuelveDatos() {

    /*En caso de que el usuario no introduzca alguno de los valores se introducen valores por
    defecto*/

    if (nombreEditText.getText().toString().matches( regex: ""))
        nombre = getString(R.string.sin_nombre);
    else
        nombre = nombreEditText.getText().toString();

    if (edadEditText.getText().toString().matches( regex: ""))
        edad = -1;
    else
        edad = Integer.parseInt(edadEditText.getText().toString());

    if (alturaEditText.getText().toString().matches( regex: ""))
        altura = -1;
    else
        altura = Integer.parseInt(alturaEditText.getText().toString());

    if (pesoEditText.getText().toString().matches( regex: ""))
        peso = -1;
    else
        peso = Integer.parseInt(pesoEditText.getText().toString());

    return new Paciente(nombre, edad, altura, peso);
}
```

Ilustración 65. Método devuelveDatos() de la ActividadInicio.

Posteriormente se llama a la función guardaDatos(), que guarda los valores de la clase que se pasa como argumento en las preferencias de la aplicación, de forma que cada vez que iniciemos la aplicación se guarden los valores de la sesión anterior. A esto se le conoce como *SharedPreferences*. Éstas se guardan a través de una llave, a la que hemos llamado

"preferencias". Una vez accedemos a ellas podemos editar sus valores a través de llaves: asociamos cada valor de la clase a sus respectivas llaves.

```
private void guardaDatos(Paciente paciente) {
    preferencias = getSharedPreferences( name: "preferencias", Context.MODE_PRIVATE);
    SharedPreferences.Editor editor = preferencias.edit();
    editor.putString("KEY_NOMBRE", paciente.getNombre());
    editor.putInt("KEY_EDAD", paciente.getEdad());
    editor.putInt("KEY_ALTURA", paciente.getAltura());
    editor.putInt("KEY_PESO", paciente.getPeso());
    editor.apply();
}
```

Ilustración 66. Método guardaDatos() que emplea SharedPreferences.

De forma análoga, el método cargaDatos() se ejecuta al inicializar las *views*, y accede a las preferencias mediante *SharedPreferences* para tomar los valores asociados a cada una de las llaves y las guarda en sus respectivas variables. Luego comprueba si los valores introducidos no están vacíos y de ser así los carga para mostrarlos en su respectivo *EditText*.

```
private void cargaDatos() {

    preferencias = getSharedPreferences( name: "preferencias", Context.MODE_PRIVATE);
    nombre = preferencias.getString( key: "KEY_NOMBRE", defValue: "Sin nombre");
    edad = preferencias.getInt( key: "KEY_EDAD", defValue: -1);
    altura = preferencias.getInt( key: "KEY_ALTURA", defValue: -1);
    peso = preferencias.getInt( key: "KEY_PESO", defValue: -1);

    if(!(nombre.equals("No name"))){
        nombreEditText.setText(nombre);
    }
    if(edad!=-1){
        edadEditText.setText(Integer.toString(edad));
    }
    if(altura!=-1){
        alturaEditText.setText(Integer.toString(altura));
    }
    if(peso!=-1){
        pesoEditText.setText(Integer.toString(peso));
    }
}
```

Ilustración 67. Método cargaDatos() que emplea SharedPreferences.

Tras recoger los datos, y puesto que el objeto de tipo Paciente no podrá ser referenciado fuera de la clase ActividadInicio, debemos llamar a un *intent* que ejecute la siguiente actividad y le envíe a ésta el objeto Paciente. Para ello se empleará el método putExtra() y startActivity(). El primero añadirá al *intent* de ejecutar la siguiente actividad un segundo comando de transferencia de información a esa segunda actividad y requerirá el uso de una llave para identificar el mensaje. Para poder enviar una clase de una

actividad a otra la clase que debemos transmitir debe heredar de *Serializable*, como se ha explicado anteriormente. Finalmente, lanzamos la ActividadConstantes a través de startActivity().

El método inicializarViews() comprende tanto la inicialización de las *views* a emplear como la carga de datos y la inicialización del *listener*. Dicho método será llamado al crear la actividad, esto es, dentro del método onCreate() de la clase ActividadInicio.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.actividad_inicio);

    inicializarViews();
}
```

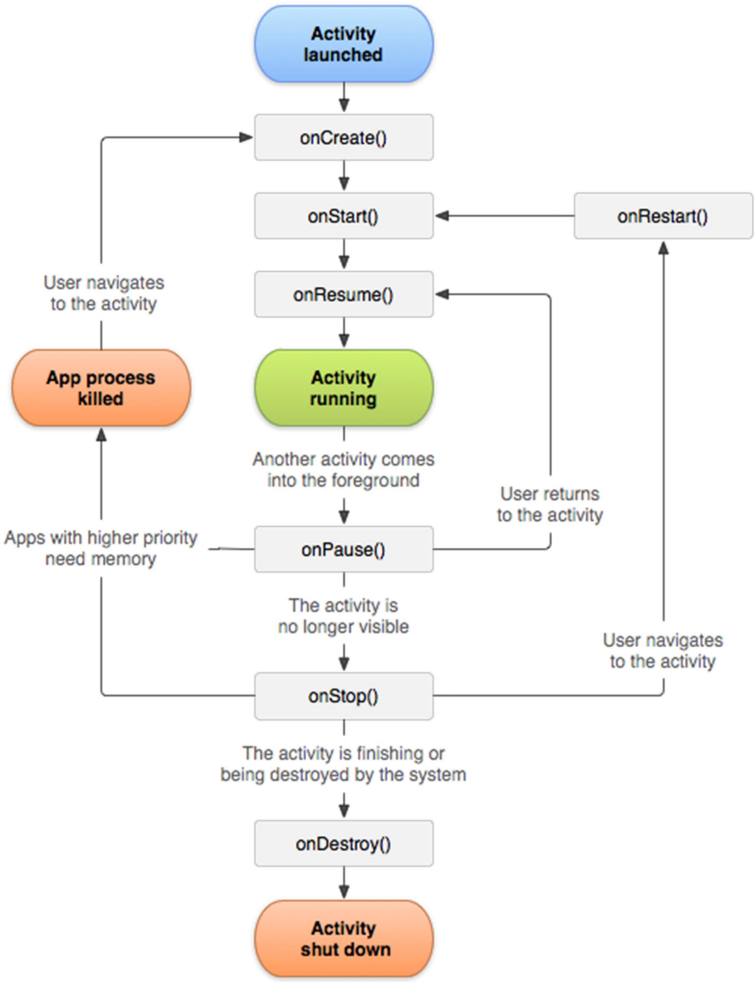
Ilustración 68. Método onCreate() de la ActividadInicio, que se ejecutará al crearse la actividad.

```
/* Inicializa todas las vistas y listeners */

private void inicializarViews(){
    nombreEditText = (EditText) findViewById(R.id.nombre);
    edadEditText = (EditText) findViewById(R.id.edad);
    alturaEditText = (EditText) findViewById(R.id.altura);
    pesoEditText = (EditText) findViewById(R.id.peso);
    continuar = (TextView) findViewById(R.id.continuar);

    /*Cargamos los datos de la sesión anterior para que aparezcan ya en las editText views*/

    cargaDatos();

    /*Creamos un listener para pasar a la siguiente actividad y enviar a esta los datos del
    paciente recogidos*/

    continuar.setOnClickListener((view) → {
            try{
                Paciente paciente = devuelveDatos();
                guardaDatos(paciente);
                Intent intent = new Intent( packageContext: ActividadInicio.this,
                        com.example.android.constantesvitales.ActividadConstantes.class);
                intent.putExtra( name: "paciente", paciente);
                startActivity(intent);
            }
            catch (Error e){
                Log.e( tag: "Conectar",  msg: "Error al inicializar el perfil del paciente. " +
                        "Introduzca los datos correctamente", e);
            }
    });
}
```

Ilustración 69. Método inicializarViews() de la ActividadInicio.

El método onCreate() de una actividad se ejecuta al lanzarse dicha actividad. Otros métodos se ejecutan en distintos momentos del ciclo de ejecución de la actividad. En el siguiente diagrama se muestran los métodos

que intervienen en las diferentes etapas del ciclo de una actividad.



5.4. ActividadConstantes

Esta actividad forma parte del grueso de la aplicación, siendo ésta la más importante. Su función es la de mostrar por pantalla la lista de todas las constantes vitales y variables a través de una *ListView*, cuyos elementos siguen el patrón establecido en el apartado referente a la clase ConstanteVital. Además, ha de disponer de un botón de conexión al dispositivo que sirva también para detener la transmisión de datos. Finalmente, se deben mostrar los datos del paciente en la cabecera, de una forma lo más compacta posible para no desplazar la lista hacia abajo, motivo

por el cual se decidió incluir la ActividadInicio.

En primer lugar, al crearse la actividad (en el método onCreate()), se llama al método inicializarViews(), que primero realiza un intent para recibir la clase Paciente enviada por la actividad anterior. Después rellena cada campo a través de la función rellenaPerfil(), que identifica cada campo a rellenar e introduce el texto correspondiente a partir de los datos de la clase Perfil. En caso de que alguno de los campos introducidos se corresponda a no haber introducido los valores, es decir, a los valores por defecto, la aplicación mostrará los caracteres "--".

```
@SuppressLint("SetTextI18n")
private void rellenaPerfil(@NotNull Paciente paciente){

    /* Tomamos los valores de la clase paciente con los datos que hemos tomado en la actividad
    del inicio y establecemos los valores de los text views correspondientes */

    nombreTextView = (TextView) findViewById(R.id.nombre);
    nombreTextView.setText(paciente.getNombre());

    edadTextView = (TextView) findViewById(R.id.edad);
    int edad = paciente.getEdad();
    if(edad==-1){
        edadTextView.setText("--");
    }
    else {
        edadTextView.setText(Integer.toString(edad));
    }

    alturaTextView = (TextView) findViewById(R.id.altura);
    int altura = paciente.getAltura();
    if(altura==-1){
        alturaTextView.setText("--");
    }
    else{
        alturaTextView.setText(Integer.toString(altura));
    }

    pesoTextView = (TextView) findViewById(R.id.peso);
    int peso = paciente.getPeso();
    if(peso==-1){
        pesoTextView.setText("--");
    }
    else{
        pesoTextView.setText(Integer.toString(peso));
    }
}
```

Ilustración 70. Método rellenaPerfil() de la ActividadCosntantes.

A continuación, el método inicializarViews() inicializa la lista de constantes a partir de un mensaje predeterminado cuyo valor constante es:

```
{"datos":{"ppg":0,"oxigeno":0,"pulso":0}}
```

Este mensaje, junto con la clase paciente obtenida, se pasan como argumento al método getListaConstantes() de la clase ObtencionDatos, y se obtiene la lista de las constantes a monitorizar. En este caso todas valdrán 0 hasta iniciar la transmisión de datos, exceptuando el índice de masa corporal, que tendrá el valor definitivo, puesto que es el único invariante.

A continuación se pasa la lista como argumento para crear un *adapter*. Un *adapter* actúa de intermediario entre un conjunto de datos, como una lista, para crear una serie de *views* a partir de ellos, de forma que éstos sólo se carguen cuando se muestren por pantalla, con el objeto de reducir costes innecesarios de CPU.

Puesto que nosotros necesitamos que el *adapter* devuelva un elemento de la lista como el antes descrito a partir de una clase de tipo ConstanteVital, debemos crear nuestro propio *adapter*. La clase ConstantesAdapter hereda de la clase ArrayAdapter, que devuelve una *view* (por ejemplo un elemento de tipo *ListView*) a partir de un recurso que se introduzca como argumento o una lista de objetos.

```
Resources mRecursos;

public ConstantesAdapter(Context context, ArrayList<ConstanteVital> constantesVitales,
                         Resources r) {
    super(context,  resource: 0, constantesVitales);
    mRecursos = r;
}
```

Ilustración 71. Constructor de la clase ConstantesAdapter y su atributo mRecursos.

A continuación, la clase va introduciendo los atributos del objeto de la lista en cuestión en sus respectivos campos dentro de la plantilla: el icono, el nombre de la variable y su valor. El método convierte los valores en *strings* para que puedan pasarse a las *TextView* del objeto lista, discriminando el valor *double* del índice de masa corporal del resto de valores de tipo *int*. En caso de que los valores del mensaje JSON valgan 0 la *view* indicará que el valor es nulo mediante los caracteres "--".

```
@Override
public View getView(int position, @Nullable View convertView, @NonNull ViewGroup parent) {
    View elementoListaView = convertView;
    if (elementoListaView == null) {
        elementoListaView = LayoutInflater.from(getContext()).inflate(R.layout.elemento_lista,
                parent,  attachToRoot: false);

    }

    ConstanteVital constanteActual = getItem(position);

    TextView descripcion = (TextView) elementoListaView.findViewById(R.id.descr_constantes);
    descripcion.setText(constanteActual.getNombre());

    if (constanteActual.getNombre()=="Body mass index") {
        TextView valorTextView = (TextView) elementoListaView.findViewById
                (R.id.valor_constantes);
        if(constanteActual.getValorDouble()<0 || constanteActual.getValorDouble()>100){
            valorTextView.setText("--");
        }
        else {
            valorTextView.setText(String.format("%.1f", constanteActual.getValorDouble()));
        }
    }
    else {
        TextView valorTextView = (TextView) elementoListaView.findViewById
                (R.id.valor_constantes);
        if(constanteActual.getValor()==0) {
            valorTextView.setText("--");
        }
        else {
            valorTextView.setText(Integer.toString(constanteActual.getValor()));;
        }
    }

    ImageView iconoView = (ImageView) elementoListaView.findViewById(R.id.icono_constantes);
    iconoView.setImageResource(constanteActual.getIcono());

    return elementoListaView;
}
```

Ilustración 72. Método getView() de la clase ConstantesAdapter.

Una vez inicializadas las secciones del perfil y la lista vemos que los datos del perfil ocupan un espacio por encima del deseado en la pantalla. Es por tanto que se optó por hacer que esta sección fuera desplegable, para poder consultar los datos cuando el usuario requiera, y poder ocultarlos mientras no se estén consultando.

Para ello, creamos un *listener* de tipo onClick en el *layout* que contiene el texto "Perfil personal". De esta forma, al pulsar en él, hacemos que el *layout* que contiene los campos de perfil se vuelva visible o no, expandiéndose o se reduciéndose. Además añadimos un pequeño icono en forma de flecha en el título de la sección, que mantenemos siempre visible, para indicar que la pestaña se reduce o se expande. Haremos variar el icono de la flecha a través del *listener* en función los campos estén expandidos o

reducidos.

```
/*Intentamos cargar los datos del paciente creados en la actividad inicial y usamos el
adapter para mostrarlos en la listView*/

try{
    paciente = (Paciente) getIntent().getExtras().getSerializable( key: "paciente");
    rellenaPerfil(paciente);

    listaConstantesVitales = ObtencionDatos.getListaConstantes(paciente,
            MENSAJE_PREDETERMINADO, getResources());
    ConstantesAdapter adapter = new ConstantesAdapter( context: this, listaConstantesVitales,
            getResources());
    listaConstantesView = (ListView) findViewById(R.id.lista);
    listaConstantesView.setAdapter(adapter);

}
catch (Error e){
    Log.e( tag: "Cargar lista",  msg: "Error al cargar la lista de datos", e);
}


/*Creamos un listener en la vista del perfil que permitira desplegar y contraer los datos */

conectarTextView = (TextView)findViewById(R.id.conectar);
datos_perfil = (LinearLayout) findViewById(R.id.datos_perfil);
detalles_perfil = (LinearLayout) findViewById(R.id.detalles_perfil);
flecha = (ImageView) findViewById(R.id.flecha);
detalles_perfil.setVisibility(View.GONE);
flecha.setBackgroundResource(R.drawable.flecha_expandir);

datos_perfil.setOnClickListener((v) -> {
        if(detalles_perfil.getVisibility()==View.GONE) {
            detalles_perfil.setVisibility(View.VISIBLE);
            flecha.setBackgroundResource(R.drawable.flecha_contraer);
        }
        else {
            detalles_perfil.setVisibility(View.GONE);
            flecha.setBackgroundResource(R.drawable.flecha_expandir);
        }
});
```

Ilustración 73. Segmento del método inicializarViews() de la ActividadConstantes, donde se inicializan la sección desplegable del perfil y la lista de constantes a monitorizar.

Queda una última *view* por inicializar, que es el botón de conexión y desconexión. Éste depende de la existencia de un adaptador bluetooth en el dispositivo móvil. La gestión de los recursos y conexión a través de bluetooth es una tarea compleja que forma parte del grueso de la actividad, y que se irá desarrollando en las siguientes páginas.

En primer lugar, al querer establecer una conexión mediante bluetooth, debemos comprobar que nuestro dispositivo móvil es capaz de recibir y enviar datos mediante conexión bluetooth, si es una característica disponible en nuestro dispositivo, esto es, comprobar la existencia de un adaptador bluetooth. A pesar de que hoy día la gran mayoría de dispositivos cuentan con adaptador bluetooth, debemos comprobar de forma protocolaria la existencia

del mismo.

Primero creamos un objeto de tipo BluetoothAdapter llamado btAdapter, a través del método getDefaultAdapter(). En el método inicializarViews() comprobamos si el objeto es nulo, en cuyo caso deshabilitamos el *TextView* que hará las veces de botón, otorgándole un color gris y el texto "BT no disponible". En caso de que el *adapter* no sea nulo, es decir, que nuestro dispositivo sí sea compatible con tecnología bluetooth, continuamos con el proceso de inicialización del *adapter*.

```
if(btAdapter == null)
{
    conectarTextView.setBackgroundResource(R.color.colorDissabled);
    conectarTextView.setText("BT unavailable");
}
```

Ilustración 74. Comprobamos si el adaptador bluetooth no existe, y en caso afirmativo bloqueamos el botón de conexión.

Primero debemos asegurarnos que nuestra actividad detecte los posibles cambios en el estado del adaptador bluetooth de forma externa. Un ejemplo de ello sería que el usuario desactivara el bluetooth desde la pestaña desplegable de su dispositivo. Para ello debemos registrar un BroadcastReceiver. Un BroadcastReceiver es una clase que nos permitirá suscribirnos a dichos cambios externos mediante el método onReceive(), dentro del cual se identificará la naturaleza del *intent* lanzado mediante el método getAction().

Comprobamos si el *intent* lanzado se corresponde a un cambio en el estado del bluetooth, y de ser así, comprobamos a qué estado se corresponde: si el bluetooth se ha conectado, si se ha desconectado, si ha pasado a buscar dispositivos, etc. A través de un switch() discriminamos los estados que nos interesan y definimos las acciones que tendrán lugar en caso de que dichos cambios de estado se produzcan. En nuestro caso, el único cambio que nos interesa es si el bluetooth se desconecta como en el ejemplo descrito anteriormente, en cuyo caso el botón volverá a mostrar el texto de conectarse al dispositivo y se establecerá como *false* una variable atómica booleana llamada estadoConexion que en todo momento indicará el estado

de la conexión.

En adición al cambio de estado del *adapter*, se comprobará si el intent lanzado corresponde a encontrar un dispositivo. En caso afirmativo, un cuadro flotante de texto nos notificará de ello junto con el nombre del dispositivo encontrado.

```
private final BroadcastReceiver broadcastReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, @NotNull Intent intent) {
        final String accion = intent.getAction();
        final int estado = intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
                BluetoothAdapter.ERROR);

        if (BluetoothAdapter.ACTION_STATE_CHANGED.equals(accion)) {
            switch (estado) {
                case BluetoothAdapter.STATE_OFF: {
                    estadoConexion.set(false);
                    ((TextView)findViewById(R.id.conectar)).setText("Connect to device");
                    break;
                }

                default:
                    break;
            }
        }

        if(BluetoothDevice.ACTION_FOUND.equals(accion)) {
            BluetoothDevice device = (BluetoothDevice) intent.getParcelableExtra
                    (BluetoothDevice.EXTRA_DEVICE);

            Toast.makeText(getApplicationContext(), "Device found: ." + device.getName(),
                    Toast.LENGTH_SHORT).show();
        }
    }

};
```

Ilustración 75.Objeto de tipo BroadcastReceiver que definimos para nuestra actividad.

Una vez definido nuestro objeto de tipo BroadcastReceiver, debemos registrarlo a través del método registrarBroadcastReceiver(). Éste crea un filtro para discriminar los dos tipos de *intent* que vamos a valorar y registra el BroadcastReceiver.

```
private void registrarBroadcastReceiver() {

    IntentFilter filtro = new IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);
    filtro.addAction(BluetoothDevice.ACTION_FOUND);
    this.registerReceiver(broadcastReceiver, filtro);
}
```

Ilustración 76. Método registrarBroadcastReceiver() de la ActividadConstantes.

Volviendo al método inicializarViews(), tras registrar el BroadcastReceiver establecemos el color de la TextView que hará las veces de botón para indicar que el botón está habilitado y comprobamos el estado

de la conexión a través de la variable atómica booleana antes mencionada. En función de si el estado es conectado o no, mostramos el texto correspondiente en el cuadro de texto.

```
registrarBroadcastReceiver();
conectarTextView.setBackgroundResource(R.color.colorPrimary);

/* Comprobamos si el Bluetooth está activo o no y cambiamos el texto acorde a ello */

if(estadoConexion.get()) {
    conectarTextView.setText(getString(R.string.desconectar));
}
else {
    conectarTextView.setText("Connect to device");
}
```

Ilustración 77. Tras registrar el BroadcastReceiver establecemos el color y texto del botón de conexión/desconexión.

A continuación, debemos inicializar el listener de tipo onClick sobre el cuadro de texto para que haga las veces de botón. En primer lugar, El botón debe funcionar únicamente si el botón está habilitado comprobando el estado de la variable atómica booleana botonConectar. En determinados momentos de la ejecución del programa debemos procurar que el usuario no intervenga, procurando que no pueda lanzar demasiadas ordenes en un corto espacio de tiempo, infiriendo en el funcionamiento normal del programa. Esto se producirá cuando el dispositivo se encuentre conectándose al dispositivo bluetooth, proceso que, como se verá más adelante, lleva unos segundos y debemos blindar el correcto funcionamiento para impedir que se sobresature la aplicación dando lugar a fallos o exista cualquier tipo de error.

Si el botón está habilitado comprobamos el estado de la conexión a través de la variable estadoConexion. Si ésta es *false* procedemos a conectarnos al dispositivo y ejecutar el hilo de conexión. En caso contrario, acabamos con dicha conexión volviendo *false* la variable estadoConexion para terminar con el hilo de conexión, y cambiamos el texto del botón a “conectar al dispositivo”, dejando listo al botón para conectarse nuevamente. Además reiniciamos el contenido de la lista para vaciar las posibles lecturas.

Para conectarnos al dispositivo primero debemos comprobar si el adaptador está conectado o no. Todo dispositivo móvil con bluetooth permite

conectar o desconectar el adaptador, bien mediante los ajustes del sistema o bien mediante un widget o pantalla desplegable con diversas opciones: deshabilitar Wi-Fi, modo avión o habilitar bluetooth.

La activación o habilitación del adaptador bluetooth se puede realizar sin necesidad de la petición de permisos por parte del usuario, pero por cuestiones protocolarias y de seguridad, debemos solicitar al usuario una confirmación de activación. Se comprueba si el adaptador bluetooth está habilitado o no mediante el método isEnabled() y si no lo está llamamos al método conectarBt(),que no sólo debe preguntar al usuario si desea conectar el adaptador bluetooth sino que además debe comenzar el hilo de conexión.

El método conectarBt() primero comprueba que, efectivamente el adaptador no está habilitado, y después lanza un *intent* para solicitar permisos al usuario e intentar obtener resultados de una actividad, en este caso de solicitud de conexión del adaptador bluetooth.

```
private  void conectarBt(@NotNull BluetoothAdapter btAdapter) {

    /* Si el dispositivo esta apagado entonces lo encendemos, buscamos dispositivos compatibles,
    y nos conectamos de ser posible */

    if (!(btAdapter.isEnabled())) {

        try {
            /* Pedimos permisos al usuario para conectar el Bluetooth */
            Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
        } catch (Error e) {
            Log.e( tag: "Cargar lista",  msg: "Error al cargar la lista de datos", e);
        }
    }
}
```

Ilustración 78. Método conectarBt() de la ActividadConstantes.

Debemos establecer el comportamiento de la actividad en función del resultado que nos devuelva el lanzamiento de la actividad de petición de habilitación del bluetooth. Para ello escribimos en el método onResult() de la actividad el código que se ejecutará en función del resultado obtenido.

Primero filtramos el tipo de resultado que debemos modificar a través de un switch que discrimine el caso en que el argumento recibido por el método sea el de REQUEST_ENABLE_BT. Para el mismo, identificamos la

respuesta obtenida. Si el resultado es RESULT_OK, indicaremos mediante un texto flotante el inicio de búsqueda del dispositivo, deshabilitaremos el botón mediante la variable botonConectar, cambiaremos el texto del botón indicando que vamos a conectarnos y lanzamos el hilo de conexión. En caso de que el usuario deniegue los permisos se emite un mensaje flotante indicando que el usuario ha denegado la petición de habilitación bluetooth.

```
@Override
protected void onActivityResult (int requestCode, int resultCode, Intent data) {
    switch(requestCode)
    {
        case REQUEST_ENABLE_BT:
        {
            if(resultCode == RESULT_OK)
            {
                Toast.makeText(getBaseContext(), getText(("Searching for devices, wait…")),
                        Toast.LENGTH_LONG).show();
                botonConectar.set(false);
                conectarTextView.setText("Connecting…");
                EstablecerConexion hiloEstablecerConexion = new EstablecerConexion();
                hiloEstablecerConexion.start();
            }
            else
            {
                Toast.makeText( context: this, "BT activation request cancelled.",
                        Toast.LENGTH_SHORT).show();
            }
            break;
        }

        default:
            break;
    }
}
```

Ilustración 79. Método onResult() de la ActividadConstantes.

Antes de proceder con la explicación de los hilos cerraremos la descripción del método inicializarViews() y del *listener* asociado al botón de conexión. Si al evaluar el estado del *adapter* éste se muestra como habilitado, en vez de llamar al método conectarBt() procedemos de la misma forma que al obtener permisos por parte del usuario para conectar el bluetooth.

```
conectarTextView.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View pulsador) {
        if (botonConectar.get()) {                          //Si está habilitado el botón
            try {
                if (!(estadoConexion.get())) {              //Si el hilo no está activo
                    if (!btAdapter.isEnabled()) {
                        conectarBt(btAdapter);
                    } else {
                        Toast.makeText(getBaseContext(), getText((R.string.buscando)),
                                Toast.LENGTH_LONG).show();
                        botonConectar.set(false);           //Deshabilitamos el botón
                        conectarTextView.setText(getString(R.string.conectando));
                        EstablecerConexion hiloEstablecerConexion =
                                new EstablecerConexion();
                        hiloEstablecerConexion.start();
                    }
                } else {                                    //Si el hilo está activo
                    estadoConexion.set(false);
                    conectarTextView.setText(getString(R.string.conectar));

                    listaConstantesVitales = ObtencionDatos.getListaConstantes(paciente,
                            MENSAJE_PREDETERMINADO, getResources());
                    ConstantesAdapter adapter = new ConstantesAdapter
                            (getApplicationContext(), listaConstantesVitales,
                                    getResources());
                    listaConstantesView = (ListView) findViewById(R.id.lista);
                    listaConstantesView.setAdapter(adapter);
                }
            } catch (Error e) {
                e.printStackTrace();
            }
        }
    }
});
```

Ilustración 80. Listener de tipo onClick() sobre el texto que hará las veces de botón en el método ActividadConstantes.

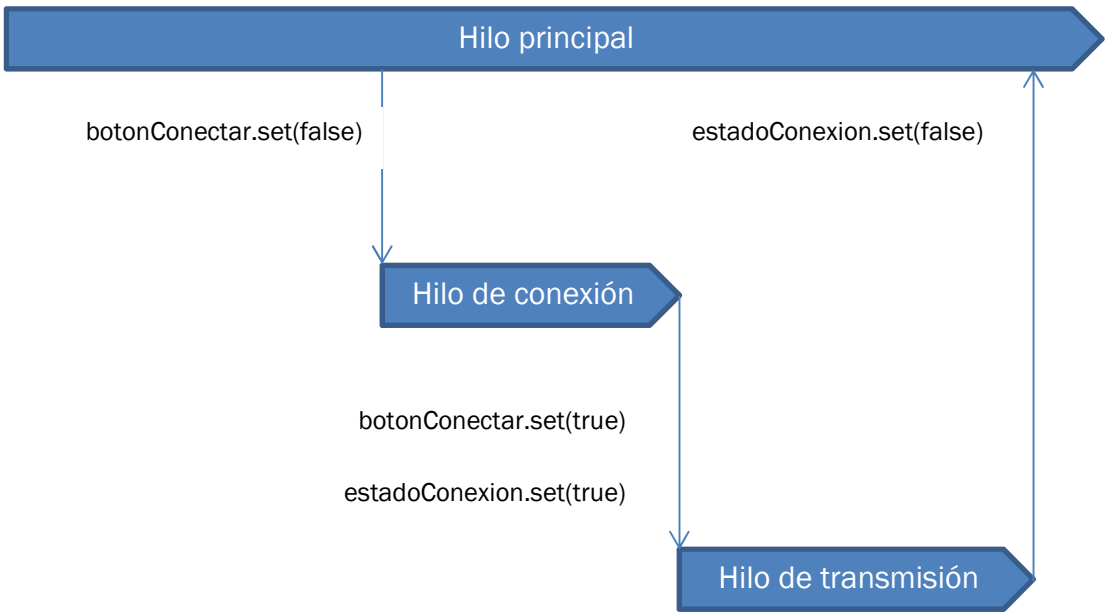
Una vez explicado el comportamiento del método inicializarViews() pasamos a detallar el comportamiento de los distintos hilos que se ejecutan en la actividad y el funcionamiento de la misma.

En primer lugar, un hilo es un tramo de un proceso o programa que se ejecuta de forma secuencial. Todo programa o proceso consta de un hilo principal, pero algunos programas requieren varios hilos paralelos que se ejecuten de forma simultánea. Por ejemplo, nuestro programa debe ser capaz de interpretar el mensaje enviado por bluetooth a la par que el programa sigue desempeñando el resto de funcionalidades con normalidad: el funcionamiento del botón de conexión/desconexión, el funcionamiento de la

pestaña desplegable del perfil, mostrar los valores recibidos en pantalla, etc.

La ActividadConstantes consta de tres hilos, como se ha mencionado anteriormente. Además del hilo principal del programa durante la ejecución del programa se lanzarán otros dos hilos paralelos. El primero es un hilo de conexión, que se lanza para que todas las operaciones para conectarse al dispositivo deseado y establecer la conexión. Inicialmente, el proceso de conexión se encontraba dentro del hilo principal pero, debido a que el proceso de conexión tarda unos segundos, su ejecución en el hilo principal bloquea el programa, congelando todas sus funcionalidades y *views*. Como norma general, ningún proceso debe bloquear el correcto funcionamiento de la aplicación. Como se ha comentado anteriormente, al lanzar el hilo, bloquearemos el uso del botón de desconexión.

El segundo hilo es el de transmisión de datos. Una vez el hilo de conexión termina y se establece la conexión de forma correcta, se lanza un segundo hilo de transmisión de datos. El hilo de transmisión se ejecuta en bucle, leyendo los datos recibidos y reconstruyendo el mensaje, comprobando que el mensaje se lee correctamente, para después pasárselo al hilo principal, que se encargará de mostrarlo por pantalla, construyendo la *ListView*.



Para crear un hilo es necesario declarar una nueva clase que herede de la clase Thread y describir su funcionamiento dentro del método run(). Para lanzar un hilo, declaramos un objeto del tipo hilo que queremos lanzar y lo ejecutamos mediante el método start().

También hay que introducir el concepto de *handler*. La clase Handler permite el envío de mensajes entre un *runnable* y el hilo principal. Por norma general, no se debe cambiar o acceder al contenido de las *views* desde otros hilos que no sean el principal. Puesto que no podemos modificar las *views* desde los hilos debemos enviar un mensaje desde el correspondiente hilo al *handler* asociado. Dicho mensaje se enviará en un paquete o *bundle*, al que se le asociará una llave para identificar el mensaje de forma correcta y evitar un posible cruce de mensajes. La actividad

- Hilo de conexión EstablecerConexion

  La clase EstablecerConexión definirá el funcionamiento del hilo de conexión. Su constructor no requiere argumentos. En primer lugar y dentro del método run() declaramos el mensaje que vamos a enviar al correspondiente *handler* asociado. Además, declaramos un conjunto o *set* de dispositivos bluetooth en el que almacenaremos los dispositivos vinculados con el dispositivo.

  Por lo general, el dispositivo pide permisos al usuario para vincular los dispositivos encontrados desde los ajustes de sistema antes de poder conectarse a ellos. Los dispositivos vinculados se guardan en el sistema y una vez que el usuario ha vinculado los dispositivos quedan registrados para su uso en usos sucesivos. De esta forma, antes de conectarnos al dispositivo debemos entrar en ajustes del sistema, reconocer el módulo bluetooth al que vamos a conectarnos (y al que se ha cambiado el nombre para facilitar su identificación) y vincularlo con nuestro dispositivo para que la

aplicación pueda identificarlo.

Almacenamos en el conjunto o set todos los dispositivos vinculados. Para ello, primero debemos asegurarnos de que el dispositivo este vacío, para que la lista de dispositivos vinculados coincida con los que se encuentran vinculados en el momento de la conexión. Recorremos el conjunto y, por cada objeto invocamos a un método para hacer nulo cada uno de ellos.

```
Message msg = handlerEstadoConexion.obtainMessage();
Bundle bundle = new Bundle();

/*Cada vez que llamemos a la función, vaciaremos el set de dispositivos para volver a
buscarlos, en caso de que el set esté vacío*/

if (listaDispositivos != null) {        //Comprobamos si es null antes de ver su tamaño
    if (listaDispositivos.size() > 0) {
        for (BluetoothDevice dispositivo : listaDispositivos) {
            try {
                Method m = dispositivo.getClass().getMethod( name: "eliminarVinculos",
                        (Class[]) null);
                m.invoke(dispositivo, (Object[]) null);

            } catch (Exception e) {
                Log.e( tag: "Vaciado de lista",  msg: "Error en el vaciado de la lista.");
            }
        }
    }
}
```

Ilustración 81. Declaración del mensaje asociado al *handler* y vaciado del *set* de dispositivos.

La conexión mediante bluetooth requerirá la creación de un socket de conexión. Antes de continuar con el proceso de conexión debemos eliminar cualquier socket existente para permitir una correcta creación del mismo.

```
if (btSocket != null) {
    try {
        btSocket.close();
    } catch (IOException e) {
        Log.e( tag: "Cierre de socket",  msg: "Error al cerrar el socket.", e);
    }
}
```

Ilustración 82. Cierre del socket de conexión.

A continuación guardamos la lista de dispositivos vinculados en nuestro conjunto, ahora vacío, mediante el método getBondedDevices(). Si el conjunto no está vacío lo recorremos, y por cada elemento verificamos si su dirección MAC coincide con la de nuestro módulo. En caso afirmativo creamos un socket con el dispositivo mediante el método creaBluetoothSocket(). En caso de falle la creación del socket se mostrará un mensaje flotante indicando que ha habido un error en el proceso.

```
private BluetoothSocket creaBluetoothSocket(BluetoothDevice dispositivo) throws IOException {

    return dispositivo.createRfcommSocketToServiceRecord(MI_UUID);
}
```

Ilustración 83. Método creaBluetoothSocket() de la ActividadConstantes.

```
for (BluetoothDevice dispositivo : listaDispositivos) {
    if (dispositivo.getAddress().equals(direccion)) {
        try {
            btSocket = creaBluetoothSocket(dispositivo);
        }catch (IOException e) {
            Toast.makeText(getBaseContext(), "Error at socket creation.",
                    Toast.LENGTH_LONG).show();
        }
        break;
    }
}
```

Ilustración 84. Buscamos el dispositivo con nuestra dirección MAC y creamos el socket.

Si el socket se ha creado con éxito y no es nulo, procedemos a conectarnos mediante el método connect(). Este método tarda unos segundos, y es el motivo principal por el que se trasladó a un hilo diferente del principal. Si el proceso de conexión falla cerramos el socket.

Si hemos conseguido conectarnos establecemos creamos y lanzamos el hilo de transmisión de datos. A continuación mandamos un mensaje al *handler* con el texto que sustituirá al texto del botón de conexión, que en este punto informa de que se está conectando al dispositivo.

En caso de que la lista esté vacía al no encontrar el dispositivo con

la dirección MAC deseada o no se pueda conectar al dispositivo, se enviará un mensaje al *handler* cambiando el texto del botón a través del método ConexionFallida().

```
if (btSocket != null) {
    try {
        btSocket.connect();
        Log.d( tag: "Conectado",  msg: "Conectado: " + btSocket.isConnected());
    } catch (IOException e) {
        try {
            btSocket.close();
            ConexionFallida(bundle, msg);
        } catch (IOException e2) {
            Log.e( tag: "Cierre de socket",  msg: "Error al cerrar el socket.", e2);
        }
    }

    if (btSocket.isConnected()) {
        HiloTransmisionDatos hiloTransmisionDatos = new HiloTransmisionDatos
                (btSocket);
        hiloTransmisionDatos.start();
        bundle.putString("llaveConexion", getString(R.string.desconectar));
        msg.setData(bundle);
        handlerEstadoConexion.sendMessage(msg);
    }
    else {
        ConexionFallida(bundle, msg);
    }
}else {
    ConexionFallida(bundle, msg);
}
```

Ilustración 85. Proceso de conexión del hilo de conexión.

```
private void ConexionFallida(Bundle bundle, Message msg){
    bundle.putString("llaveConexion", "Connect to device");
    estadoConexion.set(false);
    msg.setData(bundle);
    handlerEstadoConexion.sendMessage(msg);
}
```

Ilustración 86. Método ConexiónFallida() de la ActividadConstantes.

El handlerEstadoConexion se inicializa en el método onCreate() de la actividad. Éste recibe el *bundle* con su correspondiente llave enviado desde el hilo. El mensaje enviado a través del *bundle* contiene el texto que debe mostrar el botón en función de si se ha podido establecer la conexión o no. Volvemos a habilitar el botón y establecemos el texto en la *TextView* del botón.

```
handlerEstadoConexion = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        Bundle bundle = msg.getData();
        String estado = bundle.getString( key: "llaveConexion");
        if(estado=="Connect to device"){
            Toast.makeText(getBaseContext(), "Compatible devices not found.",
                    Toast.LENGTH_LONG).show();
        }
        conectarTextView.setText(estado);
        botonConectar.set(true);                        //Volvemos a habilitar el botón
    }
};
```

Ilustración 87. *Handler* del hilo de conexión de la ActividadConstantes.

- HiloTransmisionDatos

  En primer lugar, el constructor de la clase debe contener como argumentos el socket de conexión creado, que se guardará como atributo de la clase.

```
HiloTransmisionDatos(BluetoothSocket socket){
    mSocket = socket;
}
```

Ilustración 88. Constructor de la clase HiloTransmisionDatos.

Un segundo atributo de la clase es el correspondiente al flujo de datos que recibiremos a través del socket de conexión, un objeto de la clase InputStream. Dentro del método run() se asociará ese flujo de datos creado al flujo de datos recibido a través del socket que hemos pasado como argumento al crear el hilo mediante el uso del método getInputStream().

A continuación se establece en true la variable estadoConexion y se ejecuta un bucle que se ejecutará mientras la variable estadoConexión sea *true* y no se interrumpa de alguna forma el contexto del hilo. Si el flujo de datos es nulo, salimos del bucle, puesto que no existe la transmisión de datos hacia la aplicación.

El objetivo es extraer el mensaje en formato JSON a partir del flujo recibido. Se guarda en una cadena de caracteres el valor devuelto por el método leer(). El método leer() crea un buffer de bytes y lo

asocia con el flujo de datos que hemos creado. Tomamos del buffer una cadena, indicando que la cadena de bytes se desencriptará siguiendo un código ASCII, desde el inicio de la cadena hasta el final. El método devuelve la cadena resultante.

```
private String leer(){
    String s = "";

    try {
        byte[] buffer = new byte[1024];
        int bytes = mFlujoEntrada.read(buffer);
        s = new String(buffer,  charsetName: "ASCII");
        s = s.substring(0, bytes);
    }catch (IOException e){

    }
    return s;
}
```

Ilustración 89. Método leer() de la clase HiloTransmisionDatos.

Añadimos la cadena devuelta al mensaje ya existente y llamamos al método construirMensaje(). En primer lugar creamos un índice en el mensaje para encontrar el primer delimitador de la cadena. El delimitador es el carácter que dividirá cada uno de los mensajes que queremos obtener, en nuestro caso es un valor constante, siendo éste el carácter de retorno de carro '\n'. Si no se encuentra el carácter salimos del método para seguir añadiendo más cadenas al actual mensaje a partir del buffer hasta encontrar un delimitador.

Extraemos del mensaje desde el inicio del mismo hasta el índice encontrado en una cadena auxiliar, y luego actualizamos el valor del mensaje a partir del índice, de forma que eliminamos la cadena extraída.

Si contiene el inicio del mensaje JSON, puesto que la cadena se ha extraído hasta el carácter de retorno de carro, significa que la cadena se ha extraído de forma completa y correcta. En ese caso enviamos la cadena en el formato JSON deseado a través del *handler* mediante el método enviaHandlerConstantes(), que creará un *bundle* con el mensaje a enviar al HandlerConstantes y su

correspondiente llave. El método construirMensaje() se llama a si mismo nuevamente para comprobar si en el mensaje sigue habiendo otra cadena extraíble, existiendo otro índice en la misma. El método se llama a si mismo hasta que no quede ninguna cadena que extraer del mensaje, reiniciándose el bucle del hilo para recibir más datos del flujo establecido.

```
private void enviaHAndlerConstantes(String s) {

    Message msg = Message.obtain();
    Bundle bundle = new Bundle();
    bundle.putString("llaveConstantes", s);
    msg.setData(bundle);
    handlerConstantes.sendMessage(msg);
}
```

Ilustración 90. Método enviaHandlerConstantes() del HiloTransmisionDatos.

```
private void construirMensaje() {

    //Buscamos el primer delimitador que haya en el buffer
    int indice = mensaje.indexOf(DELIMITADOR);

    //Si no hay ninguno salimos
    if (indice == -1) {
        return;
    }

    //Construimos el mensaje completo
    String s = mensaje.substring(0, indice);

    //Eliminamos el mensaje del buffer
    mensaje = mensaje.substring(indice + 1);

    //Si empieza correctamente es que el json esta entero y lo enviamos
    if(s.contains("{\"datos\":")){
        enviaHAndlerConstantes(s);
    }

    //Seguimos buscando
    construirMensaje();
}
```

Ilustración 91. Método construirMensaje() del HiloTransmisionDatos.

```
@RequiresApi(api = Build.VERSION_CODES.N)
public void run() {

    estadoConexion.set(true);

    try{
        mFlujoEntrada = mSocket.getInputStream();
    }
    catch(IOException e){
        Log.e( tag: "Flujo de entrada",  msg: "Error al crear el flujo de entrada", e);
    }

    while (estadoConexion.get() && !this.isInterrupted() && btSocket.isConnected()) {

        if (mFlujoEntrada == null){
            break;
        }

        String s = leer();

        if (s.length() > 0)
            mensaje += s;

        construirMensaje();

    }
    if(!btSocket.isConnected() || !estadoConexion.get()){
        ((TextView)findViewById(R.id.conectar)).setText("Connect to device");
        estadoConexion.set(false);
        if (mFlujoEntrada != null) {
            try {mFlujoEntrada.close();} catch (Exception e) { e.printStackTrace(); }
        }
    }
}
```

Ilustración 92. Método run() del hilo de transmisión de datos.

Finalmente, el *handler* asociado al hilo de transmisión de datos toma el mensaje del bundle recibido por el hilo. A continuación se obtiene la lista de constantes a partir del mensaje JSON recibido y se asocia al ConstantesA*dapter* de la ListView como se vio anteriormente.

```
handlerConstantes = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        Bundle bundle = msg.getData();
        String json = bundle.getString( key: "llaveConstantes");

        listaConstantesVitales = ObtencionDatos.getListaConstantes(paciente, json,
                getResources());
        ConstantesAdapter adapter = new ConstantesAdapter(getApplicationContext(),
                listaConstantesVitales, getResources());
        listaConstantesView = (ListView) findViewById(R.id.lista);
        listaConstantesView.setAdapter(adapter);
    }
};
```

Ilustración 93. *Handler* del hilo de transmisión de datos de la ActividadConstantes.

Además, debemos liberar el broadcastReceiver y el socket dentro del método onDestroy() de la ActividadConstantes para asegurarnos de que la conexión se cierra al cerrar la aplicación.

```
@Override
public void onDestroy() {
    super.onDestroy();
    this.unregisterReceiver(broadcastReceiver);

    if (btSocket!=null) {
        try {
            btSocket.close();
        }
        catch (IOException e){
            Log.e( tag: "Cierre de socket", msg: "Error al cerrar el socket.", e);
        }
    }
}
```

Ilustración 94. Método onDestroy() de la ActividadConstantes.

Ilustración 95. Vista de la ActividadConstantes durante el proceso de transmisión de datos.

5.5. Otros ficheros

Además de las clases y actividades mencionadas anteriormente, la aplicación se compone de otros ficheros que se comentarán brevemente a continuación.

En primer lugar en el fichero XML AndroidManifest se establecerá el nombre de la aplicación, las actividades que componen la aplicación, los permisos que requerirá la aplicación como en nuestro caso los permisos de bluetooth, o el icono que la aplicación mostrará en su acceso directo.

A cada actividad se le asocia una plantilla o *layer* con la disposición y configuración de cada *view* en lenguaje XML. Además de disponer todas las *views* en el espacio, se puede variar el color de las mismas, su tamaño, estilo, añadir efectos como ondas al pulsar sobre ellas, etc. Para las *views* cuyo

contenido variará en función de los datos del programa se añadirá un identificador a las mismas para poder referenciarlas como recursos desde sus respectivas actividades asociadas. Además, los atributos a cambiar (texto, color, fondo…) se definirán como *tools* para permitir su modificación, estableciendo un valor sustituible o *placeholder*.

Se incluyen además dos carpetas dentro de recursos con los distintos archivos de imagen requeridos o *drawables*. En ellas se encuentran los iconos empleados para cada uno de los elementos de la lista de las variables a monitorizar y el icono de la flecha del desplegable del perfil personal de la ActividadConstantes. Cada imagen se encuentra repetida para las diferentes resoluciones posibles del dispositivo.

Dentro de la carpeta values se incluirán otros dos ficheros XML para códigos de colores y estilos, de forma que cada actividad dispondrá de una serie de colores y estilos en función de los definidos en estos ficheros, y podremos referenciar cada color como recurso por medio de su etiqueta.

Finalmente, las cadenas que se muestran al usuario en la aplicación tales como textos o información, en vez de escribirse directamente en el código del programa se han definido como valores referenciables desde recursos. Dentro de la carpeta values y a su vez dentro de la carpeta strings, se crearán varios ficheros, uno por cada idioma que acepte nuestra aplicación. Para ello se creará un fichero con su determinado código de idioma. Para cada fichero XML se definirá cada cadena a través de un identificador y su cadena en su respectivo idioma. Así, al emplear estas cadenas desde el código del programa se referenciará a ellas a partir de dicho identificador, siendo el resultado que se muestra por pantalla dependiente de la configuración de idiomas de nuestro dispositivo. En Android Studio, el idioma predeterminado es el inglés.

```
<resources>
    <string name="app_name">Vital Signs</string>
    <string name="visualizacion">Signs viewer</string>
    <string name="perfil">Personal profile</string>
    <string name="nombre">Name</string>
    <string name="edad">Age</string>
    <string name="altura">Height (cm)</string>
    <string name="peso">Weight (kg)</string>
    <string name="sin_nombre">No name</string>
    <string name="continuar">Continue</string>
    <string name="conectar">Connect to device</string>
    <string name="conectando">Connecting…</string>
    <string name="desconectar">Disconnect</string>
    <string name="imc">Body mass index</string>
    <string name="ppg">PPG</string>
    <string name="oxigeno">Oxygen in blood</string>
    <string name="ppm">Beats per minute</string>
    <string name="no_compatibles">Compatible devices not found.</string>
    <string name="peticion_cancelada">BT activation request cancelled.</string>
    <string name="BT_no_disponible">BT unavailable</string>
    <string name="buscando">Searching for devices, wait…</string>
    <string name="error_socket">Error at socket creation.</string>
    <string name="encontrado">Device found: .</string>

</resources>
```

```
<resources>
    <string name="app_name">Constantes Vitales</string>
    <string name="visualizacion">Visualización de Constantes</string>
    <string name="perfil">Perfil personal</string>
    <string name="nombre">Nombre</string>
    <string name="edad">Edad</string>
    <string name="altura">Altura (cm)</string>
    <string name="peso">Peso (kg)</string>
    <string name="sin_nombre">Sin nombre</string>
    <string name="continuar">Continuar</string>
    <string name="conectar">Conectarse al dispositivo</string>
    <string name="conectando">Conectando…</string>
    <string name="desconectar">Desconectar</string>
    <string name="imc">Índice de masa corporal</string>
    <string name="ppg">PPG</string>
    <string name="oxigeno">Oxígeno en sangre</string>
    <string name="ppm">Pulsaciones por minuto</string>
    <string name="no_compatibles">No se han encontrado dispositivos compatibles.</string>
    <string name="peticion_cancelada">Petición de activación BT cancelada.</string>
    <string name="BT_no_disponible">BT no disponible</string>
    <string name="buscando">Buscando dispositivos, espere un momento…</string>
    <string name="error_socket">Error en la creación del socket.</string>
    <string name="encontrado">Dispositivo encontrado: .</string>

</resources>
```

Ilustración 96.Código correspondiente al fichero de recursos de *strings* en inglés (arriba) y su homólogo en castellano (abajo).

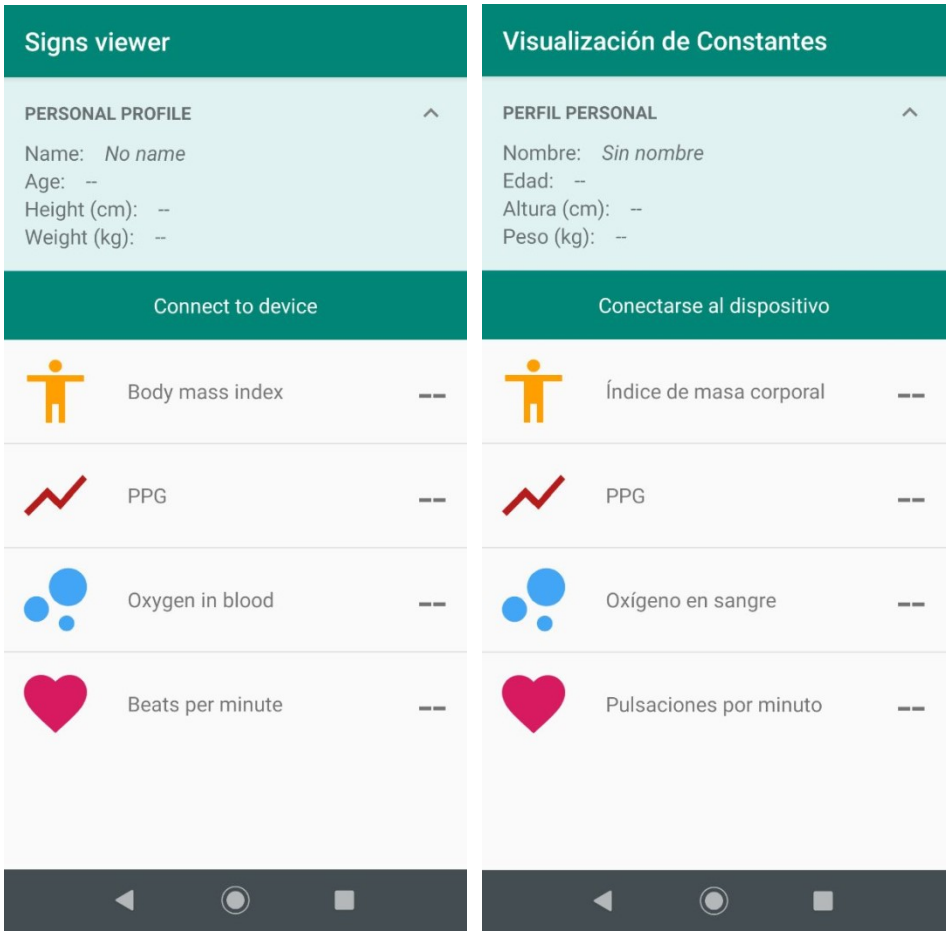


Ilustración 97. Versiones de la aplicación en inglés (izquierda) y español (derecha).

6. Conclusiones

Se han logrado los objetivos propuestos al inicio de esta memoria, se ha diseñado un dispositivo portable capaz de tomar mediciones de algunos signos vitales y transmitirlos mediante bluetooth a una aplicación Android externa. Dicha aplicación ha sido diseñada para monitorizar de forma correcta las variables que hemos tomado de nuestro dispositivo.

Uno de los principales obstáculos del desarrollo ha sido el de la adecuación de la señal para su medición. Puesto que el rango de lecturas es muy pequeño y la señal puede presentar grandes variaciones entre dos mediciones distintas, adecuar el programa y el sensor para garantizar una

lectura óptima de los niveles de oxígeno en sangre es, en todo caso, una tarea complicada debido a la naturaleza de las magnitudes a medir.

Sin embargo, este problema persiste en el sensor comercial, que muestra valores dispares a la hora de medir el nivel de oxígeno en sangre, o bien por debajo de los valores normales, o bien varían muy rápidamente en poco tiempo.

Es por tanto que cotejar los valores de nuestro dispositivo con los del oxímetro comercial que hemos tomado de referencia sea un método impreciso para determinar unos valores estrictamente precisos.

El hecho de basar nuestro proyecto en la tecnología del Arduino DUE limita el desarrollo del circuito. Hoy en día la existencia de dispositivos portables de pequeño tamaño como smartwach sugieren la posibilidad de diseñar nuestro dispositivo en un tamaño más reducido, pero dejando de lado el microcontrolador empleado, cuya capacidad y cantidad de puertos sobreexceden las necesidades de nuestro dispositivo.

Por otra parte, se ha diseñado el conjunto de forma que sea sencillo ampliar las funcionalidades del dispositivo si se quisiera, siendo sencillo añadir las funcionalidades al código del microcontrolador y al resto del circuito, y facilitando la adición de más constantes de forma sencilla gracias al empleo de clases y la normalización del mensaje de transmisión como se ha comentado en anteriores apartados, sin necesidad de editar el resto del programa Android.

El habernos decantado por fabricar un sensor más ergonómico, flexible y reducido implica una dificultad añadida, puesto que reduce la estabilidad del sensor frente a un sensor sólido y aislado como el de pinza comercial. Las pequeñas variaciones en su ajuste, así como la naturaleza dactilar de cada paciente pueden dar como resultado variaciones muy grandes en cuanto a la tensión medida a la salida del amplificador de transimpedancia. A esto se suma la problemática del rango reducido de lecturas de tensión.

Como mejora adicional, se puede modificar el programa Android para que identifique nuestro dispositivo mediante una búsqueda en vez de hacerlo a través de la dirección MAC proporcionada al código del programa. Esto permitiría al usuario detectar el dispositivo sin necesidad de conocer su dirección de antemano, y permitiendo mayor versatilidad de la aplicación frente a un dispositivo cuya dirección MAC no coincida con la introducida en el código. Como ejemplo, si el dispositivo se comercializara, cada uno de ellos se identificaría con una dirección MAC diferente, pero a cada módulo puede asignársele el mismo nombre. De esta forma, un usuario podría realizar una búsqueda de dispositivos bluetooth para encontrar su dispositivo a través del nombre.

Otra posible mejora es la de adición de usuarios, o permitir monitorizar varios dispositivos simultáneamente para, por ejemplo, monitorizar las constantes de varios pacientes dentro de un entorno médico.

Algunas otras posibles mejoras incluirían la adición de avisos mediante llamadas, alertas o correo electrónico a través de *intents*. Se puede incluir el registro de valores y guardar los datos en tablas o gráficas para estudiar la evolución de las constantes a lo largo del tiempo, permitir visualizar dichas gráficas al usuario, el envío de las mismas al personal sanitario cualificado, etc.

6.1. Errores conocidos

A continuación se enumeran una serie de errores conocidos.

- La aplicación Android interrumpía su funcionamiento al permitir el giro de pantalla. Puesto que, por motivos de diseño y funcionalidad, la aplicación se visualiza mejor desde su posición vertical (la lista de constantes vitales se despliega verticalmente una debajo de otra) se optó a impedir el giro de pantalla al modo horizontal desde su fichero manifest.

- En el programa del microcontrolador, para los valores que se emplean para tomar los valores temporales mediante las funciones millis() y micros() se han empleado variables de tipo unsigned long, de 32 bits. Esto permite almacenar un valor de hasta 4.294.967.295, o dicho de otra forma, hasta unos 50 días empleando millis() y unas 70 horas empleando micros(). El uso de delay() en un programa no es recomendable, puesto que interrumpe el programa. Igualmente, se pensó en el empleo de interrupciones para el programa, pero éstas no deben contener instrucciones muy complejas, y por tanto se descartó la opción.

  No obstante, existen métodos para solventar el sobrepaso de tamaño disponible u *overflow*, como el empleado en nuestro programa, que consiste en tomar diferencias entre lapsos de tiempo con millis() y micros() en vez de valores absolutos dentro de los bucles. De esta forma, el bucle dura mientras la diferencia de tiempo entre el actual y el tomado justo antes del bucle sea menor que la deseada. Si la variable llega a su desbordamiento durante el transcurso del bucle tomará un valor menor que el inicial, resultando un valor negativo que, al estar trabajando con variables de tipo unsigned long (millis() y micros() devuelven valores de tipo unsigned long), desbordará la inecuación, saliendo del bucle. De esta forma evitamos que la aplicación falle debido al *overflow*.

  No obstante y, puesto que trabajamos con mediciones de tiempo para hallar máximos y mínimos y el cálculo de las pulsaciones por minuto, el resultado de las mediciones puede alterarse ligeramente al sobrepasar los límites de almacenamiento de las variables. A pesar de ello y, dado que las mediciones se calculan a partir de la media de las tres últimas pulsaciones, el error no debería influir significativamente en el resultado y estableciéndose nuevamente a su valor correcto tras tres pulsos.

- El módulo bluetooth distorsiona las ondas de la señal en forma de onda cuadrada de forma períodica de forma eventual. Bien por un consumo de tensión del módulo, por una conexión insuficiente en la *protoboard* o por una inferencia al emplear dos puertos serie, uno para transmitir al PC y otro para transmitir al módulo, la presencia de esta distorsión altera el resultado de las mediciones. Si bien es cierto que un ajuste del módulo en la *protoboard* o un reinicio del sistema elimina el efecto indeseado, o a veces simplemente el propio efecto desaparece de forma independiente, el comportamiento inestable del módulo es algo que debe solucionarse. Para ello se ha intentado suministrar al módulo a través de una vía independiente y lo más cercana a la fuente posible en el diseño de la placa, intentando garantizar así un suministro de tensión estable.

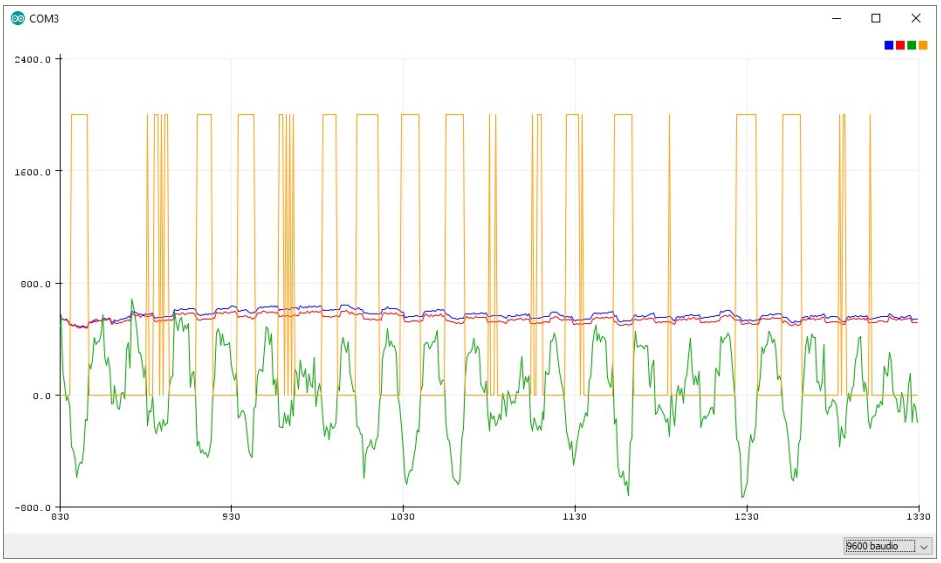


Ilustración 98. Fallo en la señal a consecuencia de la interferencia del módulo bluetooth.

Anexos

1. Código del microcontrolador

1.1. ModoContinuo

```
#define B_RATE_RAPIDO 115200
#define B_RATE_LENTO 9600

#define SENSIBILIDAD 2400

#define LED_IR 23
#define LED_R 24
#define Vo0 A0
#define Vo1 A1
#define Vo2 A2

int led_prueba = LED_IR;
float lectura[3];

////////////////////////////////////////////////////////////////////////////
void setup()
{
  Serial.begin(B_RATE_LENTO);

  pinMode(LED_IR, OUTPUT);
  pinMode(LED_R, OUTPUT);
  pinMode(Vo0, INPUT);
  pinMode(Vo1, INPUT);
  pinMode(Vo2, INPUT);

  analogReference(AR_DEFAULT);          //Referencia de AnalogRead a 3.3V
  analogWriteResolution(10);            //10 bits (0-1023) para valores de tensión
de salida
  analogReadResolution(12);             //12 bits (0-4095) para lectura de tensión

  digitalWrite(LED_IR, LOW);
  digitalWrite(LED_R, LOW);

  digitalWrite(led_prueba, HIGH);
}


////////////////////////////////////////////////////////////////////////////
void loop()
{
  lectura[0] = Lectura_Vo0();
  lectura[1] = Lectura_Vo1();
  lectura[2] = Lectura_Vo2();

  Envia_Datos();
}
```

```
////////////////////////////////////////////////////////////////////////////
void Envia_Datos()
{
  Serial.print(lectura[0]);
  Serial.print(",");
  Serial.print(lectura[1]);
  Serial.print(",");
  Serial.println(lectura[2]);
}


////////////////////////////////////////////////////////////////////////////
float Lectura_Vo0()
{
  return (analogRead(Vo0) * 3.3 * 1000 / 4095);
}

////////////////////////////////////////////////////////////////////////////
float Lectura_Vo1()
{
  return (analogRead(Vo1) * 3.3 * 1000 / 4095);
}

////////////////////////////////////////////////////////////////////////////
float Lectura_Vo2()
{
  return (analogRead(Vo2) * 3.3 * 1000 / 4095);
}
```

1.2. ModoPulsos1LED

```
#define B_RATE_RAPIDO 115200
#define B_RATE_LENTO 9600

#define T_SERVICIO 100

#define LED_IR 23
#define LED_R 24
#define Vo0 A0
#define Vo1 A1
#define Vo2 A2

unsigned long t_ciclo;
int lectura[3];

////////////////////////////////////////////////////////////////////////////
void setup()
{
  Serial.begin(B_RATE_LENTO);

  pinMode(LED_IR, OUTPUT);
  pinMode(LED_R, OUTPUT);
  pinMode(Vo0, INPUT);
  pinMode(Vo1, INPUT);
  pinMode(Vo2, INPUT);

  analogReference(AR_DEFAULT);                    //Referencia de AnalogRead a 3.3V
  analogWriteResolution(10);                      //10 bits (0-1023) para valores de
tensión de salida
  analogReadResolution(12);                       //12 bits (0-4095) para lectura de
tensión

  digitalWrite(LED_R, LOW);
  digitalWrite(LED_IR, LOW);
}

////////////////////////////////////////////////////////////////////////////
void loop()
{
  t_ciclo=micros();

  Pulso_LED(LED_IR);
  while((micros()-t_ciclo)<1000)              //PERIODO
    continue;

  Envia_Datos();
}


////////////////////////////////////////////////////////////////////////////
void Pulso_LED(int led)
```

```
{
  unsigned long t=micros();

  digitalWrite(led, HIGH);

  while((micros()-t)<T_SERVICIO)                    //TIEMPO DE SERVICIO ON
  {
    lectura[0]=analogRead(Vo0);
    lectura[1]=analogRead(Vo1);
    lectura[2]=analogRead(Vo2);
  }

  digitalWrite(led, LOW);
}


////////////////////////////////////////////////////////////////////////////
void Envia_Datos()
{
  Serial.print(lectura[0]*3.3*1000/4095);
  Serial.print(",");
  Serial.print(lectura[1]*3.3*1000/4095);
  Serial.print(",");
  Serial.println(lectura[2]*3.3*1000/4095);
}


////////////////////////////////////////////////////////////////////////////
float Lectura_Vo0()
{
  return(analogRead(Vo0)*3.3*1000/4095);
}

////////////////////////////////////////////////////////////////////////////
float Lectura_Vo1()
{
  return(analogRead(Vo1)*3.3*1000/4095);
}

////////////////////////////////////////////////////////////////////////////
float Lectura_Vo2()
{
  return(analogRead(Vo2)*3.3*1000/4095);
}
```

1.3. ModoPulsos2LED

```
#define B_RATE_RAPIDO 115200
#define B_RATE_LENTO 9600

#define T_SERVICIO 200

#define LED_IR 23
#define LED_R 24
#define Vo0 A0
#define Vo1 A1
#define Vo2 A2

unsigned long t_ciclo;
float lectura_R[3]={0,0,0};
float lectura_IR[3]={0,0,0};

////////////////////////////////////////////////////////////////////////////
void setup()
{
  Serial.begin(B_RATE_LENTO);

  pinMode(LED_IR, OUTPUT);
  pinMode(LED_R, OUTPUT);
  pinMode(Vo0, INPUT);
  pinMode(Vo1, INPUT);
  pinMode(Vo2, INPUT);

  analogReference(AR_DEFAULT);        //Referencia de AnalogRead a 3.3V
  analogWriteResolution(10);           //10 bits (0-1023) para valores de tensión
de salida
  analogReadResolution(12);           //12 bits (0-4095) para lectura de tensión

  digitalWrite(LED_IR, LOW);
  digitalWrite(LED_R, LOW);
}


////////////////////////////////////////////////////////////////////////////
void loop()
{
  t_ciclo=micros();

  Pulso_LED(LED_IR);
  while((micros()-t_ciclo)<500)
    continue;

  Pulso_LED(LED_R);
  while((micros()-t_ciclo)<1000)
    continue;
```

```
  Envia_Datos(LED_R);
}


////////////////////////////////////////////////////////////////////////////
void Pulso_LED(int led)
{
  unsigned long t=micros();
  int v0, v1, v2;

  digitalWrite(led, HIGH);

  while((micros()-t)<T_SERVICIO)
  {
      v0=analogRead(Vo0);
      v1=analogRead(Vo1);
      v2=analogRead(Vo2);
  }

  digitalWrite(led, LOW);

  if(led==LED_IR)
    {
      lectura_IR[0]=v0*3.3*1000/4095;
      lectura_IR[1]=v1*3.3*1000/4095;
      lectura_IR[1]=v2*3.3*1000/4095;
    }
    else
    {
      lectura_R[0]=v0*3.3*1000/4095;
      lectura_R[1]=v1*3.3*1000/4095;
      lectura_R[2]=v2*3.3*1000/4095;
    }
}


////////////////////////////////////////////////////////////////////////////
void Envia_Datos(int led)
{
  // lectura=(V0_IR, V1_IR, V0_R, V1_R)

      //Salida antes de los filtros
      Serial.print(lectura_IR[0]);
      Serial.print(",");
      Serial.print(lectura_R[0]);
      Serial.print(",");

      //Salida despues de los filtros
      Serial.print(lectura_IR[1]);
      Serial.print(",");
      Serial.print(lectura_R[1]);
      Serial.print(",");
      Serial.print(lectura_IR[2]);
      Serial.print(",");
      Serial.println(lectura_R[2]);
```

```
  }


////////////////////////////////////////////////////////////////////////////
float Lectura_Vo0()
{
  return(analogRead(Vo0)*3.3*1000/4095);
}

////////////////////////////////////////////////////////////////////////////
float Lectura_Vo1()
{
  return(analogRead(Vo1)*3.3*1000/4095);
}

////////////////////////////////////////////////////////////////////////////
float Lectura_Vo2()
{
  return(analogRead(Vo2)*3.3*1000/4095);
}
```

1.4. Programa principal

```
#include <ArduinoJson.h>

#define HC06 Serial3

#define B_RATE_RAPIDO 115200
#define B_RATE_LENTO 9600

#define SENSIBILIDAD_ALTA 2400
#define SENSIBILIDAD_BAJA 400

#define SIZE_VECTOR_MEDIA 10
#define SIZE_VECTOR_PENDIENTE 10

#define LED_IR 23
#define LED_R 24

#define Vo0 A0

bool pulsoAnterior;
bool pulso = false;
int ppm;
int oxigeno;
int duracion_pulso = 0;
int vectorPulsaciones[SIZE_VECTOR_MEDIA] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
unsigned long t_ciclo, t_muestreo, comienzo_pulso;
float V_IR, V_R, pendiente, limiteDeteccion, R;
float minimo_pendiente = 0;
float vector[SIZE_VECTOR_PENDIENTE] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
float extremos_IR[2] = {2400, 0};
float extremos_R[2] = {2400, 0};
float vectorR[SIZE_VECTOR_MEDIA] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

//Declaramos el mensaje con formato json

StaticJsonDocument<200> mensaje;
JsonObject datos = mensaje.createNestedObject("datos");


//////////////////////////////////////////////////////////////////////////
void setup()
{
  while (!Serial) continue;       //Esperamos a que se inicialice el puerto serie
  while (!HC06) continue;         //Esperamos a que se inicialice el puerto serie3
del bluetooth
```

```
  delay(1000);
  Serial.begin(B_RATE_LENTO);
  HC06.begin(B_RATE_RAPIDO);

  pinMode(LED_IR, OUTPUT);
  pinMode(LED_R, OUTPUT);

  pinMode(Vo0, INPUT);

  analogReference(AR_DEFAULT);    //Referencia de AnalogRead a 3.3V
  analogWriteResolution(10);       //10 bits (0-1023) para valores de tensión de
salida
  analogReadResolution(12);       //12 bits (0-4095) para lectura de tensión

  digitalWrite(LED_IR, LOW);
  digitalWrite(LED_R, LOW);


  //Inicializamos el mensaje con formato json

  datos["ppg"] = 0;
  datos["oxigeno"] = 0;
  datos["pulso"] = 0;
}


////////////////////////////////////////////////////////////////////////////
void loop()
{
  if (Check())
  {
    do
    {
      long unsigned ciclo = millis();
      while (millis() - ciclo < 1000)
      {
        t_muestreo=millis();
        Ciclo();
      }
      limiteDeteccion = minimo_pendiente * 1 / 3;
      minimo_pendiente = 0;
      enviaBluetooth();
    }
    while ((V_R < SENSIBILIDAD_ALTA) && (V_IR < SENSIBILIDAD_ALTA) && (V_R >
SENSIBILIDAD_BAJA) && (V_IR > SENSIBILIDAD_BAJA));      //Hasta que alguna de las
lecturas se salga del umbral de sensibilidad
```

```
    digitalWrite(LED_R, LOW);                    //Apagamos los dos leds
    digitalWrite(LED_IR, LOW);
  }
  else
  {
    long unsigned t = millis();

    for (int i = 0; i < SIZE_VECTOR_MEDIA; i++) //Reiniciamos los vectores
    {
      vectorPulsaciones[i] = 0;
      vectorR[i] = 0;
    }

    datos["ppg"] = 0;
    datos["oxigeno"] = 0;
    datos["pulso"] = 0;

    //enviaBluetooth();              //Enviamos un mensaje nulo para indicar que no
estamos midiendo correctamente

    while (millis() - t < 500) continue;
  }
}


////////////////////////////////////////////////////////////////////////////
void Pulso_LED(int led)
{
  unsigned long t = micros();
  float lectura;

  digitalWrite(led, HIGH);

  while ((micros() - t) < 300)
  {
    lectura = Lectura_Vo0();
  }

  digitalWrite(led, LOW);

  if (led == LED_IR)
    V_IR = lectura;
  else
    V_R = lectura;
}


////////////////////////////////////////////////////////////////////////////
```

```
void Ciclo()
{
  t_ciclo = micros();

  Pulso_LED(LED_IR);
  while ((micros() - t_ciclo) < 500) continue;                //Semiperiodo del
apagado de los leds

  Pulso_LED(LED_R);
  while ((micros() - t_ciclo) < 1000) continue;               //Semiperiodo del
apagado de los leds

  pendiente = ActualizaVectorSenal(V_IR);
  minimo_pendiente = min(pendiente, minimo_pendiente);
  extremos_IR[0] = min(extremos_IR[0], V_IR);
  extremos_IR[1] = max(extremos_IR[1], V_IR);
  extremos_R[0] = min(extremos_R[0], V_R);
  extremos_R[1] = max(extremos_R[1], V_R);
  DetectaPulso();
  Envia_Datos();
}


////////////////////////////////////////////////////////////////////////////
bool Check()
{
  unsigned long t = millis();
  float lectura_R, lectura_IR;

  lectura_R = DetectarDedo(LED_R);
  lectura_IR = DetectarDedo(LED_IR);

  if ((lectura_R < SENSIBILIDAD_ALTA) && (lectura_IR < SENSIBILIDAD_ALTA) &&
      (lectura_R  >  SENSIBILIDAD_BAJA)  &&  (lectura_IR  >  SENSIBILIDAD_BAJA))
//Si estamos dentro del umbral de sensibilidad quiere decir que el sensor esta
correctamente colocado en el dedo
  {
    return (true);
  }
  else
  {
    return (false);
  }
}


////////////////////////////////////////////////////////////////////////////
float DetectarDedo(int led)
```

```
{
  unsigned long t = millis();
  float lectura = 0;
  int i = 1;
  digitalWrite(led, HIGH);
  while ((millis() - t) < 10)
  {
    lectura = (Lectura_Vo0() + (i - 1) * lectura) / i;                //Calculamos
el valor medio de la lecturas
    i++;
  }
  digitalWrite(led, LOW);
  return (lectura);
}


////////////////////////////////////////////////////////////////////////////
void Envia_Datos()
{
  Serial.print(V_IR);                          //V_IR
  Serial.print(",");
  Serial.print(V_R);                           //V_R
  Serial.print(",");
  Serial.print(pendiente);                     //Pendiente
  Serial.print(",");
  Serial.print(limiteDeteccion);               //Limite de deteccion
  Serial.print(",");
  //Serial.print(duracion_pulso);                //Duracion de pulso
  //Serial.print(",");
  //Serial.print(extremos_IR[0]);                //Mínimo de la señal IR
  //Serial.print(",");
  //Serial.print(extremos_IR[1]);                //Máximo de la señal IR
  //Serial.print(",");
  //Serial.print(extremos_R[0]);                 //Mínimo de la señal R
  //Serial.print(",");
  //Serial.println(extremos_R[1]);               //Máximo de la señal IR
  //Serial.print(",");
  //Serial.print(ppm);                           //Pulsaciones por minuto
  //Serial.print(",");
  //Serial.print(R);                             //Relación R/IR
  //Serial.print(",");
  //Serial.print(oxigeno);                       //Saturación de oxígeno
  //Serial.print(",");
  if (pendiente < limiteDeteccion)             //Pulso
    Serial.println(2000);
  else
    Serial.println(0);
```

```
  //serializeJson(mensaje,Serial);
  //Serial.println();
  //serializeJson(mensaje, HC06);

}


////////////////////////////////////////////////////////////////////////////
float ActualizaVectorSenal(float nuevo_valor)
{
  int i;
  float aux;
    for (i = (SIZE_VECTOR_PENDIENTE - 1); i >= 0; i--)
    {
      aux = vector[i];
      vector[i] = nuevo_valor;
      nuevo_valor = aux;
    }
  return ((vector[SIZE_VECTOR_PENDIENTE - 1] - vector[0]) * 10);
}


////////////////////////////////////////////////////////////////////////////
void DetectaPulso()
{
  pulsoAnterior = pulso;

  if (pendiente < limiteDeteccion)
    pulso = true;
  else
    pulso = false;

  if ((!pulsoAnterior) && pulso)                              //Si detectamos pulso
  {
    duracion_pulso = millis() - comienzo_pulso;
    comienzo_pulso = millis();

    if ((duracion_pulso > 250) && (duracion_pulso < 1500))               //Si la
variación no es fruto de ruidos
    {
      ActualizaVectorPulso(duracion_pulso);
      ActualizaVectorR(CalculaR());
      ppm = CalculaPPM();
      R = CalculaMediaR();
      oxigeno = CalculaSpO2(R);

      if (vectorPulsaciones[0] == 0 || oxigeno > 100 || oxigeno < 90)
      {
```

```
        datos["ppg"] = V_IR;
        datos["oxigeno"] = 0;
        datos["pulso"] = 0;
      }
      else
      {
        datos["ppg"] = V_IR;
        datos["oxigeno"] = oxigeno;
        datos["pulso"] = ppm;
      }
    }
    else if (duracion_pulso > 1500)
    {
      for (int i = 0; i < SIZE_VECTOR_MEDIA; i++) //Reiniciamos los vectores
      {
        vectorPulsaciones[i] = 0;
        vectorR[i] = 0;
      }
    }
  }
}


////////////////////////////////////////////////////////////////////////////
void ActualizaVectorPulso(int nuevo_valor)
{
  int i;
  int aux;
  for (i = (SIZE_VECTOR_MEDIA - 1); i >= 0; i--)
  {
    aux = vectorPulsaciones[i];
    vectorPulsaciones[i] = nuevo_valor;
    nuevo_valor = aux;
  }
}


////////////////////////////////////////////////////////////////////////////
void ActualizaVectorR(float nuevo_valor)
{
  int i;
  float aux;
  for (i = (SIZE_VECTOR_MEDIA - 1); i >= 0; i--)
  {
    aux = vectorR[i];
    vectorR[i] = nuevo_valor;
    nuevo_valor = aux;
  }
```

```
}


//////////////////////////////////////////////////////////////////////////////
int CalculaPPM()
{
  int i;
  float pulsos[SIZE_VECTOR_MEDIA];
  float suma_pulsos = 0;

  for (i = 0; i < SIZE_VECTOR_MEDIA; i++)
  {
    pulsos[i] = 60000 / vectorPulsaciones[i];
    suma_pulsos += pulsos[i];
  }

  return (int)((suma_pulsos / SIZE_VECTOR_MEDIA) + 0.5);      //Devolvemos el
promedio de los tres pulsos, redondeando al entero más próximo
}


//////////////////////////////////////////////////////////////////////////////
float CalculaR()
{
  float r = (((extremos_R[1] - extremos_R[0]) * extremos_IR[0]) / ((extremos_IR[1]
- extremos_IR[0]) * extremos_R[0]));
  extremos_IR[0] = 2400;
  extremos_IR[1] = 0;
  extremos_R[0] = 2400;
  extremos_R[1] = 0;

  return r;
}


//////////////////////////////////////////////////////////////////////////////
float CalculaMediaR()
{
  int i;
  float suma_R;

  for (i = 0; i < SIZE_VECTOR_MEDIA; i++)
  {
    suma_R += vectorR[i];
  }

  return (suma_R / SIZE_VECTOR_MEDIA);
}
```

```
////////////////////////////////////////////////////////////////////////////
int CalculaSpO2(float r)
{
  return (int)(-8.1753 * r + 101.61 + 0.5);  //rednodeamos al entero más cercano
}


////////////////////////////////////////////////////////////////////////////
float Lectura_Vo0()
{
  return (analogRead(Vo0) * 3.3 * 1000 / 4095);
}


////////////////////////////////////////////////////////////////////////////
void enviaBluetooth()
{
  if(HC06.available())
  {
  serializeJson(mensaje, HC06);
  HC06.println();
  }
}


////////////////////////////////////////////////////////////////////////////
void actualizaMensaje(float senal, float r, int pulsaciones)
{
  datos["ppg"] = senal;
  datos["oxigeno"] = r;
  datos["pulso"] = pulsaciones;
}
```

2. Código de Android

2.1. Paciente

```
package com.example.android.constantesvitales;

import java.io.Serializable;

@SuppressWarnings("serial")
public class Paciente implements Serializable {
    private String mNombre;
    private int mEdad;
    private int mAltura;
    private int mPeso;

    Paciente(String nombre, int edad, int altura, int peso){
        mNombre=nombre;
        mEdad=edad;
        mAltura=altura;
        mPeso=peso;
    }

    public String getNombre(){

        return mNombre;
    }

    public int getEdad() {

        return mEdad;
    }

    public int getAltura() {

        return mAltura;
    }

    public int getPeso() {

        return mPeso;
    }
}
```

2.2. ConstanteVital

```
package com.example.android.constantesvitales;

public class ConstanteVital {
    private String mNombre;
    private int mValor;
    private double mValorDouble;
    private int mIcono;

    public ConstanteVital(String nombre, int valor, int icono) {
        mNombre = nombre;
        mValor = valor;
        mIcono = icono;
    }

    public ConstanteVital(String nombre, double valorDouble, int icono) {
        mNombre = nombre;
        mValorDouble=valorDouble;
        mIcono = icono;
    }

    public String getNombre() {

        return mNombre;
    }

    public int getValor() {

        return mValor;
    }

    public double getValorDouble() {

        return mValorDouble;
    }

    public int getIcono()   {

        return mIcono;
    }
}
```

2.3. ObtencionDatos

```
package com.example.android.constantesvitales;

import android.content.res.Resources;
import android.util.Log;
import org.json.JSONObject;
import org.json.JSONException;

import java.util.ArrayList;
//private static final String RESPUESTA =
"{\"datos\":{\"ppg\":1000,\"oxigeno\":98,\"pulso\":60}}";

public final class ObtencionDatos {

    private ObtencionDatos(String json){
    }

    public static ArrayList<com.example.android.constantesvitales.ConstanteVital>
getListaConstantes
            (Paciente paciente, String mensaje, Resources r){

        ArrayList<com.example.android.constantesvitales.ConstanteVital>
listaConstantesVitales =
                new ArrayList<>();

        try {
            JSONObject jsonObject = new JSONObject(mensaje);
            JSONObject datos = jsonObject.getJSONObject("datos");

            listaConstantesVitales.add(new
com.example.android.constantesvitales.ConstanteVital
                    (r.getString(R.string.imc), calculaImc(paciente),
R.drawable.ic_imc_foreground));
            listaConstantesVitales.add(new
com.example.android.constantesvitales.ConstanteVital
                    (r.getString(R.string.ppg), datos.getInt("ppg"),
                            R.drawable.ic_ppg_foreground));
            listaConstantesVitales.add(new
com.example.android.constantesvitales.ConstanteVital
                    (r.getString(R.string.oxigeno), datos.getInt("oxigeno"),
                            R.drawable.ic_oxigeno_foreground));
            listaConstantesVitales.add(new
com.example.android.constantesvitales.ConstanteVital
                    (r.getString(R.string.ppm), datos.getInt("pulso"),
                            R.drawable.ic_corazon_foreground));
        }
        catch (JSONException e){
            Log.e("ObtencionDatos", "Error en la extracción del formato JSON", e);
        }

        return listaConstantesVitales;
    }


    private static double calculaImc(@org.jetbrains.annotations.NotNull Paciente
paciente){
        double altura = paciente.getAltura();
        double peso = paciente.getPeso();
        if(altura>0 && peso>0)
            return  peso/(altura*altura/10000);
        else
            return -1;
    }
}
```

2.4. ActividadInicio

```
package com.example.android.constantesvitales;

import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class ActividadInicio extends AppCompatActivity {

    EditText nombreEditText;
    EditText edadEditText;
    EditText alturaEditText;
    EditText pesoEditText;
    TextView continuar;

    String nombre;
    int edad;
    int altura;
    int peso;

    SharedPreferences preferencias;


    /*ON CREATE*/

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.actividad_inicio);

        inicializarViews();
    }


    /* Inicializa todas las vistas y listeners */

    private void inicializarViews(){
        nombreEditText = (EditText) findViewById(R.id.nombre);
        edadEditText = (EditText) findViewById(R.id.edad);
        alturaEditText = (EditText) findViewById(R.id.altura);
        pesoEditText = (EditText) findViewById(R.id.peso);
        continuar = (TextView) findViewById(R.id.continuar);

        /*Cargamos los datos de la sesión anterior para que aparezcan ya en las
editText views*/

        cargaDatos();

        /*Creamos un listener para pasar a la siguiente actividad y enviar a esta
los datos del
        paciente recogidos*/

        continuar.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                try{
                    Paciente paciente = devuelveDatos();
                    guardaDatos(paciente);
                    Intent intent = new Intent(ActividadInicio.this,

com.example.android.constantesvitales.ActividadConstantes.class);
```

```
                    intent.putExtra("paciente", paciente);
                    startActivity(intent);
                }
                catch (Error e){
                    Log.e("Conectar", "Error al inicializar el perfil del
paciente. " +
                            "Introduzca los datos correctamente", e);
                }
            }
        });
    }


    /*Devuelve una clase Paciente con los datos que el usuario ha introducido en
los campos de
    texto*/

    private Paciente devuelveDatos() {

        /*En caso de que el usuario no introduzca alguno de los valores se
introducen valores por
        defecto*/

        if (nombreEditText.getText().toString().matches(""))
            nombre = getString(R.string.sin_nombre);
        else
            nombre = nombreEditText.getText().toString();

        if (edadEditText.getText().toString().matches(""))
            edad = -1;
        else
            edad = Integer.parseInt(edadEditText.getText().toString());

        if (alturaEditText.getText().toString().matches(""))
            altura = -1;
        else
            altura = Integer.parseInt(alturaEditText.getText().toString());

        if (pesoEditText.getText().toString().matches(""))
            peso = -1;
        else
            peso = Integer.parseInt(pesoEditText.getText().toString());

        return new Paciente(nombre, edad, altura, peso);
    }


    /*Guardamos los datos introducidos para la siguiente sesión*/

    private void guardaDatos(Paciente paciente) {
        preferencias = getSharedPreferences("preferencias", Context.MODE_PRIVATE);
        SharedPreferences.Editor editor = preferencias.edit();
        editor.putString("KEY_NOMBRE", paciente.getNombre());
        editor.putInt("KEY_EDAD", paciente.getEdad());
        editor.putInt("KEY_ALTURA", paciente.getAltura());
        editor.putInt("KEY_PESO", paciente.getPeso());
        editor.apply();
    }


    /*Cargamos las preferencias guardadas, y en caso de no encontrarse se
introducen valores por
    defecto*/

    private void cargaDatos() {

        preferencias = getSharedPreferences("preferencias", Context.MODE_PRIVATE);
        nombre = preferencias.getString("KEY_NOMBRE",
getString(R.string.sin_nombre));
        edad = preferencias.getInt("KEY_EDAD",-1);
        altura = preferencias.getInt("KEY_ALTURA",-1);
```

```
        peso = preferencias.getInt("KEY_PESO",-1);

        if(!(nombre.equals(getString(R.string.sin_nombre)))){
            nombreEditText.setText(nombre);
        }
        if(edad!=-1){
            edadEditText.setText(Integer.toString(edad));
        }
        if(altura!=-1){
            alturaEditText.setText(Integer.toString(altura));
        }
        if(peso!=-1){
            pesoEditText.setText(Integer.toString(peso));
        }
    }
}
```

2.5. ActividadConstantes

```
package com.example.android.constantesvitales;

import android.annotation.SuppressLint;
import android.content.Context;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.BroadcastReceiver;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Build;
import android.support.annotation.RequiresApi;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ImageView;
import android.widget.LinearLayout;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.Toast;
import org.jetbrains.annotations.NotNull;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.Method;
import android.os.Handler;
import android.os.Message;
import java.util.ArrayList;
import java.util.Set;
import java.util.UUID;
import java.util.concurrent.atomic.AtomicBoolean;


@SuppressWarnings("serial")
public class ActividadConstantes extends AppCompatActivity {

    TextView conectarTextView;
    TextView nombreTextView;
    TextView edadTextView;
    TextView alturaTextView;
    TextView pesoTextView;
    ListView listaConstantesView;
    LinearLayout datos_perfil;
    LinearLayout detalles_perfil;
    ImageView flecha;


    ArrayList<ConstanteVital> listaConstantesVitales;
    private BluetoothAdapter btAdapter = BluetoothAdapter.getDefaultAdapter();
    private static final int REQUEST_ENABLE_BT = 1;
    private final AtomicBoolean estadoConexion = new AtomicBoolean(false);
    private final AtomicBoolean botonConectar = new AtomicBoolean(true);
    Set<BluetoothDevice> listaDispositivos = null;
    private static final String direccion = "00:02:5B:00:A0:18";
    private static final String MENSAJE_PREDETERMINADO =
            "{\"datos\":{\"ppg\":0,\"oxigeno\":0,\"pulso\":0}}";
    public BluetoothSocket btSocket = null;
    private static final UUID MI_UUID = UUID.fromString("00001101-0000-1000-8000-
00805F9B34FB");

    Paciente paciente = new Paciente("",-1,-1,-1);
    Handler handlerConstantes;
    Handler handlerEstadoConexion;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.actividad_constantes);
        inicializarViews();

        handlerConstantes = new Handler() {
            @Override
            public void handleMessage(Message msg) {

                Bundle bundle = msg.getData();
                String json = bundle.getString("llaveConstantes");

                listaConstantesVitales =
ObtencionDatos.getListaConstantes(paciente,
                        json, getResources());
                ConstantesAdapter adapter = new
ConstantesAdapter(getApplicationContext(),
                        listaConstantesVitales, getResources());
                listaConstantesView = (ListView) findViewById(R.id.lista);
                listaConstantesView.setAdapter(adapter);

            }
        };

        handlerEstadoConexion = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                Bundle bundle = msg.getData();
                String estado = bundle.getString("llaveConexion");
                if(estado==getString(R.string.conectar)){
                    Toast.makeText(getBaseContext(),
getString(R.string.no_compatibles),
                            Toast.LENGTH_LONG).show();
                }
                conectarTextView.setText(estado);
                botonConectar.set(true);                              //Volvemos a
habilitar el botón
            }
        };
    }


    @Override
    protected void onActivityResult (int requestCode, int resultCode, Intent data)
{
        switch(requestCode)
        {
            case REQUEST_ENABLE_BT:
            {
                if(resultCode == RESULT_OK)
                {
                    Toast.makeText(getBaseContext(), getText((R.string.buscando)),
                            Toast.LENGTH_LONG).show();
                    botonConectar.set(false);
                    conectarTextView.setText(getString(R.string.conectando));
                    EstablecerConexion hiloEstablecerConexion = new
EstablecerConexion();
                    hiloEstablecerConexion.start();
                }
                else
                {
                    Toast.makeText(this, getString(R.string.peticion_cancelada),
                            Toast.LENGTH_SHORT).show();
                }
                break;
            }

            default:
                break;
        }
    }
```

```
    @Override
    public void onDestroy() {
        super.onDestroy();
        this.unregisterReceiver(broadcastReceiver);

        if (btSocket!=null) {
            try {
                btSocket.close();
                estadoConexion.set(false);
            }
            catch (IOException e){
                Log.e("Cierre de socket", "Error al cerrar el socket.", e);
            }
        }
    }


    /* Inicializa todas las vistas y listeners */

    private void inicializarViews() {

        /*Intentamos cargar los datos del paciente creados en la actividad inicial
y usamos el
        adapter para mostrarlos en la listView*/

        try{
            paciente = (Paciente)
getIntent().getExtras().getSerializable("paciente");
            rellenaPerfil(paciente);

            listaConstantesVitales = ObtencionDatos.getListaConstantes(paciente,
                    MENSAJE_PREDETERMINADO, getResources());
            ConstantesAdapter adapter = new ConstantesAdapter(this,
listaConstantesVitales,
                    getResources());
            listaConstantesView = (ListView) findViewById(R.id.lista);
            listaConstantesView.setAdapter(adapter);

        }
        catch (Error e){
            Log.e("Cargar lista", "Error al cargar la lista de datos", e);
        }


        /*Creamos un listener en la vista del perfil que permitira desplegar y
contraer los datos */

        conectarTextView = (TextView)findViewById(R.id.conectar);
        datos_perfil = (LinearLayout) findViewById(R.id.datos_perfil);
        detalles_perfil = (LinearLayout) findViewById(R.id.detalles_perfil);
        flecha = (ImageView) findViewById(R.id.flecha);
        detalles_perfil.setVisibility(View.GONE);
        flecha.setBackgroundResource(R.drawable.flecha_expandir);

        datos_perfil.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if(detalles_perfil.getVisibility()==View.GONE) {
                    detalles_perfil.setVisibility(View.VISIBLE);
                    flecha.setBackgroundResource(R.drawable.flecha_contraer);
                }
                else {
                    detalles_perfil.setVisibility(View.GONE);
                    flecha.setBackgroundResource(R.drawable.flecha_expandir);
                }
            }
        });


        /* Comprobamos si nuestro dispositivo dispone de Bluetooth. Si no es así,
el botón para
        conectarse al mismo permanecerá inhabilitado */
```

```
        if(btAdapter == null)
        {
            conectarTextView.setBackgroundResource(R.color.colorDissabled);
            conectarTextView.setText(getString(R.string.BT_no_disponible));
        }

        else {
            registrarBroadcastReceiver();
            conectarTextView.setBackgroundResource(R.color.colorPrimary);

            /* Comprobamos si el Bluetooth está activo o no y cambiamos el texto
acorde a ello */

            if(estadoConexion.get()) {
                conectarTextView.setText(getString(R.string.desconectar));
            }
            else {
                conectarTextView.setText(getString(R.string.conectar));
            }

            /* Establecemos la función del botón para conectarnos o desconectarnos
*/

            conectarTextView.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View pulsador) {
                    if (botonConectar.get()) {                          //Si está
habilitado el botón
                        try {
                            if (!(estadoConexion.get())) {              //Si el hilo
no está activo
                                if (!btAdapter.isEnabled()) {
                                    conectarBt(btAdapter);
                                } else {
                                    Toast.makeText(getBaseContext(),
getText((R.string.buscando)),
                                            Toast.LENGTH_LONG).show();
                                    botonConectar.set(false);
//Deshabilitamos el botón

conectarTextView.setText(getString(R.string.conectando));
                                    EstablecerConexion hiloEstablecerConexion =
                                            new EstablecerConexion();
                                    hiloEstablecerConexion.start();
                                }
                            } else {                                            //Si el
hilo está activo
                                estadoConexion.set(false);

conectarTextView.setText(getString(R.string.conectar));

                                listaConstantesVitales =
ObtencionDatos.getListaConstantes(paciente,
                                        MENSAJE_PREDETERMINADO, getResources());
                                ConstantesAdapter adapter = new ConstantesAdapter
                                        (getApplicationContext(),
listaConstantesVitales,
                                                getResources());
                                listaConstantesView = (ListView)
findViewById(R.id.lista);
                                listaConstantesView.setAdapter(adapter);
                            }
                        } catch (Error e) {
                            e.printStackTrace();
                        }
                    }
                }
            });
        }
    }


    /* Carga los datos del paciente y crea la vista del perfil, con la opcion de
```

```
contraer y
    expandir los detalles */

    @SuppressLint("SetTextI18n")
    private void rellenaPerfil(@NotNull Paciente paciente){

        /* Tomamos los valores de la clase paciente con los datos que hemos tomado
en la actividad
        del inicio y establecemos los valores de los text views correspondientes
*/

        nombreTextView = (TextView) findViewById(R.id.nombre);
        nombreTextView.setText(paciente.getNombre());

        edadTextView = (TextView) findViewById(R.id.edad);
        int edad = paciente.getEdad();
        if(edad==-1){
            edadTextView.setText("--");
        }
        else {
            edadTextView.setText(Integer.toString(edad));
        }

        alturaTextView = (TextView) findViewById(R.id.altura);
        int altura = paciente.getAltura();
        if(altura==-1){
            alturaTextView.setText("--");
        }
        else{
            alturaTextView.setText(Integer.toString(altura));
        }

        pesoTextView = (TextView) findViewById(R.id.peso);
        int peso = paciente.getPeso();
        if(peso==-1){
            pesoTextView.setText("--");
        }
        else{
            pesoTextView.setText(Integer.toString(peso));
        }
    }


    /*Conecta o desconecta el Bluetooth*/

    private  void conectarBt(@NotNull BluetoothAdapter btAdapter) {

        /* Si el dispositivo esta apagado entonces lo encendemos, buscamos
dispositivos compatibles,
        y nos conectamos de ser posible */

        if (!(btAdapter.isEnabled())) {

            try {
                /* Pedimos permisos al usuario para conectar el Bluetooth */
                Intent enableBtIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
                startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
            } catch (Error e) {
                Log.e("Cargar lista", "Error al cargar la lista de datos", e);
            }
        }
    }


    /* Registramos el broadcast receiver */

    private void registrarBroadcastReceiver() {

        IntentFilter filtro = new
IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);
        filtro.addAction(BluetoothDevice.ACTION_FOUND);
        this.registerReceiver(broadcastReceiver, filtro);
```

```
    }


    /*Definimos el BroadcastReceiver para detectar cambios en el adaptador
Bluetooth*/

    private final BroadcastReceiver broadcastReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, @NotNull Intent intent) {
            final String accion = intent.getAction();
            final int estado = intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
                    BluetoothAdapter.ERROR);

            if (BluetoothAdapter.ACTION_STATE_CHANGED.equals(accion)) {
                switch (estado) {
                    case BluetoothAdapter.STATE_OFF: {
                        estadoConexion.set(false);

((TextView)findViewById(R.id.conectar)).setText(getString(R.string.conectar));
                        break;
                    }

                    default:
                        break;
                }
            }

            if(BluetoothDevice.ACTION_FOUND.equals(accion)) {
                BluetoothDevice device = (BluetoothDevice)
intent.getParcelableExtra
                        (BluetoothDevice.EXTRA_DEVICE);

                Toast.makeText(getApplicationContext(),
getString(R.string.encontrado) +
                                device.getName(),
                        Toast.LENGTH_SHORT).show();
            }
        }
    };


    /*Hilo del proceso para recibir los mensajes que llegan a traves del socket de
conexión*/

    private class HiloTransmisionDatos extends Thread{

        private BluetoothSocket mSocket;
        private InputStream mFlujoEntrada;
        BufferedReader br;
        final byte delimitador = 10;           /*El caracter delimitador es el
correspondiente
                                               al salto de línea \n en código
ASCII, esto es 10*/
        private static final char DELIMITADOR = '\n';
        String mensaje  = "";

        HiloTransmisionDatos(BluetoothSocket socket){
            mSocket = socket;
        }

        private String leer(){
            String s = "";

            try {
                byte[] buffer = new byte[1024];
                int bytes = mFlujoEntrada.read(buffer);
                s = new String(buffer, "ASCII");
                s = s.substring(0, bytes);
            }catch (IOException e){

            }
            return s;
```

```
        }

        private void enviaHAndlerConstantes(String s) {

            Message msg = Message.obtain();
            Bundle bundle = new Bundle();
            bundle.putString("llaveConstantes", s);
            msg.setData(bundle);
            handlerConstantes.sendMessage(msg);
        }

        private void construirMensaje() {

            //Buscamos el primer delimitador que haya en el buffer
            int indice = mensaje.indexOf(DELIMITADOR);

            //Si no hay ninguno salimos
            if (indice == -1) {
                return;
            }

            //Construimos el mensaje completo
            String s = mensaje.substring(0, indice);

            //Eliminamos el mensaje del buffer
            mensaje = mensaje.substring(indice + 1);

            //Si empieza correctamente es que el json esta entero y lo enviamos
            if(s.contains("{\"datos\":")){
                enviaHAndlerConstantes(s);
            }

            //Seguimos buscando
            construirMensaje();
        }


        @RequiresApi(api = Build.VERSION_CODES.N)
        public void run() {

            estadoConexion.set(true);

            try{
                mFlujoEntrada = mSocket.getInputStream();
            }
            catch(IOException e){
                Log.e("Flujo de entrada", "Error al crear el flujo de entrada",
e);
            }

            while (estadoConexion.get() && !this.isInterrupted() &&
btSocket.isConnected()) {

                if (mFlujoEntrada == null){
                    break;
                }

                String s = leer();

                if (s.length() > 0)
                    mensaje += s;

                construirMensaje();

            }
            if(!btSocket.isConnected() || !estadoConexion.get()){

((TextView)findViewById(R.id.conectar)).setText(getString(R.string.conectar));
                estadoConexion.set(false);
                if (mFlujoEntrada != null) {
                    try {mFlujoEntrada.close();} catch (Exception e) {
e.printStackTrace(); }
                }
```

```
            }
        }
    }


    private class EstablecerConexion extends Thread {


        EstablecerConexion(){
        }

        public void run() {

            Message msg = handlerEstadoConexion.obtainMessage();
            Bundle bundle = new Bundle();

            /*Cada vez que llamemos a la función, vaciaremos el set de
dispositivos para volver a
            buscarlos, en caso de que el set esté vacío*/

            if (listaDispositivos != null) {        //Comprobamos si es null antes
de ver su tamaño
                if (listaDispositivos.size() > 0) {
                    for (BluetoothDevice dispositivo : listaDispositivos) {
                        try {
                            Method m =
dispositivo.getClass().getMethod("eliminarVinculos",
                                    (Class[]) null);
                            m.invoke(dispositivo, (Object[]) null);

                        } catch (Exception e) {
                            Log.e("Vaciado de lista", "Error en el vaciado de la
lista.");
                        }
                    }
                }
            }

            /*Si ya hay un socket creado, lo cerramos*/

            if (btSocket != null) {
                try {
                    btSocket.close();
                } catch (IOException e) {
                    Log.e("Cierre de socket", "Error al cerrar el socket.", e);
                }
            }

            /*Buscamos dispositivos asociados y llenamos el set de dispositivos
que hemos
            encontrado*/

            listaDispositivos = btAdapter.getBondedDevices();

            if(listaDispositivos!=null) {
                for (BluetoothDevice dispositivo : listaDispositivos) {
                    if (dispositivo.getAddress().equals(direccion)) {
                        try {
                            btSocket = creaBluetoothSocket(dispositivo);
                        }catch (IOException e) {
                            Toast.makeText(getBaseContext(),
getString(R.string.error_socket),
                                    Toast.LENGTH_LONG).show();
                        }
                        break;
                    }
                }

                /*Nos conectamos al socket. Este proceso tarda unos segundos
bloqueando al hilo
                principal, y por eso debe ejecutarse en otro hilo*/
```

```
                if (btSocket != null) {
                    try {
                        btSocket.connect();
                        Log.d("Conectado", "Conectado: " +
btSocket.isConnected());
                    } catch (IOException e) {
                        try {
                            btSocket.close();
                            //ConexionFallida(bundle, msg);
                        } catch (IOException e2) {
                            Log.e("Cierre de socket", "Error al cerrar el
socket.", e2);
                        }
                    }

                    if (btSocket.isConnected()) {
                        HiloTransmisionDatos hiloTransmisionDatos = new
HiloTransmisionDatos
                                (btSocket);
                        hiloTransmisionDatos.start();
                        bundle.putString("llaveConexion",
getString(R.string.desconectar));
                        msg.setData(bundle);
                        handlerEstadoConexion.sendMessage(msg);
                    }
                    else {
                        ConexionFallida(bundle, msg);
                    }
                }else {
                    ConexionFallida(bundle, msg);
                }
            }else {
                ConexionFallida(bundle, msg);
            }
        }
    }


    private BluetoothSocket creaBluetoothSocket(BluetoothDevice dispositivo)
throws IOException {

        return dispositivo.createRfcommSocketToServiceRecord(MI_UUID);
    }

    private void ConexionFallida(Bundle bundle, Message msg){
        bundle.putString("llaveConexion", getString(R.string.conectar));
        estadoConexion.set(false);
        msg.setData(bundle);
        handlerEstadoConexion.sendMessage(msg);
    }
}
```

2.6. ConstantesAdapter

```
package com.example.android.constantesvitales;

import android.content.Context;
import android.content.res.Resources;
import android.support.annotation.NonNull;
import android.support.annotation.Nullable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.TextView;

import java.util.ArrayList;


public class ConstantesAdapter extends ArrayAdapter<ConstanteVital> {

    Resources mRecursos;

    public ConstantesAdapter(Context context, ArrayList<ConstanteVital>
constantesVitales,
                             Resources r) {
        super(context, 0, constantesVitales);
        mRecursos = r;
    }


    @Override
    public View getView(int position, @Nullable View convertView, @NonNull
ViewGroup parent) {
        View elementoListaView = convertView;
        if (elementoListaView == null) {
            elementoListaView =
LayoutInflater.from(getContext()).inflate(R.layout.elemento_lista,
                    parent, false);

        }

        ConstanteVital constanteActual = getItem(position);

        TextView descripcion = (TextView)
elementoListaView.findViewById(R.id.descr_constantes);
        descripcion.setText(constanteActual.getNombre());

        if (constanteActual.getNombre()==mRecursos.getString(R.string.imc)) {
            TextView valorTextView = (TextView) elementoListaView.findViewById
                    (R.id.valor_constantes);
            if(constanteActual.getValorDouble()<0 ||
constanteActual.getValorDouble()>100){
                valorTextView.setText("--");
            }
            else {
                valorTextView.setText(String.format("%.1f",
constanteActual.getValorDouble()));
            }
        }
        else {
            TextView valorTextView = (TextView) elementoListaView.findViewById
                    (R.id.valor_constantes);
            if(constanteActual.getValor()==0) {
                valorTextView.setText("--");
            }
            else {

valorTextView.setText(Integer.toString(constanteActual.getValor()));;
```

```
            }
        }

        ImageView iconoView = (ImageView)
elementoListaView.findViewById(R.id.icono_constantes);
        iconoView.setImageResource(constanteActual.getIcono());

        return elementoListaView;
    }
}
```

2.7. Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.constantesvitales">


    <application

        android:allowBackup="true"
        android:icon="@drawable/ic_corazon_foreground"
        android:label="@string/app_name"
        android:roundIcon="@drawable/ic_corazon_foreground"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <activity android:name=".ActividadInicio"
            android:screenOrientation="portrait">

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".ActividadConstantes"
            android:label="@string/visualizacion"
            android:screenOrientation="portrait"/>
    </application>

    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />

</manifest>
```

2.8. Strings

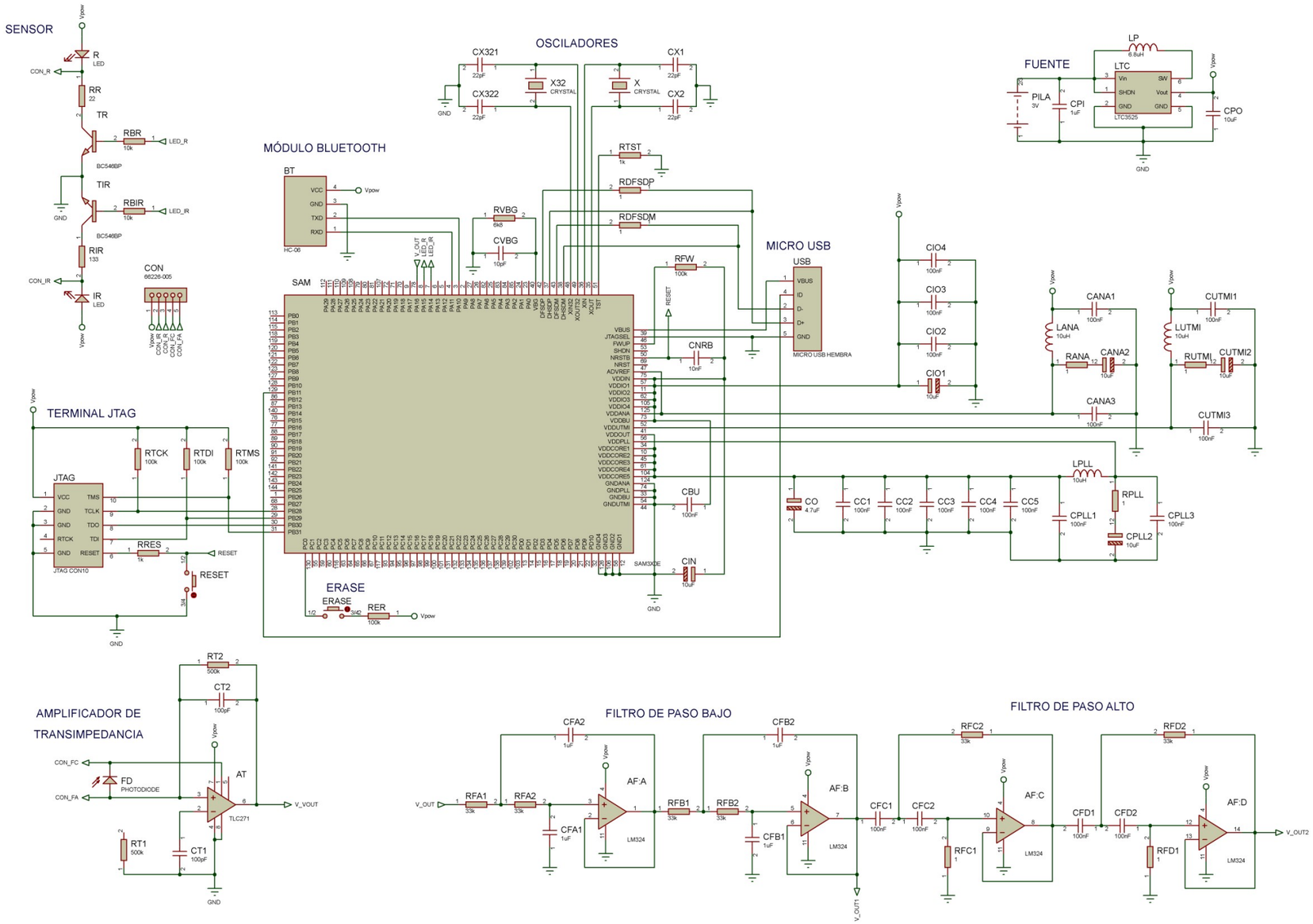
- EN

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <string name="app_name">Vital Signs</string>
    <string name="visualizacion">Signs viewer</string>
    <string name="perfil">Personal profile</string>
    <string name="nombre">Name</string>
    <string name="edad">Age</string>
    <string name="altura">Height (cm)</string>
    <string name="peso">Weight (kg)</string>
    <string name="sin_nombre">No name</string>
    <string name="continuar">Continue</string>
    <string name="conectar">Connect to device</string>
    <string name="conectando">Connecting…</string>
    <string name="desconectar">Disconnect</string>
    <string name="imc">Body mass index</string>
    <string name="ppg">PPG</string>
    <string name="oxigeno">Oxygen in blood</string>
    <string name="ppm">Beats per minute</string>
    <string name="no_compatibles">Compatible devices not found.</string>
    <string name="peticion_cancelada">BT activation request
cancelled.</string>
    <string name="BT_no_disponible">BT unavailable</string>
    <string name="buscando">Searching for devices, wait…</string>
    <string name="error_socket">Error at socket creation.</string>
    <string name="encontrado">Device found: .</string>

</resources>
```

- ES

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <string name="app_name">Constantes Vitales</string>
    <string name="visualizacion">Visualización de Constantes</string>
    <string name="perfil">Perfil personal</string>
    <string name="nombre">Nombre</string>
    <string name="edad">Edad</string>
    <string name="altura">Altura (cm)</string>
    <string name="peso">Peso (kg)</string>
    <string name="sin_nombre">Sin nombre</string>
    <string name="continuar">Continuar</string>
    <string name="conectar">Conectarse al dispositivo</string>
    <string name="conectando">Conectando…</string>
    <string name="desconectar">Desconectar</string>
    <string name="imc">Índice de masa corporal</string>
    <string name="ppg">PPG</string>
    <string name="oxigeno">Oxígeno en sangre</string>
    <string name="ppm">Pulsaciones por minuto</string>
    <string name="no_compatibles">No se han encontrado dispositivos
compatibles.</string>
    <string name="peticion_cancelada">Petición de activación BT
cancelada.</string>
    <string name="BT_no_disponible">BT no disponible</string>
    <string name="buscando">Buscando dispositivos, espere un
momento…</string>
    <string name="error_socket">Error en la creación del socket.</string>
    <string name="encontrado">Dispositivo encontrado: .</string>

</resources>
```

3. Diagrama eléctrico



4. Diseño electrónico escalado

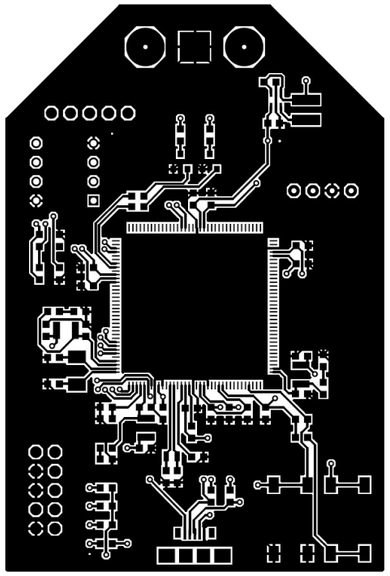


Ilustración 99. Top copper

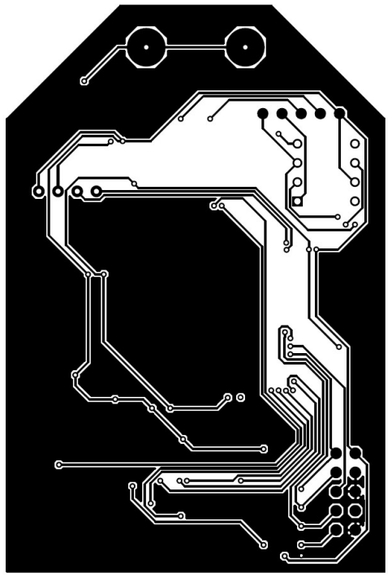


Ilustración 100. Bottom copper

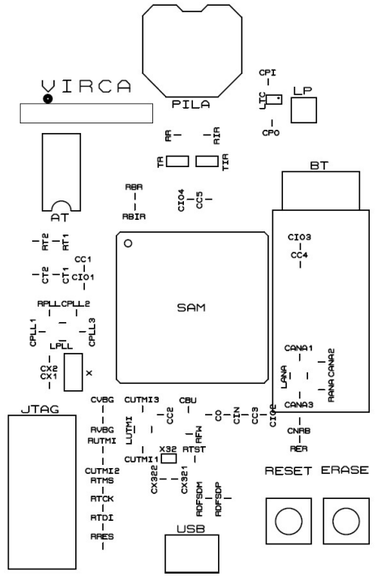


Ilustración 101. Top silk.

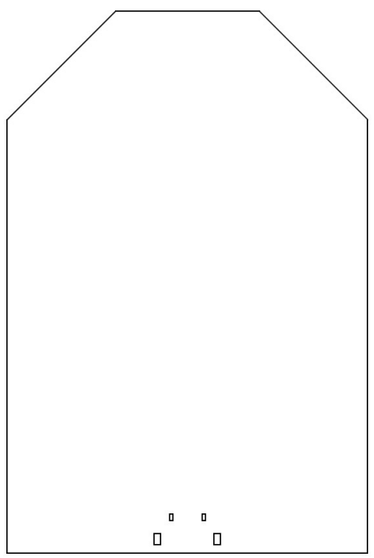


Ilustración 102. Mecanizado.

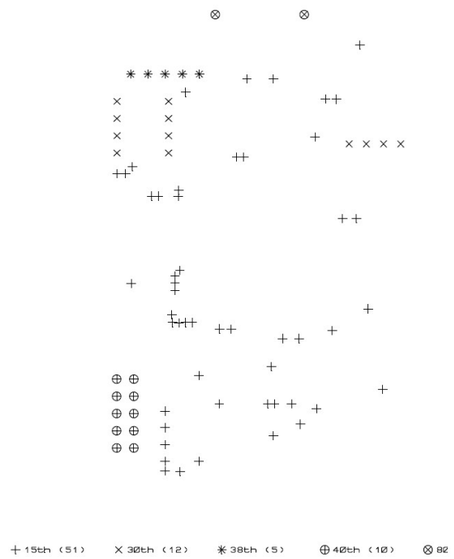


Ilustración 103. Drill.

Bibliografía

Bibliografía

AASVIK, MADS. 2018. Norwegian Creations. *Arduino Tutorial: Avoiding the Overflow Issue When Using millis() and micros().* [En línea] Norwegian Creations, 11 de Octubre de 2018. https://www.norwegiancreations.com/2018/10/arduino-tutorial-avoiding-the-overflow-issue-when-using-millis-and-micros/.

AEMPS. 2018. Agencia Española de Medicamentos y Productos Sanitarios (AEMPS). *Productos sanitarios.* [En línea] Ministerio de Sanidad del Gobierno de España, 20 de Junio de 2018. https://www.aemps.gob.es/productosSanitarios/prodSanitarios/home.htm.

ANGUITA LORENZO, SERGIO. 2016. Blog de la Universidad de Deusto. *Dispositivos médicos: regulación.* [En línea] Facultad de Ingeniería de la Universidad de Deusto, 1 de Diciembre de 2016. https://blogs.deusto.es/master-informatica/regulacion-medevs/.

ARDUINO. Arduino. *SAM3X-Arduino Pin Mapping.* [En línea] https://www.arduino.cc/en/Hacking/PinMappingSAM3X.

—. Arduino. *analogRead() reference.* [En línea] https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/.

—. Arduino Playground. *Arduino Due Bootloader Explained.* [En línea] https://playground.arduino.cc/Bootloader/DueBootloaderExplained/.

—. Arduino Store. *Arduino DUE.* [En línea] https://store.arduino.cc/arduino-due.

—. SAM3X-Arduino Pin Mapping. *Arduino Due pin mapping table.* [En línea] https://www.arduino.cc/en/Hacking/PinMappingSAM3X.

BIBLIOTECA NACIONAL DE MEDICINA DE LOS EE.UU. MedlinePlus. *Enciclopedia médica.* [En línea] https://medlineplus.gov/spanish/ency/article/002269.htm.

BLANCHON, BENOÎT. ArduinoJson. [En línea] https://arduinojson.org/.

COMISIÓN EUROPEA. Comisión Europea. *New Regulations.* [En línea] https://ec.europa.eu/growth/sectors/medical-devices/new-regulations_en.

—. Comisión Europea. *Medical Devices.* [En línea] https://ec.europa.eu/growth/sectors/medical-devices_es.

—. 1993. Directiva 93/42/CEE. *Relativa a los productos sanitarios.* [En línea] Junio de 14 de 1993. https://eur-lex.europa.eu/legal-content/ES/TXT/PDF/?uri=CELEX:01993L0042-20071011&from=EN.

—. 2017. Reglamento (UE) 2017/745. *Sobre los productos sanitarios.* [En línea] 5 de Abril de 2017. https://eur-lex.europa.eu/legal-content/ES/TXT/PDF/?uri=CELEX:02017R0745-20170505&from=EN.

DEPARTAMENTO DE TEORÍA DE LA SEÑAL Y COMUNICACIONES DE LA UNIVERSIDAD DE ALCALÁ DE HENARES. Diseño de filtros activos. [En línea]

2016. Electronics Hubs. *Basic ARM Tutorials For Beginners.* [En línea] 3 de Diciembre de 2016. https://www.electronicshub.org/arm-tutorial/.

GARCÍA, DANIEL. 2013. Let's code something up! *Bluetooth (I): Activando y desactivando el Bluetooth en Android.* [En línea] 19 de Octubre de 2013. https://danielggarcia.wordpress.com/2013/10/19/bluetooth-i-activando-y-desactivando-el-bluetooth-en-android/.

—. 2013. Let's code something up! *Bluetooth (II): Descubriendo dispositivos.* [En línea] 21 de Octubre de 2013. https://danielggarcia.wordpress.com/2013/10/21/bluetooth-ii-

descubriendo-dispositivos/.

—. 2013. Let's code something up! *Bluetooth (III): El esquema cliente-servidor.* [En línea] 23 de Octubre de 2013. https://danielggarcia.wordpress.com/2013/10/23/bluetooth-iii-el-esquema-cliente-servidor/.

—. 2013. Let's code something up! *Bluetooth (IV): Creando el hilo de conexión.* [En línea] 26 de Octubre de 2013. https://danielggarcia.wordpress.com/2013/10/26/bluetooth-iv-creando-el-hilo-de-conexion/.

—. 2013. Let's code something up! *Bluetooth (V): Creando el hilo servidor.* [En línea] 27 de Octubre de 2013. https://danielggarcia.wordpress.com/2013/10/27/bluetooth-v-creando-el-hilo-servidor/.

Geeky Theory. *Tutorial Android - 10. Paso de parámetros entre Activities.* [En línea] https://geekytheory.com/tutorial-android-10-paso-de-parametros-entre-activities.

GOOGLE. Android Developers. *View reference.* [En línea] https://developer.android.com/reference/android/view/View#implementing-a-custom-view.

—. Android Developers. *Introducción general al Bluetooth.* [En línea] https://developer.android.com/guide/topics/connectivity/bluetooth.

—. Android Developers. *Cómo interpretar el ciclo de vida de una actividad.* [En línea] https://developer.android.com/guide/components/activities/activity-lifecycle?hl=es-419.

—. Android Developers. *Handler reference.* [En línea] https://developer.android.com/reference/android/os/Handler.

—. Android Developers. *Cómo brindar compatibilidad con diferentes*

idiomas y culturas. [En línea] https://developer.android.com/training/basics/supporting-devices/languages?hl=es-419.

—. Material Design. *Design.* [En línea] https://material.io/design/.

HATHIBELAGAL, ASHRAFF. 2016. Envato Tuts+. *Android desde Cero: Entender Adaptadores y Adaptador Vista.* [En línea] Envato, 15 de Junio de 2016. https://code.tutsplus.com/es/tutorials/android-from-scratch-understanding-adapters-and-adapter-views--cms-26646.

MACHO, JUAN CARLOS. Prometec. *Módulo Bluetooth HC-06.* [En línea] https://www.prometec.net/bt-hc06/.

MAFRE. Salud MAFRE. *¿Qué son las constantes vitales?* [En línea] https://www.salud.mapfre.es/enfermedades/reportajes-enfermedades/constantes-vitales-que-son-y-cuantas-hay/.

MEZA CONTRERAS, LUIS G., LLAMOSA R., LUIS ENRIQUE y PATRICIA CEBALLOS, SILVIA. 2007. Diseño de Procedimientos para la Calibración de Pulsoxímetros. *Scientia et Technica, Año XIII, No 37.* [En línea] Diciembre de 2007. http://revistas.utp.edu.co/index.php/revistaciencia/article/viewFile/4163/2123. ISSN 0122-1701.

MEZA GONZALEZ, JUAN DAVID. Programar Ya. *Objetos, clases y constructores en Java. Crear una clase y un objeto. Class y new en Java.* [En línea] https://www.programarya.com/Cursos/Java/Objetos-y-Clases.

MIMS III, FORREST M. 2014. Makezine. *Amateur Scientist: Experimenting with Light and Dark Sensors.* [En línea] 24 de Abril de 2014. https://makezine.com/projects/make-38-cameras-and-av/light-and-dark-sensors/.

MINISTERIO DE SANIDAD Y POLÍTICA SOCIAL DEL GOBIERNO DE ESPAÑA. 2009. Boletín Oficial del Estado. *Real Decreto 1591/2009.* [En

línea] 6 de Noviembre de 2009. https://www.boe.es/buscar/doc.php?id=BOE-A-2009-17606.

Naylamp Mechatronics. *Tutorial básico del uso del módulo Bluetooth HC-06 yHC-05.* [En línea] https://www.naylampmechatronics.com/blog/12_Tutorial-B%C3%A1sico-de-Uso-del-M%C3%B3dulo-Bluetooth-H.html.

O'SULLIVAN, ANDY. 2017. Apps and Biscuits. *Saving data with SharedPreferences — Android #9.* [En línea] 15 de Marzo de 2017. https://appsandbiscuits.com/saving-data-with-sharedpreferences-android-9-9fecae19896a.

SEDATIO OFFICE. 2013. Blog de Sedatio Office. *Invención de la pulsioximetría. De History of blood gas analysis. VII. Pulse oximetry. Severinghaus JW, Honda Y. Journal of Clinical Monitoring, 3 de Abril de 1987.* [En línea] 16 de Diciembre de 2013. http://www.sedatio.es/invencion-de-la-pulsioximetria/.

SEGADOR, JON. 2012. Jon Segador. *Paso de datos/variables entre actividades en Android.* [En línea] 13 de Febrero de 2012. http://jonsegador.com/2012/02/paso-de-datos-variables-entre-actividades-android/.

Wikipedia. *Pulsioximetría.* [En línea] https://es.wikipedia.org/wiki/Pulsioximetr%C3%ADa.

Wikipedia. *Signos vitales.* [En línea] https://es.wikipedia.org/wiki/Signos_vitales.

Wikipedia. *Bluetooth.* [En línea] https://es.wikipedia.org/wiki/Bluetooth.

Wikipedia. *Internet de las cosas.* [En línea] https://es.wikipedia.org/wiki/Internet_de_las_cosas.

Wikipedia. *Hemoglobina.* [En línea]

https://es.wikipedia.org/wiki/Hemoglobina.

Wikipedia. *Photoplethysmogram.* [En línea] https://en.wikipedia.org/wiki/Photoplethysmogram.

Wikipedia. *Fotopletismografía.* [En línea] https://es.wikipedia.org/wiki/Fotopletismograf%C3%ADa.

Wikipedia. *Oxímetro de pulso (pulsoximetría).* [En línea] https://es.wikipedia.org/wiki/Ox%C3%ADmetro_de_pulso_(pulsiox%C3%ADmetro).

Wikipedia. *Ley de Beer-Lambert.* [En línea] https://es.wikipedia.org/wiki/Ley_de_Beer-Lambert.

Wikipedia. *Espectrofotometría.* [En línea] https://es.wikipedia.org/wiki/Espectrofotometr%C3%ADa.

Wikipedia. *Beer-Lambert Law.* [En línea] https://en.wikipedia.org/wiki/Beer%E2%80%93Lambert_law.

Wikipedia. *Oxygen saturation (medicine).* [En línea] https://en.wikipedia.org/wiki/Oxygen_saturation_(medicine).

Wikipedia. *Saturación de oxígeno.* [En línea] https://es.wikipedia.org/wiki/Saturaci%C3%B3n_de_ox%C3%ADgeno.

Wikipedia. *Frecuencia cardíaca.* [En línea] https://es.wikipedia.org/wiki/Frecuencia_card%C3%ADaca.

Wikipedia. *Heart rate.* [En línea] https://en.wikipedia.org/wiki/Heart_rate.

Wikipedia. *Índice de masa corporal.* [En línea] https://es.wikipedia.org/wiki/%C3%8Dndice_de_masa_corporal.

WORLD HEALTH ORGANIZATION. 2003. Medical Device Regulations. *Global overview and guiding principles.* [En línea] 2003.

https://www.who.int/medical_devices/publications/en/MD_Regulations.pdf.

Herramientas online empleadas

Simulación de filtros Sallen-Key: http://sim.okawa-denshi.jp/en/Fkeisan.htm

Arduino Create: https://create.arduino.cc/

APP Inventor: http://ai2.appinventor.mit.edu/?locale=es_ES

Autoría de imágenes

Algunas de las imágenes empleadas para la realización de esta memoria, así como algunas tablas o diagramas, han sido tomadas de las fuentes citadas en la bibliografía, las herramientas empleadas o bien pertenecen al banco de imágenes de Wikimedia Commons. La ilustración 3 es de elaboración propia así como el resto de figuras restantes.