**Universidad Nacional de Ingeniería**

**Área de Conocimiento de Tecnología de Información y Comunicación**

TRABAJO MONOGRÁFICO PARA OPTAR AL TÍTULO DE INGENIERO ELECTRÓNICO

**Tema:**

# PROTOTIPO DE ELECTROCARDIÓGRAFO DIGITAL CON COMUNICACIÓN A SERVICIO EN LA NUBE A TRAVÉS DE REDES MÓVILES PARA FINES DE TELEMEDICINA EN ZONAS RURALES DE NICARAGUA

**Autor:**

Br. Eliézer Guillermo Méndez Delgado

**Tutor:**

TkL. Marco Munguía Mena

Febrero de 2024

Managua, Nicaragua

**DEDICATORIA**

*A mis padres **Evelyn Delgado y Guillermo Méndez**, por su entrega y sacrificio. Agradezco su amor incondicional y guía constante, que han sido la base de mi perseverancia y de mis logros.*

*A mis **profesores**, cuya dedicación y compromiso con la enseñanza han sido la brújula que me ha guiado en este camino académico.*

**RESUMEN**

Desde el año 2020 Nicaragua ha presentado un aumento considerable en el número de defunciones por Infarto Agudo al Miocardio, siendo ésta la principal causa de muerte en el país. Por este motivo, la atención cardiológica se ha vuelto una necesidad para los nicaragüenses, sin embargo, a las personas que viven en zonas rurales les es difícil recibir una valoración cardiológica pues deben viajar grandes distancias para llegar a los principales hospitales del país ubicados en las ciudades, lo cual puede ser especialmente agotador para grupos de riesgo como las personas de la tercera edad.

En ese sentido, el presente trabajo monográfico plantea el desarrollo de una herramienta para facilitar la realización de electrocardiogramas para que estos sean valorados por un especialista de forma remota, a fin de dotar a centros de salud rurales con la capacidad de dar atención temprana y preventiva a enfermedades cardiacas.

Como resultado se obtuvo un sistema de electrocardiografía digital con la capacidad de capturar electrocardiogramas de una derivación y enviarlos a un médico diagnosta a través de un servicio en la nube de tal manera que pueda valorar el examen y emitir un diagnóstico. La solución propuesta consta de un prototipo electrónico para la adquisición de la señal cardíaca, una aplicación móvil que recibe esta señal del prototipo para visualización, post-procesamiento y envío hacia la nube a través de redes inalámbricas. Además, se configuró un servicio en la nube para el almacenamiento de estos exámenes que se encarga de reenviarlos hacia el médico diagnosta a través de correo electrónico y recibir su diagnóstico a través de una interfaz web. Finalmente, se compara el prototipo contra un equipo de grado clínico y se valida el funcionamiento del sistema con un paciente de prueba y la ayuda de un médico que realizó el diagnóstico.

En este informe se presentan los aspectos más relevantes para el diseño e implementación de cada componente del sistema, de tal manera que este trabajo pueda ser replicado y extendido por aquellos interesados en este campo.

**ABSTRACT**

Since 2020, Nicaragua has seen a considerable increase in the number of deaths from Acute Myocardial Infarction, this being the main cause of death in the country. For this reason, cardiological care has become a necessity for Nicaraguans, however, it is difficult for people who live in rural areas to receive a cardiological evaluation because they must travel long distances to reach the main hospitals in the country located in the cities, which can be especially exhausting for risk groups such as the elderly.

In this sense, this monographic work proposes the development of a tool to facilitate the performance of electrocardiograms so that they can be assessed remotely by a specialist, in order to provide rural health centers with the capacity to provide early and preventive care to heart diseases.

As a result, a digital electrocardiography system was obtained with the ability to capture electrocardiograms from a single lead and send them to a diagnostic doctor through a cloud service so that he can evaluate the exam and issue a diagnosis. The proposed solution consists of an electronic prototype for the acquisition of the cardiac signal, a mobile application that receives this signal from the prototype for visualization, post-processing and sending to the cloud through wireless networks. Furthermore, a cloud service was configured to store these exams that is responsible for forwarding them to the diagnostic doctor via email and receiving their diagnosis through a web interface. Finally, the prototype is compared against clinical grade equipment and the operation of the system is validated with a test patient and the help of a doctor who made the diagnosis.

This report presents the most relevant aspects for the design and implementation of each component of the system, so that this work can be replicated and extended by those interested in this field.

# ÍNDICE DE CONTENIDO

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# INTRODUCCIÓN

El desarrollo de la Telemedicina ha presentado avances significativos en la mejora de la calidad de vida en países desarrollados, particularmente respecto a la atención y los servicios de salud con los que las personas cuentan. Esta tendencia ha sido posible principalmente debido a la extensa difusión y abaratamiento de las Tecnologías de la Información y la Comunicación (TIC). En ese sentido, es posible definir a la telemedicina como el conjunto de servicios de salud apuntalados por el uso de las telecomunicaciones [1, p. 1].

El caso de los países en vías de desarrollo como Nicaragua, caracterizados por la falta de infraestructura, la implementación de estos servicios se ha visto limitada [2]. Sin embargo, con la creciente cobertura de las redes móviles, la ubicuidad de los dispositivos móviles y, en general, la democratización de las TIC plantea un nuevo escenario para que surjan soluciones tecnológicas a algunos de los problemas que aquejan la salud de los nicaragüenses.

Entre las principales enfermedades que representan una seria preocupación según el Mapa Nacional de Salud en Nicaragua [3], están las enfermedades cardiacas pues en 2021 representaron el quinto lugar de incidencia en cuanto a enfermedades crónicas, con 41,726 pacientes afectados, lo cual significa un aumento del 162% respecto al 2019. Además, el Infarto agudo al miocardio ha sido la principal causa de muerte entre la población durante los últimos años llegándose a duplicar la cifra de defunciones en el año 2021 respecto al año 2019.

Es claro que, en estas circunstancias, el cuidado de la salud del corazón cobra gran importancia para la población y para el Ministerio de Salud (MINSA). No obstante, la cobertura del sistema de salud nacional, de acuerdo con las cifras del MINSA, refleja falta de personal y de cobertura, lo que podría significar que una parte de la población se vea impedida de atenderse de manera oportuna este tipo de afecciones. Específicamente, el hecho de que haya tan solo 10 médicos por cada 10 mil habitantes, indica que una valoración por parte de un cardiólogo en

ciertos casos podría dificultarse, especialmente para las personas que deben viajar desde sus domicilios hasta uno de los 73 hospitales públicos que hay en el país, lo cual puede resultar una fatiga y un riesgo para algunos pacientes.

Teniendo en cuenta que la cobertura de redes celulares llega a muchas zonas rurales, que las redes de distribución eléctrica abarcan el 97 % del territorio nacional [4], y que los teléfonos móviles cuentan con capacidades de comunicación que podrían aprovechar para aplicaciones de telemedicina, es posible afirmar que existen condiciones para la atención remota de pacientes con afecciones cardiacas y para el diagnóstico de los casos sospechosos.

El presente trabajo monográfico aborda el desarrollo de un prototipo de electrocardiógrafo digital, con capacidad de enviar electrocardiogramas de una derivación a un servicio en la nube para hacerlos llegar a un médico diagnosta a fin de que sean valorados. Se propuso un modelo en el que el personal de un centro de salud rural utilice este dispositivo para obtener un electrocardiograma de una derivación, el cual contiene información suficiente para el diagnóstico de distintos tipos de arritmias. El sistema desarrollado consta de tres partes: un prototipo electrónico, una aplicación móvil y un servicio en la nube.

# OBJETIVOS

## Objetivo General

Desarrollar un sistema de electrocardiografía digital con acceso a servicio en la nube vía redes móviles para diagnóstico oportuno de afecciones cardiacas en zonas rurales de Nicaragua.

## Objetivos Específicos

1. Construir un dispositivo electrónico costo-efectivo para la adquisición de señales electrocardiográficas.
2. Programar el dispositivo electrónico para el muestreo, filtrado y trasmisión de la señal electrocardiográfica a un terminal móvil.
3. Elaborar una aplicación móvil para el post-procesamiento, visualización y reenvío de electrocardiogramas a un servidor en la nube a través de redes móviles.
4. Realizar la configuración de un servicio en la nube para el alojamiento de electrocardiogramas

## JUSTIFICACION

En Nicaragua, al ser un país en vías de desarrollo, no ha habido una gran difusión de los avances en Telemedicina, siendo pocos los desarrollos que facilitan el acceso a servicios de salud especializados a través de medios de telecomunicaciones, a personas que viven en zonas rurales. Sin embargo, es posible afirmar que existen condiciones sobre las cuales empezar a implementar aplicaciones de Telemedicina para acercar la atención médica hospitalaria a habitantes de zonas remotas, a través de las redes móviles y mediante el uso de las Tecnologías de la Información.

Una de las principales necesidades que existe entre la población nicaragüense de manera general, es la atención cardiológica, pues según datos del MINSA, las afecciones cardiovasculares están entre las de mayor incidencia y también la que mayor cantidad de defunciones provocan. Según datos oficiales del Ministerios de Salud, se registró un alza en la mortalidad por Infartos Agudos al Miocardio en el año 2020, marcando un aumento del 67% respecto al promedio de los 3 años anteriores, mientras que en ese mismo año se registró un 183% de aumento en los pacientes crónicos del corazón [3]. Como consecuencia, es posible considerar estas cifras con un indicador del alto el número de personas que han de requerir ser valorados por un cardiólogo en los próximos años. Por ello surge la necesidad de proveer al sistema nacional de salud, y particularmente a los centros de salud, de herramientas para poder llevar a cabo tele-diagnóstico de afectaciones cardiacas.

Si bien es cierto que en el mercado existen equipos de electrocardiografía, estos generalmente tienen un costo elevado, sin mencionar que estos equipos no han sido desarrollados para operar en escenarios con las limitaciones de recursos de un sistema de salud como el nicaragüense. De ahí, que sean necesarios encaminar esfuerzos para que el desarrollo tecnológico considere las herramientas que se tienen a disposición en el país.

En ese sentido, el presente trabajo de investigación busca ofrecer una alternativa para que las condiciones cardiacas de los pacientes alejados de los hospitales puedan ser valoradas por un especialista, sin que tengan que movilizarse, y así evitarles el gasto en transporte y la fatiga asociada, que es especialmente preocupante en pacientes de la tercera edad. Las zonas rurales donde existe cobertura celular son un buen ejemplo de los lugares donde se puede aplicar este trabajo, pero también es posible hacerlo en zonas donde no hay tal cobertura.

Por otra parte, hay que señalar que este proyecto monográfico es la continuación de un proyecto estudiantil propuesto por el Br. Yeser Morales y desarrollado en el seno de la Unidad de Proyectos del Departamento de Electrónica, que consiste en un electrocardiógrafo digital con capacidad de registrar un electrocardiograma de una derivación y además transmitirla a una computadora para su visualización. Este fue presentado en el III Congreso Bienal de Investigación y Posgrado en la Universidad San Carlos de Guatemala, en el año 2016.

# 1 CAPÍTULO I: MARCO TEÓRICO

En este capítulo se realiza una revisión de los conceptos relacionados a las aplicaciones de la Electromedicina, particularmente en el ámbito de la Electrocardiografía y la Telemedicina. En primer lugar, se aborda el concepto de Telemedicina. Luego se hace un repaso de la Electrocardiografía. A continuación, se realiza una revisión de los fundamentos de la adquisición y procesamiento de señales electrocardiográficas. Por último, se aborda los aspectos de las Tecnologías de la Información y la Comunicación que le atañen a la Telemedicina.

## 1.1 Telemedicina

R. Gupta sostiene que la Telemedicina es una “plataforma tecnológica integrada donde un paciente remoto puede ser examinado o monitoreado a través de un enlace de comunicación por un médico” [5, p. 10]. Esta definición resume lo que puede ser logrado a través de la implementación de todo un conjunto de técnicas de comunicación y manejo de información, haciendo que la falta de un entorno clínico tradicional no sea un obstáculo para prestar servicios de salud.

Si bien, la palabra «Telemedicina» es la que se utiliza más comúnmente para definir la provisión de monitoreo y diagnóstico médico de forma remota, también suele utilizarse la palabra «Telesalud» de forma intercambiable. Sin embargo, en algunas contextos, ésta última incluye un conjunto más amplio de servicios como prevención y educación.

En este sentido, se plantea un número de nuevas relaciones que pueden establecerse entre miembros del personal de salud y los pacientes. Entre estas es posible destacar algunas como la tele-consulta, cuando un paciente expone sus síntomas al médico de manera virtual, mientras que este se forma un criterio sobre la manera de proceder para dar solución a ese paciente; tele-diagnóstico, que puede darse cuando un médico consulta con un especialista sobre la condición de un determinado paciente, y en el que emplean medios de comunicación como la videoconferencia, pero también otros como el email para

compartir información necesaria; tele-monitoreo, donde los pacientes son atendidos por enfermeras para dar seguimiento a la evolución de un tratamiento; e incluso, la tele-cirugía. La Figura 1 expone esquemáticamente estas relaciones.

### 1.1.1 Modalidades

El Centro para Política de Salud Conectada (CCHP) define algunas modalidades de Telemedicina según su aplicación y el nivel de sincronía estas permiten. Se definen de la siguiente manera [6]:

- Almacenamiento y reenvío: Es de carácter asíncrono, los exámenes se almacenas y se transmiten cuando esté disponible una conexión para que sean evaluados cuando sea posible.
- Monitoreo remoto: Si bien de una modalidad asíncrona, el envío de información se realiza periódicamente y de igual forma las evaluaciones se hacen periódicamente, aunque no de forma inmediata.
- Salud Móvil: a través del uso de dispositivos móviles es posible estar en contacto con un especialista sincrónicamente para realizar consulta vía mensajes de texto o bien recibir notificaciones de epidemias.
- Videoconferencia: consiste en atención en tiempo real a través de videollamada y puede darse entre paciente y médico, o bien entre un médico y un especialista que puede guiar al primero en un procedimiento.

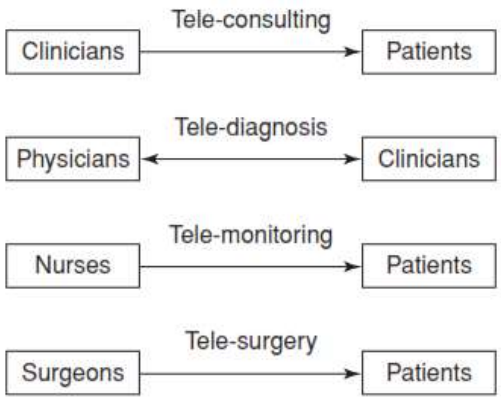



*Figura 1. Nuevas relaciones surgidas a partir de las TIC en la salud. Adaptado de B. Fong [32, p. 10]*

### 1.1.2 Telecardiología

Un caso particular que concierne al propósito de este trabajo monográfico es el de la Telecardiología, la cual se define como la aplicación de la Telemedicina en el monitoreo de la función cardiaca de un paciente remoto. En consecuencia, abarca un buen número de escenarios en la que puede ser de utilidad, como el seguimiento de pacientes de la tercera edad, chequeo pre y post hospitalarios de pacientes cardiacos, entre otros. De igual forma, los tipos de señal que pueden ser usados para monitorear los pacientes en este contexto pueden ser electrocardiogramas, ritmo cardiaco y señales fotopletismográficas (obtenidas al usar luz para detectar el flujo de sangre en un tejido).

R. Gupta propone un modelo de Telecardiología que cuenta con un sistema de 3 etapas del lado del paciente para la captura y envío de un electrocardiograma, y un sistema de 2 etapas para la recepción del electrocardiograma del lado del médico diagnosta, tal como se aprecia en la Figura 2.

Nótese que las funciones de compresión, codificación y empaquetado de la señal en cuestión está separadas del bloque que se encarga de su transmisión, sin embargo, estas funciones también se pueden unificar en una única etapa que se

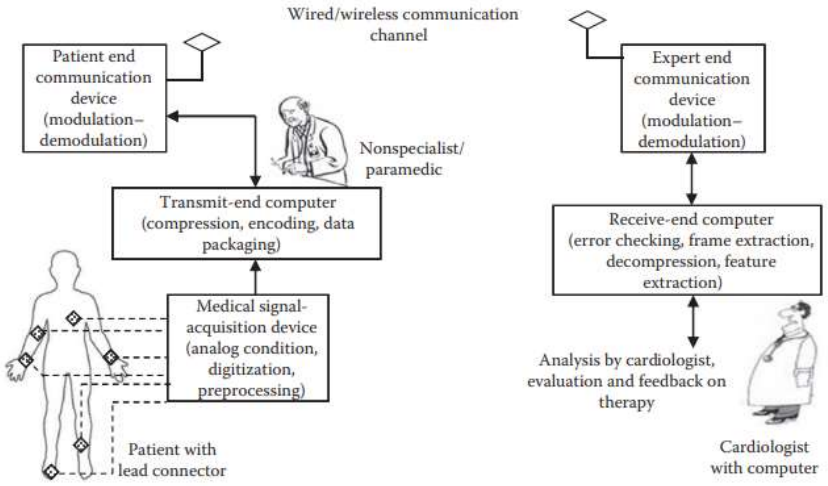



*Figura 2. Diagrama de bloques de un Sistema de Telecardiología. Adaptado de R. Gupta [5].*

puede denominar bloque de procesamiento en el dominio digital, quedando así separada la etapa de adquisición de la señal, a como muestra la Figura 3.

Por último, cabe diferenciar entre dos tipos de aparatos telecardiológicos que cumplen funciones distintas. Por un lado, están los aparatos estacionarios diseñados para estar en una instalación de salud y ser operados por personal médico no necesariamente especializado, cumpliendo con la función de examen médico de forma puntual, y por el otro, están los dispositivos de uso ambulatorio, también conocidos como *holters*, que suelen ser de pequeño tamaño operados a baterías para que puedan ser portados por los pacientes, y cumplen con la función de monitoreo en el mediano y largo plazo, especialmente en pacientes con condiciones cardiológicas ya diagnosticadas y cuya vida corre algún riesgo a raíz de ello.

## 1.2 Electrocardiografía

La Electrocardiografía es una técnica médica que se utiliza para analizar la actividad eléctrica del corazón a través de la generación e interpretación de electrocardiogramas. De esta forma, un electrocardiograma no es más que una gráfica de voltaje en función del tiempo que representa la actividad eléctrica del músculo cardíaco [7, p. 1033]. Es ampliamente utilizado en la práctica de la cardiología, ya que es examen no invasivo.

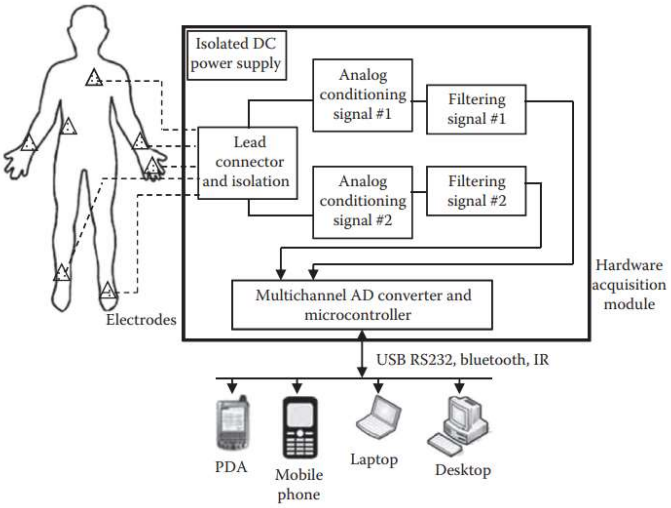



*Figura 3. Diagrama de bloques de un sistema de adquisición de señales. Adaptado de R. Gupta [5].*

### 1.2.1 El sistema cardiovascular

El sistema cardiovascular es de vital importancia para el organismo de los humanos y muchos otros seres vivos, pues se encarga del transporte rápido de moléculas entre órganos y tejidos especializados a lo largo del cuerpo, ya sea para llevar oxígeno y nutrientes a las células, o para recolectar desechos de la actividad celular [8, p. 3]. Este sistema está conformado principalmente por la sangre, los vasos sanguíneos (venas, arterias y vasos capilares) y el corazón.

La sangre está conformada por plasma, y células como glóbulos rojos, glóbulos blancos y plaquetas. El plasma es un líquido en el que están disueltas proteínas, nutrientes, productos de desecho metabólico, entre otras moléculas.

Los vasos sanguíneos, por su parte, son ductos a través de los cuales circula la sangre y que se ramifican en múltiples vasos sanguíneos cada vez más pequeños hasta llegar a lo que se conoce como capilares. Estos conforman dos sistemas de lazo cerrado [8, p. 4], a como se esquematiza en la Figura 4.

### 1.2.2 El corazón

R. Gupta define el corazón como “un órgano flexible y hueco con cuatro cavidades [...], un par de aurículas y un par de ventrículos, longitudinalmente unidos” [5, p. 1]. En la Figura 4 las aurículas aparecen marcadas como RA y LA (siglas en inglés de aurícula derecha y aurícula izquierda, respectivamente), mientras que los ventrículos son representados por RV y LV (siglas en inglés de ventrículo derecho y ventrículo izquierdo, respectivamente). De igual manera, el corazón cuenta con cuatro válvulas que aseguran el flujo unidireccional de la sangre: válvula tricúspide, válvula mitral, válvula pulmonar y válvula aórtica.

El corazón es considerado como uno de los órganos más importantes del cuerpo y su función es, *grosso modo*, generar presión para impulsar la sangre a través de los vasos sanguíneos. En ese sentido, la descripción fisiológica y funcional propuesta por A. Katz [9, p. 3] establece que el corazón puede ser considerado como dos bombas que operan en serie:  RA en conjunto con RV, que impulsan la

sangre a la circulación pulmonar; y LA que, en conjunto con LV, bombean la sangre hacia la circulación sistémica.

Sumariamente, es posible describir un ciclo cardíaco completo de la siguiente manera: la sangre pobre en oxígeno llena la aurícula derecha, luego de una pausa, la sangre pasa al ventrículo derecho, desde la que es impulsada hacia los pulmones. Una vez en los allí, la sangre es oxigenada en los alveolos pulmonares, y retorna hacia la aurícula izquierda. Cuando se alcanza la presión suficiente, la sangre rica en oxígeno pasa hacia el ventrículo izquierdo, desde donde es bombeada hacia el resto del cuerpo [5].

### 1.2.3 Naturaleza de la señal electrocardiográfica

Los latidos cardíacos son ocasionados por impulsos eléctricos que se generan en células especializadas llamadas cardiomiocitos, y que se propagan a través del resto de células musculares del corazón (células miocárdicas). La diferencia de potencial que se genera corresponde a la despolarización (contracción), y la repolarización (retorno al estado de reposo) de las fibras miocárdicas [10, pp. 1-2].

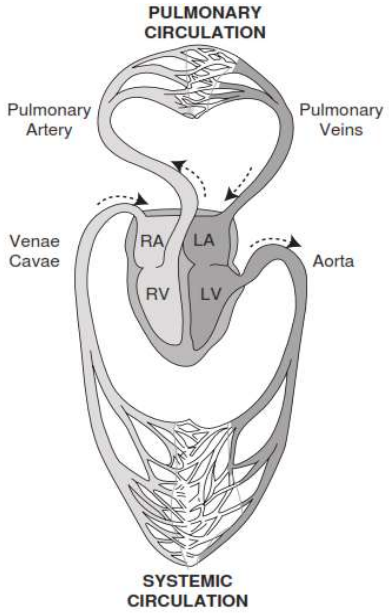



*Figura 4. Esquema simplificado del sistema cardiovascular. Adaptado de A. Katz [9].*

Físicamente, lo que ocurre es un intercambio de iones entre el interior de la célula y el exterior a través de su membrana, pues las células pueden regular su permeabilidad a ciertos iones como potasio, sodio, calcio y cloro. Este intercambio de iones modifica la distribución de carga eléctrica en la membrana, dando lugar a una onda conocida como potencial de acción.

En ese sentido, cada célula cardíaca puede ser modelada como un pequeño dipolo cuya magnitud y polaridad varía en el tiempo. La Figura 5 muestra este proceso, donde **A** representa una célula polarizada (membrana cargada positivamente) y **C** simboliza a la célula cuando está despolarizada (membrana cargada negativamente); **B** y **D** representan las transiciones entre un estado y otro, en las que se puede apreciar el potencial de acción como un frente de onda; mientras que **E** simboliza a la célula repolarizada, es decir, de vuelta a su estado de reposo. A la derecha que cada fase se muestra un bosquejo de la diferencia de potencial que tiene lugar en cada momento. Esta secuencia ocurre de manera coordinada entre todas las células que conforman el tejido cardiaco, de tal modo que el electrocardiograma no mide directamente la actividad eléctrica de cada célula, sino al vector eléctrico resultante de la suma vectorial de los potenciales de estos dipolos, a medida que se van despolarizando y repolarizando [11, p. 78].

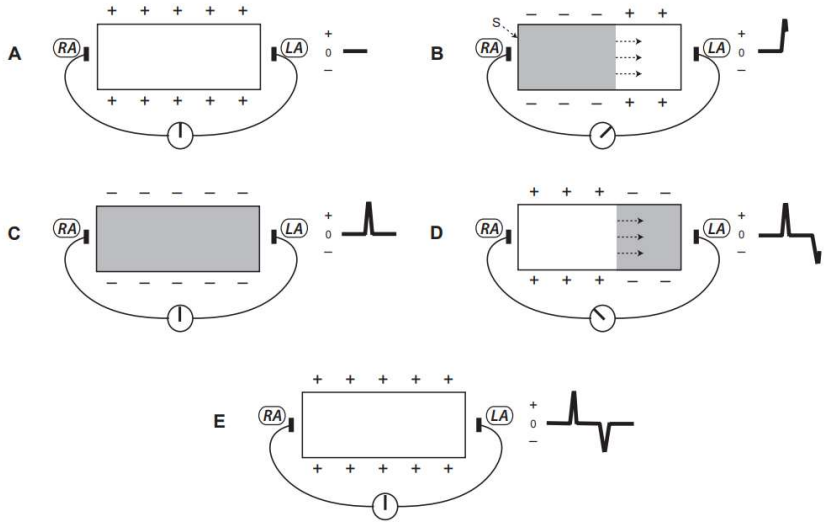



*Figura 5. Representación esquemática de la despolarización y repolarización en una célula miocárdica. Adaptado de A. Katz [9].*

Por su parte, J. Hampton describe el proceso de propagación del impulso como una descarga eléctrica que normalmente comienza en el nodo sinoauricular (SA), la cual se extiende a través de las fibras musculares de la aurícula, seguida de un retraso mientras la despolarización se propaga a través del nodo atrioventricular (AV). A partir de entonces, la onda viaja muy rápidamente hacia abajo al tejido de conducción, el 'Haz de His', que luego se divide en rama izquierda y derecha [12, p. 4]. De esta manera, en la Figura 6 es fácil observar la ruta que toma el impulso eléctrico a partir de 1) el Nodo SA, a través de 2) la aurícula derecha y luego por 3) el tabique o septum, hasta bifurcarse finalmente en 4) las dos ramas ventriculares. Una vez que termina esta secuencia, el corazón regresa a un estado de reposo por un breve periodo de tiempo antes de empezar un nuevo latido.

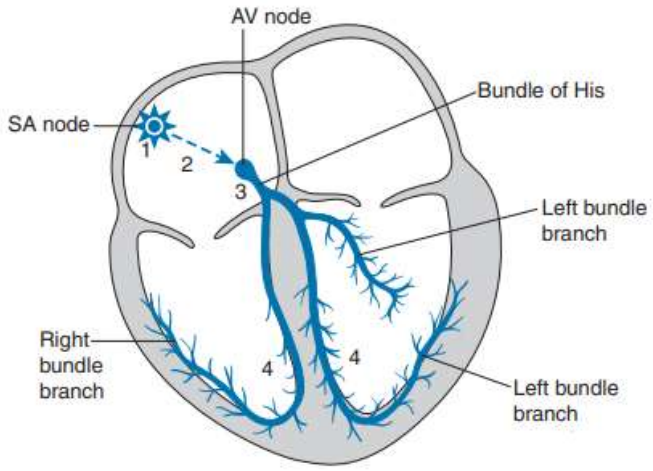



*Figura 6. Estructuras de conducción del impulso eléctrico. Adaptado de L. Lilly [11, p. 82]*

Como resultado de este proceso, se produce una corriente de naturaleza iónica en el cuerpo y se establecen diferencias de potencial en la piel. Es importante destacar que la actividad eléctrica de otras células musculares también puede generar estas diferencias de potencial, por tanto, es recomendable el estado de reposo al registrar la señal electrocardiográfica.

### 1.2.4 Electrocardiograma

Un electrocardiograma (comúnmente abreviado como ECG o EKG) es la representación de los cambios en la actividad eléctrica del corazón en el tiempo [8, p. 257]. Esta bioseñal es obtenida al adherir electrodos a ciertos puntos la superficie del cuerpo, generalmente pecho y abdomen, aunque es común también colocarlos en las extremidades. A continuación, se mide, amplifica y filtra las diferencias de potencial que se dan entre los electrodos para obtener la magnitud de interés, que también es conocida como ritmo sinusal.

Al instrumento encargado de adquirir la señal eléctrica del corazón y graficarla se le conoce como electrocardiógrafo. Estos se clasifican en mecánicos y digitales, según la técnica utilizada para capturar la señal y generar la gráfica. Asimismo, varían en el número de señales (derivaciones) que pueden capturar, siendo los más comunes los de 3 y 12 derivaciones, tal como se explica más adelante.

Para graficar la señal se suele utilizar una cuadrícula como la que se muestra en la Figura 7. El eje vertical corresponde a la amplitud de la señal y cada división equivale a 0.5 milivolts, mientras que en el eje horizontal corresponde a la magnitud del tiempo y cada división representa 0.2 segundos.

En los electrocardiógrafos mecánicos se utiliza un papel cuadriculado con una escala de 5 mm por división y éste se suele desplazar a razón de 25 mm/s. De igual forma, para ayudar en el análisis de la señal se suelen trazar subdivisiones, de modo que cada división es a su vez una cuadrícula, donde cada subdivisión de 1 milímetro en el eje vertical es igual a 0.1 milivolt y 1 milímetro son 0.04 segundos en el eje horizontal. Esta costumbre permanece en la actualidad para representar electrocardiogramas de forma digital, incluso si este no está impreso.

La utilidad de los electrocardiogramas estriba en la posibilidad de diagnosticar diferentes afecciones cardiacas de forma no invasiva, ya que, a través del estudio de la morfología de la señal, es posible detectar anormalidades en el

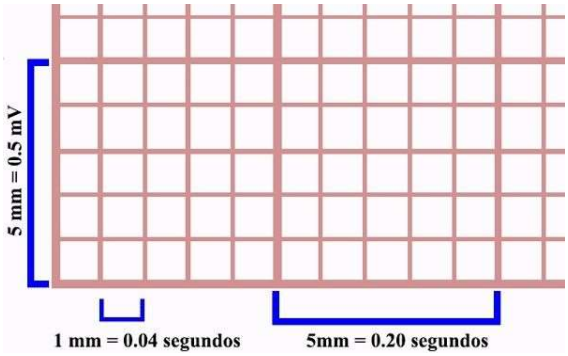



*Figura 7. Vista ampliada de una cuadrícula para graficar señales electrocardiográficas.*

funcionamiento del corazón, tales como arritmias, isquemia, hipertrofia, entre otras.

### 1.2.5 Derivaciones

Una derivación es la diferencia de potencial entre dos electrodos físicamente colocados sobre la piel, o entre un electrodo físico y uno virtual llamado electrodo de Wilson. Cada derivación es una proyección del vector eléctrico del corazón en un ángulo particular [12, p. 9]. En consecuencia, la forma del ECG tendrá un aspecto diferente en dependencia de la posición de los electrodos [13, p. 258].

Como se dijo antes, hay electrocardiógrafos de 3 y 12 derivaciones, correspondientes a algunos sistemas que se han desarrollado a lo largo de la historia como un esfuerzo por estandarizar esta técnica, sin embargo, también los hay en versiones de 6 derivaciones, que suelen ser un subconjunto de los de 12 derivaciones. Los más comunes actualmente son los de 12 derivaciones, ya que son los que se encuentran en la mayoría de los centros hospitalarios. Sin embargo, tres de estas 12 derivaciones, son las que históricamente más se han utilizado, llamadas I, II y III, debido a que la derivación II es el modelo de ECG. Además, es una de las más recomendada para análisis de ritmo cardiaco [12, p. 36].

Un primer intento por estandarizar la técnica del ECG fue propuesto por el médico holandés Wilhem Einthoven. Su método consistía es localizar los electrodos en un triángulo en cuyo centro se encuentra el corazón, tal como se muestra en la

Figura 8a, donde los electrodos reciben los nombres de RA, LA y LL, que corresponde a brazo derecho, brazo y pierna izquierdos respectivamente. A este arreglo se le conoce como el triángulo de Einthoven. Cada derivación, nombrada convencionalmente como I, II y III se obtienen al medir las diferencias de potencial correspondiente a cada la del triángulo. Los signos "+" y "–" indican la polaridad con que se debe realizar la medición y conforman las derivaciones estándares de un ECG, también llamadas derivaciones periféricas. En consecuencia, estas derivaciones se consideran bipolares y se definen como:

$$I = V_{LA} - V_{RA} \quad (1)$$

$$II = V_{LL} - V_{RA} \quad (2)$$

$$III = V_{LL} - V_{LA} \quad (3)$$

De esta manera, se busca medir el vector de despolarización del corazón (eje eléctrico) en el plano frontal [10, p. 4]. Cada derivación muestra la componente del vector en esa dirección en un instante dado.

Las demás derivaciones son conocidas como aVR, aVL, aVF, las cuales también son obtenidas a partir de los mismos tres electrodos, y V1, V2, V3, V4, V5, V6

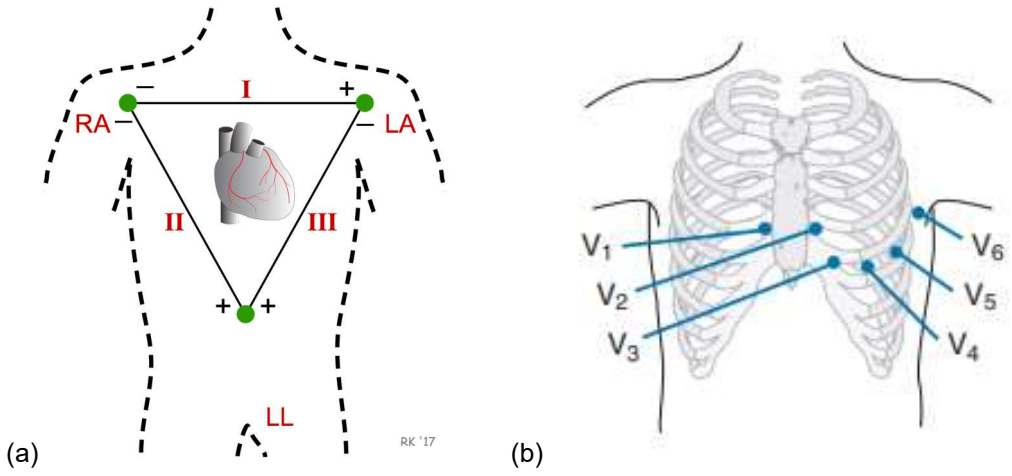



*Figura 8. a) Triángulo de Einthoven. b) Colocación de los electrodos torácicos, en las posiciones estándar. Adaptado de L. Lilly [11].*

cuya ubicación se muestra en la Figura 8b, conocidas como derivaciones precordiales. Todas son de tipo uniporlar pues su potencial se mide respecto a un derivación virtual conocida como Terminal Central de Wilson, la cual no es más que el promedio de los potenciales en los electrodos periféricos.

### 1.2.6 Morfología de la señal

La señal electrocardiográfica de un corazón sano se caracteriza por tener cinco crestas u ondas, conocidas como P, Q, R, S, T, y en ocasiones aparece una sexta cresta de menor magnitud U, que se considera normal. En la Figura 9 se muestra la forma de la señal ECG, en la que destaca el llamado complejo QRS por ser el de mayor amplitud.

El ciclo completo de un latido cardiaco puede ser interpretado en el ECG de la siguiente manera: la onda P indica la despolarización (contracción) auricular, el complejo QRS se produce durante la despolarización ventricular y debe su amplitud a la abundante musculatura que conforman los ventrículos, mientras que la onda T corresponde a la repolarización (expansión) ventricular. La repolarización auricular también ocurre durante la despolarización ventricular, pero se ve opacada dada la magnitud del complejo QRS [5, p. 4] [14, p. 117]. Adicionalmente, el intervalo R-R es nombre que recibe un latido completo.

Por otro lado, a la periodicidad con que late el corazón se le conoce como ritmo cardiaco. En un adulto sano y en reposo es usualmente de entre 60 – 100 latidos por minuto (o bpm por sus siglas en inglés) [9, p. 431] [14, p. 191]. Aunque hay

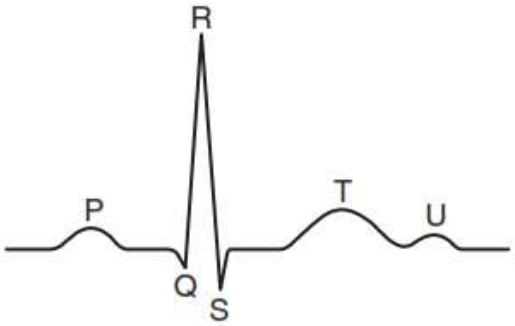



*Figura 9. Forma de una señal ECG saludable incluyendo la onda U.*

autores que afirman que el intervalo es más acotado, de entre 60 – 70 bpm [10, p. 3] [15, p. 411].

Adicionalmente, hay otros intervalos dentro de la señal electrocardiográfica cuya duración se considera normal cuando están dentro de un rango establecido como se indica en la Tabla I, dentro de los cuales destaca el intervalo QTc por ser un parámetro normalizado respecto a la frecuencia cardiaca, lo cual permite comparar directamente su duración entre diferentes observaciones.

*Tabla I – Intervalos electrocardiográficos. Adaptado de L. Lilly [11, p. 89].*

| **Intervalo** | **Definición** | **Duración normal** |
|---|---|---|
| PR | Intervalo medido desde el inicio de la onda P hasta el inicio de la onda QRS. | 0.12 – 0.20 segundos |
| QRS | Es la duración del complejo QRS. | ≤ 0.10 segundos |
| QT corregido | QTc es la duración desde el inicio de QRS hasta el final de la onda T, dividida entre la raíz cuadrada del intervalo R-R. La división se hace con el objetivo de compensar la variabilidad del ritmo cardiaco. | ≤ 0.44 segundos |

## 1.3 Tratamiento de señales electrocardiográficas

Dado que la actividad corresponde a un fenómeno electroquímico, es necesario aplicar un conjunto de técnicas encaminadas a adquirir la señal y mejorar su calidad, de tal manera que pueda ser útil para un diagnóstico médico.

### 1.3.1 Adquisición de la señal

#### 1.3.1.1 Electrodos

Los electrodos son sensores de biopotenciales. Estos convierten concentraciones iónicas a conducción electrónica a través de una reacción química [16, p. 1]. Esto los hace complicados y establece ciertas restricciones respecto al posicionamiento de estos en la superficie del cuerpo [17, p. 189].

Dado que la naturaleza de actividad eléctrica celular es de carácter iónica, es necesario obtener una corriente electrónica, a través de la interfaz electrodo-electrolito (cuerpo humano), de forma que pueda ser una señal utilizable por circuitos electrónicos. Esto, en electrocardiografía, se consigue comúnmente mediante unos electrodos superficiales de plata/cloruro de plata (Ag/AgCl), aunque existen de otros tipos.

Esta interfaz que se forma entre el tejido humano y los cables que conducen hacia el electrocardiógrafo puede ser modelado a través de un circuito equivalente. Según M. Neuman [17, p. 202] "a menudo se encuentra que las características corriente-voltaje de la interfaz electrodo-electrolito son no lineales, y, a su vez, se requiere elementos no lineales para modelar el comportamiento de los electrodos". Como resultado el comportamiento de los electrodos en general varía en función de la frecuencia y otras variables físicas. Un modelo sencillo de un electrodo es presentado en la Figura 10, donde una fuente de voltaje se encuentra en serie con un circuito paralelo de una resistencia Re y una capacitancia Ce, cuyo efecto resultante es una menor impedancia a mayores frecuencias de la señal.

#### 1.3.1.2 Amplificación

La amplificación es un paso muy importante en la adquisición de en ECG, puesto que las señales con que se trabaja tienen amplitudes que van desde las centenas de microvoltios hasta el orden de los milivoltios [18, p. 29]. Para tal fin se hace uso de un Amplificador de Instrumentación, cuyo modelo se muestra en la Figura 11.

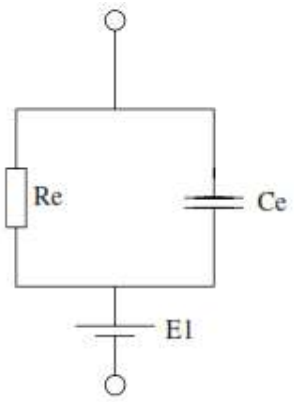



*Figura 10. Modelo eléctrico de un electrodo para biopotenciales.*

*E1 es el potencial de media celda, mientras que Re y Ce representan la interfaz electrodo electrolito.*

La razón por la que no se utiliza un amplificador operacional para este fin, es que existen unas componentes de voltaje de modo común incluidas en la señal ECG bruta. Mediante la característica diferencial del amplificador de instrumentación es posible atenuar estas señales no deseadas. Por tanto, el rechazo a señales de modo común es una de las más importantes características de un buen amplificador de instrumentación [19, p. 2].

Algunos de los parámetros deseables en un amplificador de instrumentación son los siguientes [5, pp. 55-56]:

- Ganancia (750 nominal)
- Ancho de banda 1 kHz
- Razón de Rechazo en Modo Común (1000 o 60 dB típico)
- Impedancia de entrada (10 MOhm)

La característica más distintiva de los amplificadores de instrumentación es su alta Razón de Rechazo de Modo Común (CMRR), lo que los hace una excelente opción para amplificar señales tan tenues como la del corazón.

Asimismo, hay algunas mejoras que se suelen añadir a un electrocardiógrafo como la reducción de la interferencia eléctrica, aislamiento del paciente y el filtrado. Si bien esto último se suele aplicar cuando la señal ya está digitalizada, también es necesario aplicar filtrado en la etapa de amplificación para garantizar

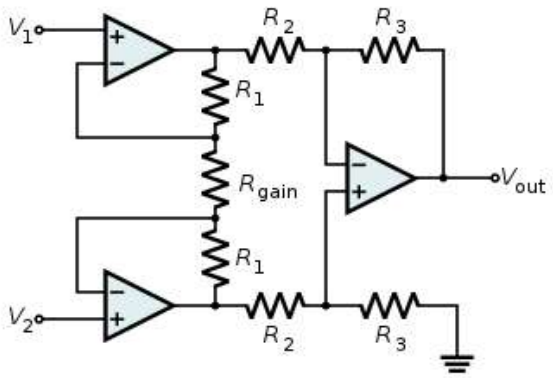



*Figura 11. Esquema de un amplificador de instrumentación.*

que el contenido espectral de la señal analógica esté por debajo de la Frecuencia de Nyquist, evitando que se produzca solapamiento (*aliasing*) al digitalizar la señal.

#### 1.3.1.3 Muestreo

El proceso de muestreo implica la conversión de la señal ECG del dominio analógico al dominio digital, mediante un conversor. Los dos parámetros más importantes en este proceso son, por un lado, la tasa de muestreo, que debe satisfacer el teorema de Nyquist, y por otro, la resolución del convertidor analógico -digital, pues de esta depende la calidad con que será cuantizada la señal.

En el caso de la frecuencia de muestreo, es importante asegurar que todas las componentes en espectro se encuentren por debajo de la frecuencia de Nyquist. De manera general, las componentes espectrales de las bioseñales se encuentran por debajo de los 1,000 Hz [20, p. 15], sin embargo, se considera que, en el caso del ECG, solo hay información clínicamente significativa en los primeros 100 Hz [5, p. 52] [14, p. 191]. Por tanto, es común encontrar aplicaciones donde se emplean tasas de muestreo de 250, 360 o hasta 500 Hz.

Por otro lado, la resolución mínima recomendada por algunos autores, para la digitalización de un ECG es de 12 bits [20, p. 15] [5, p. 54], que, luego de la etapa de amplificación, debe procurarse que la señal ocupe todo el rango dinámico del

convertidor analógico-digital (ADC). Si bien no existe, una resolución estándar, la selección de este convertidor, estará condicionadas por los requerimientos de la aplicación específica.

#### 1.3.1.4 Tipos de convertidores analógico-digital (ADC)

Existen diversos tipos de convertidores analógico-digital basados en diferentes estrategias de cuantización de señales, cada uno con sus respectivas ventajas y desventajas. A continuación, se repasan algunos de los más ampliamente utilizados según [21, pp. 826-840].

- Registro de aproximaciones sucesivas (SAR): consta de un registro con un número de bits N, el cual determina su resolución. La lógica de control modifica el contenido del registro hasta que es el equivalente digital de la entrada analógica. Una ventaja de este ADC es que presenta un tiempo de conversión relativamente rápido. Una desventaja es que para alcanzar altas resoluciones el costo puede ser prohibitivo.
- Flash: es de los que mejores tiempos de conversión presenta, pero requiere más circuitería que otros tipos. Su funcionamiento se basa en una escalera de $2^N$ resistores que forman un divisor de voltaje. Cada nodo sirve de referencia para un conjunto de $2^N - 1$ comparadores de voltaje, donde N es la resolución del ADC. La salida digital es el resultado de codificar la cantidad de comparadores que se activaron al comparar su referencia con la entrada analógica.
- Sigma-Delta: Se basa en el principio de la modulación Sigma-Delta, donde el convertidor muestrea la entrada analógica a una tasa más alta que la obtenida a la salida (*oversampling*) y produce un *stream* de un único bit a la salida. El voltaje a la entrada se corresponde con una cierta densidad de 1's lógicos proporcional a la magnitud de éste. Finalmente, la salida digital de N bits se promedia la densidad de bits de $2^N$ muestras del modulador Delta-Sigma.

### 1.3.2 Procesamiento digital

Luego de la etapa de adquisición, la señal aun contiene componentes indeseables o espurias, llamadas artefactos, de las que se debe librar la señal de interés. Además, una vez que se obtiene una señal con calidad aceptable, es necesario conocer algunos parámetros, en aras de ayudar al diagnosta en la interpretación del ECG. Por último, es conveniente aplicar compresión a la señal para hacer un uso eficiente del espacio de almacenamiento y/o ancho de banda, según sea el caso. Para ello se hace uso de técnicas de procesamiento de señales.

#### 1.3.2.1 Filtrado de ruido

Algunas de las fuentes de ruido que contaminan una señal ECG bruta, son las siguientes: ruido electromiográfico, interferencia de línea eléctrica, ruido de contacto, artefactos motrices, ruido de amplificación y ruido de cuantización. Algunos de ellos pueden ser atenuados mediante filtros pasa banda, sin embargo, los espectros de algunos otros tipos de ruido se traslapan con el espectro del propio ECG, por lo tanto, más difícil librarse de ellos. Según R. Gupta [5, p. 53], hoy en día es imposible eliminarlos completamente, sin embargo, con el uso métodos adecuados se pueden obtener resultados satisfactorios.

En este sentido, algunas de las técnicas que más se utilizan, son filtros digitales, análisis de ondeleta (*wavelet*), entre otros. El desempeño de estos métodos puede ser medido a través de parámetros de desempeño, tales como: Error Cuadrático Medio (MSE) y Razón Señal a Ruido (SNR).

Dentro de los filtros digitales existen múltiples formas de clasificarlos. En la Tabla II se aborda una de las más importantes, que es la clasificación por la respuesta en frecuencia, la cual determina si el filtro atenúa o no ciertos rangos de frecuencias.

*Tabla II – Clasificación de los filtros digitales según su respuesta en frecuencia.*

| Tipo | Características |
|---|---|
| Pasa-bajas | Deja pasar solamente frecuencias entre 0 y una frecuencia de corte $f_0$, a partir de la cual empieza a atenuar las frecuencias mayores. |
| Pasa-banda | Permite el paso de frecuencias comprendidas entre $f_1$ y $f_2$. |
| Pasa-altas | Deja pasar solamente frecuencias entre una frecuencia de corte $f_0$ y la frecuencia de Nyquist del sistema $f_N$. |
| Rechaza-banda | Atenúa solamente frecuencias comprendidas entre $f_1$ y $f_2$ siendo $f_1 < f_2$. Un caso particular es el filtro Notch que atenúa una única frecuencia y es útil en la supresión de ruido de la red eléctrica. |
| Pasa-todo | Es útil para cambiar la fase de una señal. |

Si bien es cierto que la clasificación por respuesta en frecuencia es universal para los filtros electrónicos en general, hay algunas de gran importancia que solo les conciernen a los filtros digitales como es el caso de la clasificación por su respuesta al impulso.

En ese sentido existen dos tipos [22, p. 114]:

- Respuesta Finita al Impulso (FIR): su salida se anula después de un determinado número de muestras, luego de aplicar un impulso a su entrada. Se caracterizan por tener únicamente ceros en el plano de la transformada Z y por ser un sistema sin retroalimentación, es decir, cuya salida depende únicamente de su entrada y de muestras pasadas de la señal de entradas.
- Respuesta Infinita al Impulso (IIR): su salida nunca se convierte en cero luego de aplicar un impulso, aunque correctamente diseñado esta

respuesta debería decrecer para garantizar que la estabilidad del filtro. Constan de polos y ceros en el plano Z y es un sistema con retroalimentación, es decir, su salida depende de la señal de entrada y de muestras retrasadas de su señal de salida.

Las ventajas y desventajas de cada tipo se enumeran en la Tabla III.

*Tabla III – Clasificación de los filtros digitales por su respuesta al impulso.*

| Tipo | Ventajas | Desventajas |
|---|---|---|
| Respuesta Finita al Impulso (FIR) | • Siempre son estables.<br>• Pueden tener fase lineal.<br>• Son fáciles de diseñar. | • Son computacionalmente más demandantes.<br>• Suelen introducir mayor latencia en la señal. |
| Respuesta Infinita al Impulso (IIR) | • Son computacionalmente eficientes. | • Pueden ser inestables.<br>• Introducen distorsiones de fase. |

#### 1.3.2.2 Estimación de características

La mayoría de las características que resultan de interés para la evaluación médica son las medidas de ciertos intervalos y amplitudes. Entre ellos están anchura de la onda P, del complejo QRS, y de los segmentos QTc y PQ/PR, por el lado del dominio del tiempo, mientras que en el caso de las magnitudes de interés están la de las ondas P y T, así como la altura del complejo QRS y el nivel ST [23, p. 62].

Para la medición de intervalos R-R, a partir del cual se puede obtener la frecuencia cardiaca, se han propuesto varios métodos que pasan por la detección del complejo QRS, entre los que destacan la transformada ondeleta (*wavelet*) y otras transformaciones no lineales, al mismo tiempo que redes neuronales. Sin

embargo, un modelo general para la detección de complejo QRS se esquematiza en la Figura 12.

Un ejemplo clásico de estos métodos es el algoritmo de Pan-Tompkins [24], propuesto en 1985, que implementa el modelo descrito en la Figura 9. En pocas palabras utiliza filtros digitales sobre la señal ECG para luego elevarla al cuadrado, de tal forma que la contribución del complejo QRS se vea amplificada, para posteriormente aplicar una regla de umbral a para detectar los picos R de la señal.

Por muchos años el algoritmo de Pan-Tompkins fue uno de los pocos existentes para este fin, sin embargo, a partir del año 2000, se empezaron a proponer métodos nuevos, como es el caso del algoritmo Hamilton [25], propuesto en el año 2002, que es una versión mejorada del primero, siendo capaz de adaptarse a diferentes frecuencias de muestreo.

Por su parte, el algoritmo de Christov [26], propuesto en 2004, consiste en un método de umbral adaptativo basado en 3 estimadores de umbral, que presentan una precisión igual o superior que otros algoritmos propuestos hasta la fecha, además de ser auto sincronizado con la señal ECG, permitiéndole operar a distintas tasas de muestreo.

Del mismo modo, es posible emplear otros métodos avanzados para estimar características de señales ECG, por ejemplo: filtrado adaptivo; análisis tiempo-frecuencia, usando transformada ondeleta y otras transformaciones no lineales; y Análisis de Componente Principal (PCA), redes neuronales, entre otros [5, pp. 24-25].

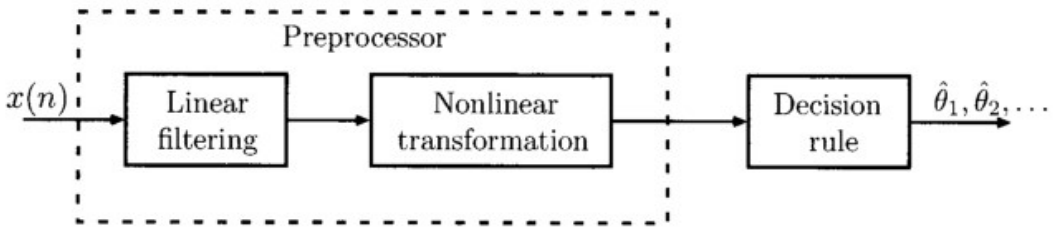



*Figura 12. Diagrama de bloques de una estructura comúnmente utilizada en la detección QRS.*

#### 1.3.2.3 Compresión de la señal

El objetivo de la compresión es representar la información con un menor número de bits, para lo que se cuenta con dos tipos de algoritmos: con pérdidas y sin pérdidas. Ambos tipos toman ventaja de la redundancia que existe en las señales, sin embargo, difieren en la razón de compresión que pueden alcanzar.

En el caso de la compresión sin pérdidas, la aproximación es encontrar patrones repetitivos o información redundante en la señal y tratar de codificarla en una manera más eficiente. De forma general, hay dos tipos de métodos para lograr este objetivo. Por un lado, están los métodos basados en diccionarios (v.g. algoritmo de Lempel-Ziv), que funcionan al tomar una cadena de símbolos y codificándola como un *token* usando un diccionario [27, p. 171]; por otro lado, están los métodos de codificación de entropía o métodos estadísticos (v.g. codificación Huffman), cuyo principio es la asignación de códigos de menor longitud a los símbolos de mayor ocurrencia (menor entropía) en la fuente de datos. Un ejemplo notable dentro de los métodos de codificación sin pérdidas es el algoritmo DEFLATE, el cual combina una variante de Lempel-Ziv llamada LZ77 y codificación Huffman [27, pp. 230-231].

Por su parte, en la compresión con pérdidas, figuran los métodos directos y los métodos basados en transformadas. Los primeros operan sobre las muestras de la señal original, seleccionando un numero de muestras significativas y descartando las redundantes. AZTEC y SAPA son ejemplos muy utilizados. Los últimos métodos, se basan en la transformación de la señal original, donde se descartan los coeficientes cercanos a cero, y solo se conservan los de menor peso, por ejemplo, el formato de compresión de imágenes JPEG que se basa en la transformada discreta del coseno (DCT).

Teniendo en cuenta la necesidad creciente de almacenar y transmitir señales ECG, resulta imperativo implementar métodos de compresión, pues no se puede subestimar la cantidad de datos producidos por un ECG, lo cual está directamente

relacionado con la tasa de muestreo y resolución del Convertidor Analógico-Digital con que se adquiere a señal y el número de derivaciones empleadas.

## 1.4 Aspectos tecnológicos

### 1.4.1 Aspectos computacionales

#### 1.4.1.1 Computación en la Nube

La Computación en la Nube (*Cloud Computing*, por sus siglas en inglés) se define como un conjunto de recursos computacionales, tales como almacenamiento y potencia de cómputo, puestos a disposición por un proveedor de servicios a través de internet para su consumo bajo demanda.

La Computación en la nube se suele categorizar en 3 modelos de servicio [28]:

- Software como servicio (SaaS), el cual consiste en un producto terminado, totalmente administrado por el proveedor de servicio.
- Plataforma como servicio (Paas), donde el proveedor de servicio ofrece la administración de una infraestructura sobre la cual el usuario puede montar sus propias aplicaciones.
- Infraestructura como Servicio (IaaS), en el que el proveedor pone a disposición los recursos básicos de cómputo, brindando mayor flexibilidad y control sobre estos.

#### 1.4.1.2 Computación en el Borde

Es un paradigma de computación que se enfoca en llevar el procesamiento y análisis de datos lo más cerca posible de la fuente de esos datos, en lugar de realizarlo en centros de datos remotos o en la nube. *Edge computing*, acerca la capacidad de cómputo al "borde" de la red, más cerca de donde se generan y utilizan los datos. Este paradigma juega un rol clave en aplicaciones de baja latencia, privacidad de datos y tecnologías emergentes como Internet de las Cosas (IoT) [29].

### 1.4.2 Redes de comunicación

El objetivo de una red de comunicación es mover una cierto volumen de datos desde un punto A hasta un punto B. A la rapidez con que los datos se transfieren en un sistema de comunicación se le conoce como velocidad de transferencia de datos, la cual se mide en bits por segundo y sus múltiplos. Por otro lado, de la distancia que abarca una red de comunicación depende si se clasifica en una de las siguientes categorías:

- Red de Área Corporal (BAN): pocos centímetros
- Red de Área Personal (PAN): algunos metros
- Red de Área Local (LAN): decenas de metros
- Red de Área Metropolitana (MAN): algunos kilómetros
- Red de Área Extensa (WAN): decenas de kilómetros

A modo de ejemplo, algunas tecnologías inalámbricas se muestran en la Figura 13, en función de las tasas de transferencia y su área de cobertura.

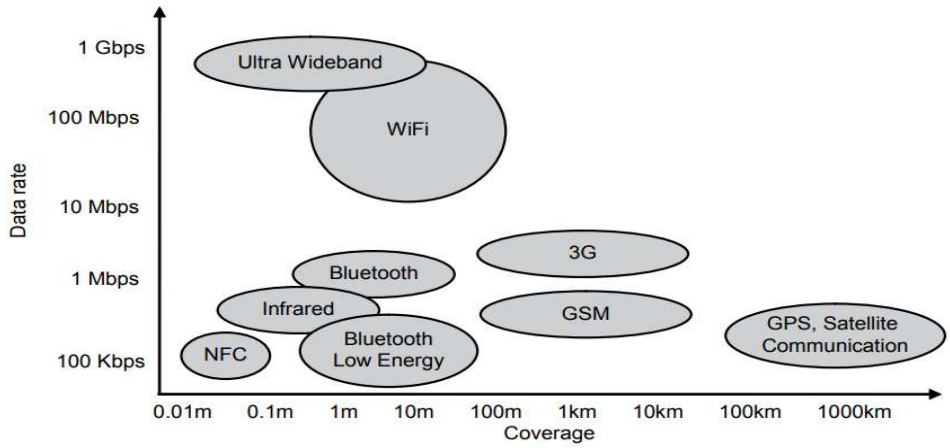



*Figura 13. Tasas de transferencia y cobertura típicas de varias tecnologías inalámbricas. Adaptado de N. Gupta [30, p. 3]*

#### 1.4.2.1 Tecnologías Inalámbricas

A continuación, se describen algunos protocolos de comunicación de interés en Telemedicina, haciendo énfasis en las redes inalámbricas, ya que suelen ser ideales para estas aplicaciones [30, p. 10].

**Bluetooth**

Es un protocolo de Red de Área Personal, se caracteriza por su corto rango de alcance bajo consumo, bajo costo y pequeño factor de forma. Fue creado como un reemplazo para cables seriales y actualmente es ideal para conectar dispositivos móviles [30, p. 18]. Opera en la banda de frecuencia industrial, científica y médica (ISM) 2,4 GHz, es decir, utiliza una banda no licenciada. Una de sus principales características es el uso de técnicas de espectro ensanchado, particularmente el mecanismo de Espectro Ensanchado por Salto de Frecuencia (FHSS) para combatir interferencia [30, p. 23] y su capacidad para formar redes *ad-hoc*.

Las dos variantes de la tecnología Bluetooth son:

- Bluetooth *basic rate/enhanced data rate* (BR/EDR) o Bluetooth clásico
- Bluetooth *low energy* (LE) o Bluetooth *Smart*, introducida en la versión 4.0 del protocolo.

Bluetooth BR/EDR está diseñado principalmente para operaciones de bajo consumo y alto rendimiento de datos, tales como audífonos inalámbricos. Por su parte Bluetooth LE está optimizado para admitir casos de uso que tienen un *duty cycle* relativamente bajo. Para disminuir el consumo de energía su radio es encendido durante un período de tiempo muy corto.

La Tabla IV se resume y compara diferentes características de Bluetooth BR/EDR y Bluetooth LE.

**Redes Celulares**

Son sistemas de comunicación inalámbricos que dividen grandes área geográficas en pequeñas secciones llamadas células, con el propósito de optimizar el uso de un limitado número de frecuencias. Cada dispositivo móvil utiliza 2 frecuencias, una para transmitir y otra para recibir. Estas redes se

encuentran casi en todas partes, lo que significa que son un punto de acceso a internet casi garantizado en cualquier lugar [31, p. 223].

*Tabla IV – Comparación entre Bluetooth BR/EDR y Bluetooth LE.*

| **Característica** | **Bluetooth BR/EDR** | **Bluetooth LE** |
|---|---|---|
| Banda de frecuencia | 2,400 MHz – 2,483.5 MHz (79 canales) | 2,400 MHz – 2,483.5 MHz (40 canales) |
| Ancho de banda | 1 MHz | 2 MHz |
| Esquema de modulación | • Desplazamiento de frecuencia gaussiana (GFSK)<br>• Desplazamiento de fase en cuadratura diferencial (DQPSK)<br>• Desplazamiento de fase diferencial (DPSK) | Desplazamiento de frecuencia gaussiana (GFSK) |
| Consumo energético | 1 W (valor de referencia) | 0.01 W – 0.5 W |
| Tasa de transferencia | 3 Mb/s (máxima) | 2 Mb/s (máxima) |

Un sistema de comunicación celular consta de los siguientes cuatro componentes principales:

- Una red telefónica pública conmutada (PSTN)
- Una oficina de conmutación de telefonía móvil (MTSO)
- Sitios celulares con sistemas de antena
- Unidades de abonado móvil (MSU)

Los avances en tecnología celular se han agrupado en generaciones (1G, 2G, 3G, etc.). Algunos de las tecnologías celulares más utilizadas son GSM (2G), GPRS (2.5G), UMTS (3G), HSDPA (3.5 G) y LTE (4G) [32, pp. 25-26].

La Tabla V hace una comparativa entre las diferentes generaciones.

A medida que han pasado las generaciones de redes de telefonía móvil, se ha venido mejorando el ancho de banda y los tiempos de latencia de los que los usuarios pueden disfrutar, sin embargo, la cobertura se ha visto comprometida debido al uso de frecuencias cada vez mayores. No obstante, la coexistencia de diferentes generaciones en las áreas de cobertura asegura que cualquier usuario de esta red puede acceder a ella con un ancho de banda suficiente para muchas aplicaciones en telemedicina.

*Tabla V – Comparativa de las generaciones de redes celulares. Adaptado de C. Beard et al [38, p. 471].*

| Technology | 1G | 2G | 2.5G | 3G | 4G |
|---|---|---|---|---|---|
| Design began | 1970 | 1980 | 1985 | 1990 | 2000 |
| Implementation | 1984 | 1991 | 1999 | 2002 | 2012 |
| Services | Analog voice | Digital voice | Higher capacity packetized data | Higher capacity, broadband | Completely IP based |
| Data rate | 1.9. kbps | 14.4 kbps | 384 kbps | 2 Mbps | 200 Mbps |
| Multiplexing | FDMA | TDMA, CDMA | TDMA, CDMA | CDMA | OFDMA, SC-FDMA |
| Core network | PSTN | PSTN | PSTN, packet network | Packet network | IP backbone |

### 1.4.3 Consideraciones de Seguridad

Se debe tener en cuenta que la información médica es un tipo de información sensible, pues su pérdida, corrupción o robo puede vulnerar gravemente la privacidad y poner en riesgo la salud de un paciente. En ese sentido I. Tsai [33, p. 39] recomienda que, como primera medida, se implemente encriptación en el envío de datos sensibles, sin embargo, también es recomendable emplear otros métodos asociados con la protección de accesos no autorizados en aras de proteger la integridad el Registro Electrónico del Paciente (EPR).

# 2 CAPÍTULO II: DISEÑO E IMPLEMENTACIÓN

En este capítulo son presentados los aspectos técnicos relacionados con el diseño e implementación del sistema desarrollado, incluyendo los requerimientos del prototipo acorde de acuerdo con los objetivos específicos de este trabajo monográfico, la selección de componentes y de las herramientas de desarrollo, así como las decisiones de integración de las partes constitutivas del sistema.

## 2.1 Etapa de Análisis

En esta etapa se establecen los requerimientos con los que debe cumplir el sistema a fin de dar una solución al problema planteado, los cuales sirven de guía en la etapa de diseño.

### 2.1.1 Requerimientos del sistema

Para demostrar la viabilidad de capturar y transmitir un electrocardiograma para que este pueda ser evaluado por un médico fue necesario realizar un revisión de literatura para conocer el estado del arte de la electrocardiografía digital y las aproximaciones que otros investigadores han empleado para aplicaciones de telemedicina. Del mismo modo, se realizó un análisis de las capacidades técnicas que actualmente existen en Nicaragua en materia de Telecomunicaciones, teniendo en cuenta las limitaciones y oportunidades presentes en zonas rurales del país.

En términos generales el sistema propuesto cuenta con 3 componentes:

- Un prototipo electrónico para la adquisición de señales electrocardiográficas (comúnmente llamado DAQ), con capacidad de transmisión inalámbrica.
- Una aplicación móvil para la visualización, procesamiento y envío de los exámenes a través de internet, la cual también sea capaz de recibir los diagnósticos.

- Un servicio en la nube para el almacenamiento y el redireccionamiento a su destino de los exámenes, así como para el manejo de los diagnósticos por parte del médico diagnosta.

El diagrama funcional de la solución propuesta se muestra en la Figura 14.

Con base en este esquema general es posible enumerar la siguiente lista de requerimientos del sistema:

- Prototipo electrónico:
    - Para este primer prototipo se ha decidido capturar una señal electrocardiográfica de una derivación, específicamente una de las derivaciones bipolares conformadas por el triángulo de Einthoven, con un rango de frecuencia de 0.5 Hz hasta 40 Hz, una Razón de Rechazo de Modo Común de al menos 80 dB.
    - Emplear una tasa de muestreo de entre 250 sps y 500 sps, que es el rango que usualmente se utiliza para aplicaciones electrocardiográficas. Si bien se podría emplear una tasa de muestreo menor conservando el contenido frecuencial de la señal, realizar sobre-muestreo (*oversampling*) ayudará a conservar la forma de onda en el dominio del tiempo, para su posterior análisis por un médico diagnosta.

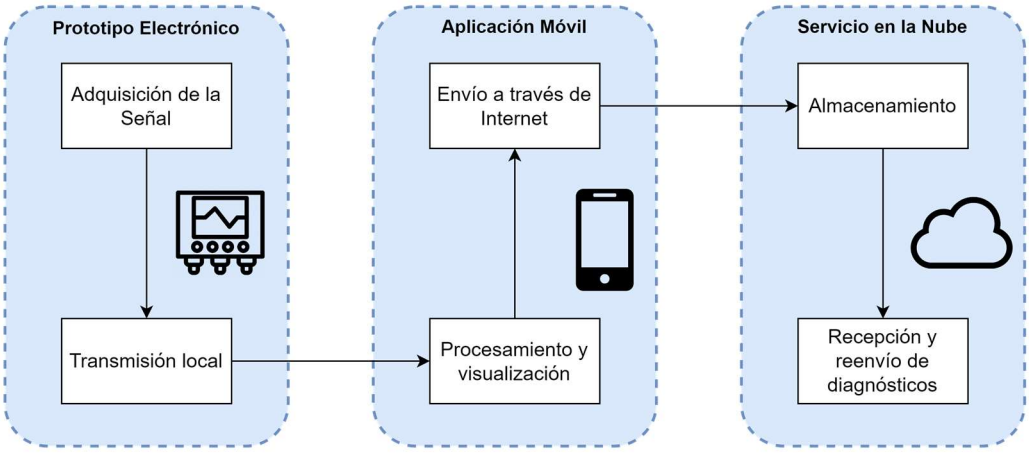



*Figura 14. Diagrama funcional del sistema propuesto.*

- Utilizar un ADC con al menos 12 bits de resolución, con capacidad de manejar una tasa de muestreo dentro del rango propuesto en el punto anterior. De igual manera el prototipo debe ser capaz de operar con esta resolución al momento de realizar el procesamiento digital de la señal.
- Dado que el prototipo debe tener autonomía de la red eléctrica, se debe emplear un protocolo de comunicación de bajo consumo.
- Los electrocardiogramas generados deben tener una duración de 10 segundos que es aproximadamente el promedio de duración de dicho examen en un entorno clínico.
- Para el procesamiento digital de señales se debe utilizar únicamente filtros FIR dado que, entre otras ventajas, presentan una respuesta de fase lineal, es decir, que su retraso de fase en constante y por lo tanto no introducen distorsiones en la forma de onda de la señal en el dominio del tiempo, a diferencia de los filtros IIR que tienen respuesta de fase no lineal.

- Aplicación móvil:
  - La aplicación debe operar en sistema operativo Android y demandar pocos recursos, de tal forma que sea capaz de correr en equipos de bajo costo.
  - Debe ser capaz de utilizar cualquiera de las interfaces inalámbricas del dispositivo en que es ejecutada para adaptarse a diferentes escenarios de conectividad disponible, a fin de enviar los exámenes de forma remota.
  - Para enriquecer la información enviada debe capturar datos adicionales como signos vitales, datos del paciente y metadatos del examen.
  - En la aplicación móvil se debe visualizar la señal electrocardiográfica tanto en el momento que está siendo capturada como de los exámenes que fueron grabados anteriormente.

    - Debe procesar la señal para detección de características como la frecuencia cardiaca y comprimirla para su almacenamiento tanto local como en la nube.
    - Debe ser capaz de recibir los diagnósticos de regreso y asociarlos con el examen original.
- Servicio en la nube:
    - El servicio en la nube que actuará como *backend* de la solución debe brindar la capacidad de almacenamiento de los exámenes en una base de datos, donde también estará contenido la información de destino hacia donde se deben redirigir los exámenes una vez recibidos del lugar de origen.
    - Debe contar con capacidad de autenticación ante la escritura en la base de datos, para asegurar los datos en tránsito y en reposo.
    - Debe implementar un mecanismo de notificaciones para informar al médico diagnosta cuando un nuevo examen sea enviado y también debe informar al lugar de origen cuando un diagnóstico sea emitido, con un enlace web para el formulario de envío de diagnósticos.
    - El servicio alojado en la nube debe exponer una interfaz web para el diagnóstico de los exámenes de tal forma que al recibir el diagnóstico se asocie al examen correspondiente en la base de datos.
    - Para garantizar que la solución sea mantenible se debe elegir una solución de alojamiento en la nube que brinde facilidades de implementación a fin de disminuir al mínimo la necesidad de intervenir manualmente para realizar tareas como la actualización del software y sistema operativo subyacentes.

Como resultado de esta lista de requerimientos es posible establecer un esquema de funcionamiento de la solución a través del uso de un diagrama de secuencia, donde se abarca de forma general las interacciones de los usuarios con el sistema. Esto se detalla en la Figura 15.

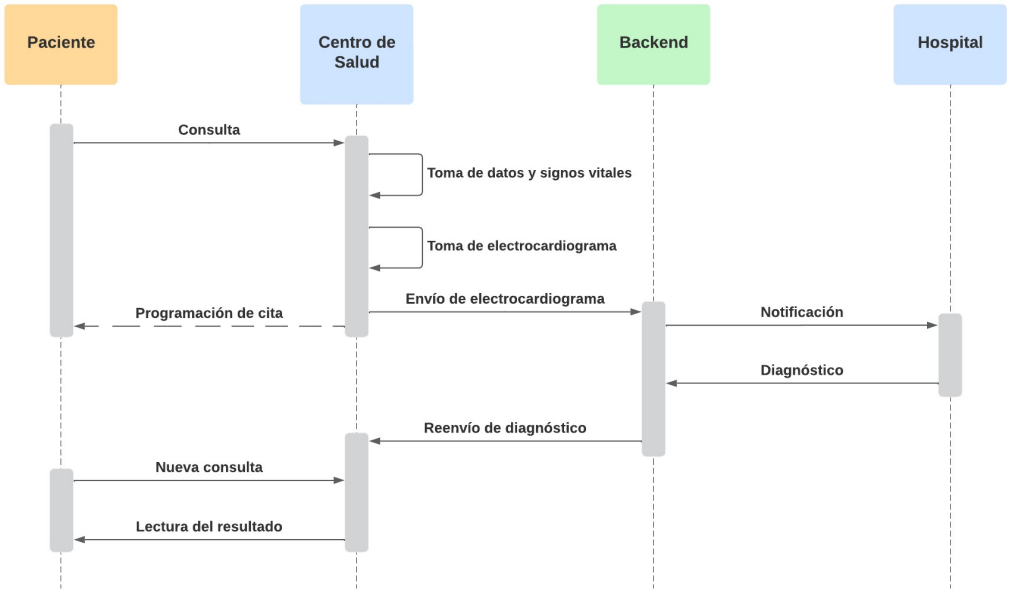



*Figura 15. Diagrama de secuencias de la solución propuesta.*

### 2.1.2 Análisis funcional

En la Figura 14 se propuso un diagrama general de la solución con las funciones esenciales de cada componente en el sistema. A continuación, se propone un diagrama de bloques donde se refleja a nivel interno los componentes de cada bloque funcional en la Figura 16.

En el caso del prototipo, algunos bloques como el de alimentación y periféricos indicadores de estado se omiten por simplicidad, sin embargo, los bloques principales se describen a continuación:

- La interfaz analógica, que no es más que un circuito de señal analógica cuyo propósito es realizar el acondicionamiento de la señal, es decir, pre-amplificarla, filtrarla al rango de frecuencias en que se encuentra las componentes espectrales de interés y amplificarla a los niveles de voltaje de operación del sistema, de tal manera que pueda ser digitalizada.
- El circuito de señal mixta se encarga de tomar la señal analógica acondicionada para digitalizarla mediante un convertidor analógico digital, para posteriormente transformarla aplicando técnicas de procesamiento

digital de señales y empaquetarla en un formato apropiado para su transmisión de forma inalámbrica.

- La interfaz inalámbrica se encarga de recibir señales de control por parte de la aplicación móvil y de transmitir las muestras digitalizadas y procesadas de la señal electrocardiográfica. Para este fin se ha seleccionado el protocolo de comunicación Bluetooth debido a su flexibilidad, alta tasa de transferencia y principalmente a su ubicuidad en los dispositivos móviles de consumo, por tanto, no es necesario considerar otros protocolos menos comunes como ZigBee.

En el caso de la aplicación móvil no es más que una pieza de software que se encargará de presentar una interfaz de usuario (UI) que le permita al personal médico interactuar con el prototipo electrónico a través de la interfaz bluetooth, guardar en el almacenamiento local los exámenes realizados y utilizar las interfaces de acceso a la red para la comunicación con el *backend*.

Por otro lado, para el *backend* se decidió implementarlo en una nube pública utilizando el modelo *Platform as a Service* (PaaS), de acuerdo con el

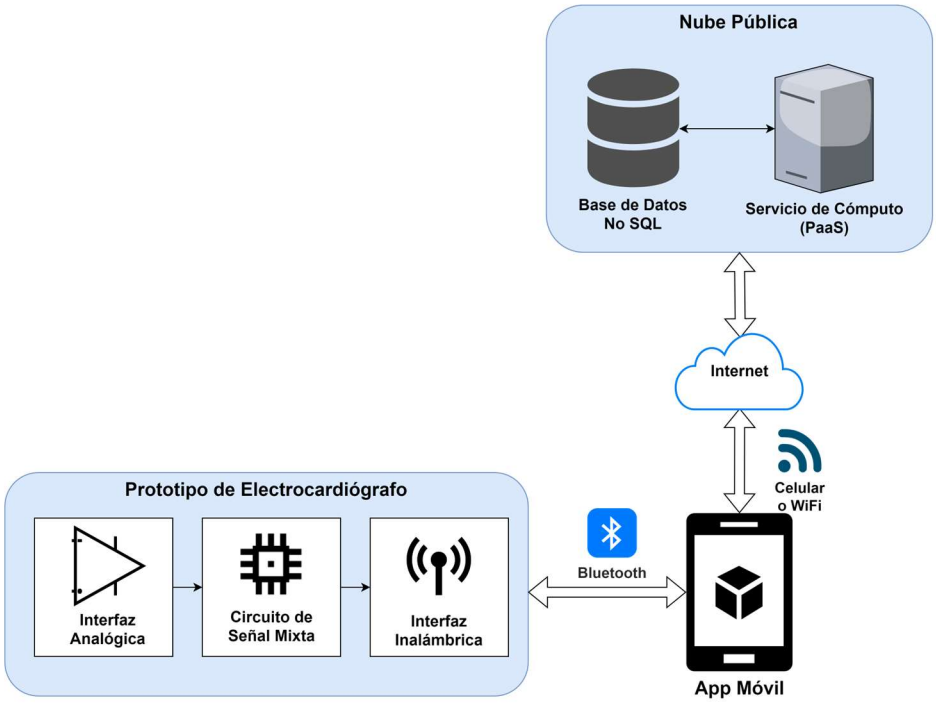



*Figura 16. Diagrama de bloques general del sistema.*

requerimiento de facilidad de mantenimiento de la solución, contrario al modelo *Infrastructure as a Service* (IaaS) que demanda un mayor esfuerzo de mantenimiento. Particularmente se propuso los siguientes componentes:

- Una instancia de Base de Datos escalable y administrada por el proveedor de servicios.
- Un servicio de cómputo sobre el que se ejecute el servicio de notificaciones y la interfaz web.

## 2.2 Etapa de Diseño

En este apartado se explica el proceso realizado para el diseño de cada uno de los bloques que conforma la solución, partiendo de los requerimientos establecidos en la etapa de análisis. De igual manera se aborda el proceso de la selección de componentes electrónicos y herramientas de desarrollo, así como el razonamiento detrás de las decisiones de diseño de la solución propuesta.

### 2.2.1 Diseño del Prototipo Electrónico

Esta sección se profundiza en el diseño del prototipo electrocardiográfico, desde la selección de componentes electrónicos, diseño de circuitos, y se describe la funcionalidad que ha de llevar a cabo su programación.

En primer lugar, se propone un diagrama de bloques con todos componentes que debe poseer el prototipo desde el punto de vista electrónico. La Figura 17 muestra los bloques descritos a continuación:

- Nuevamente se muestra la interfaz analógica, también conocida como *frontend* analógico, el cual puede ser implementado como un único circuito integrado, pues en las últimas décadas ha habido un considerable progreso en la tecnología de adquisición de señales fisiológicas.

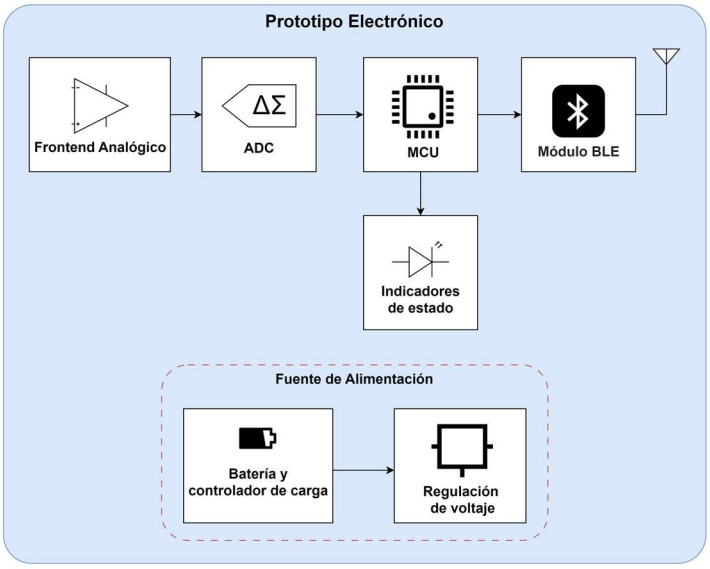



*Figura 17. Diagrama de bloques del prototipo electrónico.*

- Para la etapa de conversión analógico-digital se propone un convertidor de tipo Sigma-Delta, ya que estos se caracterizan por ofrecer alta resolución a cambio de sacrificar tasa de muestreo efectiva. Dado que la tasa de muestreo máxima establecida como requerimiento se encuentra dentro del rango usual para este tipo de convertidores, son la opción más adecuada. Asimismo, se optó por un protocolo de comunicación serial ya que de esta manera se reduce la necesidad de reservar pines de comunicación en el bloque de procesamiento.
- En el bloque de procesamiento se optó por la opción un microcontrolador con capacidad para realizar procesamiento digital de señales, así como periféricos de entrada salida y periféricos de comunicación serial. En el mercado existen varias opciones lo que simplifica la búsqueda de una opción con estas prestaciones.
- La interfaz inalámbrica, como ya se mencionó anteriormente, es un módulo de comunicación de Bluetooth, correspondiente al estándar IEEE 802.15.1, particularmente se ha seleccionado Bluetooth Low Energy (BLE) que corresponde a la versión del estándar 4.0 y superiores. Esta decisión se debe al menor consumo de energía comparado con el protocolo Bluetooth

Classic y tasa de transferencia equiparable, tal como se mostró en la Tabla IV.

- El bloque de alimentación consiste en una batería de iones de Litio (Li-Ion) con su correspondiente controlador de carga y regulador de tensión para alimentar el circuito. El voltaje de salida estará determinado por los componentes seleccionados en la próxima sección.
- Para el bloque de Indicadores, se propone una simple interfaz compuesta por diodos LED para indicar si el prototipo está encendido y para indicar el estado de la comunicación por Bluetooth.

#### 2.2.1.1 Selección del Frontend analógico y ADC

Para la selección del Frontend analógico hay que tomar en cuenta dos parámetros principales, a saber, la razón de rechazo de modo común (CMRR) y el ancho de banda (BW). De estos parámetros dependen las características específicas de los demás componentes a elegir. Del CMRR depende la robustez ante el ruido de la señal resultante y del ancho de banda.

Luego de realizar una búsqueda en la web de distribuidores de componentes, se encontró en el mercado 2 opciones que presentan buenas prestaciones y un costo bajo relativamente bajo para adquisición de señales electrocardiográficas, de los cuales existen tarjetas de prototipado disponibles comercialmente:

- El circuito integrado AD8232 del fabricante Analog Devices.
- El circuito integrado ADS1293 del fabricante Texas Instruments.

Antes de comparar ambas opciones es importante destacar que el IC ADS1293 cuenta con un Convertidor analógico-digital incorporado, por tanto, es necesario encontrar un convertidor analógico digital que sea adecuado para el integrado AD8232 a fin de poder comparar apropiadamente las opciones. En este sentido, existen 2 opciones de convertidores analógico-digitales de tipo Delta-Sigma que cumplen con tener una resolución mayor a 12 bits y una tasa de muestreo superior a las 300 sps, de los cuales existen módulos de prototipado:

- El circuito integrado MCP3424 del fabricante Texas Instruments.
- El circuito integrado ADS1115 del fabricante Texas Instruments.

De esta forma, se enumeran las 3 soluciones evaluadas y se realiza la comparación de sus características a continuación en la Tabla VI, donde los parámetros analógicos corresponden al IC AD8232 y al *frontend* del IC ADS1293, mientras que los parámetros digitales corresponden a los convertidores mencionados y a sus respectivas interfaces de comunicación serial.

*Tabla VI – Comparativa de opciones para Frontend Analógico y Conversor Analógico Digital.*

| **Parámetro** | **AD8232 + MCP3424** | **AD8232 + ADS1115** | **ADS1293** |
|---|---|---|---|
| CMRR | 80 dB | 80 dB | 100 dB |
| Máximo ancho de banda | 1000 Hz | 1000 Hz | 1280 Hz |
| Bloque de filtro variable | Sí | Sí | Sí |
| Ganancia | 100 | 100 | 3.5 |
| Derivaciones | 1 | 1 | 3 |
| Voltaje de operación | 2.0 V – 3.5 V | 2.0 V – 3.5 V | 2.7 V – 5.5 V |
| Resolución (bits) | 18 | 16 | 24 |
| Máxima tasa de muestreo | 240 sps | 860 sps | 2560 sps |
| Bus de comunicación | I2C | I2C | SPI |
| Costo total | $25 - $28 | $18 - $21 | $66 |

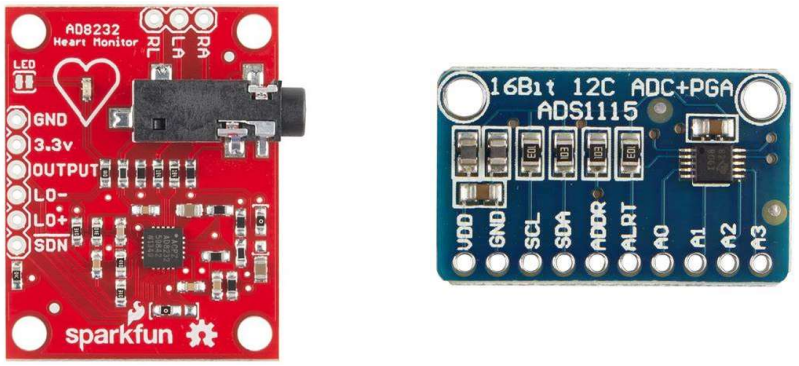
En cuanto a los parámetros de CMRR y ancho de banda máximo las tres opciones presentan números aceptables, de igual manera tienen características como bloques de filtro variable que pueden ser configurados para obtener el ancho de banda deseado por la solución y ganancias aceptables, destacando el ADS1293 por tener 3 derivaciones y mejor CMRR. Si bien es cierto que en términos de

parámetros este último resulta ser el mejor perfilado, el AD8232 igualmente cumple con los requerimientos mínimos.

Por otro lado, en el caso de la opción con el convertidor MCP3424, pese a tener una resolución bastante buena, tiene una tasa de muestreo máxima ligeramente menor a la requerida y de igual forma es una opción más costosa que ADS1115. En este punto también destaca el ADS1293 con sus 24 bits de resolución y alta tasa de muestreo, sin embargo, el ADS1115 cumple con los requerimientos mínimos que demanda la solución.

Por tanto, la decisión depende del costo de los componentes, cuyos precios se consultaron en varias plataformas en línea, específicamente se buscó el formato de módulo de prototipado para simplificar la construcción del prototipo. En ese sentido, la ventaja está del lado de la combinación AD8232 + ADS1115, con un costo tres veces menor en comparación con el ADS1293. Los módulos seleccionados se muestran en la Figura 18. Donde (a) es el módulo AD8232 y (b) es el módulo ADS1115.

Dentro de las características del módulo AD8232 seleccionado, está la circuitería de apoyo alrededor del integrado, lo que le permite al IC hacer uso de su bloque de filtro variable para realizar un filtrado de la señal y una amplificación a la salida. De esta manera la señal queda restringida un rango de 0.5 Hz a 40 Hz y tiene una




(a) (b)

*Figura 18. Módulos de Frontend analógico y Convertidor analógico-digital. (a) AD8232. (b) ADS1115.*

ganancia total de 1100 V/V. De igual forma, está configurado para utilizar 3 electrodos, donde dos de ellos perciben la señal cardíaca y el tercero actúa como un controlador de pierna derecha (RLD) cuyo propósito es cancelar ruido proveniente del exterior y actuar como electrodo de referencia. El esquemático del circuito implementado en este módulo se muestra en la Figura 19.

Asimismo, este módulo cuenta con un *jack* para la conexión con el cable de electrodos y una cabecera de pines para conectarse al resto del sistema. Los pines VCC y GND son los de alimentación, OUTPUT contiene la señal de salida y debe ser conectado a la entrada analógica ADC. Por su parte, los pines LO-, LO+, que indican si alguno de los electrodos está desconectado, y ~SDN, que sirve para apagar el módulo, no son utilizados en el prototipo.

Por su parte, el módulo ADC es un poco más sencillo en su construcción ya que además del integrado ADS1115 solo requiere de algunos resistores para el bus de comunicación y algunos capacitores de desacople de DC. Los pines VDD y GND son los pines de alimentación; SDA y SCL son los pines del BUS I2C, mientras que los pines de A0 a A3 son los 4 canales de entrada analógica. Adicionalmente este IC cuenta con un Amplificador de Ganancia Programable (PGA), lo que permite seleccionar los voltajes de referencia con que opera y de los cuales depende la equivalencia entre el voltaje medido en su entrada y el valor binario a su salida.

De esta manera quedan fijas los siguientes parámetros a tomar en cuenta en la siguiente sección:

- Voltaje de operación: 2.0 V – 3.5 V
- Resolución: 16 bits
- Bus de comunicación: I2C

#### 2.2.1.2 Selección del Microcontrolador

Para la selección del microcontrolador se tienen dos alternativas: por un lado, está la utilización de un procesador digital de señales cuya características es contar

con arquitectura de hardware especializada para realizar operaciones multiplicación y acumulación (MAC), y por el otro lado está la utilización de un microcontrolador de arquitectura convencional para realizar todo el procesamiento en software.

Si bien es cierto que los primeros son más eficientes computacional y energéticamente, también es cierto que suelen ser relativamente más costosos. Por su parte los microcontroladores convencionales han experimentado mejoras sostenidas a lo largo de las últimas décadas que les permiten acercarse en desempeño un Procesador Digital de Señales, por ejemplo, a través del uso de mayores frecuencias de reloj y de la implementación de unidades de co-procesamiento matemático como las Unidades de Punto Flotante (FPU).

En ese sentido, para este proyecto se ha optado por la última opción ya que también esta cuenta con la ventaja de ser más comercial y, por ende, se facilita la búsqueda de una tarjeta de desarrollo adecuada para prototipado. Adicionalmente, hay algunos fabricantes que han dotado a sus microcontroladores con conectividad inalámbrica, específicamente con tecnología Bluetooth, lo que resulta en la posibilidad de unificar en un único módulo, también llamado *System-on-Chip* (SoC), tanto la Unidad de Procesamiento Digital como la Conectividad inalámbrica.

Con este fin se han evaluado los siguientes SoC, debido a su amplia disponibilidad, soporte del protocolo Bluetooth Low Energy y diversidad de formatos:

- La tarjeta de desarrollo ESP32 del fabricante Espressif.
- El tarjeta de desarrollo nRF52 del fabricante Nórdico Semiconductor.

Tras revisar las hojas de datos de ambas opciones se procede a realizar una comparación de sus características más importantes en la Tabla VII.

*Tabla VII – Comparativa de opciones de Microcontroladores con comunicación Bluetooth.*

| **Parámetro** | **nRF52** | **ESP32** |
|---|---|---|
| Modelo de SoC | nRF52832 | ESP32 WROVER B |
| Arquitectura de Nucleo | ARM Cortex-M4 | Dual-core Tensilica LX6 |
| Frecuencia de Reloj | 64 MHz | 240 MHz |
| Unidad de Punto Flotante | SÍ | Sí |
| Conectividad Inalámbrica | Bluetooth 5.4, Zigbee, Thread, ANT+ | Bluetooth 4.2, Wi-Fi 802.11 b/g/n |
| Memoria Flash | 512 kB | 4 MB |
| Memoria RAM | 64 kB | 520 kB + 8 MB |
| Pines GPIO | 32 | 34 |
| Interfaces Seriales | 2 | 3 |
| Interfaces SPI | 2 | 4 |
| Voltaje de Operación | 1.7 V – 3.6 V | 2.2 V – 3.6 V |
| *Framework* de desarrollo | Nordic SDK, Arduino | SPD-IDF, Arduino, PlatformIO |
| Costo aprox. / unidad | $20 | $15 |

En cuanto a las características computacionales, si bien ambos Microcontroladores cuentan con una Unidad de Punto Flotante, lo cual les permitirá realizar cálculos matemáticos con facilidad, claramente la ventaja la tiene el ESP32 con su procesador de 2 núcleos a 240 MHz y 520 kB de RAM. De igual manera el ESP32 cuenta con mayor capacidad de almacenamiento con 4 MB de memoria Flash.

En el apartado de comunicación inalámbrica, tanto el nRF52 y el ESP32 operan en la banda ISM de 2.4 GHz y soportan Bluetooth Low Energy, sin embargo, en este particular la diferencia entre ellos estriba en que nRF52 implementa la versión 5.4 del protocolo Bluetooth mientras que el ESP32 implementa la versión 4.2. De forma resumida es posible decir que BLE 5.4 ofrece mejoras significativas en

términos de rendimiento de transferencia de datos, seguridad, y eficiencia energética en comparación con BLE 4.2. Sin embargo, para decidir entre uno u otro el parámetro más importante es la tasa de transferencia, por tanto, es necesario calcular la tasa de transferencia necesaria para lograr enviar 500 muestras por segundo, que es la máxima tasa de muestreo propuesta para este prototipo. Usando la Ecuación (4) se calcula esta tasa de transferencia necesaria para el sistema partiendo de la tasa de muestreo y la resolución del ADC.

Tasa de transferencia = Número de bits * Frecuencia (4)

Tasa de transferencia = 16 bits * 500 Hz

Tasa de transferencia = 8000 bits/s

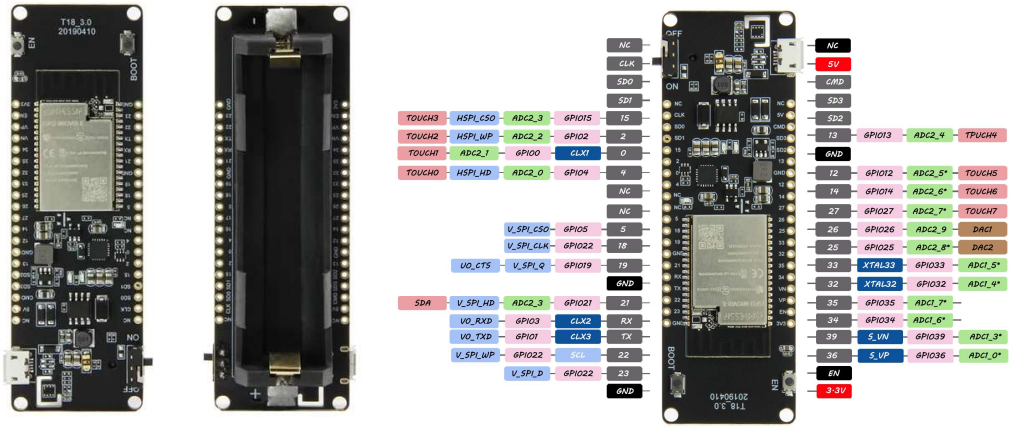
Siendo BLE 4.2 capaz de transferir un máximo de 1 Mbps, es más que suficiente para suplir el requerimiento de 8 kbps que se calculó. De igual manera, si el sistema se hubiera diseñado para implementar un electrocardiógrafo de 12 derivaciones, BLE 4.2 seguiría siendo suficiente pues en ese caso la tasa de transferencia de 96 kbps seguiría estando por debajo de la máxima tasa de transferencia permitida por el protocolo.

En consecuencia, la opción de utilizar el microcontrolador ESP32 es la que mejor se perfila, especialmente considerando que el costo aproximado de este es en general menor que el del nRF52. Si bien es cierto, que el nRF52 tiene mejores características de consumo energético, por tener menor frecuencia de reloj e implementar una versión protocolo BLE más eficiente, el ESP32 tiene capacidades de ahorro de energía al hacer uso de *sleep modes*. Por tanto, el ESP32 es la mejor opción.

Una vez elegido el microcontrolador, es necesario elegir una tarjeta de desarrollo adecuada para este prototipo. En el mercado existen múltiples opciones que tiene diversos factores de forma. En ese sentido, es necesario encontrar una que presente ventajas para este prototipo que debe ser alimentado por batería. Es por ello por lo que se ha elegido la tarjeta de desarrollo T-Energy del fabricante LilyGO,

cuya principal característica es la de poseer un soporte para batería de Litio con factor de forma 18650, además de contar con circuitería para control de carga y regulación de voltaje, siendo su voltaje de operación de 3.3 V. Esto la hace compatible con los demás componentes ya seleccionados, siendo también posible alimentar todo el sistema a partir del regulador del voltaje de la tarjeta. En la Figura 19 se muestra la fotografía de la tarjeta T-Energy (a) y su respectivo pinout (b), en el cual destaca los pines del bus I2C que serán utilizados en la interconexión con el ADC.

Por último, la batería elegida para alimentar el circuito es una celda de Litio de 3000 mAh a 3.7 V nominales, modelo IMR 18650, que es el factor de forma correspondiente al soporte de la tarjeta de desarrollo. Gracias al circuito controlador de carga




(a) (b)

*Figura 19. Tarjeta de Desarrollo ESP32 T-Energy de LilyGO.*

### 2.2.1.3 Circuito esquemático

Habiendo elegido todos los componentes del prototipo electrónico, se propone el siguiente diagrama esquemático mostrado en la Figura 20, en cual se han omitido los pines no utilizados de cada componentes por simplicidad.

La conexión entre el AD8232 y el módulo ADS1115 se realiza a través del canal 1 del ADC, mientras que el canal 0 se utiliza para tener referencia de la línea de

alimentación, lo cual es útil al momento de procesar la componente DC de la línea de alimentación. Asimismo, el módulo ADS1115 está conectado a la tarjeta de desarrollo ESP32 a través de los terminales SDA y SCL del bus de comunicación I2C. Los resistores *pull-up* de este bus se han omitido, sin embargo, estos están presentes como parte del módulo ADS1115.

La tarjeta ESP32, por su parte, engloba los bloques de procesamiento, comunicación Bluetooth y alimentación que fueron propuestos en la Figura 17. Ésta provee de voltaje a la línea de alimentación del circuito a través de su regulador de voltaje incorporado. Para indicar el estado encendido o apagado del circuito se añadió el LED2, mientras que para indicar el estado de la comunicación inalámbrica se incorporó el LED1, ambos con sus correspondientes resistores.

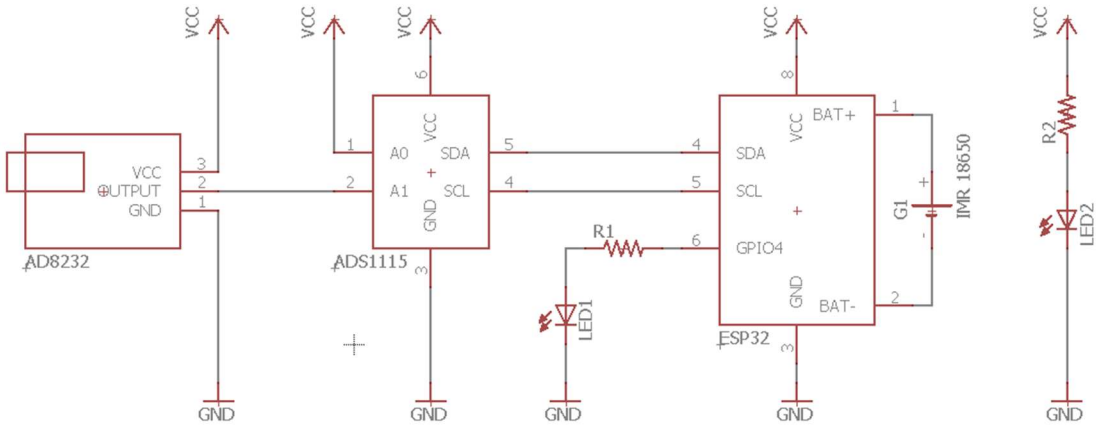



*Figura 20. Diagrama esquemático del circuito propuesto.*

### 2.2.1.4 Diseño del firmware

Para la realización de un programa que controle la lógica de operación del prototipo electrónico fue necesario considerar que el prototipo actúa como un Periférico dentro del protocolo BLE, donde un dispositivo Android actúa como Central, que envía comandos para controlar el estado del Periférico y recibe la señal electrocardiográfica. En ese sentido, se propone una máquina de estado finito que modela la lógica de la comunicación, mostrada en la Figura 21.

En esta se definen 3 estados:

- IDLE, cuando el prototipo está encendido y a la espera de una conexión.
- CONN, cuando el prototipo está conectado a un dispositivo Central, a la espera de comandos.
- SEND, cuando el prototipo está enviando muestras de la señal electrocardiográfica.

En consecuencia, el funcionamiento específico del prototipo se diseñó alrededor de estos estados y sus respectivas condiciones de transición, lo cual permite separar claramente las funciones que necesitan ser ejecutadas en cada estado. En la Figura 22 se muestra el flujograma con la lógica de programación empleada.

El programa empieza con la configuración de las comunicaciones BLE, Serial e I2C, luego procede a configurar el ADC con los parámetros adecuados, y termina por configurar el funcionamiento de un *timer* de los que están integrados en el ESP32, que sirve para temporizar una rutina de interrupción (ISR) que es la encargada desencadenar la generación de una nueva muestra de la señal. De igual manera se inicializa el estado de la máquina de estados implementada y se define una subrutina que controla la salida del LED indicador de estado, cuyo patrón cambia de acuerdo con la máquina de estado finito.

Tal como se describe en la Figura 21, en el estado IDLE el prototipo emite paquetes de *advertising* a la espera de ser escaneado por un dispositivo Central e iniciada la conexión, en cuyo caso se realiza la transición al siguiente estado. De lo contrario, permanece en el mismo. Mientras el prototipo se encuentra en este estado, el LED permanece apagado.

En el estado CONN ya se ha recibido una conexión y se está a la espera de recibir nuevos comandos, que en este estado son 3 posibles, representados como un byte:  00h para iniciar la transmisión de la señal electrocardiográfica, 10h para desactivar el filtro digital y 11h para activarlo. En caso de recibir el comando de inicio de envío, se realiza la transición al siguiente estado, se actualiza la

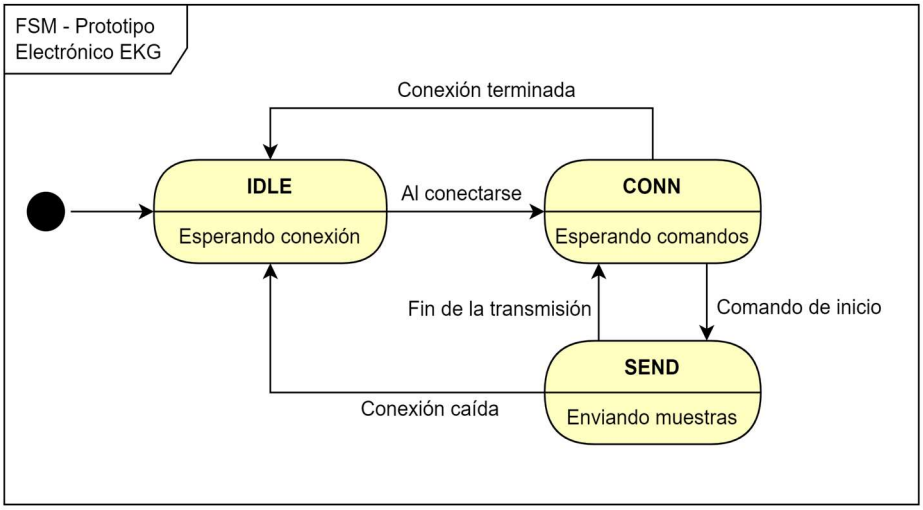



*Figura 21. Diagrama de máquina de estado finito implementada en el prototipo.*

información del estado del dispositivo como el estado de la batería que será accesible a la aplicación para lectura, y se habilita la interrupción del *timer* a través de una bandera booleana. El indicador LED parpadea en este estado para hacer saber al usuario que el electrocardiógrafo está conectado y listo para recibir comandos.

Por otro lado, en el estado SEND, se está a la espera de la habilitación de la bandera request_read por parte de la ISR, que señaliza la toma de una nueva muestra. Inmediatamente se realiza la lectura del ADC, de igual forma se substrae la componente de DC proveniente del *frontend* analógico, la cual se calcula a partir de la lectura de la línea de alimentación. A continuación, se hace pasar la muestra recién adquirida a través del filtro FIR, cuyo flujograma se muestra en la Figura 23. A las muestras filtradas se las acumula en un array con el fin de enviarlas en un solo paquete BLE, lo cual es preferible a enviar cada muestra por separado debido a la forma de operación del protocolo, cuyo radio solo es encendido a intervalos de tiempo definidos, no a demanda (véase la Sección 1.4.2.1). Cuando el array está lleno, lo que conforma una trama de muestras, este es enviado hacia el dispositivo Central y reinicia el contador de muestras. Por último, se comprueba si la conexión BLE sigue activa, de lo contrario se hace la transición a estado IDLE, y también se comprueba si se ha recibido el comando de finalización de la

comunicación FFh, en cuyo caso se hace la transición a estado CONN. El indicador LED permanece encendido mientras esta en este estado.

El flujograma del filtro FIR describe de forma esencial el funcionamiento del bloque de proceso que fue indicado en la Figura 22, usando notación de lenguaje C. Es este se verifica si el filtro debe estar funcionando o no a través de la bandera filter_active. En caso de que sí lo esté, se reinicia la variable acumulador y contador, y se establece los punteros de entrada y del arreglo de coeficientes del filtro. En seguida se entra en un bucle que realiza el producto punto entre el buffer de entrada y el array de coeficientes. Una vez terminado se asigna el valor del acumulador a variable de salida del filtro y se desplaza una posición en el buffer de entrada para dar paso a una nueva muestra en la próxima llamada a esta subrutina (véase la Figura 23). Por el otro lado, si se ha desactivado el filtro, las muestras de entra son retornadas a la salida del proceso intactas.

Para el desarrollo del firmware del prototipo, se ha elegido PlatformIO IDE como plataforma de desarrollo, dado su amplio gestor de librerías, y su capacidad de integrarse con otras herramientas como git. Se optó por el uso del lenguaje C++ y el uso de librerías Arduino compatibles.

#### 2.2.1.5 Adquisición y procesamiento de la señal

Para lograr poder caracterizar la señal de interés se procedió a realizar una implementación mínima con el circuito de adquisición de señal en tabla de nodos, lo cual permitió obtener una primera señal cardiaca y realizar un análisis en el dominio del tiempo y en el dominio de la frecuencia. De esta manera, se realizó la señal usando como herramienta Jupyter Notebooks que permite, entre otras aplicaciones, hacer análisis de señales a través de las librerías numpy y scipy, las cuales son un excelente reemplazo a herramientas como Matlab.

Para obtener una señal electrocardiográfica inicial se configuró un circuito de adquisición mínimo, con los cables terminales y electrodos mostrados el Anexo A. De igual forma se configuró una tasa de muestreo de 480 sps y se removió la

componente de DC proveniente del frontend analógico. Acerca de esto se profundiza en la sección 2.3.1.1.

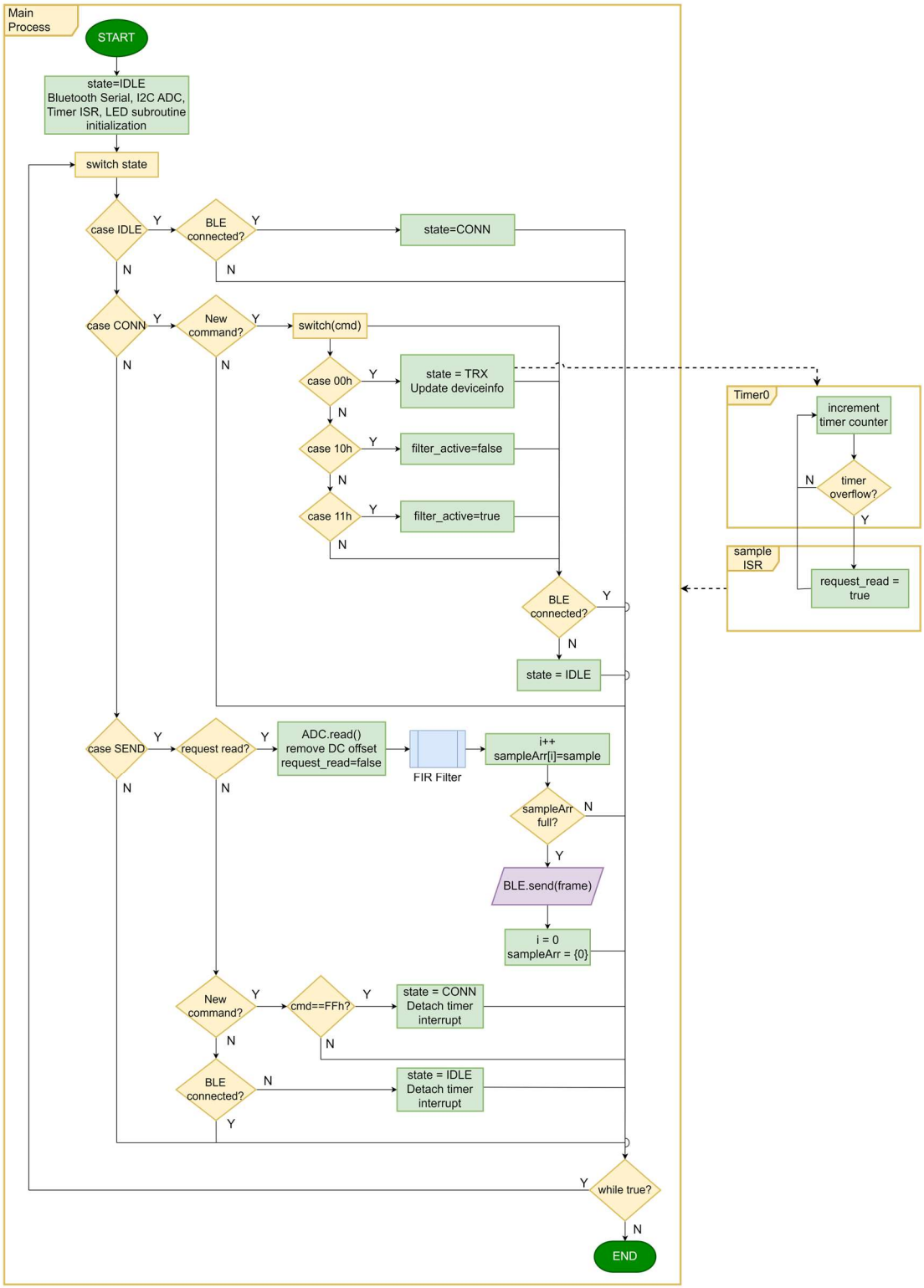



*Figura 22. Flujograma de la programación del prototipo.*

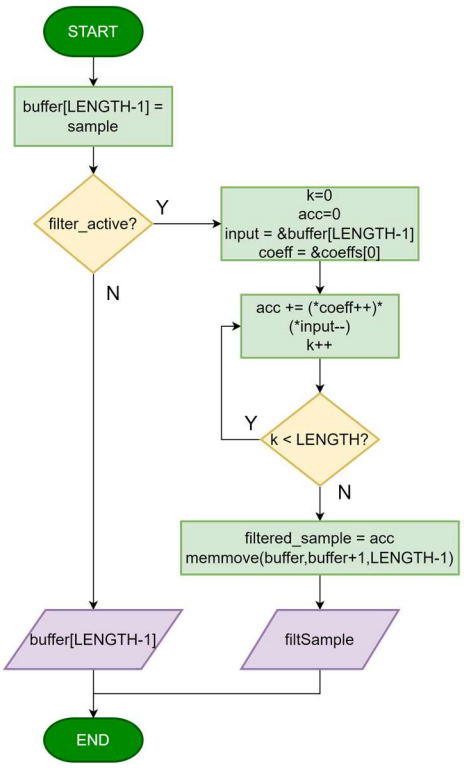



*Figura 23. Flujograma del proceso de filtrado FIR. Fuente: Autor.*

De esta manera, el resultado de las pruebas iniciales arrojó una señal donde se puede apreciar la forma de onda de la señal esperada, con sus picos R claramente destacados en amplitud. De igual modo, se puede apreciar ruido eléctrico que contamina la señal de interés. Éste presenta una forma sinusoidal armónica, en contraste con otras fuentes de ruido que son de naturaleza gaussiana, por tanto, es por tanto que se asumió que la principal fuente de ruido inducido por la red eléctrica comercial, tal como se puede apreciar en la Figura 24.

Asimismo, se realizó la transformación de la señal al dominio de la frecuencia haciendo uso de la librería scipy, a fin de encontrar cuales son las características del espectro de la señal y del ruido. En la Figura 25 es posible apreciar varios picos de frecuencias que están entre los 0 y 40 Hz, correspondientes a la señal de interés. Estos picos son consecuencia de su naturaleza periódica de la señal electrocardiográfica. Luego, hay dos picos de frecuencia en los 60 Hz y 120 Hz con una amplitud considerable, lo que coincidió con la suposición acerca de la naturaleza del ruido contaminando la señal.

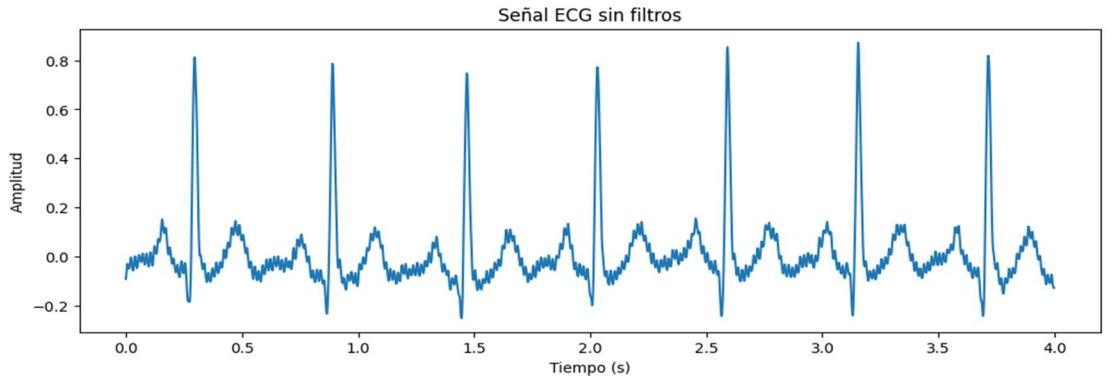



*Figura 24. Señal electrocardiográfica sin procesar. Fuente: Autor.*

Por tanto, se procedió con el diseño del filtro digital que se utilizaría el prototipo para procesar la señal. En este caso, dado que la señal se encuentra restringida a los primeros 40 Hz del espectro, y la principal componente de ruido se encuentra en 60 Hz se optó por un filtro con las características de la Tabla VIII. Estos parámetros están esquematizados gráficamente en la Figura 26. Nótese que para el diseño de filtros digitales las magnitudes frecuencias deben ser normalizadas.

Para el cálculo de los coeficientes de filtros digitales se tienen algunos métodos tales como, método de Remez o algoritmo de Parks-McClellan, método de muestreo de frecuencias y método de ventanas. Para este trabajo monográfico seleccionó el método de Remez debido a que presenta las siguientes ventajas.

- Precisión Controlada: El método de Remez permite un control preciso sobre la respuesta de frecuencia del filtro. Puede especificar las bandas de paso y las bandas de atenuación, así como los valores deseados en esas

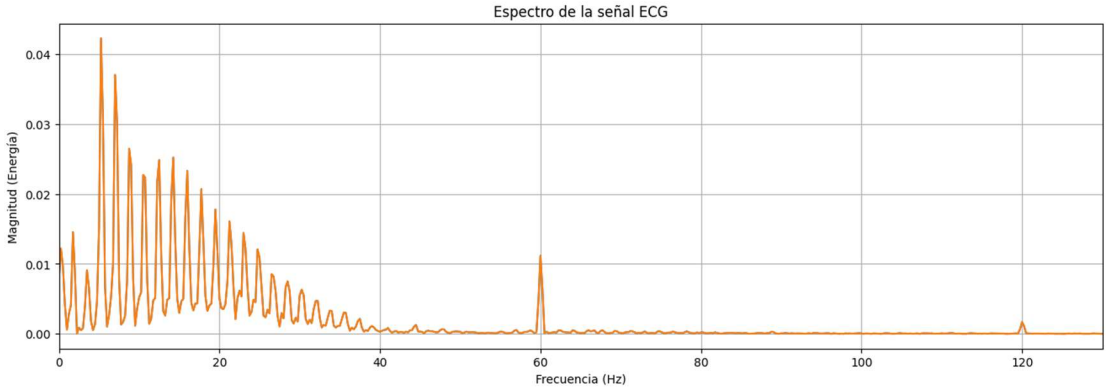



*Figura 25. Espectro de frecuencias de la señal original sin procesar. Fuente: Autor.*

*Tabla VIII – Parámetros de diseño del filtro digital pasabajas.*

| **Parámetro** | **Valor** |
|---|---|
| Frecuencia de corte | 40 Hz |
| Banda de transición | 19.5 Hz |
| Rizado en la banda de paso | 1 dB |
| Atenuación en la banda de rechazo | 40 dB |

bandas, y el algoritmo buscará los coeficientes del filtro que minimicen el error máximo en la respuesta de frecuencia.

- Óptimo: El método de Remez busca minimizar el error máximo en la respuesta de frecuencia, lo que resulta en un filtro óptimo en el sentido de Chebyshev. Esto significa que se logra la máxima atenuación de las bandas laterales para una respuesta de frecuencia dada, al mismo tiempo que minimiza el rizado en la banda de paso.

Aunque una de las desventajas que se suele asociar con este método es que tiene una mayor complejidad computacional, ya que tiende a ser más intensivo en términos de recursos computacionales que algunos otros métodos, especialmente cuando se busca una alta precisión y especificaciones estrictas, lo cierto es que las especificaciones planteadas son bastante laxas para el cálculo del filtro paso-bajo y por ende no es una limitante en ese sentido.

Se implementó un script en Jupyter Notebooks para la realización de los cálculos del filtro y las correspondientes gráficas con ayuda de las librerías scipy y matplotlib. Luego de algunas iteraciones para probar con diferentes longitudes de filtro, se determinó que la menor longitud de filtro que cumple con los requerimientos establecidos fue de N = 52, por tanto, el orden del filtro N-1 es 51.

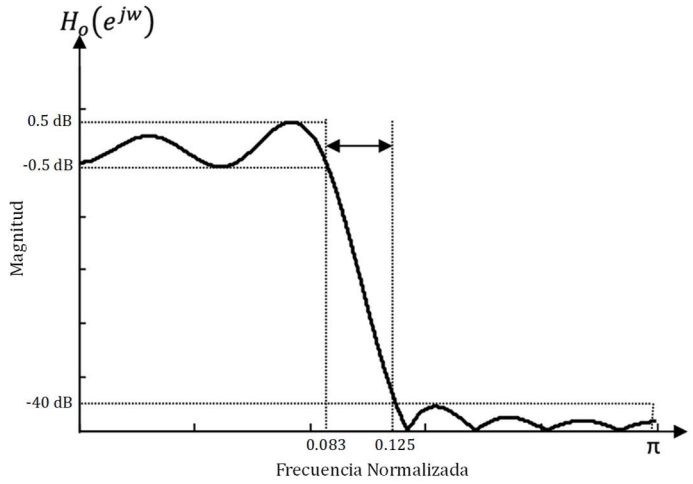



*Figura 26. Diagrama de parámetros de diseño del filtro digital.*

A continuación, se muestra en la Figura 27 la respuesta en frecuencia resultante del filtro, tanto en amplitud como en fase. Se señalaron con líneas rojas los parámetros de diseño para confirmar visualmente que el resultado fuera el correcto, validando que en la banda de paso el rizado es menor a 1 dB y la atenuación en la banda de rechazo de 40 dB.

De igual manera, se graficó la respuesta al impulso del filtro, cuyos valores corresponden a los coeficientes del filtro, a como es mostrado en la Figura 28. La duración de esta respuesta es equivalente a la longitud del filtro dividido por la

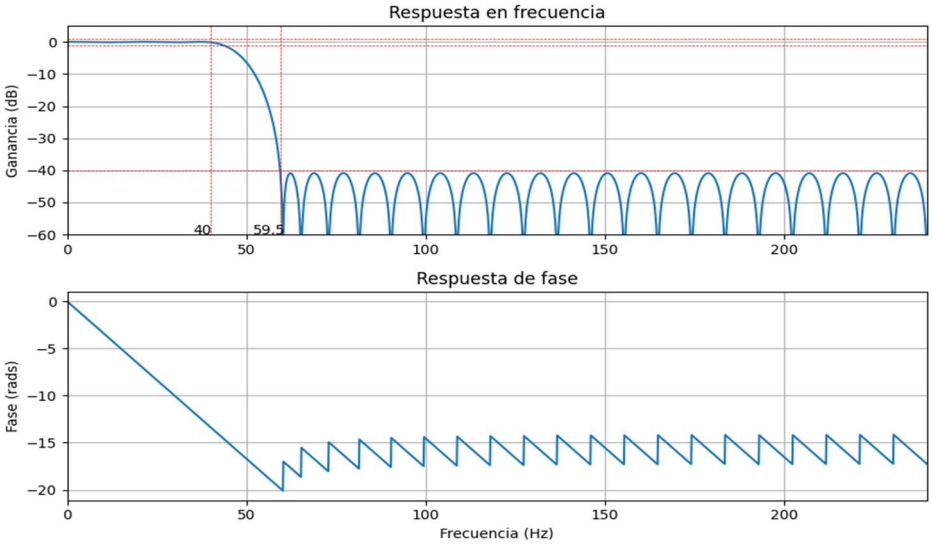



*Figura 27. Respuesta en frecuencia (amplitud y fase) del filtro digital.*

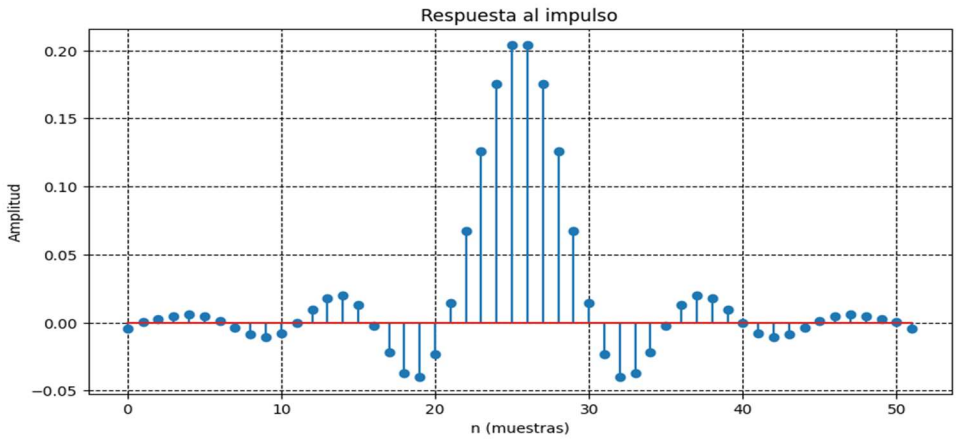



*Figura 28. Respuesta al impulso del filtro digital.*

tasa de muestreo, el retraso de grupo es la magnitud que mide cuánto tiempo de retraso aplicará el filtro sobre la señal procesada, la cual se calcula a través de la Ecuación (8) igualmente medido en milisegundos.

$$D_g = (N - 1) / (2 * f_s) = 53 \text{ ms} \qquad (8)$$

Donde:

N: longitud del filtro

$F_s$: tasa de muestreo

Para validar la efectividad del filtro se realizó el filtrado de la señal tomada originalmente y se aplicó el filtro calculado utilizando la librería nuevamente la librería scipy y mostrando ambas graficas en la Figura 29. Nótese que la forma de onda resultante se ve mucho más definida.

#### 2.2.1.6 Comunicación Bluetooth Low Energy

La especificación de Bluetooth Low Energy establece un mecanismo de intercambio de información llamado Protocolo de Atributos (ATT), el cual define la manera en que un servidor, que en este caso sería el electrocardiógrafo, expone sus datos a un cliente, que en este caso sería la aplicación móvil. En ese sentido, un atributo es un término genérico para cualquier dato expuesto por el servidor, cuya estructura es mostrada en la Figura 30.

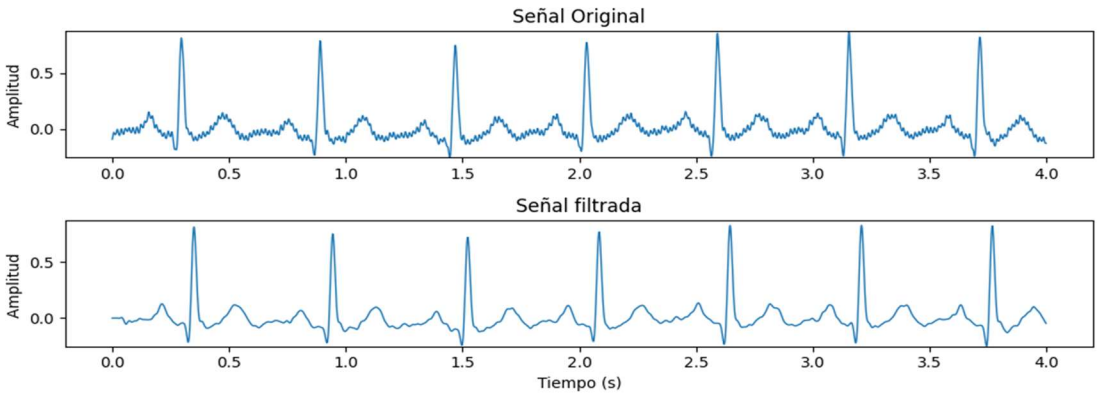



*Figura 29. Comparación entre la señal electrocardiográfica original y filtrada.*

A partir del protocolo ATT, la especificación Bluetooth establece dos perfiles genéricos para uso en aplicaciones embebidas de propósitos específicos, los cuales son el Perfil de Acceso Genérico (GAP) y el Perfil de Atributos Genéricos (GATT). El primero es una capa de control de alto nivel que define procedimientos, modos de operación y roles entre dos dispositivos Bluetooth. El segundo es una capa de datos de alto nivel para intercambio de información que estructura los atributos en una jerarquía de servicios y características, para que los dispositivos puedan leer y escribir datos entre sí.

En sentido, un servicio GATT es una colección de características. Cada característica es la unidad básica de información y es un atributo que se identifica por un Identificador Único Universal (UUID) y está conformada por lo siguiente [34]:

- Descriptores: son atributos que funcionan como metadatos asociados a una característica que proporcionan información adicional sobre la característica o modifican su comportamiento. Los descriptores pueden utilizarse para describir el formato del valor.

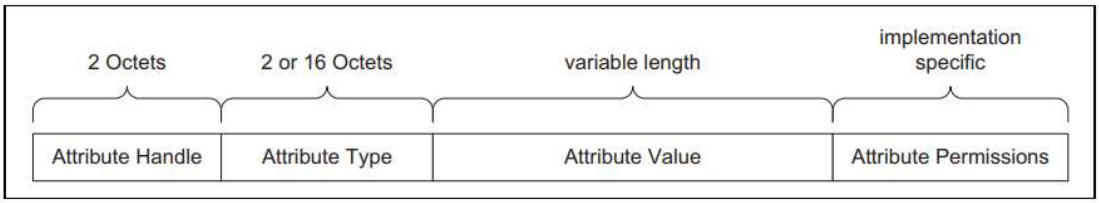



*Figura 30. Formato binario de un atributo en Bluetooth Low Energy. Adaptado de [40].*

- Valor: es el dato real que representa la característica. Puede ser un número, una cadena de texto o una cadena de bytes, dependiendo de la característica.
- Propiedad: define el comportamiento de una característica y cómo se puede acceder o modificar su valor (lectura, escritura, notificación, etc.)

De esta manera, se propone el uso de un único servicio GATT. Este servicio se conforma a su vez de dos características, una de ellas para el control del dispositivo donde se escriban los comandos y se lea el estado del dispositivo, mientras que la otra característica se encarga de exponer la muestras de la señal electrocardiográfica. La Figura 31 muestra la estructura propuesta con las características mencionadas.

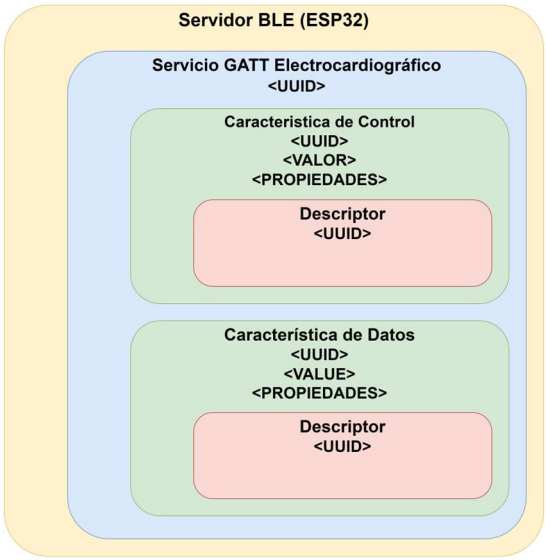



*Figura 31. Diseño del servidor GATT para implementación en el microcontrolador. Fuente: Autor.*

### 2.2.2 Diseño de la aplicación móvil

La aplicación móvil que se propone aquí sirve como interfaz para la toma de electrocardiogramas, por ende, debe ser capaz de manejar la comunicación bluetooth a través de la interfaz, al mismo tiempo, almacena la información de los pacientes y los exámenes para posteriormente ser enviados.

En ese sentido se proponen dos corrutinas que se ejecutan de manera concurrente: una corrutina controla la comunicación BLE, cuyo diseño es una

máquina de estado finito complementaria a la del prototipo, mientras que la otra se encarga de manejar la interfaz con el usuario, incluyendo el almacenamiento de la información en la base de datos y las tareas de envío y recepción de la información.

En el caso de la comunicación por BLE la máquina de estados a implementar se muestra en la Figura 32. De los cuatro estados hay tres que se corresponden directamente con los de la máquina de estado finito del prototipo, mientras que el cuarto, SCANNING, es un estado transitorio mientras se escanea el medio circundante en busca del prototipo para luego iniciar la conexión en el estado CONNECTED. De no hallarse el prototipo se regresa al estado IDLE. Mientras el usuario permanece en el estado CONNECTED, la aplicación estará a la espera de que se presione el botón de inicio del examen, en cuyo caso se enviará el comando apropiado al prototipo y se pasará al estado SENDING.

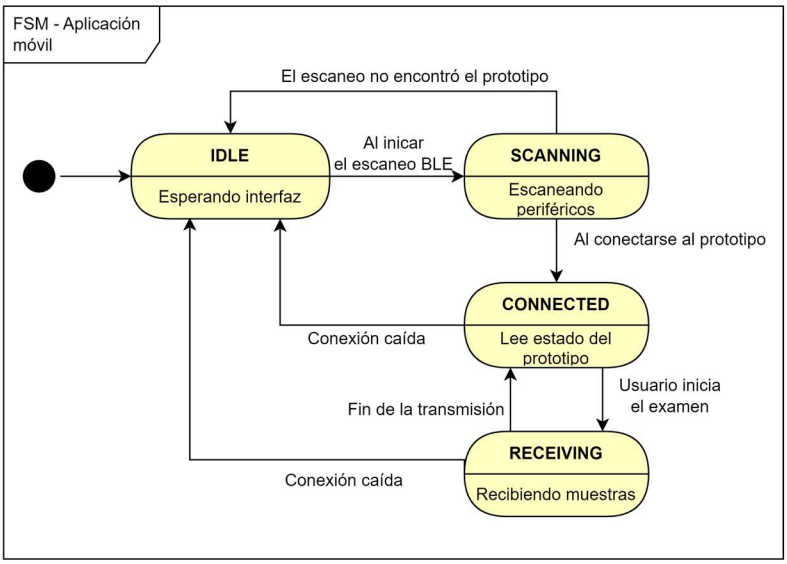



*Figura 32. Máquina de estado finito para la corrutina de comunicación BLE de la aplicación Móvil.*

La interfaz de usuario por su parte se compone de 5 apartados principales, a través de las cuales el usuario podrá interactuar con el sistema:

- Inicio: muestra algunas estadísticas y notificaciones de nuevos diagnósticos.

- Exámenes: tiene la lista de exámenes guardados, enviados y diagnósticos recibidos. Aquí se pueden crear nuevos exámenes y revisar los existentes.
- Pacientes: contiene la lista de pacientes, detalle de estos y se puede registrar nuevos pacientes.
- Configuración: contiene algunas opciones para modificar el comportamiento de la aplicación o del prototipo. Aquí también se registra el usuario
- Ayuda: muestra instrucciones de uso e información de soporte.

La Figura 33 muestra las diferentes vistas con que contará cada apartado dentro de la interfaz. Éstas serán accesibles a través de un menú de navegación. Para pasar de una vista a la siguiente en el árbol de navegación hay que seleccionar opción de una lista o bien presionar un botón de acción. Para regresar a la anterior se hace uso de un botón en la parte superior izquierda de la pantalla como es costumbre en aplicaciones móviles.

En una base de datos local se almacena los detalles de pacientes y los exámenes, los cuales son accedidos a través de las vistas “Detalle de paciente” y “Detalle de

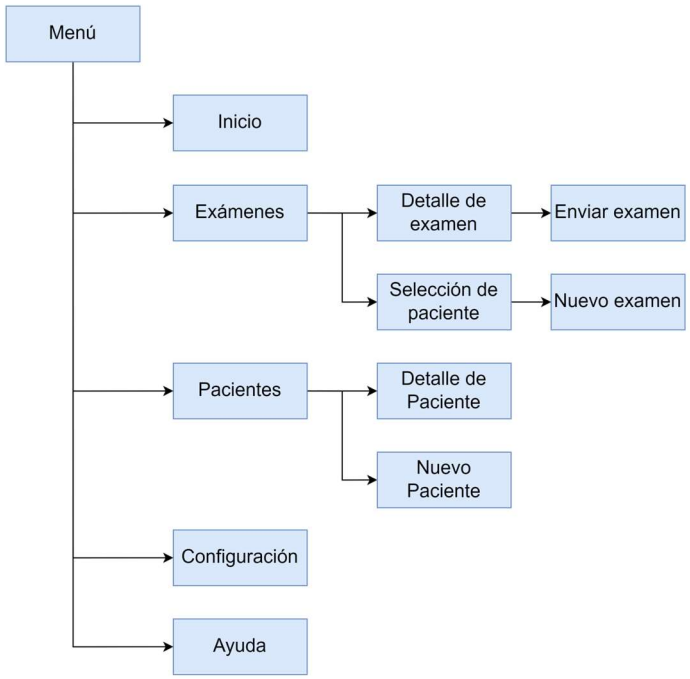



*Figura 33. Diagrama de navegación de la aplicación móvil.*

examen” respectivamente. Lo pacientes se ingresan a la base de datos a través de un formulario en la vista “Nuevo paciente”, mientras que para registrar un electrocardiograma la aplicación solicita al usuario que indique a qué paciente pertenece el examen, por lo cual será necesario ingresar al paciente en la aplicación antes de empezar. Finalmente, se muestra una vista donde se puede ingresar signos vitales, comentarios que también una sección para controlar el estado del prototipo y un área donde se grafica la señal a medida que se están recibiendo las muestras.

El modelo de datos a implementar en la base de datos local gira en torno a la clase Examen, al cual corresponde un Electrocardiograma, un Paciente, y un Hospital de destino. En la Figura 34 se muestras esquemáticamente estas relaciones, los campos usados y el tipo de dato de cada uno. La tabla de exámenes (Exam) contiene metadatos como origen y destino, al igual que las marcas de tiempo de cuando fue creado, enviado y diagnosticado. Guarda el peso, presión arterial y saturación de oxígeno del paciente, al igual que los comentarios y el contenido del diagnóstico.

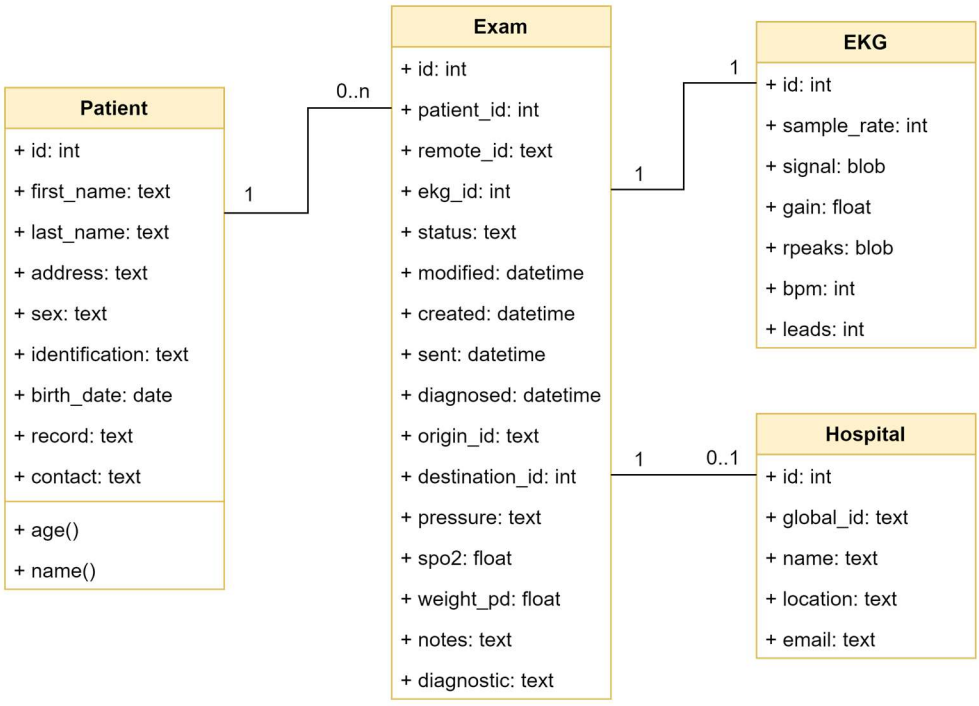



*Figura 34. Modelo de datos de la aplicación móvil.*

Los tabla de electrocardiogramas (EKG) guardan una relación de cardinalidad 1 a 1 con cada examen. Cada registro en esta tabla contiene la señal de interés comprimida con un formato de compresión sin pérdidas y almacenada como un BLOB (Objecto Binario Grande), además de información adicional como la ganancia del equipo con que se capturó, la cantidad de derivaciones, la tasa de muestreo, así también algunas características extraídas como la frecuencia cardiaca y las marcas de tiempos en los que se detectaron los picos R, que es una lista de enteros, los cuales se almacenan de igual manera como un BLOB.

La tabla de hospitales (Hospital) es la lista de destinos a los cuales puede remitirse un examen. La relación con cada examen tiene una cardinalidad de 1 a 0 cuando no se ha enviado el examen y de 1 a 1 cuando el examen se haya enviado. De igual forma, la tabla de pacientes (Patient) contiene los datos de identificación de cada paciente y se relaciona con los exámenes con una cardinalidad de 1 a N.

El *framework* de desarrollo elegido para implementar la aplicación móvil fue Kivy bajo el lenguaje de programación Python, debido a que es *framework open-source*, multiplataforma y Python por su parte, es un lenguaje de fácil aprendizaje. Para la base de datos local se decidió el uso de SQLite debido a que ofrece soporte para lenguaje SQL sin la necesidad de implementar un servidor de base de datos convencional, ya que opera únicamente con archivos locales.

### 2.2.3 Diseño de solución en la Nube

Como ya se mencionó, la solución se compone de una base de datos y un servicio de cómputo. Este último consta de dos partes esenciales. se propone el uso de un servicio de hosting para servir la página web del frontend a través del cual los médicos pueden enviar el diagnóstico de los exámenes. La página web es de tipo estática para que sea renderizada en el lado del cliente. Éste se comunicará a través de solicitudes HTTP hacia el *backend*.

En segundo lugar, para el backend se propone el uso de funciones lambda que son un tipo de servicio computacional *serverless*, lo que significa que no se

requiere el despliegue de un servidor ni del software adicional para que un programa pueda ejecutarse. En ese sentido, estas funciones ejecutarán por separado el servicio de notificaciones y el servicio de recepción de diagnósticos, así como realizar validaciones al momento de insertar o actualizar registros en la base de datos (véase en la Figura 35).

Las funciones específicas que serían implementadas, como ya se mencionó en los requerimientos, notifican de exámenes nuevos, validan el envío de diagnósticos y adicionalmente permiten al frontend validar si el enlace para un diagnóstico es válido.

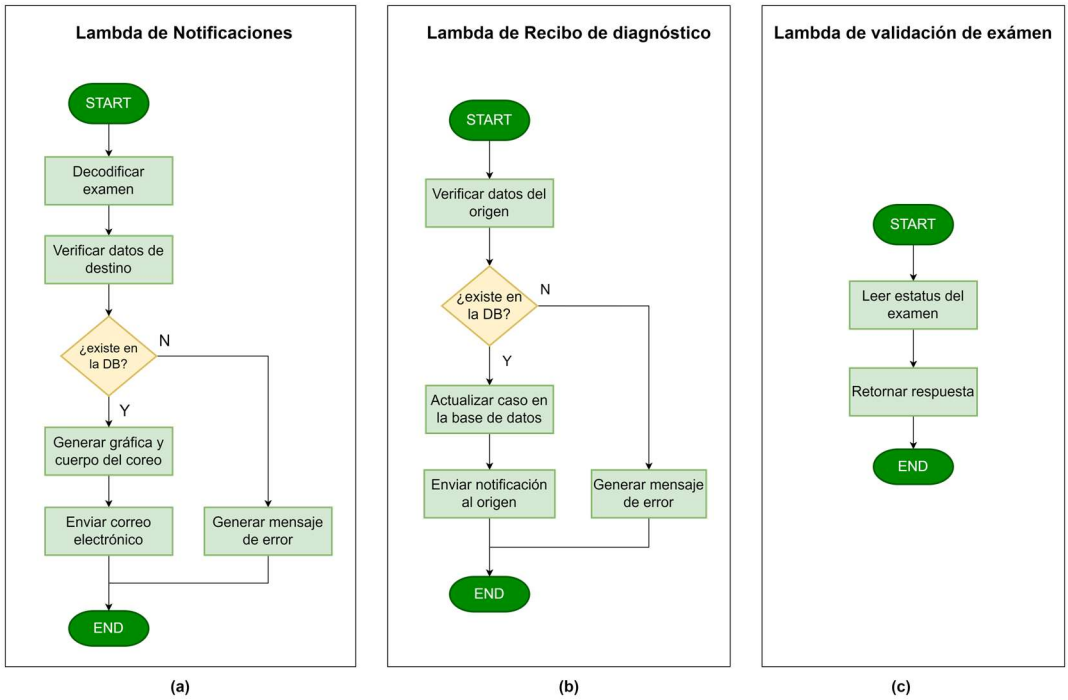



*Figura 35. Flujogramas de las funciones lambda.*

Por otro lado, se eligió una base de datos no relacional de documentos que almacena 2 colecciones de documentos en formato JSON. Las colecciones definidas son las siguientes:

- Casos: Es el nombre que reciben los exámenes una vez que se han enviado hacia un hospital de destino para su diagnóstico.
- Hospitales: contiene la información de localización y contacto de los hospitales o clínicas de destino.

- Centros de Salud: contiene la información de localización y contacto de los centros de salud de origen.

La plataforma de servicios en la nube seleccionada para implementar este diseño fue Firebase, del proveedor Google, debido a que ofrece una gran variedad de servicios, incluyendo los que se han planteado en este apartado. Del mismo modo, ofrece una capa de servicio gratuita, lo cual la hace ideal para el desarrollo de este proyecto monográfico. La Figura 36 muestra la arquitectura a implementar.

## 2.3 Etapa de Implementación

Durante la etapa de implementación se procedió a plasmar en código los diseños funcionales de la etapa de diseño tanto en el prototipo electrónico, como en la aplicación móvil y servicios en la nube. De igual manera se configuraron los parámetros más adecuados en cada componente del sistema para su correcto funcionamiento, de acuerdo con los requerimientos, características de operación y parámetros de diseño.

Para efectos de identificación y documentación del desarrollo del sistema en su conjunto, se le dio el nombre de OpenKardio y se creó un repositorio en GitHub del mismo nombre[1].

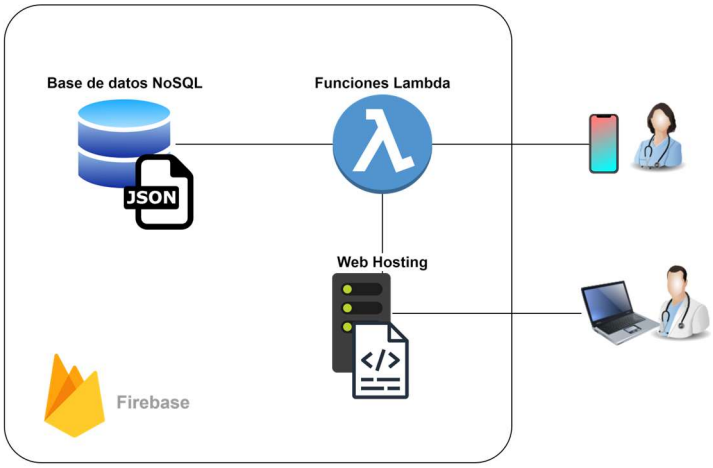



*Figura 36. Arquitectura de la solución en la nube de Firebase.*

[1] El URL del repositorio es https://github.com/eliezer42/openkardio

### 2.3.1 Implementación del Prototipo Electrónico

Durante la implementación del prototipo electrónico se procedió a realizar pruebas de funcionamiento a los componentes individuales en aras de conocer a detalle sus características de operación y determinar los parámetros más adecuados de configuración. A continuación, se detalla el proceso de implementación del prototipo electrónico.

#### 2.3.1.1 Configuración y programación del firmware

De las características eléctricas del *frontend* la más relevante para la configuración del circuito de adquisición de señal es el rango dinámico de la señal a la salida del módulo AD8232, que va de 0 a $V_{CC}$. A pesar de que la naturaleza diferencial de la señal electrocardiográfica hace ésta que tenga valores negativos en determinados momentos, el circuito del módulo agrega un voltaje de compensación equivalente a la mitad del voltaje de alimentación con el finde evitar los voltajes negativos de la señal y poder operar con una fuente de alimentación única. Como resultado la señal tiene los siguientes parámetros, expresados en las Ecuaciones (5) y (6), con base en los cuales se debe seleccionar los parámetros de configuración del ADC.

$$DR = V_{max} - V_{min} = V_{CC} = 3.3\ V \qquad (5)$$

$$V_{Offset} = V_{CC} / 2 = 1.65\ V \qquad (6)$$

Donde

DR: rango dinámico de la señal

$V_{max}$, $V_{min}$: Voltaje máximo y mínimo de la señal, respectivamente.

$V_{Offset}$: Voltaje de compensación DC

$V_{CC}$: Voltaje de alimentación

Por su parte el IC ADS1115 cuenta con un registro de configuración (véase la Figura 37) que permite configurar entre otras cosas, el Amplificador de Ganancia

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
|---|---|---|---|---|---|---|---|
| OS | MUX[2:0] | | | PGA[2:0] | | | MODE |
| R/W-1h | R/W-0h | | | R/W-2h | | | R/W-1h |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| DR[2:0] | | | COMP_MODE | COMP_POL | COMP_LAT | COMP_QUE[1:0] | |
| R/W-4h | | | R/W-0h | R/W-0h | R/W-0h | R/W-3h | |

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

*Figura 37. Registro de configuración del IC ADS1115. Adaptado de [39].*

Programable (PGA) que controla el rango valor del Rango de Escala Completa (FSR) del ADC, así como la tasa de muestreo.

Dado que la resolución del ADC es de 16 bits, esto define 65,536 niveles de cuantización repartidos a lo largo del FSR, cuyo valor se puede modificar al escribir en los bits PGA[2:0] según la Tabla IX, en donde también está expresado el valor del voltaje entre dos niveles de cuantización consecutivos, también llamado tamaño de Bit Menos Significativo (LSB).

Nótese que, a mayor rango, mayor es el tamaño del LSB y viceversa, siendo los dos rangos superiores los únicos capaces de muestrear el rango de la señal de interés. Sin embargo, para no perder resolución es necesario configurar el rango de +/-4.096 V, dando como resultado un tamaño de LSB de 125 μV. Esto se traduce en que el rango de la señal digitalizada en el dominio del tiempo discreto irá desde 0 hasta 26,400 niveles de cuantización, que en binario con complemento a 2 sería de 0000h a 6720h. Por tanto, es posible emplear un entero con signo de 16 bits para representar cada muestra en el microcontrolador.

Por otra parte, el ADC tiene dos modos de conversión, uno es de disparo único, donde la conversión ocurre únicamente cuando el microcontrolador lo solicita, y el otro modo es de conversión continua. Dado que se necesita tener control sobre la tasa de conversión, se hizo uso del modo de disparo único, al establecer el bit MODE del registro de configuración, lo cual también contribuye al ahorro de energía pues el dispositivo se apaga internamente una vez que se ha completado una conversión y se vuelve a encender hasta la siguiente petición. Sin embargo, también fue necesario configurar el tiempo que dura la conversión al más corto posible, lo cual se logró configurando la máxima tasa de muestreo de 860 sps, al

*Tabla IX – Rangos de Escala Completa configurables y su correspondiente tamaño LSB. Adaptado de [39].*

| FSR | LSB SIZE |
|---|---|
| ±6.144 V(1) | 187.5 µV |
| ±4.096 V(1) | 125 µV |
| ±2.048 V | 62.5 µV |
| ±1.024 V | 31.25 µV |
| ±0.512 V | 15.625 µV |
| ±0.256 V | 7.8125 µV |

establecer los bits DR[2:0], dando como resultado un tiempo de conversión máximo de acuerdo con la Ecuación (7) indicada en la hoja de datos.

$$T_{Conv} = 1/DR = 1.16 \text{ ms}, \qquad (7)$$

donde DR es la tasa de muestreo seleccionada en el módulo.

Adicionalmente, la comunicación I2C se configuró en *fast-mode* que corresponde a una frecuencia en la señal de reloj de 400 kHz, lo que significa que para transferir los 16 bits de una única muestra al bus le toma 40 µs, dando como resultado un tiempo total para obtener una muestra de 1.2 ms.

En la sección 2.1.1 se estipuló una tasa muestreo para el electrocardiógrafo de entre 250 sps y 500 sps. De igual forma, en la sección 2.2.1.3 se propuso un criterio de diseño para que el envío de las muestras de la señal se realice a través de tramas que llevan un grupo de muestras consecutivas a la vez. Teniendo lo anterior en cuenta se implementó una tasa de muestreo de 480 sps ya que este número, al ser múltiplo de 2, 3 y 5, permite muchas combinaciones al momento de elegir el número de muestras a enviar en cada trama, según resultase conveniente durante la realización de pruebas.

En ese sentido, el estado SEND es el más importante ya que es ahí cuando se realiza el muestreo de la señal, el filtrado y el envío de esta. Cada N muestras capturadas se enviará una trama de muestras al dispositivo Android que ejecuta la aplicación móvil. En la Figura 38 se muestra la estructura de esta trama. Dado

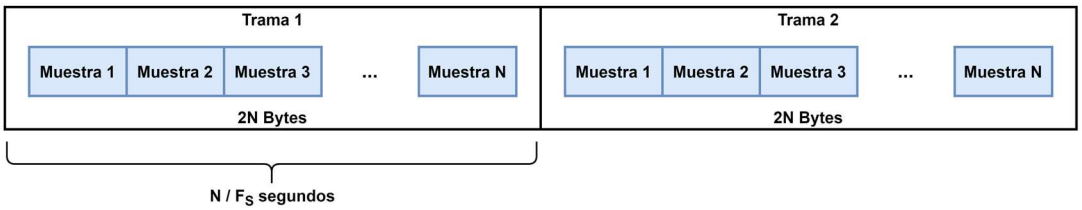



*Figura 38. Estructura de la trama de muestras a enviar.*

que cada muestra es de 2 bytes, el tamaño de cada trama es de 2N bytes y el intervalo de tiempo entre el envío de una trama y la siguiente es de N dividido entre la tasa de muestreo $F_s$. El número N se calcula más adelante a partir de las restricciones del estándar BLE.

A fin de orquestar la operación del muestreo y envío de la señal, se configuró el timer0 en modo temporizador con recarga automática. Este temporizador cuenta con un contador de 64 bits y un pre-escalador de 16 bits. En la tarjeta de desarrollo la señal de reloj que alimenta este timer es de 80 MHz, por tanto, se decidió establecer el pre-escalador en un valor de 8 para que la señal de reloj efectiva a la entrada del contador igual a 10 MHz. La ISR del interruptor es disparada cuando el timer alcanza el valor dado por la Ecuación (9).

$$C_{INT} = F_{IN} / F_S = (10\ MHz) / 480\ sps = 20{,}833 \qquad (9)$$

Donde:

$C_{INT}$: Conteo de interrupción del timer

$F_{IN}$: Frecuencia efectiva a la entrada del contador

$F_S$: Tasa de muestreo

Para el desarrollo del firmware se creó un nuevo proyecto en PlatformIO usando el entorno de desarrollo Visual Studio Code, y se añadió al repositorio dentro de una carpeta llamada “firmware”. En este cual se incluyeron algunas librerías de dominio público como la librería Arduino.h que permite utilizar el *framework* de Arduino para tener acceso a las definiciones, las API y librerías de Arduino dentro del lenguaje C++. De esta manera también es posible aprovechar librerías hechas por la comunidad de Arduino y que no están disponibles en C/C++ directamente.

Asimismo, se incluyó el paquete de librería Arduino-esp32, el cual es soportado directamente por el fabricante Espressif y por miembros de la comunidad, cuyo propósito es brindar acceso a los periféricos del módulo ESP32 a través de librerías compatibles con Arduino. En la Figura 39(a) se muestra la estructura de directorios utilizada, mientras que en la Figura 39(b) está reflejado el contenido del archivo de configuración platformio.ini con las versiones de cada librería.

En el caso de la implementación de la máquina de estados que controla el comportamiento del prototipo, se optó por la librería StateMachine.h del usuario de GitHub llamado jrullan. Esta permite definir estados, condiciones de transición e implementar una mecánica para ejecutar una pieza de código una única vez al hacer la transición de un estado a otro. Cabe destacar se realizó una modificación a la librería para poder leer el estado actual de la máquina lo cual fue útil para determinar si la máquina debe cambiar de estado ante la llegada de un nuevo comando. Esta librería a su vez emplea una librería de listas enlazadas como dependencia, llamada LinkedLists.h.

En cuanto a la interfaz con el módulo ADC se empleó la librería ADS1X15.h del usuario robtillaart en GitHub. Esta librería expone a través de funciones la configuración interna del IC ADS1115 haciendo uso de comunicación I2C, lo cual ocurre de forma transparente para el desarrollador. Convenientemente ofrece un funciones de lectura de muestras de forma síncrona y asíncrona. Las anteriores librerías se encuentran dentro de la carpeta “libdeps” mostrada en la Figura 39(a) y también están indicadas como dependencias en la Figura 39(b).

Por otro lado, la implementación del algoritmo de filtrado descrito en la sección 2.2.1.3 en el microcontrolador se utilizó la librería de lenguaje C provista por el software de diseño de filtros digitales MicroModeler, cuya licencia permite el uso gratuito para propósitos académicos y sin fines de lucro [35]. Se eligió esta librería debido a que permite instanciar tantos filtros digitales como sean necesarios, con poco esfuerzo, al encapsular el estado de cada filtro en su propia estructura de datos (struct) y definiendo funciones para operar sobre cada filtro de forma

independiente. Esto en consecuencia haría más sencilla la adición de más derivaciones en el electrocardiógrafo. Esta librería se incluye dentro de la carpeta “libs”, tal como se muestra en la Figura 39(a), como lpfilter.cpp.

La implementación de la comunicación Bluetooth se realizó a través de la librería BLE del framework Arduino para ESP32, la cual expone varias interfaces (clases) para implementar un servicio GATT en el microcontrolador. El proceso consiste en instanciar un Servidor, añadir características y descriptores a éste, tal como se propuso en la sección 2.2.1.5. Se definen también las propiedades de cada característica: para el caso de la característica de control se le otorgaron las propiedades de lectura y escritura para permitir el la recepción de comandos y el envío de información desde el prototipo, por otro lado, la característica de datos se le asignó las propiedades de lectura y notificación, a fin de poder emplear el mecanismo de notificación al comunicarse con la aplicación, puesto que este mecanismo es más consistente para transmitir las muestras de la señal, en comparación con leer constantemente la característica en el prototipo desde la aplicación.

Por medio de la característica de control se reciben los comandos para el prototipo, al enviar un byte con el valor definido para cada uno de los 4 comandos definidos en la sección 2.2.1.5. Sin embargo, cual es prototipo ha establecido una


(a) (b)

*Figura 39. Configuración del Proyecto en PlatformIO.*

conexión con un cliente, esta característica contiene la información del estado del prototipo. El estado del prototipo esta codificado en una trama binaria (véase la Figura 40) y contiene los valores de las siguientes variables, con su tipo de dato entre paréntesis:

- battery_level (uint8_t): El porcentaje de la batería del dispositivo calculado al medir el voltaje de la celda de litio a través de un divisor de voltaje conectado al pin analógico en la tarjeta de desarrollo.
- sample_rate (uint8_t): es la tasa de muestreo del prototipo.
- samples_per_frame (uint16_t): es la cantidad de muestras que contendrá cada trama enviada a través de Bluetooth (Ver Figura 38).
- lead_count (uint8_t): el número de derivaciones que contiene la señal.
- resolution (uint8_t): la resolución en bits del ADC.
- fw_major_version (uint8_t): dígito más significativo de la versión del firmware.
- fw_minor_version (uint8_t): dígito menos significativo de la versión del firmware.
- conv_factor (double): el factor de conversión para transformar el valor digital de la señal en valores de voltaje (milivoltios).

La característica de datos encargada del envío de la señal se describió como una trama de muestras. Para determinar valores adecuados del número N de muestras en cada trama fue necesario considerar el máximo tamaño de atributo admitido por el estándar BLE, que es de 512 bytes. El tamaño máximo de un atributo impone a su vez un máximo para N, ya que todas las muestras deben caber en un único atributo, que en este caso es la característica de Datos. De esta

| Byte 15 | Byte 14 | Byte 13 | Byte 12 | Byte 11 | Byte 10 | Byte 9 | Byte 8 |
|---|---|---|---|---|---|---|---|
| Factor de conversión | | | | | | | |
| | | | | | | | |
| Byte 7 | Byte 6 | Byte 5 | Byte 4 | Byte 3 | Byte 2 | Byte 1 | Byte 0 |
| FW MSD | FW LSD | Resolución | Derivaciones | Tasa de muestreo | | muestras/trama | % Batería |

*Figura 40. Trama binaria del estado del prototipo..*

manera en las Ecuación (10) y se tiene el límite a los posibles valores que puede adoptar N.

$$N_{MAX} = ATT_{MAX} / S = 256 \qquad (10)$$

Donde

$ATT_{MAX}$: es el máximo tamaño de atributo
S: El tamaño de una muestra (2 bytes)

Si bien el rango de valores que puede adoptar N es amplio, se seleccionó un valor de N = 15 para el tamaño de las tramas, lo cual implica que se envían 32 tramas por segundo, siendo este un numero de adecuado para que se actualice la gráfica de la señal en la pantalla de la aplicación. El tamaño resultante del valor de la característica de datos es 30 bytes, sin embargo, el tamaño de los paquetes (MTU) de la capa L2CAP de Bluetooth, debe tener un valor igual o mayor al tamaño del atributo máximo a enviar más 3 bytes de cabecera, por tanto, éste se configuró en 33 bytes dentro del archivo OKConfig.h donde además se encuentran algunas otras configuraciones como la tasa de muestreo.

Una vez terminada la programación del firmware se realizaron pruebas y corrección de errores haciendo uso de la consola para depurar el código y la aplicación móvil nRF Connect para depurar la comunicación Bluetooth. En la Figura se muestran capturas captura de pantalla de la aplicación nRF Connect mostrando el prototipo nombrado “OKDevice” anunciándose, listo para establecer la conexión y el servicio implementado con sus respectivas características. Nótese que el UUID del servicio y los UUID de las características se usan para interactuar con la funcionalidad del prototipo fueron definidos como:

- UUID Servicio OpenKardio: "5e6c**5000**-05d8-463b-b21d-eed2204c2002"
    - UUID Datos: "5e6c**5001**-05d8-463b-b21d-eed2204c2002"
    - UUID Control: "5e6c**5002**-05d8-463b-b21d-eed2204c2002"

El código fuente del firmware puede ser hallado en el Anexo G.

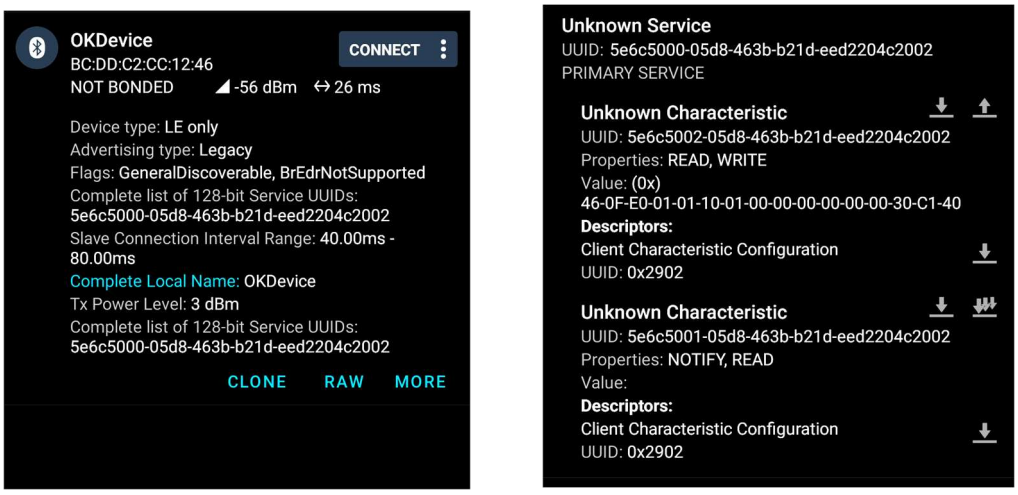



*Figura 41. Capturas de la aplicación nRF24 mostrando las características BLE del prototipo.*

### 2.3.1.2 Construcción Física del prototipo

Una vez que se validó la operación del prototipo se procedió con el diseño de la PCB para posteriormente hacer el montaje físico del prototipo, haciendo uso del *software* Eagle 7.6. En este se creó primeramente un esquemático para una tarjeta que sirve como soporte o *backplane* a los módulos que conforman todo el circuito (véase la Figura 42). Se emplearon cabeceras de pines para modelar las conexiones de los módulos. De igual forma se realizó el diseño de las pistas a partir del esquemático.

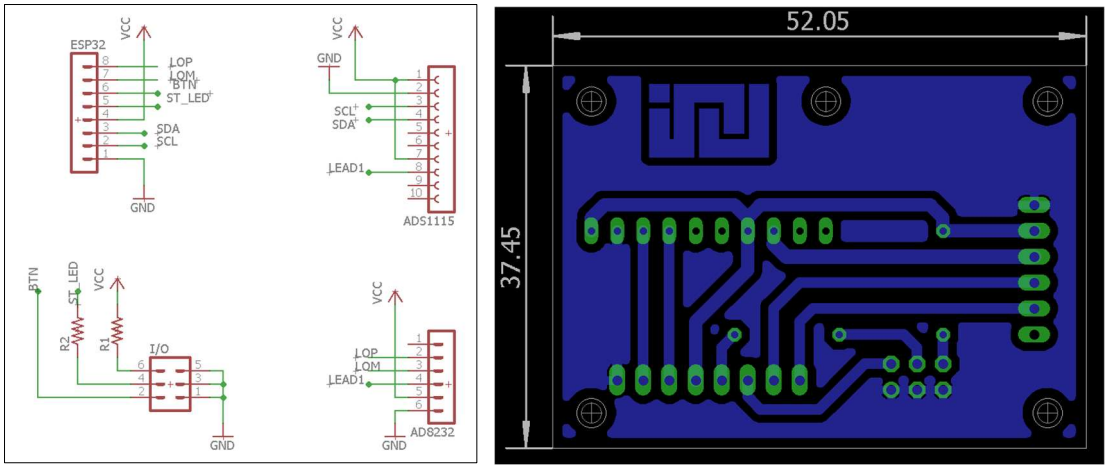



*Figura 42. Esquemático y diseño de PCB del prototipo.*

Para la construcción de la tarjeta se optó por el método de transferencia térmica sobre una tarjeta virgen de cobre. Se imprimió el diseño de las pistas y la máscara de componentes en papel fotográfico. Una vez transferido el diseño a la tarjeta de cobre se sometió a ácido nítrico diluido a una concentración aproximada de 1 parte de ácido por 3 partes de agua, para que las pistas quedaran grabadas en el cobre. A continuación, se removió la tinta remanente con acetona y se taladró los agujeros para los componentes. Por último, se procedió a soldar las cabeceras de pines, tanto hembra como macho según correspondiese y los demás componentes discretos, resultado una PCB de dimensiones 52.05 x 37.45 mm. El proceso se muestra en el Anexo B.

Asimismo, se construyó una tarjeta de apoyo para servir de extensión al puerto USB del módulo ESP32, cuyo propósito es el de sostener un puerto USB tipo B en un punto conveniente de la carcasa para permitir la alimentación y transferencia de datos a través de él y el microcontrolador. El diseño de esta PCB se muestra en la Figura 43.

El prototipo se encapsuló en una caja de plástico ABS con dimensiones 151 x 83 x 53 mm. En total, el interior consta de 5 tarjetas individuales, donde el módulo ADS1115 está inserto directamente en la PCB de conexiones, mientras que los demás módulos se conectan a través de cables con terminales para cabeceras de pines. El interior del prototipo se muestra en la Figura 44. Se realizaron perforaciones para la conexión USB, el conector de electrodos del frontend analógico, y en la tapa se colocó un interruptor de encendido y dos LED indicadores.

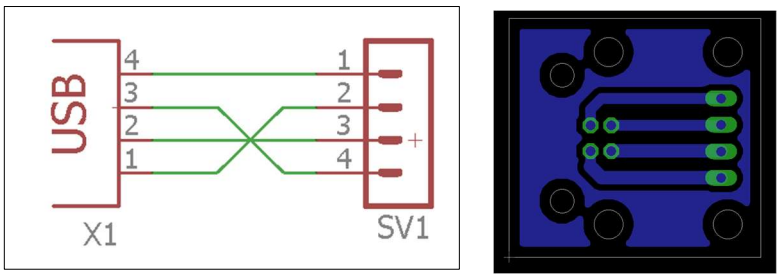



*Figura 43. Esquemático y diseño de PCB del puerto USB..*

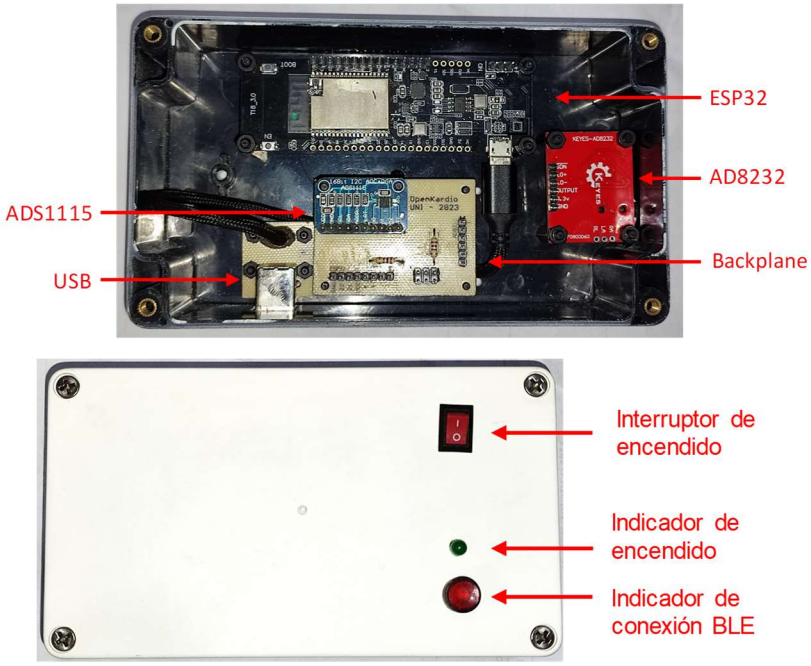



*Figura 44. Vista exterior e interior del prototipo.*

### 2.3.2 Implementación de la Aplicación en Android

El framework utilizado para construir la aplicación móvil es Kivy, que permite definir la lógica de funcionamiento de la aplicación usando el lenguaje Python, así como la interfaz gráfica usando un lenguaje propio denominado lenguaje KV. Este *framework* puede crear aplicaciones que se ejecutan sistemas operativos Linux, Windows, iOS y Android y tiene la flexibilidad de que se puede emplear paquetes de librerías estándares del ecosistema de Python del mismo modo que puede hacer uso de las APIs propias de cada sistema operativo, siendo en el caso de Android a través de la librería PyJNIus, que permite usar clases Java desde el código en Python.

En el caso de las aplicaciones compiladas para Android, el ejecutable empaqueta todo lo necesario para que el código en Python y KV funcione, incluyendo el intérprete de Python, las librerías y las dependencias de lenguaje Java, lo cual es logrado a través de la herramienta de línea de comando conocida como Buildozer. De igual forma esta herramienta, permite configurar versiones de Android

compatibles, dependencias, entre otras configuraciones relacionadas al aspecto visual de la aplicación.

El proceso para desarrollar la aplicación constó de tres fases: a) desarrollo de interfaz de usuario y base de datos local, b) integración entre los elementos de la interfaz de usuario y la comunicación con la nube y c) implementación de la comunicación Bluetooth Low Energy y post procesamiento de la señal. El ambiente de desarrollo elegido fue el Subsistema Windows para Linux (WSL), que es una implementación de Linux para ser ejecutada dentro del sistema operativo Windows sin necesidad de aprovisionar una máquina virtual. Esto ofrece las ventajas de Linux desde la facilidad de uso del escritorio de Windows. Para ejecutar la aplicación en Windows durante el desarrollo se hizo uso del servidor de ventanas VcXsrv para Windows, el cual permite ejecutar gráficamente en Windows una aplicación que esté corriendo sobre WSL. La distribución de Linux específica utilizada sobre WSL fue Ubuntu 22.04 LTS y la versión de Python fue 3.10. Las versiones de las librerías de Python utilizadas se listan en el Anexo H, en el contenido del fichero requirements.txt.

Del mismo modo en que se hizo para el desarrollo del *firmware* del prototipo, la aplicación fue programada a través de Visual Studio Code, procediendo a crear una carpeta llamada “app” dentro del repositorio. La estructura de esta carpeta al culminar la programación se muestra en la Figura 45, la cual servirá para la explicación de las tres fases del desarrollo de la aplicación ya mencionadas.

Durante la primera fase se procedió con la creación de los diseños visuales de las vistas y su implementación en lenguaje KV, para lo cual se empleó la librería KivyMD, construida sobre Kivy y que permite emplear objetos con estilos basados en los estilos de *Material Design* de Google. Se utilizó la clase llamada ScreenManager para organizar las diferentes vistas lo cual está definido en el fichero main.kv. De igual manera, se creó un estilo de vista personalizado con una barra de herramientas en la parte superior, implementado en el fichero toolbarscreen.kv y una gaveta de navegación (NavDrawer) en la parte izquierda,

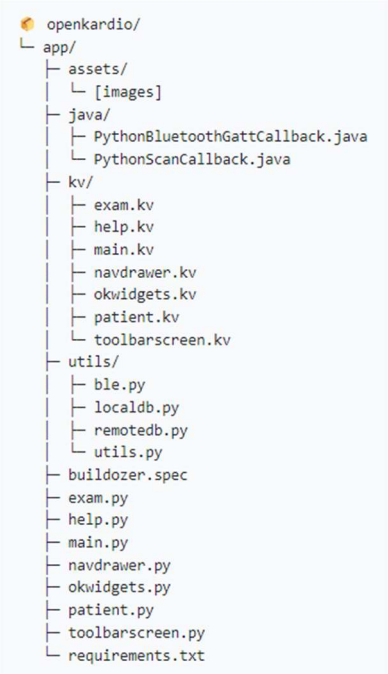
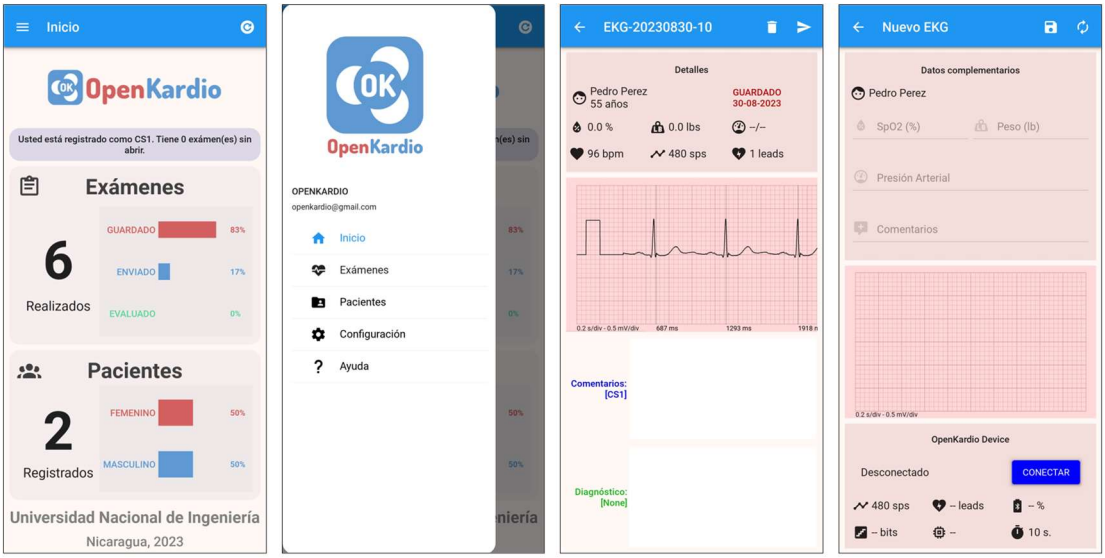



*Figura 45. Estructura del Código Fuente de la aplicación móvil.*

definido en navdrawer.kv. A partir de estos diseños base, se implementaron las vistas que fueron definidas en la sección 2.2.2. En la Figura 46(a) se muestra a modo de ejemplo la vista de Inicio, mientras que en la Figura 46(b) se muestra la gaveta de navegación que permite desplazarse entre diferentes vistas. En la carpeta “kv” (véase la Figura 45) se incluyen los elementos visuales que conforman las demás vistas. Por ejemplo, el fichero exam.kv incluye los componentes visuales que se utilizan en las vistas de detalles [véase la Figura 46(c)] y creación de exámenes [véase la Figura 46(d)]. En el Anexo D se muestran todas las vistas de la aplicación.

En esta primera fase también se implementó la base de datos local haciendo uso de SQLite, el cual es un motor de bases de datos basado en archivos de gran utilidad, ya que no requiere de servidor. La conexión hacia la base de datos es manejada a través de la librería SQLAlchemy y las clases que funcionan de interfaz con el modelo de datos se incluyeron el fichero localdb.py de la carpeta “utils”.




(a) (b) (c) (d)

*Figura 46Vistas implementadas en la aplicación móvil. Fuente: Autor.*

En la siguiente fase se continuó con la implementación de la funcionalidad de los elementos visuales desarrollados en la fase anterior del desarrollo de la aplicación móvil. Estas funcionalidades están contenidas en los ficheros fuente que se corresponden con sus contrapartes en lenguaje KV. Por ejemplo, exam.py define las mecánicas para crear, graficar y consultar exámenes en la base de datos para mostrarlos sobre los elementos de las vistas correspondientes, del mismo modo que patient.py realiza la misma funcionalidad para los pacientes en la base de datos; navdrawer.py y okwidgets.py definen las mecánicas de varios elementos comunes de la interfaz gráfica. El diseño visual de la aplicación, incluyendo los logotipos, se desarrolló en esta fase y se incluyeron las imágenes necesarias en la carpeta “assets”.

De igual manera, la integración con la base de datos de la nube se logró al crear una instancia de pruebas en la consola de Firebase (lo cual se aborda más adelante), y a continuación configurar el protocolo de seguridad de Google a través de la librería GoogleAuth para Python haciendo uso de una llave privada, llamada archivo de cuenta de servicio, en formato JSON. Esta llave privada permite solicitar un token a Firebase cada vez que se realice una petición, contra el cual la base de datos se asegura que la aplicación está autorizada a editar

datos. Esto es importante, ya que de este token depende que los datos sensibles de los usuarios viajen encriptados a través de internet. Las funciones para leer y escribir en la base de datos remota se definieron el fichero remotedb.py, al hacer uso de la librería requests para manejar la comunicación a través de peticiones HTTPS. Esta librería hace uso de la pila de protocolos de red disponibles en el sistema operativo, ya sea WiFi o celular. En ese sentido también se implementó la posibilidad de poder diagnosticar exámenes desde la aplicación móvil si se cambia la configuración adecuada, como un método alternativo a diagnosticar desde el portal web que se desarrolla el siguiente sección.

En este punto cabe destacar una limitante encontrada durante esta fase respecto a la comunicación con la base de datos remota, pues si bien es cierto que es posible utilizar librerías de Java en Python a través de PyJNIus, para adaptar las librerías de Firebase se encontró poca o nula documentación al respecto. La poca documentación encontrada en foros de desarrolladores estuvo sujeta a cambios disruptivos entre sucesivas versiones de Python y PyJNIus, por lo que se desistió de este enfoque y se optó por uno puramente implementado en Python, cambiando la estabilidad que ofrece Python por la velocidad de ejecución que habría ofrecido el uso de las librerías de Java.

Durante la tercera Fase se integró la comunicación con el prototipo mediante Bluetooth Low Energy, haciendo uso de la librería Bleak de Python. Se creó la clase BleHandler en el fichero ble.py, la cual implementa la máquina de estado finito definida en 2.2.2. Haciendo uso de la librería Bleak, esta clase escanea los dispositivos disponibles en busca de uno llamado “OKDevice”, una vez encontrado intenta conectarse al servicio definido por el UUID definido en la sección anterior. De igual manera, se encarga de leer la característica de control para conocer la información que ofrece el prototipo. Una vez que la interfaz gráfica recibe la orden de empezar la transmisión, la instancia de la clase BleHandler se suscribe al mecanismo de notificaciones de la característica de datos del prototipo, para en seguida enviar el comando 0x00 hacia la característica de control. Una vez que el

examen termina se escribe el comando de finalización correspondiente 0xFF en la característica de control del prototipo.

A fin de que la aplicación móvil sea capaz de recibir las muestras sin que otros componentes de código interfieran, fue necesario ejecutar este proceso de forma concurrente. En ese sentido, se optó por implementar el hilo principal de la aplicación y el manejo de la comunicación BLE de forma asíncrona a través de la librería asyncio, ya que esto resultó ser el método más sencillo, en comparación con implementar un nuevo hilo de procesamiento haciendo uso de las librerías nativas de Java en Android, debido a las limitantes del *framework* utilizado.

Asimismo, cabe destacar que la librería Bleak que utiliza clases de Java para manejar la comunicación presentó problemas para incorporar estas librerías al compilar la aplicación. Este inconveniente se resolvió al insertar en el código fuente de la aplicación las definiciones de clases Java necesarias, razón por la cual se creó la carpeta “java”.

Posteriormente, se continuó con la fase de post-procesamiento de la señal, a fin de poder identificar características de la señal y comprimirla para su envío y almacenamiento. En el caso de la identificación de características se optó por la detección de picos R de la señal electrocardiográfica, por ser uno de los principales indicadores que se pueden extraer para análisis de arritmias cardiacas y para detectar el ritmo cardiaco. En ese sentido, se realizaron pruebas con 3 algoritmos de detección de picos R: Algoritmo de Pan-Tompkins, Algoritmo de Hamilton, y Algoritmo de Christov. Las pruebas se hicieron a través de Jupyter Notebooks haciendo uso de la librería py-ecg-detectors. En la Figura 47 se muestra la comparación entre las capacidades de detección de los tres algoritmos.

De las tres opciones sometidas a comparación la que mejor se perfila es el algoritmo de Christov que logra identificar con bastante precisión la posición de los picos, indicados por puntos rojos, superando al algoritmo de Pan-Tompkins que es el segundo mejor y al algoritmo de Hamilton en tercero. Nótese que se empleó una señal sin filtrar con el fin de asegurar que el algoritmo seleccionado

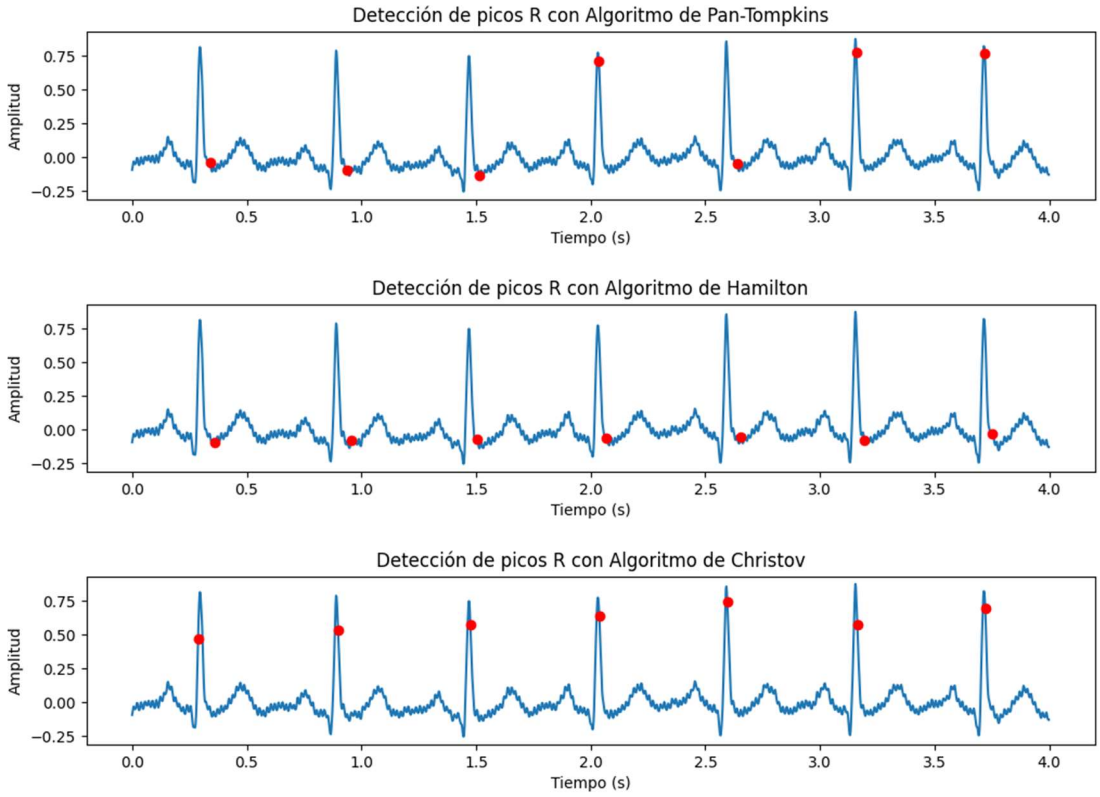



*Figura 47. Comparación entre algoritmos de detección de picos R.*

es robusto incluso en condiciones de ruido. La implementación de este algoritmo se incluyó dentro del fichero utils.py.

Por otro lado, para realizar la compresión de la señal se optó por un método de compresión sin pérdidas, debido a que la forma de onda de la señal debe ser preservada a fin de poder ser inspeccionada visualmente por el médico diagnosta. En ese sentido se seleccionó la librería zlib que implementa al algoritmo DEFLATE. Para realizar la compresión se convierte la señal de una lista de enteros a un arreglo de bytes, lo cual se incluyó dentro del fichero exam.py.

Por último, la compilación de la aplicación se realizó configurando el fichero buildozer.spec y ejecutando los comandos correspondientes de la herramienta Buildozer, indicados en su documentación [36]. El contenido del instalador APK se esquematiza en la Figura 48, donde es posible apreciar las librerías que interactúan con recursos del sistema operativo Android, así como los ficheros de

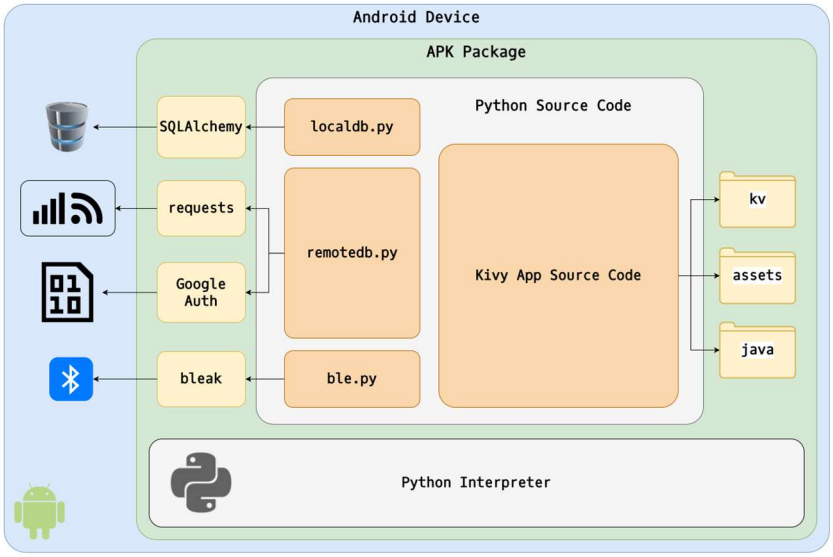



*Figura 48. Contenido del instalador de la App móvil y su interacción con los recursos de sistema operativo.*

apoyo a la aplicación, que sirven de interfaz con las base de datos local, la base de datos remota y la comunicación BLE.

### 2.3.3 Implementación de los servicios en Firebase

Esta etapa se incluye el aprovisionamiento de una base de datos no relacional, funciones sin servidor (*serverless*) y un alojamiento web, todo en la plataforma Firebase, para lo cual se creó una cuenta, que sirve para administrar todos estos servicios como parte de un mismo proyecto. De igual forma se creó una carpeta dentro del repositorio GitHub llamada “firebase” (véase la Figura 49).

El aprovisionamiento de la base de datos consistió en inicializar desde la consola de administración el servicio *Realtime Database*, con tres colecciones de archivos:

- “Cases”, contiene los casos de los pacientes, desde que son enviados por los centros de salud.
- “HCenters”, almacena datos de contacto e identificación de los centros de salud.
- “Hospitals”, contiene los datos de los Hospitales o clínicas de destino hacia los que se envían los casos.

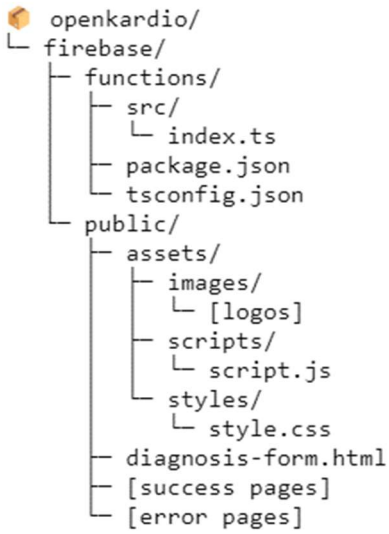



*Figura 49. Estructura del código fuente de las funciones serverless y el portal web de diagnóstico.*

Los datos que contiene cada documento en la colección “Cases”, es una versión aplanada (i.e. contrario a anidada) del modelo de datos que se implementa en la base de datos local de la aplicación móvil. Esto se hace para fines de escalabilidad, tal como lo sugiere Firebase en su documentación [37].

Para garantizar la seguridad de la escritura y lectura de la base de datos, se implementó una regla de acceso para asegurar que todas las peticiones HTTPS lleguen debidamente acompañadas por el token de acceso. De igual forma, esta regla aplica índices de búsqueda en la colección de casos para mejorar el rendimiento de las búsquedas (véase la Figura 50).

Por otro lado, las Funciones Lambda o *serverless* fueron programadas desde el entorno de desarrollo Visual Studio Code al igual que los otros dos componentes del proyecto. El lenguaje seleccionado fue Typescript, ya que por su popularidad

```
{
  "rules": {
    ".read": "auth != null && auth.token.firebase != null",
    ".write": "auth != null && auth.token.firebase != null",
    "Cases": {
      ".indexOn": ["destination_id","origin_id"]
    }
  }
}
```

*Figura 50. Reglas de la base de datos en la nube de Firebase.*

cuanta con amplio soporte y documentación. Al usar Typescript, *Functions* de Firebase permite emplear node.js como plataforma para ejecutar el código.

A continuación, se detalla el enfoque usado para lograr la implementación de los flujogramas establecidos en la sección 2.2.3.

- Lambda de Notificaciones: es disparada cada vez que se inserta un nuevo caso en la base de datos. Decodifica el objeto insertado para extraer de él la información del paciente y del examen, incluyendo la descompresión de la señal. A continuación, se formatea el cuerpo de un correo en HTML, que incluye la información ya decodificada y de igual forma se crea una imagen con la información de la señal, para que sea esta la representación que evalúa el médico diagnosta. Por último, usando la librería nodemailer se envía un *email* al hospital o clínica de destino acompañado de un enlace para realizar el diagnóstico, obteniendo la dirección de correo del destinatario en la base de datos.
- Lambda de recibo de diagnósticos: esta función se dispara cuando el URL asignado por Firebase recibe una petición HTTP del *frontend* con el contenido del diagnóstico. De esta manera la función actualiza el caso correspondiente en la base de datos y envía una notificación vía *email* al centro de salud de origen.
- Lambda de validación de exámenes: esta función se dispara cuando el *frontend* de verificar a través de una petición HTTP si el examen que se intenta diagnosticar existe y no ha sido diagnosticado todavía en la base de datos. De no ser así le hace saber al *frontend* para que este a su vez le muestre el mensaje de error correspondiente.

El código fuente de estas funciones fue implementado en un único fichero llamado index.ts dentro una subcarpeta del repositorio llamada “functions”, la cual también incluye las dependencias requeridas para ejecutar las funciones.

Por su parte, el portal web para diagnosticar exámenes se implementó dentro de la subcarpeta “public”. En esta se ubican los estilos, las imágenes y los scripts del portal, dentro de “assets” y las siguientes páginas HTML:

- diagnosis-form.html, es el formulario principal de diagnóstico.
- submission-success.html, muestra un mensaje de envío exitoso.
- already-diagnosed.html, submission-failure.html y no-exam-selected.html muestran mensajes de errores.

En el Anexo E se han incluido las capturas de pantalla de estas páginas renderizadas en el navegador.

El enlace que redirige al formulario correspondiente a un examen tiene el siguiente formato https://<openkardio_address>/diagnosis-form.html?examId=<caseId> donde se incluye como parámetro de *query* al final del URL, el identificador del caso que se desea diagnosticar.

El despliegue tanto de las funciones lambda como del portal web de diagnóstico, se realizó a través de la herramienta para línea de comando llamado firebase-tools. Una vez hecho el despliegue, se procedió con la realización de pruebas de funcionalidad. En el Anexo A se muestran capturas de pantallas de los resultados obtenidos durante las pruebas.

# 3 CAPITULO III: PRESENTACIÓN DE RESULTADOS

El presente trabajo monográfico produjo como resultado un sistema de Tele-cardiografía digital que consta de tres componentes, según lo propuesto en los Objetivos Específicos. En este capítulo se procede a presentar las pruebas de desempeño del prototipo, así como la validación del uso de la aplicación móvil y la interfaz web realizada. Del mismo modo, se evalúa la señal obtenida de la derivación periférica II, en aras de caracterizar el sistema.

## 3.1 Métricas de Desempeño

Una vez construido el prototipo, mostrado en la Figura 44, se procedió a realizar la conexión con la aplicación móvil y capturar electrocardiogramas para medir algunas características de desempeño como los tiempos de ejecución de las subrutinas del *firmware* y la mejora de la razón señal a ruido (SNR) proporcionada por el filtro digital.

En primer lugar, se realizó la medición de los tiempos de las subrutinas de conversión, filtrado y envío de tramas. Para estas mediciones se realizó una prueba de toma de electrocardiograma por cada intervalo a medir, haciendo uso de la función micros() dentro del código del *firmware*. En la Tabla V se resume el resultado de estas mediciones en microsegundos.

*Tabla X – Medición de tiempos de las subrutinas del firmware.*

| Parámetros | Conversión ADC | Filtrado | Envío de Trama |
|---|---|---|---|
| Muestra (N) | 4800 | 4800 | 200 |
| Mínimo | 824 µs | 5 µs | 607 µs |
| Máximo | 1,001 µs | 19 µs | 974 µs |
| Promedio | 844 µs | 5.9 µs | 871.2 µs |
| Desviación Estándar | 26.1 µs | 0.86 µs | 31.6 µs |

Esta tabla describe la variabilidad de la ejecución de estas subrutinas, la cual depende de varios factores como por ejemplo inicialización de variables al inicio de la ejecución, así como variabilidad en el medio electromagnético en el caso del envío de tramas a través de Bluetooth Low Energy. Se puede apreciar que, en el peor de los casos, la suma de los tiempos máximos de las 3 subrutinas tomaría ejecutarse 1.994 ms lo cual está por debajo de los 2.08 ms de intervalo entre cada muestra para cumplir con 480 muestras por segundo. Sin embargo, en promedio la ejecución de las 3 subrutinas toma 1.72 ms. En la Figura 51 se muestra esquemáticamente los tiempos de ejecución ya mencionados.

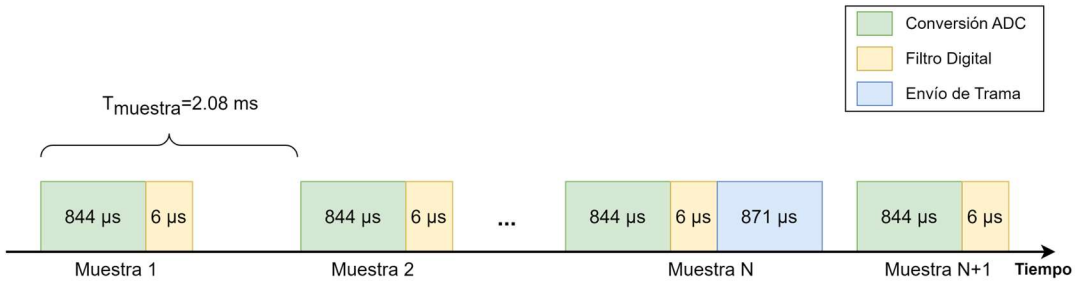



*Figura 51. Esquema de tiempos de ejecución de las subrutinas del firmware (No a escala).*

Asimismo, se realizó la estimación de la SNR cuando la señal se captura sin filtro digital y se la comparó con la SNR de la señal filtrada a fin de encontrar el incremento brindado por el filtro digital. En la Figura 25 se mostró el espectro de la señal electrocardiográfica sin filtrar. Si se toma la potencia del espectro desde 0 Hz a 40 Hz como la potencia de la señal y el resto del espectro como el ruido, se obtiene una SBR de 16.1 dB. Por otro lado, la Figura 52 se observa el espectro de la señal filtrada por el prototipo, donde se observa que las señales por encima de 40 Hz fueron filtradas. Para esta señal, la SNR se calculó de igual forma utilizando un script en jupyter notebooks (véase el Anexo F) obteniendo 23.5 dB de SNR. En consecuencia, el incremento de la SNR está dado por la Ecuación (11).

$$SNR_{DIF} = SNR_F - SNR_O \qquad (11)$$

$$SNR_{DIF} = 23.5\ dB - 16.1\ dB = 7.4\ dB$$

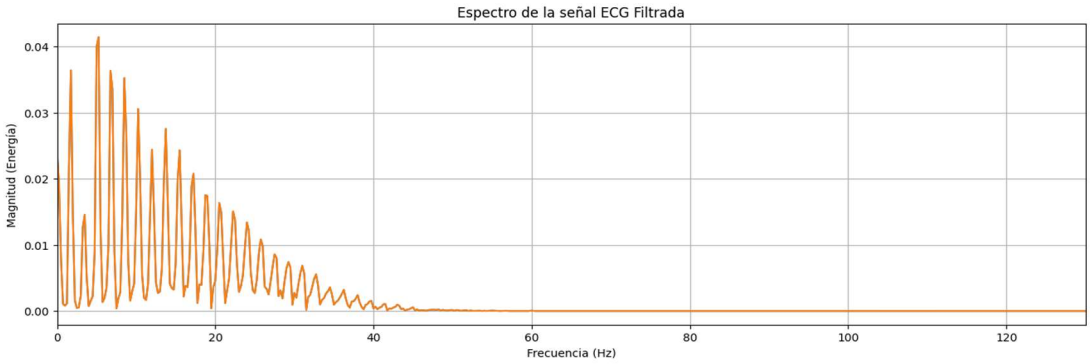



*Figura 52. Espectro del electrocardiograma filtrado en el prototipo.*

Por su parte la compresión de la señal, que se realiza en la aplicación móvil, es otro factor para tener en cuenta para transferencia y almacenamiento. Para calcular la razón de compresión alcanzada en el sistema basta con conocer el tamaño original de la señal y el tamaño luego de comprimirla. En este caso, por definición el tamaño en bytes de la señal viene dado por la Ecuación (12):

$$T_S = (N_D) * (T_M) * (F_S) * (D_S) \quad (12)$$

$$T_S = (1) * (2 \text{ bytes/muestra}) * (480 \text{ sps}) * (10 \text{ s})$$

$$T_S = 9600 \text{ bytes}$$

Donde:

$T_S$: Tamaño de la señal en bytes
$N_D$: Número de derivaciones
$T_M$: Tamaño de una muestra en bytes
$F_S$: Tasa de muestreo
$D_S$: Duración de la señal en segundos.

Por su parte, el tamaño de la señal comprimida promedia los 6,709 bytes. Por tanto, la razón de compresión alcanzada está dada por la Ecuación (13).

$$C_R = T_O / T_C$$

$$C_R = 9{,}600 \text{ bytes} / 6{,}709 \text{ bytes}$$

$C_R = 1.43$

Este factor de compresión significa una disminución de un 30% en el espacio que ocupa la señal comprimida en comparación con la señal original.

Otro factor importante para garantizar que el sistema pueda ser utilizado en un ambiente rural es su comportamiento en condiciones de red adversas. Por ese motivo, se midió el desempeño de la aplicación para comunicarse con la base de datos en condiciones de ancho de banda limitado. Para ese fin se configuró una red Wi-Fi en un *router* Mikrotik RB951 con conexión a Internet usando diferentes anchos de bandas. Se utilizó un teléfono celular Xiaomi Note 12 Pro conectado a red Wifi para realizar las pruebas. En la Figura 53 se muestra un ejemplo de la Cola Simple (Simple Queue) aplicada al *router* para introducir un límite de ancho de banda, mientras que en la Figura 54 se muestran los resultados.

Se consideraron anchos de banda de 256 kbps, 128 kbps y 64 kbps los cuales son valores típicos en redes EDGE y UMTS, cuya cobertura abarca amplias zonas rurales del Nicaragua. A como es de esperarse a mayor ancho de banda, menor es el tiempo de transmisión de cada examen con la base de datos. Si bien, en el caso de un ancho de banda de 64 kbps el envío se toma aproximadamente 19 segundos, la comunicación termina de forma satisfactoria.

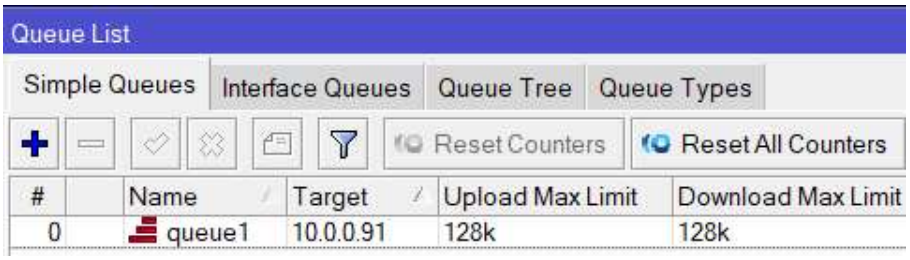



*Figura 53. Configuración de ancho de banda máximo en router marca Mikrotik.*

## 3.2 Comparación con Electrocardiograma Clínico

Con el propósito de verificar la validez de la forma de onda obtenida a través del prototipo, se comparó un electrocardiograma generado por este contra un electrocardiograma de grado clínico (véase el Anexo A) el cual fue realizado al autor. Cabe mencionar que para la toma de ambos exámenes fue necesario

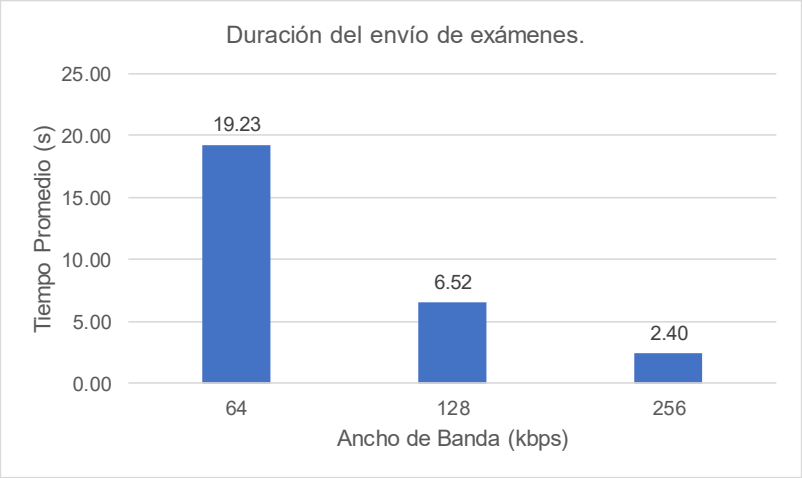
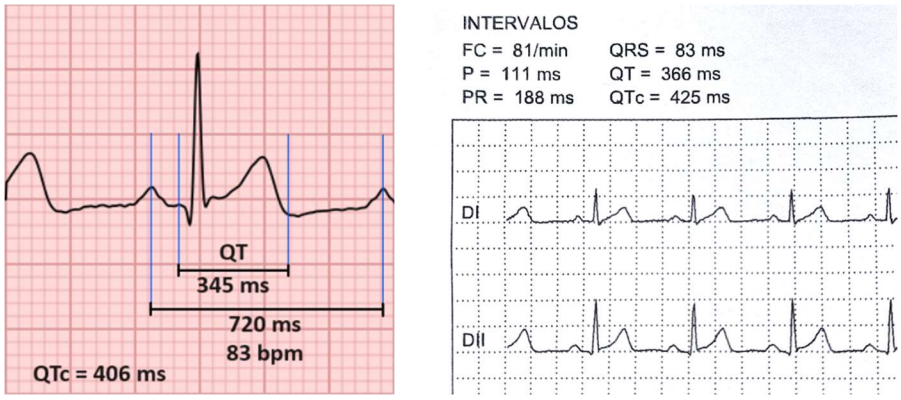



*Figura 54. Gráfica de duración de envíos de exámenes.*

mantenerse en reposo y en una postura inmóvil para evitar la interferencia de las señales eléctricas de otros grupos musculares del cuerpo, pues si bien es cierto que una de las terminales tiene como propósito reducir el ruido de modo común que pueda presentarse, la amplitud de estas señales ruidosas llega a tener una amplitud considerable.

Al evaluar visualmente las forma de onda de la derivación II[2], mostrada en la Figura 55(a), que también es la derivación capturada por el prototipo, mostrada en la Figura 55(b) es posible constatar la correspondencia morfológica en los segmentos PQ, el complejo QRS y la onda T. Del mismo modo, al medir la




(a) (b)

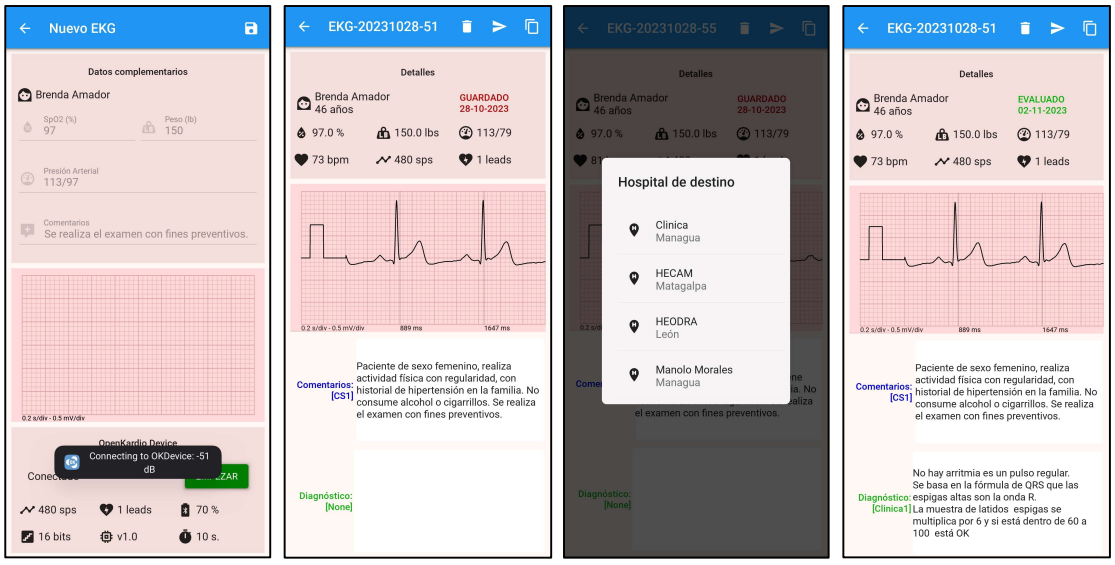
*Figura 55. Comparación del EKG capturado contra un EKG de grado clínico.*

[2] El electrocardiograma clínico completo está incluido en el Anexo A.

duración del segmento QTc, que es indiferente a la frecuencia cardiaca, se obtiene un valor de 406 ms al aplicar la fórmula descrita en la Tabla I, que coincide con el valor de 425 ms reportado el examen clínico realizado dentro de un margen de 3% de diferencia.

## 3.3 Validación del Funcionamiento

Para la validación del funcionamiento general del sistema, se realizó un electrocardiograma a un sujeto de pruebas, localizado en la ciudad de Managua, para posteriormente enviar el examen a un médico para su evaluación. La Figura 56(a) muestra la vista de creación de exámenes, justo luego de establecer la comunicación con el prototipo, donde ya se ha seleccionado al paciente registrado previamente en la aplicación para realizar el examen y también se ha llenado la información médica adicional; nótese que también se muestra la información del estado del prototipo en la sección inferior. La Figura 56(b) corresponde al electrocardiograma una vez que ha sido capturado y guardado. Para el envío del examen es posible seleccionar el centro asistencial de destino, como lo muestra la Figura 56(c), tras lo cual, el examen queda en estado “ENVIADO” a la espera de ser diagnosticado.




(a) (b) (c) (d)

*Figura 56. Vistas de la toma de un electrocardiograma a sujeto de pruebas desde la aplicación móvil.*

Posteriormente, se envió el electrocardiograma al Dr. José René Amador, quien reside en la ciudad de Managua. El examen fue recibido por correo a través del servicio de notificaciones alojado en Firebase, donde el médico logró revisar los datos básicos del sujeto de pruebas, datos de salud adicionales como el peso y la presión arterial, así como interpretar la señal electrocardiográfica. Su diagnóstico fue ingresado en la interfaz web y recibido de regreso en la aplicación móvil, completando el proceso de forma satisfactoria. El resultado se observa en la Figura 56(d).

De esta manera, se culminó la fase de validación funcional del sistema habiendo demostrado exitosamente el funcionamiento propuesto en este proyecto monográfico.

## 3.4 Análisis de costos

El prototipo tuvo un costo de USD 114.45[3], sin incluir la mano de obra, siendo la principal fuente de costos el módulo AD8232, la tarjeta de desarrollo ESP32 y algunos otros componentes que no se encuentran en el mercado local, por lo que tuvieron que ser importados. Por otro lado, los componentes discretos y los materiales para fabricar la PCB fueron adquiridos en el mercado local.

Dado que se utilizó en gran medida software y librerías de código abierto para la realización de este proyecto, no se incurrió en ningún gasto de licenciamiento. En el caso del alojamiento en la nube, la plataforma Firebase ofrece una capa gratuita que permitiría alojar hasta 100,000 exámenes de manera gratuita. Una vez sobrepasado ese límite, el alojamiento tendría un costo de $5 por cada 100,000 exámenes considerando que cada examen tiene un tamaño menor a 10 kB.

La Tabla XI muestra una comparación del costo de este prototipo con otros equipos ofrecidos por empresas extranjeras que tienen un propósito similar, que consiste en ofrecer un equipo de electrocardiografía con alguna capacidad de compartir remotamente el resultado del examen con un médico. Sin embargo,

[3] Véase el Anexo C para un detalle completo de los costos

cabe resaltar que para hacer uso de esas funciones se debe pagar una suscripción y permitir que los datos personales y médicos del usuario sean alojados en servidores de terceros sin una garantía de que estos sean debidamente encriptados y custodiados.

En ese particular es posible afirmar que se logró un prototipo costo efectivo dentro del rango de precios de productos con capacidades similares en el mercado internacional, desde el punto de vista del dispositivo electrónico y también desde el punto de vista de la solución para el envío y almacenamiento de exámenes en la nube.

*Tabla XI – Comparación de precios con equipos disponibles en el mercado internacional.*

| **Equipo** | **Costo USD** |
|---|---|
| Prototipo OpenKardio | 114.45 |
| KardiaMobile | 79.00 + 9.99 mensual |
| Omron BP7900 | 139.98 + 9.99 mensual |

# 4 CAPÍTULO IV: CONCLUSIONES Y RECOMENDACIONES

## 4.1 Conclusiones

El presente trabajo monográfico tuvo como resultado un prototipo de electrocardiógrafo de una derivación con capacidad de comunicación inalámbrica con dispositivos móviles Android, al igual que una aplicación móvil capaz de almacenar y transmitir los electrocardiogramas capturados por el prototipo. Del mismo modo se configuró la plataforma Firebase para recibir y almacenar electrocardiogramas, así como recibir el diagnóstico a través de una interfaz web.

En el capítulo anterior se realizaron pruebas de validación del funcionamiento del sistema, cuyos resultados demostraron satisfactoriamente que este prototipo podría ser aplicado en escenarios rurales para la práctica de la Telemedicina, específicamente como ayuda al diagnóstico de condiciones cardiacas de forma remota.

Con base en los objetivos específicos que fueron propuestos y en los resultados obtenidos, se enuncian las siguientes conclusiones:

- Se demostró la viabilidad de construir un prototipo electrónico con capacidad de adquisición de electrocardiogramas de una derivación de manera costo-efectiva a través del uso de un Módulo *frontend* analógico AD8232, una Conversor Analógico Digital de tipo Sigma-Delta ADS1115 y una tarjeta de desarrollo ESP32.
- Se logró programar exitosamente el prototipo para la captura de electrocardiogramas a una tasa de 480 muestras por segundo, aplicando un filtro digital pasa-bajas obteniendo una mejora sensible en la Razón Señal a Ruido de la señal. Asimismo, se logró la transmisión de grupos de muestras de la señal filtrada a través del protocolo de comunicación Bluetooth Low Energy.
- Se desarrolló una aplicación móvil para sistema operativo Android con capacidad de utilizar las interfaces inalámbricas de un dispositivo móvil

para comunicarse con el prototipo a través de Bluetooth Low Energy, y para enviar los exámenes y recibir diagnósticos a través de Wi-Fi o redes celulares disponibles, tomando un tiempo razonable en la transmisión, lo cual permitiría su uso en zonas rurales con cobertura celular. Del mismo modo, la aplicación permite visualizar la señal, detectar el ritmo cardiaco y comprimir los electrocardiogramas.

- Se configuró la plataforma Firebase como alojamiento en la nube para los exámenes, permitiendo la posibilidad enviar notificaciones a través de Email y la recepción del diagnóstico de estos a través una interfaz web alojada en la misma plataforma.

## 4.2 Recomendaciones

El sistema que se ha desarrollado en este trabajo monográfico satisface los objetivos propuestos, dado que se logró obtener una señal electrocardiográfica, procesarla y transmitirla a un servicio en la nube, demostrando de esta manera la viabilidad de emplear un prototipo con las características aquí descritas para servir como herramienta en la práctica de la Telemedicina en Nicaragua.

No obstante, cabe mencionar algunas recomendaciones y oportunidades de mejora para una futura iteración de este proyecto:

- Aumentar el número de derivaciones al emplear un frontend analógico de un mayor número de canales, con el fin de incrementar la capacidad de diagnóstico a los médicos.
- Considerar el uso de electrodos alternativos a los electrodos de gel, como los electrodos de abrazadera y electrodos de succión, los cuales son reutilizables y tienen una vida útil mayor, con el fin de minimizar el costo de tomar cada electrocardiograma.
- El *framework* Kivy aún no está lo suficientemente maduro para el desarrollo de aplicaciones que demandan interacción intensiva con los periféricos de comunicación y el sistema operativo de un dispositivo móvil, por lo tanto, es recomendable emplear lenguajes más convencionales como Kotlin o Java para este tipo de aplicaciones.
- Se implementó una funcionalidad para detectar los picos R de la señal electrocardiográfica, a partir de los cuales fue posible calcular la frecuencia cardiaca, sin embargo, considerando que las señales se almacenan en un base de datos, se podría utilizar como fuente de datos para alimentar modelos de detección automática de afecciones cardiacas, basados en algoritmo de aprendizaje automático.
- Considerar la fabricación de la PCB a través de una empresa especializada, pues los precios de este servicio se han hecho bastante accesible incluso para pocas unidades.

# BIBLIOGRAFÍA


[1] B. Fong, A. C. M. Fong y C. K. Li, Telemedicine Technologies, West Sussex: John Wiley & Sons, 2011.

[2] A. Martínez, V. Villaroel, J. Seoane y F. del Pozo, «Analysis of Information and Communication Needs in Rural Primary Health Care in Developing Countries,» *IEEE TRANSACTIONS ON INFORMATION TECHNOLOGY IN BIOMEDICINE,* vol. 9, nº 1, pp. 66-72, 2005.

[3] Ministerio de Salud, «Mapa Nacional de la Salud en Nicaragua,» [En línea]. Available: http://mapasalud.minsa.gob.ni/mapa-de-padecimientos-de-salud-de-nicaragua/. [Último acceso: 6 Junio 2022].

[4] Ministerio de Energía y Minas; ENATREL, «Programa Nacional de Electrificación Sostenible y Energía Renovable (PNESER),» Managua, 2020.

[5] R. Gupta, M. Mitra y B. Jitendranath, ECG Acquisition and Automated Remote Processing, New Delhi: Springer India, 2014.

[6] Center for Connected Health Policy, «What is telehealth?,» [En línea]. Available: https://www.cchpca.org/what-is-telehealth/. [Último acceso: 05 09 2023].

[7] A. Feather et al., Clinical Medicine, Londres: Elsevier, 2021.

[8] Springer, Handbook of Cardiac Anatomy, Physiology, and Devices, P. A. Iaizzo, Ed., Nueva York: Springer Sciences+Business Media, 2009, pp. 3-12.

[9] A. M. Katz, Physiology of the Heart, Philadelphia: LIPPINCOTT WILLIAMS & WILKINS, 2011.

[10] J. Wasilewski y L. Polonski, «An Introduction to ECG Interpretation,» de *ECG Signal Processing, Classification and Interpretation*, Londres, Springer-Verlag, 2012, pp. 1-20.

[11] L. S. Lilly, Pathophysiology of Heart Disease, Hong Kong: Lippincott Williams & Wilkins, 2011.

[12] J. R. Hampton, The ECG Made Easy, ELSEVIER, 2013.

[13] A. Dupre, S. Vieau y P. A. Iaizzo, «Basic ECG Theory, 12-Lead Recordings and Their Interpretation,» de *Handbook of Cardiac Anatomy, Physiology, and Devices*, Springer Sciences+Business Media, 2009, pp. 257-269.

[14] K. Najarian y R. Splinter, Biomedical Signal and Image Processing, CRC Press, 2012.

[15] X.-H. Li y F. Lu, «Catheter Ablation of Cardiac Arrhythmias,» de *Handbook of Cardiac Anatomy, Physiology, and Devices*, Springer Sciences+Business Media, 2009, pp. 411-441.

[16] A. Naït-Ali y P. Karasinski, «Biosignals: Acquisition and General Properties,» de *Advanced Biosignal Processing*, Berlin, Springer-Verlag, 2009, pp. 1-14.

[17] M. R. Neuman, «Biopotential Electrodes,» de *Medical Instrumentation. Applications and Design.*, Wiley, 2010, pp. 189-240.

[18] A. Gacek, «An Introduction to ECG Signal Processing and Analysis,» de *ECG Signal Processing, Classification and Interpretation*, Londres, Springer-Verlag, 2012, pp. 21-46.

[19] J. H. Nagel, «Biopotential Amplifiers,» de *Medical Instruments and Devices*, CRC Press, 2016, pp. 1-16.

[20] L. Sörnmo y P. Laguna, BIOELECTRICAL SIGNAL PROCESSING IN CARDIAC AND NEUROLOGICAL APPLICATIONS, Elsevier, 2005.

[21] N. S. Widmer, G. L. Moss y R. J. Rocci, Digital Systems, Nueva Jersey: Pearson, 2017.

[22] G. Blanchet y M. Charbit, Digital Signal and image Processing Using MATLAB., Londres: ISTE, 2006.

[23] G. D. Clifford, «ECG Statistics, Noise, Artifacts, and Missing Data,» de *Advanced Methods and Tools for ECG Data Analysis*, Norwood, Artech House, 2006, pp. 55-93.

[24] J. Pan y W. J. Tompkins, «A Real-Time QRS Detection Algorithm,» *IEEE Transactions on Biomedical Engineering,* vol. 32, nº 3, p. 230–236, 1985.

[25] P. S. Hamilton, «Open source ECG analysis,» de *Proc. Computers in Cardiology*, Memphis, 2002.

[26] I. I. Christov, «Real time electrocardiogram QRS detection using combined adaptive threshold,» *BioMedical Engineering OnLine,* vol. 3, nº 28, 2004.

[27] D. Salomon, Data Compression, Londres: Springer, 2007.

[28] Amazon Web Services, «What is Cloud Computing,» Amazon Web Services, [En línea]. Available: https://aws.amazon.com/what-is-cloud-computing/. [Último acceso: 15 02 2022].

[29] W. Yu et al, «A Survey on the Edge Computing for the Internet of Things,» *IEEE Access,* vol. 6, pp. 6900-6919, 2018.

[30] N. Gupta, Inside Bluetooth Low Energy, Boston: Artech House, 2016.

[31] E. Wright et al, Practical Telecommunications and Wireless Communications, Burlington: Elsevier, 2004.

[32] B. Fong et al., Telemedicine Technologies, Chichester: John Wiley & Sons, 2011.

[33] I. Tsai, «Readying Medical Applications for Telehealth,» de *Telemedicne and Electronics Medicine*, CRC Press, 2016, pp. 33-50.

[34] K. Townsend, C. Cufí, A. Davidson y R. Davidson, Getting Started with Bluettoth Low Energy, O'Reilly, 2014.

[35] MicroModeler, «MicroModeler DSP,» [En línea]. Available: https://www.micromodeler.com/dsp/. [Último acceso: 10 October 2023].

[36] Kivy's Developers, «Quickstart - Buildozer 0.11 documentation,» [En línea]. Available: https://buildozer.readthedocs.io/en/latest/quickstart.html. [Último acceso: 10 October 2023].

[37] Firebase, «Structure Your Database | Firebase Realtime Database,» Google, [En línea]. Available: https://firebase.google.com/docs/database/rest/structure-data. [Último acceso: 10 October 2023].

[38] C. Beard y W. Stallings, Wireless Communication Networks and Systems, Harlow: Pearson, 2016.

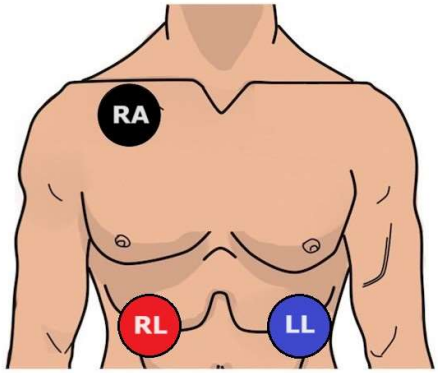
[39] Texas Instruments, *ADS111x Ultra-Small, Low-Power, I2C-Compatible, 860-SPS, 16-Bit ADCs,* 2018.

[40] Bluetooth Interest Group, Specification of the Bluetooth System (version 4.2), 2014.

# ANEXOS

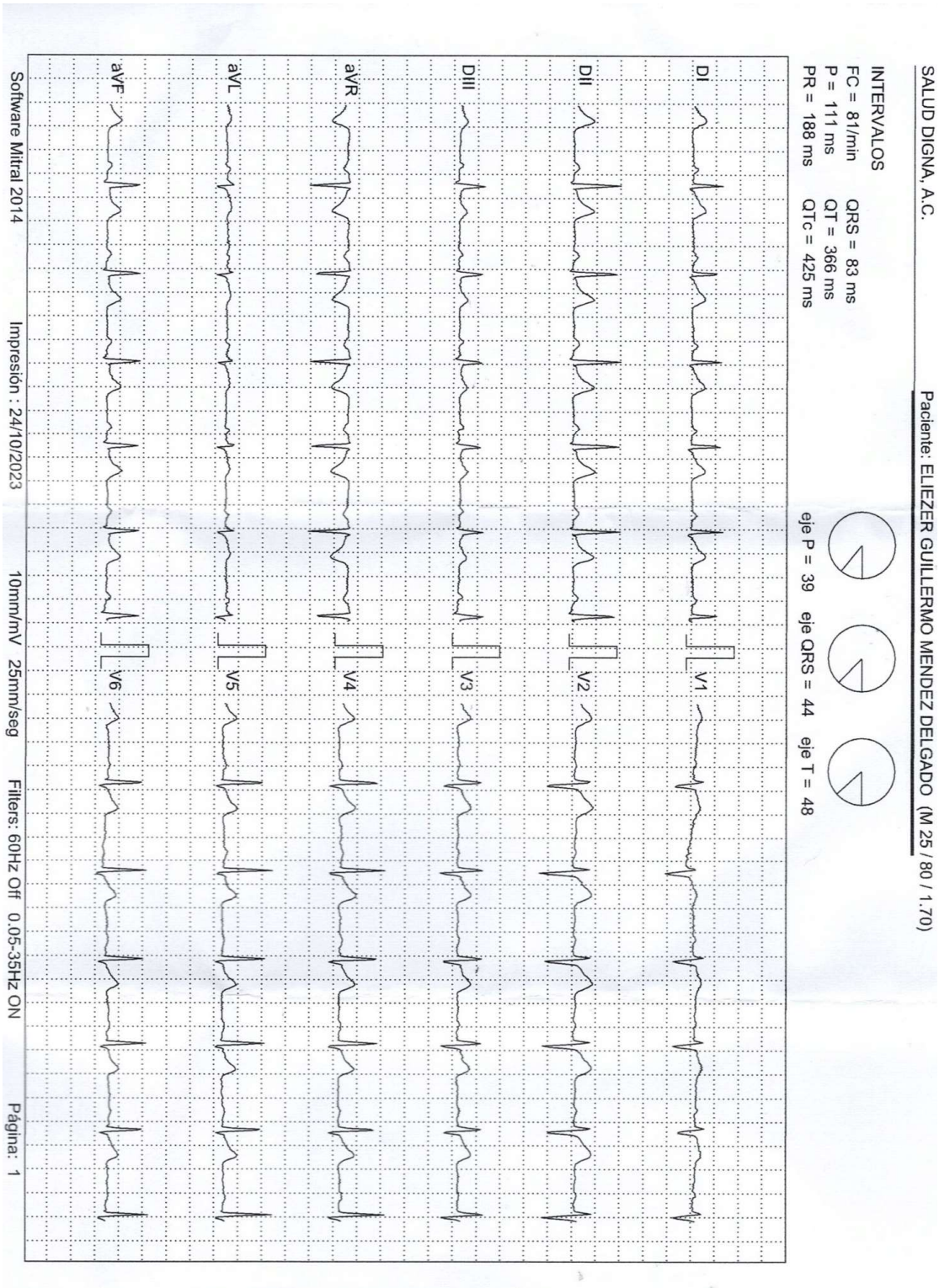
## Anexo A. Pruebas Realizadas

A1. Cables de terminales y electrodos utilizados en la realización de pruebas.



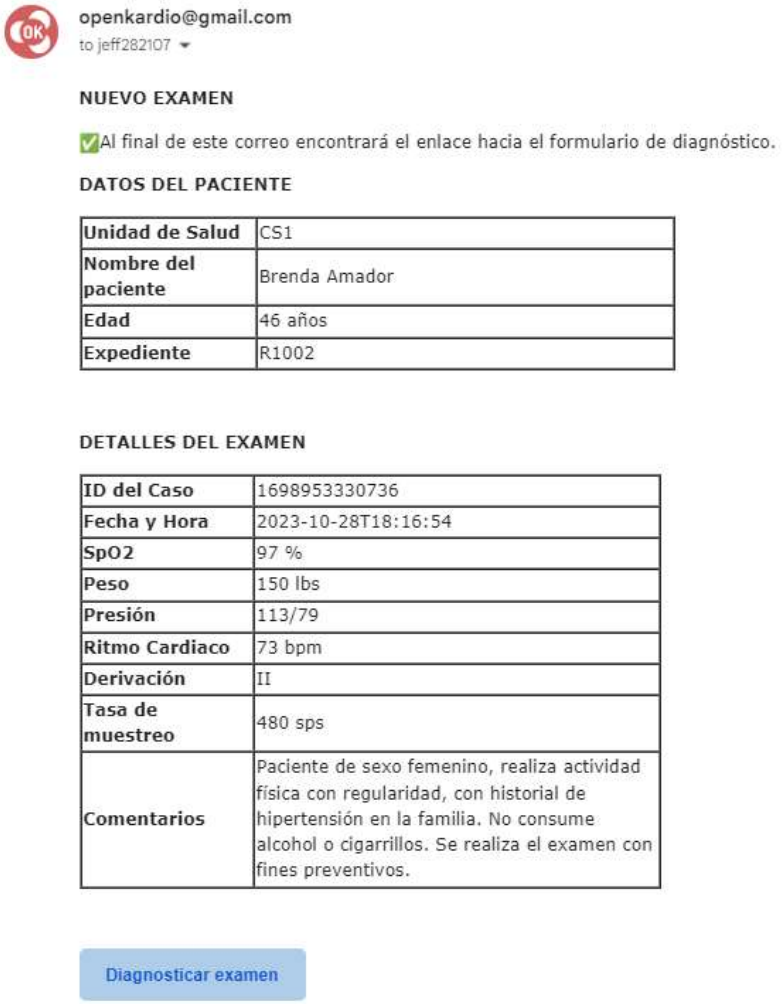


A3. Ejemplo de Electrocardiograma (derivación II) capturado por el prototipo.




A4. Electrocardiograma de 12 derivaciones realizado en una clínica.

A5. Fotografía de la realización de un examen.



openkardio@gmail.com
to jeff282107

**NUEVO EXAMEN**

✅Al final de este correo encontrará el enlace hacia el formulario de diagnóstico.

**DATOS DEL PACIENTE**

| | |
|---|---|
| **Unidad de Salud** | CS1 |
| **Nombre del paciente** | Brenda Amador |
| **Edad** | 46 años |
| **Expediente** | R1002 |

**DETALLES DEL EXAMEN**

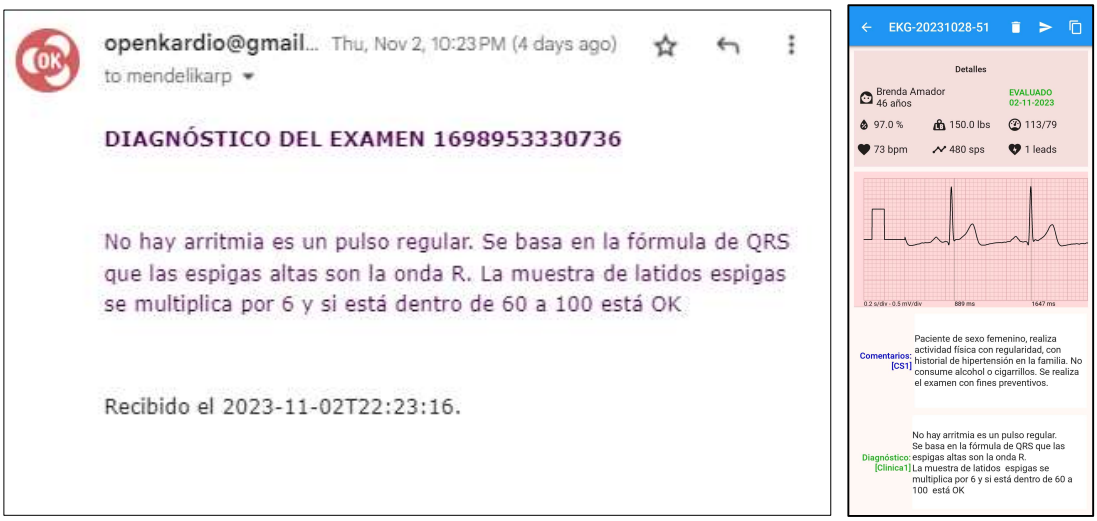
| | |
|---|---|
| **ID del Caso** | 1698953330736 |
| **Fecha y Hora** | 2023-10-28T18:16:54 |
| **SpO2** | 97 % |
| **Peso** | 150 lbs |
| **Presión** | 113/79 |
| **Ritmo Cardiaco** | 73 bpm |
| **Derivación** | II |
| **Tasa de muestreo** | 480 sps |
| **Comentarios** | Paciente de sexo femenino, realiza actividad física con regularidad, con historial de hipertensión en la familia. No consume alcohol o cigarrillos. Se realiza el examen con fines preventivos. |

Diagnosticar examen

A6. Correo electrónico enviado para su diagnóstico por el Dr. José René Amador.

```
2023-11-02 13:28:53.737  sendNotificationOnCreation  8ad0sqjjecij  Function execution started
2023-11-02 13:28:53.846  sendNotificationOnCreation  8ad0sqjjecij  New case 1698953330736
2023-11-02 13:28:53.848  sendNotificationOnCreation  8ad0sqjjecij  Compressed bytes length: 7509
2023-11-02 13:28:53.848  sendNotificationOnCreation  8ad0sqjjecij  Peaks: [
2023-11-02 13:28:53.848  sendNotificationOnCreation  8ad0sqjjecij    138,  235,  599,  992,
2023-11-02 13:28:53.848  sendNotificationOnCreation  8ad0sqjjecij   1397, 1805, 2206, 2608,
2023-11-02 13:28:53.848  sendNotificationOnCreation  8ad0sqjjecij   3013, 3414, 3818, 4204,
2023-11-02 13:28:53.848  sendNotificationOnCreation  8ad0sqjjecij   4590
2023-11-02 13:28:53.848  sendNotificationOnCreation  8ad0sqjjecij  ]
2023-11-02 13:28:55.782  sendNotificationOnCreation  8ad0sqjjecij  Email sent successfully
```

A7. Registro de la función de envío de notificación al usuario destinatario (médico).



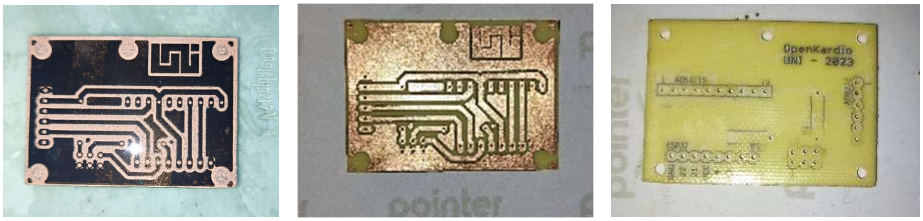

(a) (b)

A8. Recepción del diagnóstico a través de notificación por (a) correo electrónico y en (b) la aplicación móvil.

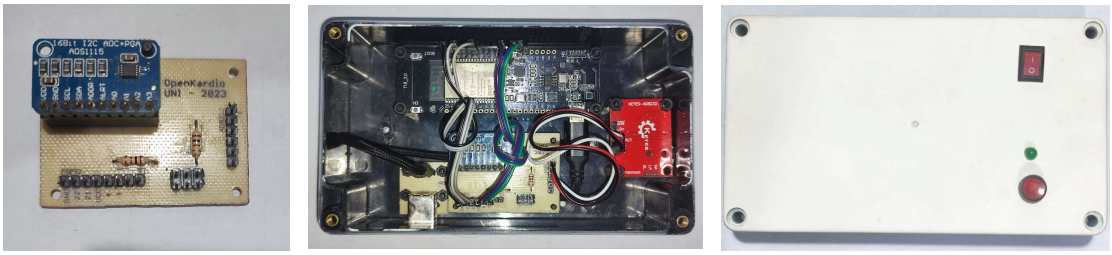
| Ancho de Banda (kbps) | Tiempo (segundos) |
|---|---|
| 64 | 19.05949671 |
| 64 | 19.11952307 |
| 64 | 19.51684812 |
| 128 | 6.631819322 |
| 128 | 6.005819321 |
| 128 | 6.929484424 |
| 256 | 2.408006822 |
| 256 | 2.402856457 |
| 256 | 2.377956875 |

A10. Mediciones de tiempos de envío de exámenes

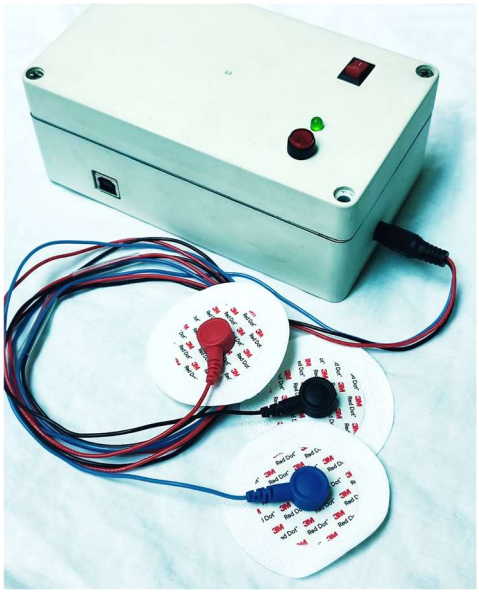
## Anexo B. Proceso de construcción del prototipo.




B1. Proceso de fabricación de la PCB.




B2. Ensamblaje de los componentes en la carcasa.

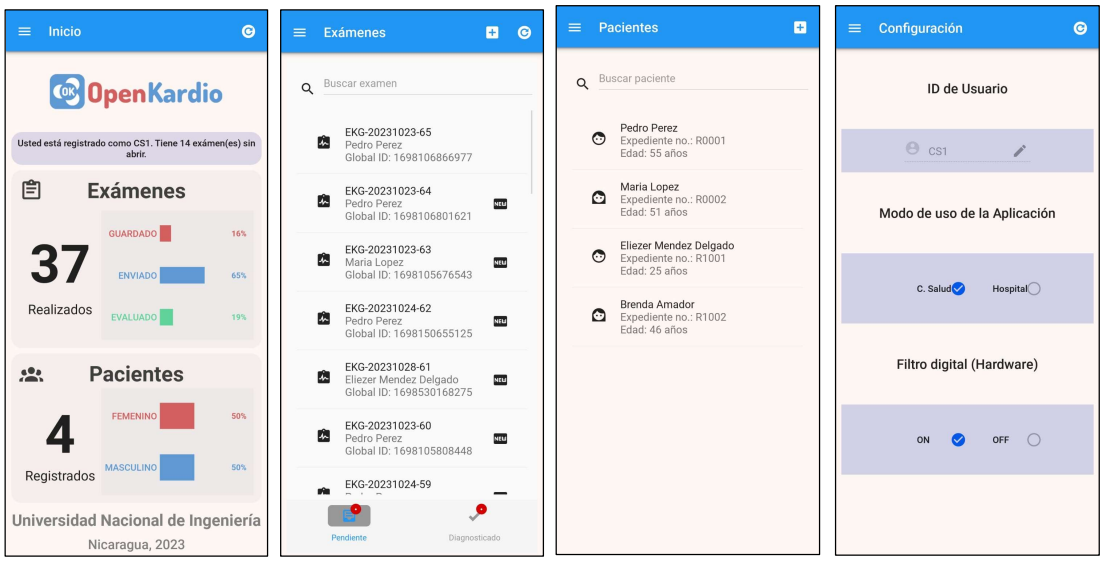


B3. Prototipo totalmente ensamblado con los electrodos conectados.

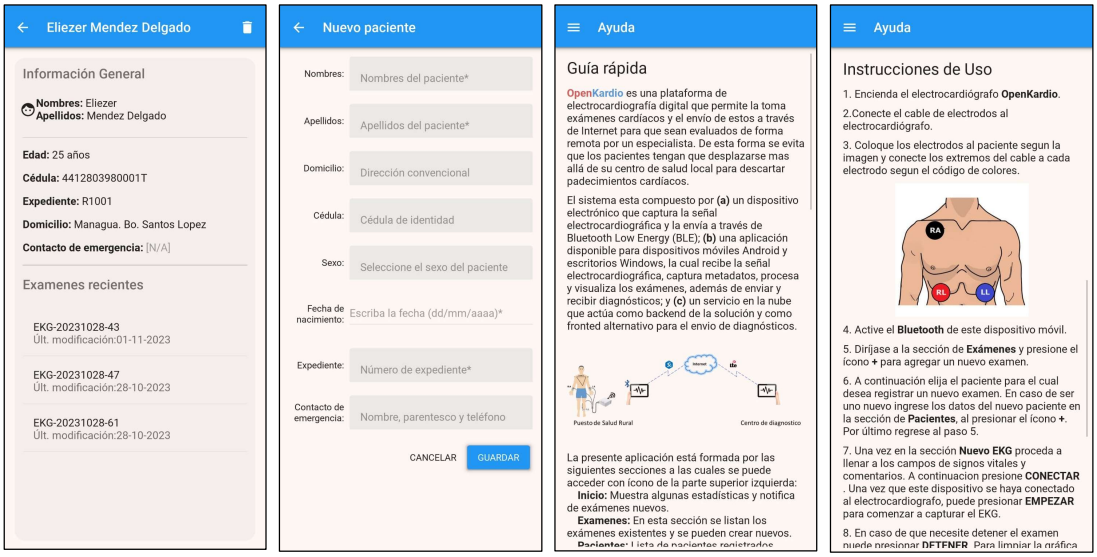
**Anexo C. Tabla de costos del prototipo.**

| Item | Componente | Cant. | Monto |
|---|---|---|---|
| 1 | Tarjeta de desarrollo ESP32 | 1 und. | $24.00 |
| 2 | Modulo ADC ADS1115 | 1 und. | $6.45 |
| 3 | Modulo ECG AD8232 | 1 und. | $18.00 |
| 4 | Cable de terminales | 1 und. | $5.50 |
| 5 | Electrodos descartables | 50 und | $14.00 |
| 6 | Batería Li-Ion 18650 3 Ah | 1 und. | $5.20 |
| 7 | Jumpers 24 AWG | 1 set | $1.40 |
| 8 | Caja plástica 158x90x60 mm | 1 und. | $7.40 |
| 9 | Soportes plásticos hexagonales | 1 set | $16.00 |
| 10 | Tarejta Virgen 10x15 cm | 1 und. | $1.80 |
| 11 | Impresión en papel satinado | 1 und. | $1.00 |
| 12 | Acido Nitrico | 0.5 ltr | $1.80 |
| 13 | Conector USB B Hembra | 1 set | $1.00 |
| 14 | Componentes discretos | N/A | $1.50 |
| 15 | Pintura en aerosol | 1 und. | $8.00 |
| 16 | Cabeceras de pines | 2 und | $1.40 |
| | **Total** | | **$114.45** |

## Anexo D. Pantallas de la Aplicación Móvil.




D1. Vistas generales de Inicio, Exámenes, Pacientes y Configuración.




D2. Vistas de Detalle de Paciente, formulario de Creación de Pacientes y sección de Ayuda.

## Anexo E. Vistas de la interfaz de diagnóstico.


E1. Formulario de diagnóstico cuyo enlace se envía en el cuerpo del correo.


E2. Mensaje de éxito al enviar un diagnóstico.


E3. Ejemplo de mensaje de error al no encontrar un ID de caso válido.

## Anexo F. Scripts de Jupyter Notebooks.

### F1. Diseño del filtro digital FIR pasa-bajas

```
from pylab import *
import scipy.signal as signal

#Plot step and impulse response
def impz(b,a=1):
    rcParams['figure.figsize'] = 12, 3
    l = len(b)
    impulse = repeat(0.,l); impulse[0] =1.
    x = arange(0,l)
    response = signal.lfilter(b,a,impulse)
    subplot(111)
    stem(x, response)
    ylabel('Amplitud')
    xlabel(r'n (muestras)')
    title(r'Respuesta al impulso')
    grid(color='k', linestyle='--')
    # subplot(212)
    # step = cumsum(response)
    # stem(x, step)
    # ylabel('Amplitude')
    # xlabel(r'n (samples)')
    # title(r'Step response')
    # grid(color='k', linestyle='--')
    # subplots_adjust(hspace=0.5)

import matplotlib.pyplot as plt

def plot_response(fig, w, h):
    "Utility function to plot response functions"
    ax = plt.subplot(2, 1, 1)
    ax.plot(w, 20*np.log10(np.abs(h)))
    ax.plot([22.4,22.4],[-80,10],"r--",linewidth=0.5)
    ax.plot([59.5,59.5],[-80,10],"r--",linewidth=0.5)
    ax.plot([240,240],[-80,10],"r--",linewidth=0.5)
    ax.plot([0,250],[-1,-1],"r--",linewidth=0.5)
    ax.plot([0,250],[1,1],"r--",linewidth=0.5)
    ax.plot([0,250],[-40,-40],"r--",linewidth=0.5)
    ax.set_ylim(-80, 5)
    ax.set_xlim(0,240)
    ax.grid(True)
    ax.annotate(text="40",xy=(35,-60))
    ax.annotate(text="59.5",xy=(52,-60))
    ax.set_ylabel('Ganancia (dB)')
    ax.set_title("Respuesta en frecuencia")
    ax2 = plt.subplot(2, 1, 2, sharex=ax)
    h_Phase = unwrap(arctan2(imag(h),real(h)))
    ax2.plot(w,h_Phase)
    # grid(color='k', linestyle='--')
    subplots_adjust(hspace=0.5)
    ax2.set_xlabel('Frecuencia (Hz)')
    ax2.set_ylabel('Fase (rads)')
    ax2.set_title("Respuesta de fase")
    ax2.set_xlim(0,240)
    ax2.grid(True)
    plt.tight_layout()
```

```
fs = 480
cutoff = 40   # Desired cutoff frequency, Hz
trans_width = 19.5 # Width of transition from pass to stop, Hz
numtaps1 = 52      # Size of the FIR filter.
lpf_coeff = signal.remez(numtaps1, [0, cutoff, cutoff + trans_width, 0.5*fs],
                   [1, 0], fs=fs)

fig = plt.figure(figsize=(9, 6))
w, h = signal.freqz(lpf_coeff, [1], worN=4096, fs=fs)
plot_response(fig, w, h)
plt.show()
figure(2)
impz(lpf_coeff)
```

## F2. Análisis del filtro diseñado aplicado a la señal original

```
import json
import array
import zlib
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fs = 480

with open('ecg/original_ecg.json','r') as f:
    rawdata = json.loads(f.read())
ekg_samples = array.array('h',zlib.decompress(bytes(rawdata['signal']))).tolist()
ekg_signal = (np.array(ekg_samples)/8800)[120:2040]

plt.figure(figsize=(12, 4))

plt.plot(np.array([i/fs for i in range(len(ekg_signal))]), ekg_signal)
# plt.plot(np.array(r_peaks)/fs,ekg_signal[r_peaks],'ro')

# Add labels and title
plt.xlabel('Tiempo (s)')
plt.ylabel('Amplitud')
plt.title('Señal ECG sin filtros')

# Display the chart
plt.show()

fig, ax = plt.subplots(figsize=(16, 5))
s,f,l = ax.magnitude_spectrum(ekg_signal,fs)
ax.plot(f,s)
print(ax)
ax.grid(True)
ax.set_xlim(0,130)
ax.set_xlabel("Frecuencia (Hz)")
ax.set_ylabel("Magnitud (Energía)")
ax.set_title("Espectro de la señal ECG")
plt.show()

coeff = np.array([-0.004656318004405627, 0.0004569009298227957, 0.0024217966033922757,
0.004684861893815379, 0.005911428270733918, 0.004923963289002614,
0.0013597792995145826, -0.003883667265059867, -0.008744938489793033, -
0.010699039238104122, -0.007916746969875156, -0.0003759665269553883,
0.009622486426806236, 0.017948344854431718, 0.020062780114192214, 0.013064145908364373,
```

```
-0.0025077187661627347, -0.022013763386495768, -0.03751156643160482, -
0.04018234286406491, -0.023470458899338402, 0.014053817803264725, 0.06734006004832002,
0.1256143549458254, 0.17523965924627843, 0.20375892388594258, 0.20375892388594258,
0.17523965924627843, 0.1256143549458254, 0.06734006004832002, 0.014053817803264725, -
0.023470458899338402, -0.04018234286406491, -0.03751156643160482, -
0.022013763386495768, -0.0025077187661627347, 0.013064145908364373,
0.020062780114192214, 0.017948344854431718, 0.009622486426806236, -
0.0003759665269553883, -0.007916746969875156, -0.010699039238104122, -
0.008744938489793033, -0.003883667265059867, 0.0013597792995145826,
0.004923963289002614, 0.005911428270733918, 0.004684861893815379,
0.0024217966033922757, 0.0004569009298227957, -0.004656318004405627])

from scipy import signal
t =np.array([i/fs for i in range(len(ekg_signal))])
filtered_signal = signal.lfilter(coeff, 1.0, ekg_signal)
# Determine the y-axis range for both subplots
y_min = min(np.min(ekg_signal), np.min(filtered_signal))
y_max = max(np.max(ekg_signal), np.max(filtered_signal))

plt.figure(figsize=(10, 4))
ax1 = plt.subplot(2, 1, 1)
ax1.plot(t, ekg_signal, linewidth=1)
ax1.set_title('Señal Original')
ax1.set_ylabel('Amplitud')
ax1.set_ylim(y_min, y_max)  # Set y-axis limits

ax2 = plt.subplot(2, 1, 2, sharex=ax1)
ax2.plot(t, filtered_signal, linewidth=1)
ax2.set_title('Señal filtrada')
ax2.set_xlabel('Tiempo (s)')
ax2.set_ylabel('Amplitud ')
ax2.set_ylim(y_min, y_max)  # Set y-axis limits

# Adjust layout to prevent overlap
plt.tight_layout()

# Show the plot
plt.show()

plt.figure(figsize=(15, 5))
fig, ax = plt.subplots(figsize=(16, 5))
s,f,l = ax.magnitude_spectrum(filtered_signal,fs)
ax.plot(f,s)
print(ax)
ax.grid(True)
ax.set_xlim(0,130)
plt.show()
```

F3. Análisis de la Razón Señal a Ruido antes y después de aplicar el filtrado en el prototipo.

```
import numpy as np
import scipy.signal as signal
from scipy.fft import fft
import json
import array
```

```
import zlib

fs = 480  # Sample rate (Hz)
t = np.linspace(0, 1, fs, endpoint=False)

with open('ecg/original_ecg.json','r') as f:
    rawdata = json.loads(f.read())
original_ekg_samples =
array.array('h',zlib.decompress(bytes(rawdata['signal']))).tolist()
original_ekg_signal = (np.array(original_ekg_samples)/8800)
with open('ecg/filtered_ecg.json','r') as f:
    rawdata = json.loads(f.read())
filtered_ekg_samples =
array.array('h',zlib.decompress(bytes(rawdata['signal']))).tolist()
filtered_ekg_signal = (np.array(filtered_ekg_samples)/8800)
# Calculate the FFT of the combined signal
original_signal_fft = fft(original_ekg_signal)
# Calculate the frequency axis
freq = np.fft.fftfreq(len(ekg_signal), 1 / fs)
# Find the indices for the signal and noise frequency ranges
signal_freq_range = (0, 40)  # Frequency range for the signal of interest
noise_freq_range = (40, fs / 2)  # Frequency range for the noise
signal_mask = (freq >= signal_freq_range[0]) & (freq <= signal_freq_range[1])
noise_mask = (freq >= noise_freq_range[0]) & (freq <= noise_freq_range[1])
# Calculate the power in the signal and noise frequency ranges
signal_power = np.sum(np.abs(original_signal_fft[signal_mask])**2) / len(signal_mask)
noise_power = np.sum(np.abs(original_signal_fft[noise_mask])**2) / len(noise_mask)
original_snr_db = 10 * np.log10(signal_power / noise_power)
print(original_snr_db)
# Calculate the FFT of the combined signal
filtered_signal_fft = fft(filtered_ekg_signal)

# Calculate the frequency axis
freq = np.fft.fftfreq(len(ekg_signal), 1 / fs)
# Find the indices for the signal and noise frequency ranges
signal_freq_range = (0, 40)  # Frequency range for the signal of interest
noise_freq_range = (40, fs / 2)  # Frequency range for the noise
signal_mask = (freq >= signal_freq_range[0]) & (freq <= signal_freq_range[1])
noise_mask = (freq >= noise_freq_range[0]) & (freq <= noise_freq_range[1])
# Calculate the power in the signal and noise frequency ranges
signal_power = np.sum(np.abs(filtered_signal_fft[signal_mask])**2) / len(signal_mask)
noise_power = np.sum(np.abs(filtered_signal_fft[noise_mask])**2) / len(noise_mask)
filtered_snr_db = 10 * np.log10(signal_power / noise_power)
print(filtered_snr_db)
```

## F4. Análisis de algoritmos de Detección de picos R

```
from ecgdetectors import Detectors
import json
import array
import zlib
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fs = 480
detectors = Detectors(fs)
with open('ecg/original_ecg.json','r') as f:
    rawdata = json.loads(f.read())
```

```
ekg_samples = array.array('h',zlib.decompress(bytes(rawdata['signal']))).tolist()

ekg_signal = (np.array(ekg_samples)/8800)[120:2040]
plt.figure(figsize=(12, 2))
r_peaks = detectors.christov_detector(ekg_signal)

plt.plot(np.array([i/fs for i in range(len(ekg_signal))]), ekg_signal)
plt.plot(np.array(r_peaks)/fs,ekg_signal[r_peaks],'ro')

# Add labels and title
plt.xlabel('Tiempo (s)')
plt.ylabel('Amplitud')
plt.title('Detección de picos R con Algoritmo de Christov')

# Display the chart
plt.show()
plt.figure(figsize=(12, 2))
r_peaks = detectors.hamilton_detector(ekg_signal)

plt.plot(np.array([i/fs for i in range(len(ekg_signal))]), ekg_signal)
plt.plot(np.array(r_peaks)/fs,ekg_signal[r_peaks],'ro')

# Add labels and title
plt.xlabel('Tiempo (s)')
plt.ylabel('Amplitud')
plt.title('Detección de picos R con Algoritmo de Hamilton')

# Display the chart
plt.show()
plt.figure(figsize=(12, 2))
r_peaks = detectors.pan_tompkins_detector(ekg_signal)

plt.plot(np.array([i/fs for i in range(len(ekg_signal))]), ekg_signal)
plt.plot(np.array(r_peaks)/fs,ekg_signal[r_peaks],'ro')

# Add labels and title
plt.xlabel('Tiempo (s)')
plt.ylabel('Amplitud')
plt.title('Detección de picos R con Algoritmo de Pan-Tompkins')

# Display the chart
plt.show()
```

## Anexo G. Código fuente del *firmware* del prototipo.

### G1. main.cpp

```
#include <Arduino.h>
#include <StateMachine.h>
#include <Ticker.h>
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <ADS1X15.h>
#include "OKConfig.h"
#include "lpfilter.h"

BLECharacteristic OKDataCharacteristic(OK_DATA_UUID, BLECharacteristic::PROPERTY_READ |
BLECharacteristic::PROPERTY_NOTIFY);
BLECharacteristic OKCtrlCharacteristic(OK_CTRL_UUID, BLECharacteristic::PROPERTY_READ |
BLECharacteristic::PROPERTY_WRITE);
uint8_t error_code = 0x00;
void send_flag(uint8_t flag){
  uint8_t temp[1];
  temp[0] = flag;
  OKDataCharacteristic.setValue(temp,1);
  OKDataCharacteristic.notify();
}

//// Variables declaration
// Timer reference variable
hw_timer_t *sampling_clock = NULL;

// ISR variables
volatile bool request = false;
volatile long tic, toc = 0L;
volatile long prebuffer_counter = 0;
long sample_times[5000] = {0};
long max_time = 0;
long min_time = 0x7FFFFFFF;
int frame_counter = 0;

// Filter reference
lpfilterType *pFilter = lpfilter_create();
bool bFilterActive = false;
int16_t filter(int16_t sample){
  if(bFilterActive){
    float input = float(sample);
    lpfilter_writeInput(pFilter, input);
    float output = lpfilter_readOutput(pFilter);
    return int16_t(output);
  }
  return sample;
}
// Ekg processing variables
int byte_count = 0;
sample ekg_sample;
uint8_t sample_buffer[BUFFER_LENGTH];
int16_t signal_offset = 0;
int16_t raw_sample = 0;

// Device info
info device_info = {
  100,
  SAMPLES_PER_FRAME,
  EKG_SAMPLE_RATE,
  FRONTEND_LEADS,
  ADC_RESOLUTION,
  FIRMWARE_MAJOR_VERSION,
```

```
  FIRMWARE_MINOR_VERSION,
  FRONTEND_GAIN*ADC_GAIN,
};
bool dev_connected = false;
bool exam_running = false;
bool adc_connected = false;

//ADS declaration and setup definition
ADS1115 ADS(0x48);
void adc_setup(void){
  adc_connected = ADS.begin();
  ADS.setWireClock(400000U);
  ADS.setGain(1);     // 4.096 volt max range
  ADS.setDataRate(7); // 1.16 ms per conversion
}

// Ticker declarations
Ticker led_blinker;
Ticker batt_monitor;

void blink(int pattern) {
  static int counter = 0;
  digitalWrite(STAT_LED_PIN, (pattern>>counter++)&0x01);
  if(counter>15) counter = 0;
}

// FSM definitions and flags
StateMachine machine = StateMachine();
void state_idle_cb(void){
  if(machine.executeOnce){
    digitalWrite(STAT_LED_PIN, LOW);
    Serial.println("State: IDLE");
  }
}
void state_conn_cb(void){
  if(machine.executeOnce){
    led_blinker.attach_ms(125, blink, 0x000A);  // 2 pulses every 2 segundos
    OKCtrlCharacteristic.setValue((uint8_t*)&device_info,sizeof(info));
    Serial.println("State: CONN");
  }
}
void on_conn_exit(void){
  led_blinker.detach();
  digitalWrite(STAT_LED_PIN, LOW);
}
void state_send_cb(){
  int16_t newSample = 0;
  if(machine.executeOnce){
    byte_count = 0;
    prebuffer_counter = 0;
    memset(sample_times,0,sizeof(sample_times));
    frame_counter = 0;
    max_time = 0;
    min_time = 0x7FFFFFFF;
    signal_offset = ADS.readADC(0)>>1;
    timerAlarmEnable(sampling_clock);
    digitalWrite(STAT_LED_PIN, HIGH);
    if(!adc_connected){
      error_code = NO_ADC;
      OKDataCharacteristic.setValue(&error_code,1);
      OKDataCharacteristic.notify();
    }
    Serial.println("State: SEND");
  }
  if(request){

    raw_sample = ADS.readADC(1);
    newSample = raw_sample-signal_offset;
```

```
    ekg_sample.raw = filter(newSample);

    sample_buffer[byte_count++] = ekg_sample.bytes[0];
    sample_buffer[byte_count++] = ekg_sample.bytes[1];
    request = false;

    if(byte_count/2 == SAMPLES_PER_FRAME){
      if (prebuffer_counter > lpfilter_length){
        tic = micros();
        OKDataCharacteristic.setValue(sample_buffer,byte_count);
        OKDataCharacteristic.notify();
        toc = micros();

        frame_counter++;

        sample_times[frame_counter-1] = toc - tic;
        max_time = max(max_time, sample_times[frame_counter-1]);
        min_time = min(min_time, sample_times[frame_counter-1]);
      }

      byte_count = 0;
    }
  }
}

void on_send_exit(void){
  long sum = 0;
  double average = 0;

  timerAlarmDisable(sampling_clock);
  digitalWrite(STAT_LED_PIN, LOW);

  if (frame_counter <= 5000 && frame_counter > 0){
    for (int i = 0; i < frame_counter; i++) {
        sum += sample_times[i];
    }
    average = (double)sum / frame_counter;
    double squaredDifferencesSum = 0.0;

    for (int i = 0; i < frame_counter; i++) {
        double difference = sample_times[i] - average;
        squaredDifferencesSum += difference * difference;
    }

    double variance = squaredDifferencesSum / (frame_counter);
    double sampleStdDeviation = sqrt(variance);

    Serial.println("-----");
    Serial.print("Time AVG: ");
    Serial.println(average);
    Serial.print("Time MAX: ");
    Serial.println(max_time);
    Serial.print("Time MIN: ");
    Serial.println(min_time);
    Serial.print("Time STD: ");
    Serial.println(sampleStdDeviation);
  }
  else {
    Serial.println("There was a problem.");
  }
  Serial.println("-----");
}

State* IDLE = machine.addState(&state_idle_cb);
State* CONN = machine.addState(&state_conn_cb);
State* SEND = machine.addState(&state_send_cb);

void IRAM_ATTR onSamplingClock(){
    prebuffer_counter++;
```

```
    request = true;
}

void timer_setup(void){
  sampling_clock = timerBegin(0, 8, true);
  timerAttachInterrupt(sampling_clock, &onSamplingClock, true);
  timerAlarmWrite(sampling_clock, (uint64_t)10000000/EKG_SAMPLE_RATE, true);
}

void measure_batt(void){
  float voltage_level;
  voltage_level = analogReadMilliVolts(35)*2.0/1000.0; // factor 2 for compensating the voltage
divider the pin 35 is attached to
  device_info.battery_level = 10 * min(10,int(floor(9.49 * (voltage_level - MIN_BATTERY_VOLTAGE) /
(MAX_BATTERY_VOLTAGE - MIN_BATTERY_VOLTAGE))));
  if(machine.isInState(CONN)) OKCtrlCharacteristic.setValue((uint8_t*)&device_info,sizeof(info));
}

bool idle_to_conn(){
  return dev_connected;
}

bool conn_to_idle(){
  if(!dev_connected){
    on_conn_exit();
    return true;
  }
  return false;
}

bool conn_to_send(){
  if(exam_running){
    on_conn_exit();
    return true;
  }
  return false;
}

bool send_to_idle(){
  if(!dev_connected){
    exam_running = false;
    return true;
  }
  return false;
}

bool send_to_conn(){
  if(!exam_running){
    on_send_exit();
    return true;
  }
  return false;
}

void sm_setup(void){
  IDLE->addTransition(&idle_to_conn,CONN);
  CONN->addTransition(&conn_to_idle,IDLE);
  CONN->addTransition(&conn_to_send,SEND);
  SEND->addTransition(&send_to_idle,IDLE);
  SEND->addTransition(&send_to_conn,CONN);
}

// BLE Callbacks
class OKServerCallbacks: public BLEServerCallbacks {
  void onConnect(BLEServer* pServer) {
    dev_connected = true;
  }
  void onDisconnect(BLEServer* pServer) {
    dev_connected = false;
```

```
    exam_running = false;
    pServer->getAdvertising()->start();
    Serial.println("Waiting a client connection to notify...");
  }
};

class OKCtrlCallbacks: public BLECharacteristicCallbacks {
  void onWrite(BLECharacteristic *pCharacteristic) {
    uint8_t* pData = pCharacteristic->getData();
    Serial.print("CMD:  ");
    Serial.println(pData[COMMAND],HEX);
    switch(pData[COMMAND]){
      case 0x00:
      exam_running = true;
      break;
      case 0x10:
      bFilterActive = false;
      break;
      case 0x11:
      bFilterActive = true;
      break;
      case 0xFF:
      if(machine.isInState(SEND)) exam_running = false;
      break;
      default:
      break;
    }
  }
};

void ble_setup(void){
    // Create the BLE Device
  BLEDevice::init(BLE_SERVER_NAME);
  BLEDevice::setMTU(BLE_L2CAP_MTU);
  // Create the BLE Server
  BLEServer *pServer = BLEDevice::createServer();
  pServer->setCallbacks(new OKServerCallbacks());
  // Create the BLE Service
  BLEService *pService = pServer->createService(SERVICE_UUID);
  // Create and add Descriptors
  BLEDescriptor *pOKDataDescriptor = new BLE2902();
  OKDataCharacteristic.addDescriptor(pOKDataDescriptor);
  BLEDescriptor *pOKCtrlDescriptor = new BLE2902();
  OKCtrlCharacteristic.addDescriptor(pOKCtrlDescriptor);
  // Add Characteristics
  pService->addCharacteristic(&OKDataCharacteristic);
  pService->addCharacteristic(&OKCtrlCharacteristic);
  // Set Callbacks for characteristics
  OKCtrlCharacteristic.setCallbacks(new OKCtrlCallbacks());
  // Start advertising
  pService->start();
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pServer->getAdvertising()->start();
  Serial.println("Waiting a client connection..");
}

void setup() {
  Serial.begin(115200);
  pinMode(STAT_LED_PIN, OUTPUT);
  pinMode(PUSH_BTN_PIN, INPUT_PULLUP);
  pinMode(5,OUTPUT);
  digitalWrite(5,HIGH);
  lpfilter_init(pFilter);
  timer_setup();
  adc_setup();
  sm_setup();
  ble_setup();
  batt_monitor.attach(6, measure_batt);
```

```
}

void loop() {
  machine.run();
}
```

## G2. OKConfig.h

```
#include <Arduino.h>
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <ADS1X15.h>

#define FIRMWARE_MAJOR_VERSION 1
#define FIRMWARE_MINOR_VERSION 0
#define MAX_BATTERY_VOLTAGE 4.2     // V
#define MIN_BATTERY_VOLTAGE 3.0     // V
#define BLE_MIN_CONN_INTERVAL 40    // ms
#define BLE_MAX_CONN_INTERVAL 80    // ms

#define BATTERY_PIN 35
#define AD8232_LODM 2
#define AD8232_LODP 15
#define PUSH_BTN_PIN 0
#define STAT_LED_PIN 4

#define ADC_RESOLUTION 16
#define ADC_GAIN 8.0 // Steps/mV
#define FRONTEND_GAIN 1100.0 // mV/mV
#define FRONTEND_LEADS 1

#define EKG_SAMPLE_RATE 480
#define SAMPLES_PER_FRAME 15
#define BUFFER_LENGTH 30
#define BLE_L2CAP_MTU BUFFER_LENGTH + 3 // Bytes
#define COMMAND 0
#define PAYLOAD 1

const uint8_t ERROR_FLAG = 0xFF;
const uint8_t LEADS_OFF = 0xF1;
const uint8_t NO_ADC = 0xE1;
const uint8_t WAITING = 0x00;

//// BLE definitions
#define BLE_SERVER_NAME "OKDevice"
#define SERVICE_UUID "5e6c5000-05d8-463b-b21d-eed2204c2002"
#define OK_DATA_UUID "5e6c5001-05d8-463b-b21d-eed2204c2002"
#define OK_CTRL_UUID "5e6c5002-05d8-463b-b21d-eed2204c2002"

//// Type definitions
union sample{
  volatile int16_t raw;
  volatile uint8_t bytes[2];
};
struct info {
  uint8_t battery_level;
  uint8_t samples_per_frame;
  uint16_t sample_rate;
  const uint8_t lead_count;
  const uint8_t resolution;
  const uint8_t fw_major_version;
  const uint8_t fw_minor_version;
  const double conv_factor;
};
```

## G3. lpfilter.h

```
#ifndef LPFILTER_H_ // Include guards
#define LPFILTER_H_

static const int lpfilter_length = 52;
extern float lpfilter_coefficients[52];

typedef struct
{
  float * pointer;
  float state[104];
  float output;
} lpfilterType;


lpfilterType *lpfilter_create( void );
void lpfilter_destroy( lpfilterType *pObject );
void lpfilter_init( lpfilterType * pThis );
void lpfilter_reset( lpfilterType * pThis );
#define lpfilter_writeInput( pThis, input )  \
    lpfilter_filterBlock( pThis, &(input), &(pThis)->output, 1 );

#define lpfilter_readOutput( pThis )  \
    (pThis)->output

int lpfilter_filterBlock( lpfilterType * pThis, float * pInput, float * pOutput, unsigned int
count );
#define lpfilter_outputToFloat( output )  \
    (output)

#define lpfilter_inputFromFloat( input )  \
    (input)

void lpfilter_dotProduct( float * pInput, float * pKernel, float * pAccumulator, short count );
#endif // LPFILTER_H_
```

## G4. lpfilter.cpp

```
#include "lpfilter.h"
#include <stdlib.h> // For malloc/free
#include <string.h> // For memset

float lpfilter_coefficients[52] =
{
    -0.004656318004405627, 0.0004569009298227957, 0.0024217966033922757, 0.004684861893815379,
0.005911428270733918, 0.004923963289002614, 0.0013597792995145826, -0.003883667265059867, -
0.008744938489793033, -0.010699039238104122, -0.007916746969875156, -0.0003759665269553883,
0.009622486426806236, 0.017948344854431718, 0.020062780114192214, 0.013064145908364373, -
0.0025077187661627347, -0.022013763386495768, -0.03751156643160482, -0.04018234286406491, -
0.023470458899338402, 0.014053817803264725, 0.06734006004832002, 0.1256143549458254,
0.17523965924627843, 0.20375892388594258, 0.20375892388594258, 0.17523965924627843,
0.1256143549458254, 0.06734006004832002, 0.014053817803264725, -0.023470458899338402, -
0.04018234286406491, -0.03751156643160482, -0.022013763386495768, -0.0025077187661627347,
0.013064145908364373, 0.020062780114192214, 0.017948344854431718, 0.009622486426806236, -
0.0003759665269553883, -0.007916746969875156, -0.010699039238104122, -0.008744938489793033, -
0.003883667265059867, 0.0013597792995145826, 0.004923963289002614, 0.005911428270733918,
0.004684861893815379, 0.0024217966033922757, 0.0004569009298227957, -0.004656318004405627
};

lpfilterType *lpfilter_create( void )
{
    lpfilterType *result = (lpfilterType *)malloc( sizeof( lpfilterType ) ); // Allocate memory
for the object
    lpfilter_init( result );                                          // Initialize it
    return result;                                                   // Return the result
```

```
}

void lpfilter_destroy( lpfilterType *pObject )
{
    free( pObject );
}

void lpfilter_init( lpfilterType * pThis )
{
    lpfilter_reset( pThis );
}

void lpfilter_reset( lpfilterType * pThis )
{
    memset( &pThis->state, 0, sizeof( pThis->state ) ); // Reset state to 0
    pThis->pointer = pThis->state;                      // History buffer points to start of state
buffer
    pThis->output = 0;                                  // Reset output
}

int lpfilter_filterBlock( lpfilterType * pThis, float * pInput, float * pOutput, unsigned int
count )
{
    float *pOriginalOutput = pOutput;               // Save original output so we can track the
number of samples processed
    float accumulator;

    for( ;count; --count )
    {
        pThis->pointer[lpfilter_length] = *pInput;                  // Copy sample to top of
history buffer
        *(pThis->pointer++) = *(pInput++);                         // Copy sample to bottom of
history buffer

        if( pThis->pointer >= pThis->state + lpfilter_length )      // Handle wrap-around
            pThis->pointer -= lpfilter_length;

        accumulator = 0;
        lpfilter_dotProduct( pThis->pointer, lpfilter_coefficients, &accumulator, lpfilter_length
);

        *(pOutput++) = accumulator;  // Store the result
    }

    return pOutput - pOriginalOutput;
}

void lpfilter_dotProduct( float * pInput, float * pKernel, float * pAccumulator, short count )
{
    float accumulator = *pAccumulator;
    while( count-- )
        accumulator += ((float)*(pKernel++)) * *(pInput++);
    *pAccumulator = accumulator;
}
```

## platformio.ini

```
[env:ttgo-t-energy]
platform = espressif32
board = ttgo-t7-v14-mini32
framework = arduino
monitor_speed = 115200
build_type = debug
lib_deps =
    ivanseidel/LinkedList@0.0.0-alpha+sha.dac3874d28
    jrullan/StateMachine@^1.0.11
    robtillaart/ADS1X15@^0.3.7
```

## Anexo H. Código Fuente de la Aplicación Móvil.

### H1. requirements.txt

```
appdirs==1.4.4
async-timeout==4.0.2
autopep8==1.7.0
bleak==0.20.2
buildozer==1.5.0
cachetools==5.3.1
certifi==2022.5.18.1
charset-normalizer==2.0.12
colorama==0.4.4
contourpy==1.1.0
cycler==0.11.0
Cython==0.29.33
dbus-fast==1.85.0
distlib==0.3.6
docutils==0.18.1
filelock==3.12.0
fonttools==4.40.0
gatspy==0.3
google-auth==2.20.0
greenlet==1.1.2
heartpy==1.2.7
idna==3.3
importlib-metadata==4.11.4
Jinja2==3.1.2
Kivy==2.2.0
Kivy-examples==2.2.0
Kivy-Garden==0.1.5
kivymd==1.1.1
kiwisolver==1.4.4
MarkupSafe==2.1.1
matplotlib==3.7.1
numpy==1.22.4
packaging==23.1
pep517==0.6.0
pexpect==4.8.0
Pillow==9.1.1
pip-autoremove==0.10.0
pipdeptree==2.7.1
platformdirs==3.5.0
ptyprocess==0.7.0
py-ecg-detectors==1.3.4
pyasn1==0.5.0
pyasn1-modules==0.3.0
pycodestyle==2.9.1
Pygments==2.12.0
pyjnius==1.4.2
pyparsing==3.0.9
python-dateutil==2.8.2
pytoml==0.1.21
PyWavelets==1.4.1
requests==2.27.1
rsa==4.9
scipy==1.10.1
sh==1.14.2
six==1.16.0
SQLAlchemy==1.4.40
SQLAlchemy-Utils==0.38.3
toml==0.10.2
typing_extensions==4.5.0
urllib3==1.26.9
virtualenv==20.23.0
zipp==3.8.0
```

### H2. buildozer.spec

```
[app]

title = OpenKardio

package.name = com.openkardio

package.domain = org.ok

source.dir = .

source.include_exts = py,png,jpg,kv,atlas,json

version = 0.1

requirements =
python3,kivy==2.2.0,pillow,numpy,dateutil,sqlalchemy==1.4.40,sqlalchemy_utils==0.38.3,bleak,async_
to_sync,async_timeout,typing_extensions,kivymd==1.2.0,google-
auth==2.20.0,cachetools==5.3.1,pyasn1-
modules==0.3.0,pyasn1==0.5.0,rsa==4.9,six==1.16.0,urllib3==1.26.9

presplash.filename = assets/isotipo-padded.png

icon.filename = assets/isotipo.png

orientation = portrait
fullscreen = 0

android.presplash_color = #FFFFFF
```

```
android.permissions =
INTERNET,BLUETOOTH,BLUETOOTH_ADMIN,ACCESS_FINE_LOCATION,ACCESS_COARSE_LOCATION,ACCESS_BACKGROUND_L
OCATION,BLUETOOTH_SCAN,BLUETOOTH_CONNECT,WRITE_EXTERNAL_STORAGE,READ_EXTERNAL_STORAGE

android.add_src = ./java
```

### H3. main.py

```
import asyncio
import array
import json
from kivy.logger import Logger
import logging
import utils.remotedb as rdb
import utils.localdb as ldb
import utils.ble as ble
from kivymd.app import MDApp
from datetime import date, datetime
from kivy.lang import Builder
from kivy.storage.jsonstore import JsonStore
from kivy.metrics import dp
from kivymd.theming import ThemeManager
from kivymd.uix.menu import MDDropdownMenu
from kivymd.uix.dialog import MDDialog
from kivymd.uix.button import MDRaisedButton, MDFlatButton
from kivymd.toast import toast
from navdrawer import ItemDrawer
from okwidgets import OKHospitalSelectorItem, OKCommentWidget
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy.orm import make_transient
from kivy.core.window import Window
from kivy.utils import platform
from kivy.clock import Clock
from kivy.base import EventLoop
from os.path import join
from time import perf_counter

if platform != 'android':
    Window.size = (800, 800)
    Window.minimum_width, Window.minimum_height = Window.size

logging.Logger.manager.root = Logger

class OpenKardioApp(MDApp):
    title = "OpenKardio"
    theme_cls = ThemeManager()
    icons = {'M': 'face-man', 'F': 'face-woman'}
    dialog = None

    def __init__(self):
        super().__init__()
        self.ble = ble.BleHandler(frame_handler=self.frame_handler)
        self.running = True

    def build(self):
        logging.basicConfig(format='%(asctime)s %(message)s')
        self.theme_cls.material_style = "M3"
        self.icon = 'assets/isotipo-mini.png'

        if __name__ == '__main__':
            data_dir = getattr(self, 'user_data_dir')
            Logger.info(f"DATADIR: {data_dir}")
            try:
                self.store = JsonStore(join(data_dir,'config.json'))
                self.user_id = self.store["user"]["id"]
```

```
            except KeyError as e:
                Logger.critical(e)
                config_data = {
                    "app": {"mode": "C"},
                    "user": {"id": "CS1"},
                    "device": {"duration": 10},
                    "filter": {"state":"on"}
                }
                with open(join(data_dir,'config.json'), 'w') as config_file:
                    json.dump(config_data, config_file)
            finally:
                self.store = JsonStore(join(data_dir,'config.json'))
                self.user_id = self.store["user"]["id"]


            return Builder.load_file("kv/main.kv")

    def on_start(self):
        # attaching keyboard hook when app starts
        EventLoop.window.bind(on_keyboard=self.hook_keyboard)
        self.session = ldb.session_init(getattr(self, 'user_data_dir'))
        if self.session.query(ldb.Patient).first() is None:
                patient1 = ldb.Patient(first_name="Pedro",
                                       last_name="Perez",
                                       sex="M",
                                       birth_date=date(1968, 8, 28),
                                       record="R0001",
                                       identification="441-280868-0006T",
                                       address="San Ramón, Matagalpa",
                                       emergency_contact="505 8765 4321")
                patient2 = ldb.Patient(first_name="Maria",
                                       last_name="Lopez",
                                       sex="F",
                                       birth_date=date(1972, 4, 5),
                                       record="R0002",
                                       identification="321-050472-0001K",
                                       address="Camoapa, Boaco",
                                       emergency_contact="505 8765 4321")
                self.session.add_all([patient1, patient2])
                self.session.commit()
        self.root.ids.patient_form.sex_menu = MDDropdownMenu(caller =
self.root.ids.patient_form.ids.sex,
                                                             width_mult = 3,
                                                             items = [{"viewclass":
"OneLineListItem",
                                                                       "text": i,
                                                                       "height": dp(50),
                                                                       "on_release": lambda x=i:
self.root.ids.patient_form.set_sex(x)} for i in ["Femenino", "Masculino"]])

        drawer_items = {
            "home": {"text": "Inicio", "target": "homescreen"},
            "heart-pulse": {"text": "Exámenes", "target": "ekg_list_view"},
            "folder-account": {"text": "Pacientes", "target": "patient_list_view"},
            "cog": {"text": "Configuración", "target": "config"},
            "help": {"text": "Ayuda", "target": "help"}
            }
        for item_name in drawer_items.keys():
            self.root.ids.content_drawer.ids.md_list.add_widget(
                ItemDrawer(icon=item_name, text=drawer_items[item_name]["text"],
screen=drawer_items[item_name]["target"])
            )
        self.populate_start()

        Clock.schedule_once(lambda dt: self.populate_hospitals(), 5)

    def hook_keyboard(self, window, key, *largs):

        # key == 27 means it is waiting for back button to be pressed
```

```
            if key == 27:
                # return True means do nothing
                return True

        def on_stop(self):
            self.running = False
            self.session.close()

        def ble_disconnect(self):
            self.ble.transition(ble.ConnState.IDLE)

        def frame_handler(self, sender, data):

            Logger.info("SENDER: {0}: {1} [{2} bytes]".format(sender, data, len(data)))
            if len(data) == 1:
                self.ble.toggle_reception()
                Logger.error(f"Frame: Error while receiving frame. Error code={data}")
                toast("Ocurrió un error desconocido")
            else:
                try:
                    samples = array.array('h', data).tolist()
                    self.root.ids.new_ekg.next_samples = samples
                    if len(self.root.ids.new_ekg.ekg_samples) >= self.ble.sample_rate*9.6:
                            self.ble.transition(ble.ConnState.CONNECTED)
                except Exception as e:
                    Logger.error(f"Frame: Error while decoding frame.")
                    Logger.error(f"Frame: {e}")
                    self.ble.toggle_reception()

        def go_back(self, previous):
            self.root.ids.screen_manager.current = previous
            self.root.ids.screen_manager.transition.direction = "right"

        def go_to(self, target):
            self.root.ids.screen_manager.current = target
            self.root.ids.screen_manager.transition.direction = "left"

        def mark_exam_as_opened(self, id):
            self.session.query(ldb.Exam).filter(ldb.Exam.id == id).update({ldb.Exam.unopened: False})
            self.session.commit()

        def populate_hospitals(self):
            try:
                hospitals = rdb.retrieve_all_objects("Hospitals")
                for gid in hospitals:
                    if self.session.query(ldb.Hospital).filter(ldb.Hospital.global_id == gid).first()
    is None:
                        hosp_data = hospitals[gid]
                        hosp_data["global_id"] = gid
                        new_hosp = ldb.Hospital(**hosp_data)
                        self.session.add(new_hosp)
                        self.session.commit()
                        Logger.info(f"Hospital ID: {new_hosp.id}")

            except SQLAlchemyError as e:
                Logger.error(e)
                toast("Error en la base de datos local.")
            except rdb.RDBException as e:
                Logger.error(f"RDB:{str(e)}")
                toast(str(e))
            except Exception as e:
                Logger.error(f"Hospitals:{str(e)}")
                toast("Error desconocido.")

        def upsert_local_cases(self):
            try:
                sent_exams = self.session.query(ldb.Exam.remote_id).filter(ldb.Exam.remote_id != "")
                sent_exam_remote_ids = [exam.remote_id for exam in sent_exams]
                remote_cases = rdb.retrieve_objects("Cases","origin_id",self.store["user"]["id"])
```

```
            Logger.debug(f"Remote ID:{str(sent_exam_remote_ids)}")
            for gid in remote_cases:
                case_dict = remote_cases[gid]
                if gid in sent_exam_remote_ids:
                    local_exam = self.session.query(ldb.Exam).filter(ldb.Exam.remote_id ==
gid).one()
                    if datetime.fromisoformat(case_dict['modified']) > local_exam.modified:
                        timestamp = datetime.now().replace(microsecond=0)
                        Logger.debug(f"REMOTE:{case_dict['modified']}")
                        local_exam.diagnostic = case_dict["diagnostic"]
                        local_exam.status = case_dict['status']
                        local_exam.modified = timestamp
                        local_exam.diagnosed = timestamp
                        local_exam.unopened = True
                        self.session.commit()
                else:


                    ekg = {
                        "bpm":int(case_dict.get('bpm')),
                        "sample_rate":int(case_dict.get('sample_rate')),
                        "gain":float(case_dict.get('gain')),
                        "rpeaks":bytes(array.array('I',list(case_dict.get('rpeaks')))),
                        "signal":bytes(case_dict.get('signal'))
                    }
                    new_ekg = ldb.Ekg(**ekg)
                    self.session.add(new_ekg)
                    self.session.commit()
                    local_patients =
self.session.query(ldb.Patient.identification).filter(ldb.Patient.identification != "")
                    patient_ids = [patient.identification for patient in local_patients]
                    if case_dict['patient_identification'] not in patient_ids:
                        patient = {
                            "first_name":case_dict.get('patient_first_name'),
                            "last_name":case_dict.get('patient_last_name'),
                            "birth_date":datetime.strptime(case_dict.get('patient_birth_date'),'%d
-%m-%Y').date(),
                            "sex":case_dict.get('patient_sex'),
                            "identification":case_dict.get('patient_identification'),
                            "record":case_dict.get('patient_record'),
                            "address":case_dict.get('ptient_address')
                        }
                        new_patient = ldb.Patient(**patient)
                        self.session.add(new_patient)
                        self.session.commit()
                    exam_data = {
                        'remote_id':gid,
                        'name':'EKG-' +
datetime.fromisoformat(case_dict.get('created')).strftime('%Y%m%d') + '-' + str(new_ekg.id),
                        'ekg_id':new_ekg.id,
                        'created':datetime.fromisoformat(case_dict.get('created')),
                        'sent':datetime.fromisoformat(case_dict.get('sent')),
                        'modified':datetime.fromisoformat(case_dict.get('modified')),
                        'origin_id':self.store['user']['id'],
                        'destination_id':case_dict.get('destination_id'),
                        'spo2':float(case_dict.get('spo2')),
                        'weight_pd':float(case_dict.get('weight_pd')),
                        'pressure':case_dict.get('pressure'),
                        'notes':case_dict.get('notes'),
                        'status':case_dict.get('status'),
                        'patient_id':self.session.query(ldb.Patient.id).filter(ldb.Patient.identif
ication==case_dict['patient_identification']).one()[0]
                    }
                    if case_dict.get('status') == 'EVALUADO':
                        exam_data.update(
                            {
                                'diagnosed':datetime.fromisoformat(case_dict.get('diagnosed')),
                                'diagnostic':case_dict.get('diagnostic')
                            }
```

```
                    )
                new_exam = ldb.Exam(**exam_data)
                self.session.add(new_exam)
                self.session.commit()

        except SQLAlchemyError as e:
            Logger.error(e)
            toast("Error en la base de datos local.")

        except rdb.RDBException as e:
            Logger.error(e)
            toast(str(e))

        except Exception as e:
            Logger.error(e)
            toast("Error desconocido")

        finally:
            self.session.rollback()


    def download_cases(self):
        if self.store['app']['mode'] == 'C':
            self.upsert_local_cases()
        elif self.store['app']['mode'] == 'H':
            self.retrieve_foreign_cases()
        self.root.ids.pending.badge_icon =
self.root.ids.exam_pending_list.populate("",False,False)
        self.root.ids.done.badge_icon = self.root.ids.exam_done_list.populate("",False,True)

    def try_exam_creation(self):
        if self.store['app']['mode'] == 'H':
            toast("Modo 'Hospital' activo. No puede crear exámenes.")
        else:
            self.go_to('select_patient_view')

    def save_exam(self):

        def save():
            ekg = self.root.ids.new_ekg.get_ekg()
            new_ekg = ldb.Ekg(**ekg)
            self.session.add(new_ekg)
            self.session.commit()
            exam_data = {
                'name':'EKG-' + date.today().strftime("%Y%m%d") + '-' + str(new_ekg.id),
                'ekg_id':new_ekg.id,
                'created':timestamp,
                'modified':timestamp,
                'origin_id':self.store['user']['id']
            }
            exam_data.update(metadata)
            new_exam = ldb.Exam(**exam_data)
            self.session.add(new_exam)
            self.session.commit()
            Logger.info("Commited Ekg")
            self.root.ids.screen_manager.get_screen("ekg_detail_view").object_id = new_exam.id
            self.root.ids.screen_manager.get_screen("ekg_detail_view").title = new_exam.name
            self.root.ids.screen_manager.current = "ekg_detail_view"

        try:
            timestamp = datetime.now().replace(microsecond=0)
            metadata = self.root.ids.new_exam_metadata.save()

            field_map = {
                "spo2":"SpO2 (%)",
                "weight_pd":"Peso (lb)",
                "pressure":"Presión"
            }
```

```
            wrong_field = ""

            for k, v in metadata.items():

                if v is None:

                    wrong_field = field_map[k]
                    break

            if wrong_field:

                self.dialog = MDDialog(
                    text=f"El campo {wrong_field} tiene un valor incorrecto.",
                    buttons=[
                        MDRaisedButton(
                            text="ACEPTAR",
                            on_release= lambda _: self.dialog.dismiss()
                        ),
                    ],
                )
                self.dialog.open()

            elif metadata['notes'] == "":

                self.dialog = MDDialog(
                    text=f"Los comentarios están vacíos. ¿Desea continuar?",
                    buttons=[
                        MDFlatButton(
                            text="CANCELAR",
                            on_release= lambda _: self.dialog.dismiss()
                        ),
                        MDRaisedButton(
                            text="ACEPTAR",
                            on_release= lambda _: (save(), self.dialog.dismiss())
                        ),
                    ],
                )
                self.dialog.open()

            else:

                save()

        except Exception as e:
            Logger.error(e)
            self.session.rollback()

    def copy_exam(self, exam_id):
        exam = self.session.query(ldb.Exam).filter(ldb.Exam.id == exam_id).one()
        make_transient(exam)
        exam.id = None
        exam.remote_id = None
        exam.sent = None
        exam.status = "GUARDADO"
        self.session.add(exam)
        self.session.commit()
        Logger.info(f"COPY: Exam {exam.id} with status {exam.status}")

    def send_exam(self, exam_id):

        exam = self.session.query(ldb.Exam).filter(ldb.Exam.id == exam_id).one()

        def send_diagnostic(diagnostic:str):
            try:
                timestamp = datetime.now().replace(microsecond=0)
                updated_case = {'diagnostic':diagnostic,
                                'diagnosed':timestamp.isoformat(),
                                'modified':timestamp.isoformat(),
                                'status':'EVALUADO'
```

```
                        }
                rdb.update_object('Cases',exam.remote_id,updated_case)
                updated_case['diagnosed'] = timestamp
                updated_case['modified'] = timestamp
                for key, value in updated_case.items():
                    setattr(exam, key, value)
                self.session.commit()
                self.root.ids.exam_metadata.populate(exam_id)

            except SQLAlchemyError as e:
                Logger.error(e)
                toast("Error en la base de datos.")

            except rdb.RDBException as e:
                Logger.error(e)
                toast(str(e))

            except Exception as e:
                toast("Error desconocido")
                Logger.error(f"Exam:{e}")

            finally:
                self.session.rollback()
                self.dialog.dismiss()

        def send_to_firebase(global_id:str):
            Logger.debug(f"Global ID: {global_id}")
            try:
                timestamp = datetime.now().replace(microsecond=0)
                remote_exam_id = rdb.create_object("Cases",
                    {
                        'exam_id':exam.id,
                        'origin_id':exam.origin_id,
                        'status':"ENVIADO",
                        'destination_id':global_id,
                        'created':exam.created.isoformat(),
                        'modified':timestamp.isoformat(),
                        'sent':timestamp.isoformat(),
                        'diagnosed':'',
                        'patient_first_name':exam.patient.first_name,
                        'patient_last_name':exam.patient.last_name,
                        'patient_birth_date':exam.patient.birth_date.strftime('%d-%m-%Y'),
                        'patient_identification':exam.patient.identification,
                        'patient_record':exam.patient.record,
                        'patient_sex':exam.patient.sex,
                        'patient_address':exam.patient.address,
                        'pressure':exam.pressure,
                        'spo2':exam.spo2,
                        'weight_pd':exam.weight_pd,
                        'sample_rate':exam.ekg.sample_rate,
                        'bpm':exam.ekg.bpm,
                        'leads':exam.ekg.leads,
                        'gain':exam.ekg.gain,
                        'rpeaks':array.array('I',exam.ekg.rpeaks).tolist(),
                        'signal':list(exam.ekg.signal),
                        'notes':exam.notes,
                        'diagnostic':exam.diagnostic
                    }
                )
                exam.status = "ENVIADO"
                exam.sent = timestamp
                exam.modified = timestamp
                exam.remote_id = remote_exam_id
                self.session.commit()
                return remote_exam_id

            except SQLAlchemyError as e:
                Logger.error(f"SQL Error: {e}")
```

```
            except rdb.RDBException as e:
                Logger.error(e)
                toast(str(e))

            except Exception as e:
                Logger.error(e)
                toast("Error desconocido.")

            finally:
                self.session.rollback()

        def send_callback(global_id):
            tic = perf_counter()
            remote_id = send_to_firebase(global_id)
            toc = perf_counter()
            Logger.info(f"[Send] Time elapsed: {toc - tic} s")
            if remote_id is not None:
                self.dialog.dismiss()
                self.root.ids.screen_manager.get_screen("ekg_detail_view").title = exam.name
                exam.destination_id = global_id
                self.session.commit()
                self.root.ids.exam_metadata.populate(exam_id)
                Logger.debug(f"Remote ID:{remote_id}")
                return
            self.dialog.dismiss()

        if "GUARDADO" != exam.status and self.store['app']['mode'] == 'C':
            toast(f"Este examen ya fue enviado.")
            return

        if self.store['app']['mode'] != 'H':
            hosp_list = self.session.query(ldb.Hospital).order_by(ldb.Hospital.name).all()
            Logger.debug([hosp.global_id for hosp in hosp_list])
            self.dialog = MDDialog(
                title="Hospital de destino",
                type="simple",
                items=[
                    OKHospitalSelectorItem(text=item.name, secondary_text=item.location,
global_id=item.global_id, on_release=lambda x: send_callback(x.global_id)) for item in hosp_list
                ]
            )
        else:
            self.dialog = MDDialog(
                title="Diagnóstico",
                type="custom",
                content_cls=OKCommentWidget(hint="Agregue su diagnóstico"),
                buttons=[
                    MDFlatButton(
                        text="CANCELAR",
                        on_release=lambda _: self.dialog.dismiss()
                    ),
                    MDRaisedButton(
                        text="GUARDAR",
                        on_release=lambda _:
send_diagnostic(self.dialog.content_cls.ids.text_field.text)
                    )
                ]
            )
        self.dialog.open()

    def delete_patient(self, patient_id):

        def delete_from_db():
            obj = self.session.query(ldb.Patient).get(patient_id)
            try:
                self.session.delete(obj)
                self.session.commit()
                Logger.info(f"Patient {patient_id} deleted successfully.")
            except SQLAlchemyError as e:
```

```
                self.session.rollback()
                Logger.error(e)

        def delete_callback():
            delete_from_db()
            self.dialog.dismiss()
            self.go_back("patient_list_view")

        self.dialog = MDDialog(
            text="¿Estás seguro(a) que deseas borrar este paciente?",
            buttons=[
                MDFlatButton(
                    text="CANCELAR",
                    on_release= lambda _: self.dialog.dismiss()
                ),
                MDRaisedButton(
                    text="ACEPTAR",
                    on_release= lambda _: delete_callback()
                ),
            ],
        )
        self.dialog.open()

    def delete_exam(self, exam_id):

        def delete_from_db():
            try:
                obj = self.session.query(ldb.Exam).get(exam_id)
                self.session.delete(obj)
                self.session.commit()
                Logger.info(f"Exam {exam_id} deleted successfully.")
            except SQLAlchemyError as e:
                self.session.rollback()
                Logger.error(e)

        def delete_callback():
            delete_from_db()
            self.dialog.dismiss()
            self.go_back("ekg_list_view")

        self.dialog = MDDialog(
            text="¿Estás seguro(a) que deseas borrar este examen?",
            buttons=[
                MDFlatButton(
                    text="CANCELAR",
                    on_release= lambda _: self.dialog.dismiss()
                ),
                MDRaisedButton(
                    text="ACEPTAR",
                    on_release= lambda _: delete_callback()
                ),
            ],
        )
        self.dialog.open()

    def populate_start(self):
        unopened_exams = self.session.query(ldb.Exam).filter(ldb.Exam.unopened == True).count()
        info_msg = f"Usted está registrado como {self.store['user']['id']}. Tiene {unopened_exams}
exámen(es) sin abrir."
        self.root.ids["info"].text = info_msg
        field = ldb.Exam.destination_id if (app.store['app']['mode'] == 'H') else
ldb.Exam.origin_id
        exam_count = self.session.query(ldb.Exam).filter(field == app.store['user']['id']).count()
        patient_count = self.session.query(ldb.Patient).count()
        self.root.ids["start_exams"].count = exam_count
        self.root.ids["start_patients"].count = patient_count
        self.root.ids["start_exams"].clear()
        if self.store['app']['mode'] == 'C':
            self.root.ids["start_exams"]\
```

```
                .add_bar("GUARDADO","#d35f5f",self.session.query(ldb.Exam)\
                .filter(ldb.Exam.status == "GUARDADO")\
                .filter(field == app.store['user']['id'])\
                .count()/max(exam_count,1))
            self.root.ids["start_exams"]\
                .add_bar("ENVIADO","#5f99d3",self.session.query(ldb.Exam)\
                .filter(ldb.Exam.status == "ENVIADO")\
                .filter(field == app.store['user']['id'])\
                .count()/max(exam_count,1))
            self.root.ids["start_exams"]\
                .add_bar("EVALUADO","#5fd399",self.session.query(ldb.Exam)\
                .filter(ldb.Exam.status == "EVALUADO")\
                .filter(field == app.store['user']['id'])\
                .count()/max(exam_count,1))
        else:
            self.root.ids["start_exams"]\
                .add_bar("RECIBIDO","#5f99d3",self.session.query(ldb.Exam)\
                .filter(ldb.Exam.status == "ENVIADO")\
                .filter(field == app.store['user']['id'])\
                .count()/max(exam_count,1))
            self.root.ids["start_exams"]\
                .add_bar("EVALUADO","#5fd399",self.session.query(ldb.Exam)\
                .filter(ldb.Exam.status == "EVALUADO")\
                .filter(field == app.store['user']['id'])\
                .count()/max(exam_count,1))

        self.root.ids["start_patients"].clear()
        self.root.ids["start_patients"].add_bar("FEMENINO","#d35f5f",self.session.query(ldb.Patien
t).filter(ldb.Patient.sex == "F").count()/max(patient_count,1))
        self.root.ids["start_patients"].add_bar("MASCULINO","#5f99d3",self.session.query(ldb.Patie
nt).filter(ldb.Patient.sex == "M").count()/max(patient_count,1))

    def refresh_home(self):
        self.download_cases()
        self.populate_hospitals()
        self.populate_start()

async def main(app):
    await asyncio.gather(app.async_run("asyncio"), app.ble.connection_handler())

if __name__ == "__main__":
    Logger.setLevel(logging.INFO)
    # app running on one thread with two async coroutines
    app = OpenKardioApp()
    asyncio.run(main(app))
```

## H4. exam.py

```
from kivy.logger import Logger
from sqlalchemy.exc import SQLAlchemyError
import utils.localdb as ldb
import utils.ble as ble
import numpy as np
import utils.utils as utils
from kivy.uix.widget import Widget
from kivy.core.text import Label as CoreLabel
from kivy.graphics import Color, Line, Rectangle
from kivy.properties import DictProperty, StringProperty, ListProperty, ObjectProperty,
BooleanProperty, NumericProperty, ColorProperty
from kivymd.app import MDApp
from kivymd.uix.boxlayout import MDBoxLayout
from kivymd.toast import toast
from kivy.metrics import dp
from kivy.utils import platform
import zlib
import array
```

```
class Plot(Widget):
    sample_rate = NumericProperty(480)
    top_of_scale = NumericProperty(26400)
    gain = NumericProperty(1)
    run_samples = BooleanProperty(False)
    ekg_samples = ListProperty([])
    next_samples = ListProperty([])
    bpm = NumericProperty(0)
    MARGINS = dp(16)
    GRID_SUBDIVS = 5
    SEC_PER_SUBDIV = 0.04
    MV_PER_SUBDIV = 0.1
    SEC_PER_DIV = GRID_SUBDIVS*SEC_PER_SUBDIV
    MV_PER_DIV = GRID_SUBDIVS*MV_PER_SUBDIV
    GRID_SUBDIV_LINE_WIDTH = 0.5
    GRID_DIV_LINE_WIDTH = 1.2
    app = MDApp.get_running_app()

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Clock.schedule_interval(self.update_plot,1/60.0)
        self.time_generator = utils.time_gen(self.sample_rate)
        self.peaks = []
        with self.canvas.after:
            Color(0, 0, 0, 1)
            self.preamble = Line(points=[], width = 1.75 if platform == 'android' else 1)
            self.line = Line(points=[], width = 1.75 if platform == 'android' else 1)

    def on_sample_rate(self, instance, value):
        self.time_generator = utils.time_gen(self.sample_rate)

    def on_size(self,*args):
        try:
            self.grid_y_divs = 6

            self.y_subdiv_count = self.grid_y_divs*self.GRID_SUBDIVS
            self.subdiv_size = int((self.height - self.MARGINS)//(self.y_subdiv_count))
            self.div_size = self.subdiv_size*self.GRID_SUBDIVS

            self.grid_height = self.div_size*self.grid_y_divs
            self.grid_y_offset = (self.height - self.grid_height)/2
            self.grid_y = self.y + self.grid_y_offset
            self.grid_top = self.grid_y + self.grid_height

            self.grid_x_divs = int((self.width - self.MARGINS)//(self.div_size))
            self.grid_width = self.grid_x_divs*self.div_size
            self.grid_x = self.x + self.MARGINS
            self.grid_right = self.grid_x + self.grid_width
            self.canvas.clear()
            self.plot_grid()
        except ZeroDivisionError:
            pass

    def on_ekg_samples(self,*args):
        self.width =
max((len(self.ekg_samples)*self.subdiv_size/(self.sample_rate*self.SEC_PER_SUBDIV))+8*self.MARGINS
, self.parent.width)

    def on_next_samples(self, *args):
        if len(self.next_samples):
            if len(self.preamble.points) == 0:
                pre_points = [float(point) for sample in range(193) for point in [self.grid_x +
next(self.time_generator)*self.subdiv_size/self.SEC_PER_SUBDIV,
                                                    self.grid_y_offset + np.interp(13200 if (sample <=
38 or sample > 96) else 19780,[0,self.top_of_scale],[0,self.grid_height])]]
                pre_points += [self.grid_x +
next(self.time_generator)*self.subdiv_size/self.SEC_PER_SUBDIV,self.grid_y_offset +
np.interp(13200,[0,self.top_of_scale],[0,self.grid_height])]
                self.preamble.points = pre_points
```

```
                self.line.points = pre_points

            next_points = [float(point) for sample in self.next_samples for point in [self.grid_x
+ next(self.time_generator)*self.subdiv_size/self.SEC_PER_SUBDIV,
                                                    self.grid_y_offset + np.interp(sample/self.gain,[-
(self.grid_y_divs*self.MV_PER_DIV)/2,(self.grid_y_divs*self.MV_PER_DIV)/2],[0,self.grid_height])]]

            if len(self.line.points) <= 2*4632: #max points per line
                self.line.points += next_points

            self.ekg_samples.extend(self.next_samples)
            self.next_samples = []

            if self.peaks:

                for i, peak in enumerate(self.peaks):

                    if i == 0:
                        continue

                    x_peak = self.preamble.points[-2] +
(peak/self.sample_rate)*self.subdiv_size/self.SEC_PER_SUBDIV
                    mylabel = CoreLabel(text=f"{int(400.0+peak*1000/self.sample_rate)} ms",
font_size=dp(10), color=(0, 0, 0, 1))
                    mylabel.refresh()
                    texture = mylabel.texture
                    texture_size = list(texture.size)
                    with self.canvas:
                        Color(0.1,0.1,0.1,0.2)
                        Line(points=[x_peak,self.grid_y,x_peak,self.y_subdiv_count*self.subdiv_siz
e + self.grid_y], width=2)
                        Color(0.1,0.1,0.1,1)
                        Rectangle(texture=texture, size=texture_size, pos=[x_peak,0])

                self.peaks = []

    def plot_grid(self, *args):
        mylabel = CoreLabel(text="0.2 s/div - 0.5 mV/div", font_size=dp(10), color=(0, 0, 0, 1))
        # Force refresh to compute things and generate the texture
        mylabel.refresh()
        # Get the texture and the texture size
        texture = mylabel.texture
        texture_size = list(texture.size)
        with self.canvas.before:
            Color(rgba=[1.0, 0.85, 0.85, 1])
            Rectangle(pos=self.pos, size=self.size)
        with self.canvas:
            # eff_height = round(self.height/amp_divisions)*10 + 1
            Color(rgba=[0.86,0.59,0.59,0.9])
            try:
                for i in range(0, self.GRID_SUBDIVS*self.grid_x_divs + 1):
                    # vertical lines
                    pos = i*self.subdiv_size + self.grid_x
                    line_width = self.GRID_SUBDIV_LINE_WIDTH if (i % self.GRID_SUBDIVS > 0) else
self.GRID_DIV_LINE_WIDTH
                    Line(points=[pos, self.grid_y, pos, self.grid_top], width=line_width)
                for i in range(0, self.y_subdiv_count + 1):
                    # horizontal lines
                    pos = i*self.subdiv_size + self.grid_y
                    line_width = self.GRID_SUBDIV_LINE_WIDTH if (i % self.GRID_SUBDIVS > 0) else
self.GRID_DIV_LINE_WIDTH
                    Line(points=[self.grid_x, pos, self.grid_right, pos], width=line_width)
                Rectangle(texture=texture, size=texture_size, pos=[self.grid_x,0])
            except AttributeError:
                pass

    def reset(self):
        self.canvas.clear()
        self.ekg_samples = []
```

```
        self.next_samples = []
        self.line.points = []
        self.preamble.points = []
        self.time_generator = utils.time_gen(self.sample_rate)
        self.plot_grid()

    def populate(self, ekg_id):
        self.reset()
        try:
            ekg = self.app.session.query(ldb.Ekg).filter(ldb.Ekg.id == ekg_id).one()
            self.sample_rate = ekg.sample_rate
            self.gain = ekg.gain
            signal = array.array('h',zlib.decompress(ekg.signal)).tolist()
            self.peaks = utils.christov_detector(signal,self.sample_rate)
            self.next_samples = signal
        except Exception as e:
            Logger.error(e)

    def get_ekg(self):
        if len(self.ekg_samples):
            Logger.info(f"EKG: {len(self.ekg_samples)} samples")
            return {
                'bpm':
round(60/np.average(np.diff(utils.christov_detector(np.array(self.ekg_samples),self.sample_rate)[1
:])/self.sample_rate)),
                'sample_rate': self.sample_rate,
                'gain': self.app.ble.conv_factor,
                'rpeaks':
bytes(array.array('I',utils.christov_detector(self.ekg_samples,self.sample_rate))),
                'signal':
zlib.compress(bytearray(array.array('h',list(self.ekg_samples)[:int(self.sample_rate*9.6)])))
            }
        return {
                'bpm': round(60/np.average(np.diff(utils.christov_detector(np.array([item*16 for
item in
        utils.single_pulse*10])-936,self.sample_rate)[1:])/self.sample_rate)),
                'sample_rate': self.sample_rate,
                'gain': 8800,
                'rpeaks': bytes(array.array('I',utils.christov_detector([(item-936)*16 for item in
        utils.single_pulse*10],self.sample_rate))),
                'signal': zlib.compress(bytearray(array.array('h',[(item-936)*16 for item in
        utils.single_pulse*10])))
            }

class ExamList(MDBoxLayout):
    def populate(self, text="", search=False, diagnosed=False):
        '''Builds a list of icons for the screen MDIcons.'''

        def add_item(instance):
            self.ids.rv.data.append(
                {
                    "viewclass": "OKListItem",
                    "object_id": instance.id,
                    "text": instance.name,
                    "secondary_text": instance.patient.first_name + " " +
instance.patient.last_name,
                    "tertiary_text": f"Global ID: {instance.remote_id}",
                    "avatar": "new-box" if instance.unopened else "",
                    "icon": "clipboard-pulse",
                    "screen": "ekg_detail_view",
                    "propagate": True
                }
            )

        self.ids.rv.data = []
        app = MDApp.get_running_app()
        field = ldb.Exam.destination_id if (app.store['app']['mode'] == 'H') else
ldb.Exam.origin_id
        status_list = ["EVALUADO"] if diagnosed else ["GUARDADO","ENVIADO"]
```

```
        query = app.session.query(ldb.Exam)\
            .filter(field == app.store['user']['id'])\
            .filter(ldb.Exam.status.in_(status_list))\
            .order_by(-ldb.Exam.id)
        avatar = ""
        for instance in query:
            if search:
                if text.lower() in instance.patient.first_name.lower() or\
                    text.lower() in instance.patient.last_name.lower() or\
                    text.lower() in instance.patient.record.lower() or \
                    text.lower() in str(instance.remote_id):
                    add_item(instance)
            else:
                add_item(instance)
            if instance.unopened:
                avatar = "circle-small"
        return avatar

class PatientSelector(MDBoxLayout):
    def populate(self, text="", search=False):
        '''Builds a list of icons for the screen MDIcons.'''

        def add_patient_item(instance):
            self.ids.rv.data.append(
                {
                    "viewclass": "OKSelectorItem",
                    "object_id": instance.id,
                    "text": instance.first_name + " " + instance.last_name,
                    "secondary_text": f"Expediente no.: {instance.record}",
                    "icon": MDApp.get_running_app().icons.get(instance.sex),
                    "screen": "ekg_create_view"
                }
            )

        self.ids.rv.data = []
        app_session = MDApp.get_running_app().session
        for instance in app_session.query(ldb.Patient).order_by(ldb.Patient.id):
            if search:
                if text in instance.first_name or text in instance.last_name or text in
instance.record:
                    add_patient_item(instance)
            else:
                add_patient_item(instance)

class ExamMetadataForm(MDBoxLayout):
    data = DictProperty({})
    title = StringProperty("Title")

    def populate(self, patient_id):
        app_session = MDApp.get_running_app().session
        self.title = "Datos complementarios"

        try:
            self.patient = app_session.query(ldb.Patient).filter(ldb.Patient.id ==
patient_id).one()
            self.ids.info.text = f"{self.patient.first_name} {self.patient.last_name}"
            self.ids.info.icon = MDApp.get_running_app().icons.get(self.patient.sex[0])
            self.data['patient_id'] = patient_id
            self.ids.spo2.text = ""
            self.ids.weight_pd.text = ""
            self.ids.pressure.text = ""
            self.ids.notes.text = ""
        except SQLAlchemyError as e:
            Logger.error(e)
            toast('Hubo un error al buscar el paciente.')

    def save(self):
        self.data["spo2"] = (lambda s: 0.0 if s == "" else (float(s) if s.replace('.', '',
1).isdigit() else None))(self.ids.spo2.text)
```

```
        self.data["weight_pd"] = (lambda s: 0.0 if s == "" else (float(s) if s.replace('.', '',
1).isdigit() else None))(self.ids.weight_pd.text)
        self.data["pressure"] = (lambda s: "--/--" if s == "" else (s if all(part.isdigit() for
part in s.split('/')) else None))(self.ids.pressure.text)
        self.data["notes"] = self.ids.notes.text
        return self.data

class OKDevicePanel(MDBoxLayout):
    title = StringProperty("Title")
    sample_rate = NumericProperty(0)
    leads = NumericProperty(0)
    battery = NumericProperty(0)
    resolution = NumericProperty(0)
    fw_version = StringProperty("")
    button_color = ColorProperty("blue")
    button_text = StringProperty("CONECTAR")
    ble_state = ObjectProperty()
    status = StringProperty("Desconectado")
    duration = NumericProperty(10)
    app = MDApp.get_running_app()

    def set_state(self):
        if self.app.ble.state == ble.ConnState.IDLE:
            self.app.ble.transition(ble.ConnState.SCANNING)
        elif self.app.ble.state == ble.ConnState.SCANNING:
            pass
        elif self.app.ble.state == ble.ConnState.CONNECTED:
            self.app.ble.transition(ble.ConnState.RECEIVING)
        elif self.app.ble.state == ble.ConnState.RECEIVING:
            self.app.ble.transition(ble.ConnState.CONNECTED)

    def on_ble_state(self, instance, value):
        Logger.info(f"OKDevice: {self.ble_state}")
        if value == ble.ConnState.IDLE:
            self.button_text = "CONECTAR"
            self.button_color = "blue"
            self.status = "Desconectado"
        elif value == ble.ConnState.SCANNING:
            self.status = "Escaneando..."
        elif value == ble.ConnState.CONNECTED:
            self.button_text = "EMPEZAR"
            self.button_color = "green"
            self.status = "Conectado"
        elif value == ble.ConnState.RECEIVING:
            self.button_text = "DETENER"
            self.button_color = "red"
            self.status = "Transmitiendo"

    def on_sample_rate(self, instance, value):
        self.ids.sample_rate.text = str(value) + " sps"

    def on_leads(self, instance, value):
        self.ids.leads.text = str(value) + " leads"

    def on_battery(self, instance, value):
        self.ids.battery.text = str(value) + " %"

    def on_resolution(self, instance, value):
        self.ids.resolution.text = str(value) + " bits"

    def on_fw_version(self, instance, value):
        self.ids.fwv.text = "v" + value
```

## Anexo I. Código fuente desplegado en Firebase Functions.

### I1. Funciones Lambda implementadas.

```
/* eslint-disable camelcase */
/* eslint-disable require-jsdoc */
/* eslint-disable max-len */
import * as functions from "firebase-functions";
import * as admin from "firebase-admin";
import * as nodemailer from "nodemailer";
import * as canvas from "canvas";
import * as zlib from "zlib";
import * as moment from "moment-timezone";
import * as cors from "cors";

admin.initializeApp();

function parseDateString(dateString: string): Date {
  const parts = dateString.split("-");

  const day = parseInt(parts[0]);
  const month = parseInt(parts[1]) - 1; // Months in JavaScript are 0-indexed
  const year = parseInt(parts[2]);

  return new Date(year, month, day);
}

function calculateAge(birthdate: Date): number {
  const currentDate = new Date();

  const yearsDiff = currentDate.getFullYear() - birthdate.getFullYear();
  const currentMonth = currentDate.getMonth();
  const birthMonth = birthdate.getMonth();

  // Adjust age based on birth month
  if (currentMonth < birthMonth || (currentMonth === birthMonth && currentDate.getDate() <
birthdate.getDate())) {
    return yearsDiff - 1;
  }

  return yearsDiff;
}

function normalizeIntArray(array: Int16Array, gain: number, pixelsPerMv: number): Float32Array {
  const normalizedArray = new Float32Array(array.length);

  for (let i = 0; i < array.length; i++) {
    normalizedArray[i] = (array[i]*pixelsPerMv) / gain;
  }

  return normalizedArray;
}

function generateTimestamp(): string {
  // Get the current timestamp in UTC
  const utcTimestamp = Date.now();

  // Convert to the desired timezone (UTC-6)
  const utcMinus6Timestamp = moment(utcTimestamp).tz("America/Managua");

  // Format the timestamp in ISO format
  const isoTimestamp = utcMinus6Timestamp.format("YYYY-MM-DDTHH:mm:ss");

  return isoTimestamp;
```

```
}

exports.sendNotificationOnCreation = functions.database
  .ref("Cases/{caseId}")
  .onCreate(async (snapshot, context) => {
    const caseId = context.params.caseId;
    const caseData = snapshot.val();
    console.log(`New case ${caseId}`);

    // Get the destination ID
    const destId = caseData.destination_id;
    const origId = caseData.origin_id;
    const created = caseData.created;
    const patientName = caseData.patient_first_name + " " + caseData.patient_last_name;
    const patientRecord = caseData.patient_record;
    const patientAge = calculateAge(parseDateString(caseData.patient_birth_date)).toString();
    // const patientIdentification = caseData.patient_identification;
    const sampleRate = caseData.sample_rate;
    const spo2 = caseData.spo2;
    const weight = caseData.weight_pd;
    const pressure = caseData.pressure;
    const bpm = caseData.bpm;
    const signalGain = caseData.gain;
    const notes = caseData.notes;
    const rPeaks = caseData.rpeaks;

    // Graph Constants
    const graphMargin = 20;
    const graphWidth = 2500;
    const graphHeight = 300;
    const pixelsPerSubdiv = 10;

    const compressedSignalBytes = caseData.signal;
    const compressedSignalBuffer = Buffer.from(compressedSignalBytes);
    const decompressedSignalBuffer = zlib.inflateSync(compressedSignalBuffer);
    const originalSignal = new Int16Array(decompressedSignalBuffer.buffer).slice(0,
Math.trunc(decompressedSignalBuffer.length/2));
    const normalizedSignal = normalizeIntArray(originalSignal, signalGain, 10*pixelsPerSubdiv);

    console.log("Compressed bytes length:", compressedSignalBytes.length);
    // console.log("Compressed buffer length:", compressedSignalBuffer.byteLength);
    // console.log("Decompressed length:", decompressedSignalBuffer.buffer.byteLength);
    // console.log("Original length:", new Int16Array(decompressedSignalBuffer.buffer));
    // console.log("Original Minimum:", Math.min(...originalSignal));
    // console.log("Original Maximum:", Math.max(...originalSignal));
    // console.log("Normalized Minimum:", Math.min(...normalizedSignal));
    // console.log("Normalized Maximum:", Math.max(...normalizedSignal));
    console.log("Peaks:", rPeaks);

    // Create a canvas
    const graphCanvas = canvas.createCanvas(graphWidth + 2*graphMargin, graphHeight +
2*graphMargin);
    const ctx = graphCanvas.getContext("2d");
    ctx.fillStyle = "rgb(255,217,217)";
    ctx.fillRect(0, 0, graphWidth + 2*graphMargin, graphHeight + 2*graphMargin);

    ctx.strokeStyle = "rgba(220,153,153,0.9)";

    for (let i = 0; i <= Math.floor(graphWidth/pixelsPerSubdiv); i++) {
      // Vertical Lines
      ctx.beginPath();
      ctx.lineWidth = i%5 == 0 ? 2 : 0.75;
      ctx.moveTo(i*pixelsPerSubdiv+graphMargin, graphMargin);
      ctx.lineTo(i*pixelsPerSubdiv+graphMargin, graphHeight + graphMargin);
      ctx.stroke();
    }
    for (let i = 0; i <= Math.floor(graphHeight/pixelsPerSubdiv); i++) {
      // Horizontal Lines
      ctx.beginPath();
```

```
      ctx.lineWidth = i%5 == 0 ? 2 : 0.75;
      ctx.moveTo(graphMargin, i*pixelsPerSubdiv+graphMargin);
      ctx.lineTo(graphWidth + graphMargin, i*pixelsPerSubdiv+graphMargin);
      ctx.stroke();
    }

    ctx.strokeStyle = "rgba(25,25,25,0.8)";
    ctx.fillStyle = "rgb(0,0,0)";
    ctx.font = "11pt sans-serif";
    for (let i = 1; i < rPeaks.length - 1; i++) {
      ctx.beginPath();
      ctx.lineWidth = 3;
      ctx.moveTo(((rPeaks[i + 1] + 193)*(graphCanvas.width - 2*graphMargin) / (sampleRate*10.0)) +
graphMargin, graphHeight + graphMargin + 2);
      ctx.lineTo(((rPeaks[i + 1] + 193)*(graphCanvas.width - 2*graphMargin) / (sampleRate*10.0)) +
graphMargin, graphHeight + graphMargin + 17);
      ctx.stroke();
      if (i > 1) {
        ctx.fillText(`${Math.floor((rPeaks[i + 1] - rPeaks[i])*1000/sampleRate)} ms`, (rPeaks[i] +
0.5*(rPeaks[i + 1] - rPeaks[i]) + 193)*(graphCanvas.width - 40) / (sampleRate*10.0),
graphCanvas.height - 4);
      }
    }

    ctx.strokeStyle = "black";
    ctx.lineWidth = 1.9;
    ctx.beginPath();
    ctx.moveTo(20, graphCanvas.height/2);
    for (let i = 0; i < 193; i++) {
      ctx.lineTo(i * ((graphWidth) / (sampleRate*10)) + graphMargin, i <= 38 || i > 96 ?
graphCanvas.height/2 : graphCanvas.height/2 - 10*pixelsPerSubdiv);
    }
    for (let i = 1; i < normalizedSignal.length; i++) {
      ctx.lineTo(i * ((graphCanvas.width - 40) / (sampleRate*10)) + 20 + 100, graphCanvas.height -
20 - (normalizedSignal[i] + graphHeight/2));
    }
    ctx.stroke();

    ctx.fillStyle = "rgb(0,0,0)";
    ctx.font = "11pt sans-serif";
    ctx.fillText(`Paciente: ${patientName} |  Caso #${caseId}  |  Fecha: ${created}`, 20, 16);
    ctx.fillText("0.2 s/div - 0.5 mV/div\t\tIntervalos RR ->", 20, graphCanvas.height - 4);

    // Convert canvas to PNG buffer
    const graphImageBuffer = graphCanvas.toBuffer();

    // Get the email address
    const emailRef = admin.database().ref(`Hospitals/${destId}/email`);
    const emailSnapshot = await emailRef.once("value");
    const email = emailSnapshot.val();

    // Send email notification
    const transporter = nodemailer.createTransport({
      service: "gmail",
      auth: {
        user: functions.config().gmail.address,
        pass: functions.config().gmail.password,
      },
    });

    const htmlContent = `
<html>
<head>
  <style>
    .button {
      display: inline-block;
      background-color: #afcce9;
      color: #0e0e0e;
      padding: 10px 20px;
```

```
      text-decoration: none;
      border-radius: 5px;
    }
  </style>
</head>
<body>

  <p><span style="font-family:Verdana,Geneva,sans-serif"><strong>NUEVO EXAMEN</strong></span></p>

  <p>☑<span style="font-family:Verdana,Geneva,sans-serif">Al final de este correo encontrará el
enlace hacia el formulario de diagnóstico.</span></p>

  <p><span style="font-family:Verdana,Geneva,sans-serif"><strong>DATOS DEL
PACIENTE</strong></span></p>

  <table border="1" cellpadding="1" cellspacing="1" style="width:450px">
  <tbody>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-serif"><strong>Unidad de
Salud</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-
serif">${origId}</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-serif"><strong>Nombre del
paciente</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-
serif">${patientName}</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-
serif"><strong>Edad</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-serif">${patientAge}
años</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-
serif"><strong>Expediente</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-
serif">${patientRecord}</span></td>
  </tr>
  </tbody>
  </table>

  <p> </p>

  <p><span style="font-family:Verdana,Geneva,sans-serif"><strong>DETALLES DEL
EXAMEN</strong></span></p>

  <table border="1" cellpadding="1" cellspacing="1" style="width:438.6px">
  <tbody>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-serif"><strong>ID del
Caso</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-
serif">${caseId}</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-serif"><strong>Fecha y
Hora</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-
serif">${created}</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-
serif"><strong>SpO2</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-serif">${spo2.toString()}
%</span></td>
  </tr>
```

```
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-
serif"><strong>Peso</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-serif">${weight.toString()}
lbs</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-
serif"><strong>Presión</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-
serif">${pressure}</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-serif"><strong>Ritmo
Cardiaco</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-serif">${bpm.toString()}
bpm</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-
serif"><strong>Derivación</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-serif">II</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-serif"><strong>Tasa de
muestreo</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-
serif">${sampleRate.toString()} sps</span></td>
  </tr>
  <tr>
  <td style="width:130px"><span style="font-family:Verdana,Geneva,sans-
serif"><strong>Comentarios</strong></span></td>
  <td style="width:320px"><span style="font-family:Verdana,Geneva,sans-serif">${notes}</span></td>
  </tr>
  </tbody>
  </table>

  <p> </p>
  <a class="button" href="${functions.config().form.url}/diagnosis-
form.html?examId=${caseId}"><strong>Diagnosticar examen</strong></a>
  <p> </p>
  <p><span style="font-family:Verdana,Geneva,sans-serif; font-size: 10px; color:
#2c2c2c">OpenKardio es una plataforma de Telecardiografía digital en fase de desarrollo. Su uso
aún no está recomendando para diagnóstico clínico.</span></p>
</body>
    `;

    let origEmail = "";

    try {
      const hcSnapshot = await admin.database().ref(`/HCenters/${origId}`).once("value");
      const hcData = hcSnapshot.val();

      if (hcData) {
        origEmail = hcData.email;
      } else {
        console.error("Health Center not found");
      }
    } catch (error) {
      console.error("Error getting Health center:", error);
    }

    const mailOptions = {
      from: functions.config().gmail.address,
      to: email,
      replyTo: origEmail.length ? origEmail : functions.config().gmail.address,
      subject: `Nuevo Electrocardiograma para ${destId}`,
      html: htmlContent,
      attachments: [{
```

```
        filename: `ekg_${caseId}.png`,
        content: graphImageBuffer,
      }],
    };

    await transporter.sendMail(mailOptions);

    console.log("Email sent successfully");
  });

// Firebase Cloud Function to handle diagnosis submission
exports.submitDiagnosis = functions.https.onRequest(async (req, res) => {
  cors({origin: true})(req, res, async () => {
    const examId = req.body.examId;
    const diagnosis = req.body.diagnosis;
    const timestamp = generateTimestamp();

    const updateObj = {
      diagnostic: diagnosis,
      status: "EVALUADO",
      modified: timestamp,
      diagnosed: timestamp,
    };

    // Update the diagnosis field in the Realtime Database
    const examRef = admin.database().ref(`/Cases/${examId}`);
    examRef.update(updateObj)
      .then(() => {
        res.status(200).send("Diagnosis submitted successfully");
      })
      .catch((error) => {
        console.error("Error receiving diagnostic:", error);
        res.status(500).send("Error submitting diagnosis");
      });

    try {
      const origId = (await examRef.once("value")).val().origin_id;
      const hcSnapshot = await admin.database().ref(`/HCenters/${origId}`).once("value");
      const hcData = hcSnapshot.val();

      if (hcData) {
        const origEmail = hcData.email;
        // Send email notification
        const transporter = nodemailer.createTransport({
          service: "gmail",
          auth: {
            user: functions.config().gmail.address,
            pass: functions.config().gmail.password,
          },
        });

        const htmlContent = `
        <p><span style="font-family:Verdana,Geneva,sans-serif"><strong>DIAGNÓSTICO DEL EXAMEN
${examId}</strong></span></p>
        <p> </p>
        <p><span style="font-family:Verdana,Geneva,sans-serif">${diagnosis}</span></p>
        <p> </p>
        <p><span style="font-family:Verdana,Geneva,sans-serif">Recibido el
${timestamp}.</span></p>`;

        const mailOptions = {
          from: functions.config().gmail.address,
          to: origEmail,
          subject: `Nuevo Diagnóstico para ${origId}`,
          html: htmlContent,
        };

        await transporter.sendMail(mailOptions);
```

```
        console.log("Email sent successfully");
      } else {
        console.error("Health Center not found");
      }
    } catch (error) {
      console.error("Error getting Health center:", error);
    }
  });
});

exports.checkDiagnosisStatus = functions.https.onRequest((req, res) => {
  cors({origin: true})(req, res, async () => {
    try {
      const examId = req.query.examId;

      // Fetch exam data from Realtime Database
      const examSnapshot = await admin
        .database()
        .ref(`/Cases/${examId}`)
        .once("value");
      const exam = examSnapshot.val();

      if (!exam) {
        res.status(404).send("Exam not found");
        return;
      }

      const response = {
        isDiagnosed: exam.status === "EVALUADO",
        diagnostic: exam.diagnostic || null,
      };

      res.json(response);
    } catch (error) {
      console.error("Error:", error);
      res.status(500).send("An error occurred");
    }
  });
});
```